Exploiting the Power of Multiprocessors for Real-Time Systems

by
Michele Cirinei

Ph.D. Thesis

Scuola Superiore Sant'Anna, Pisa
2007

Pensavo
che avrei cambiato il mondo
con ciò che scoprivo.

I thought
I would have changed the world
with what I discovered.

Ora so
che il mondo si cambia
anche spegnendo la luce in una stanza vuota.

Now I know
that the world is changed
even turning off the light in an empty room.

Continuerò
a cambiare il mondo
a modo mio.

I'll continue
to change the world
my way.

Thanks to

The Pisa Boss: **Giuseppe Lipari**
The Roma Boss: **Alberto Ferrari**
The USA Boss: **Ted Baker**

The LaTeX Man: **Enrico Bini**,
source of constant help, and author, among the rest, of this document's style

The Pusher: **Marko Bertogna**,
walking by my side in this research, he "pushed" me to improve every day

The whole **ReTiS Lab**

and too many others to remember them one by one:
they just know I'm grateful to them.

And of course my sweet lady: **Alice**

M.C.

Signature Page

Prof. Giuseppe Lipari

Dott. Alberto Ferrari

Prof. Theodore P. Baker

Prof. Giorgio C. Buttazzo

Vita

Michele Cirinei was born on January 12th 1979, at Arezzo in Italy. He attended the "Michelangelo Buonarroti" Accounting High School in Arezzo, class of Programming and Computer Science and he got the diploma in 1998, with the highest grade.

From 1998 to 2003 he attended the Università "Il Cherubino" in Pisa, where, in 2004, he graduated cum laude in *Ingegneria Informatica*. In the same year he entered the Ph.D. program at Scuola Superiore S.Anna. His scholarship was funded by **Parades GEIE**, in Roma.

During 2006, Michele spent six months researching at **Florida State University** in Tallahassee, collaborating with **prof. Theodore P. Baker**.

During his Ph.D. he has been mostly researching with **prof. Giuseppe Lipari** and **dr. Alberto Ferrari**, and collaborated also with **ing. Marko Bertogna** and **dr. Enrico Bini**. His publications are (only papers in journals and in refereed conference proceedings are reported):

- Marko Bertogna, Michele Cirinei and Giuseppe Lipari, *Improved Schedulability Analysis of EDF on Multiprocessor Platforms*, Proceedings of the 17th Euromicro Conference on Real-Time Systems, ECRTS '05, 209–218, July 2005, Palma de Mallorca, Spain. Best Paper Award.
- Marko Bertogna, Michele Cirinei and Giuseppe Lipari, *New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors*, Proceedings of the 9th International Conference On Principles Of Distributed Systems, OPODIS '05, 306–321, December 2005, Pisa, Italy.
- Theodore P. Baker and Michele Cirinei, *A Necessary and Sometimes Sufficient Condition for the Feasibility of Sets of Sporadic Hard-Deadline Tasks*, Proceedings of the 27th Real-Time Systems Symposium, RTSS '06, 178–187, Rio de Janeiro, Brazil.
- Michele Cirinei, Enrico Bini, Giuseppe Lipari and Alberto Ferrari, *A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors*, Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium, IPDPS '06, Long Beach (CA), U.S.A..
- Theodore P. Baker and Michele Cirinei, *A Unified Analysis of Global EDF and fixed-task-priority Schedulability of Sporadic Task Systems on Multiprocessors*, accepted for the Journal of Embedded Computing. To appear.

- Michele Cirinei and Theodore P. Baker, *EDZL Schedulability Analysis*, accepted for the 19$^{\text{th}}$ Euromicro Conference on Real-Time Systems, ECRTS '07, July 2007, Pisa, Italy. To appear.

He defended this thesis on July the 2$^{\text{nd}}$ 2007.

Contents

iii

Signature Page v

Vita vii

List of Figures xi

Notation xiii

Chapter 1. Introduction 1
1. Technology invasion 1
2. Real-time systems 3
3. Taxonomy 9
4. Contributions and summary 15

Chapter 2. Performance Problem: schedulability analysis 17
1. Overview 17
2. System model 18
3. Summary of existing results 22
4. Predictability 27
5. Schedulability analysis 28
6. Schedulability tests 43
7. Improving the analysis through the slack 46
8. Schedulability tests: recursive approach 50
9. Experimental results 53
10. Conclusions 65

Chapter 3. Performance Problem: feasibility analysis 67
1. Overview 67
2. System model 69
3. The maxmin load 70
4. How to compute maxmin demand 76
5. How to approximate maxmin load 78
6. Empirical performance 83
7. Improvements in simulations 86
8. Conclusion 87

Chapter 4. Integrity Problem: the partitioned approach 89
1. Overview 89
2. System model 92
3. Design methodology 97
4. Example of Application 103

5. Partitioning tasks among processors 106
6. Conclusions 108

Chapter 5. Conclusions 109
1. Performance 109
2. Integrity 110
3. Future works 111
4. Conclusion 111

Appendix A. Integrity Problem: hints on the global approach 113
1. Overview 113
2. Global scheduling algorithms for fault-tolerance 115
3. Schedulability analysis 119
4. Comments and future research 121

Appendix. Bibliography 123

List of Figures

1.1	Example of task, job and related parameters.	4
2.1	Necessary condition for a deadline miss.	30
2.2	Worst-case workload under EDF	33
2.3	Worst-case workload under FP	34
2.4	Worst-case workload under EDZL	36
2.5	Deadline positions under FP	38
2.6	Worst-case execution of τ_i under EDF and EDZL	41
2.7	Example of slack	47
2.8	Core of the schedulability test for EDF and EDZL, above, and FP, below.	51
2.9	Pseudo-code for the Recursive Test.	52
2.10	EDF: 4 processors, $D_i \leq T_i$.	55
2.11	EDF: 8 processors, $D_i \leq T_i$.	55
2.12	FP: 4 processors, $D_i \leq T_i$.	56
2.13	FP: 8 processors, $D_i \leq T_i$.	56
2.14	EDF: 2 processors, $D_i \leq 4T_i$.	57
2.15	EDF: 8 processors, $D_i \leq 4T_i$.	57
2.16	FP: 4 processors, $D_i \leq 2T_i$.	58
2.17	FP: 4 processors, $D_i \leq 4T_i$.	59
2.18	Improvements for EDF-DM: 4 processors, $D_i \leq 2T_i$.	59
2.19	EDZL: 2 processors, $D_i \leq T_i$.	60
2.20	EDZL: 8 processors, $D_i \leq 4T_i$.	60
2.21	Global comparison: 2 processors, $D_i \leq T_i$.	61
2.22	Global comparison: 4 processors, $D_i \leq 2T_i$.	61
2.23	Global comparison: 8 processors, $D_i \leq 4T_i$.	62
2.24	Comparison of results for REDF when the number of step is limited.	64
3.1	Minimum demand examples.	70
3.2	The two cases of the proof of Theorem 4.1.	77
3.3	$md(\tau_2, t)$ with lower and upper bounds.	77
3.4	$md(\tau_1, t)$, $md(\tau_2, t)$, and their sum.	79
3.5	$\frac{md(\tau_1,t)}{t}$, $\frac{md(\tau_2,t)}{t}$, and their sum.	79

3.6 Pseudo-code for approximate computation of $m\ell$ from below. 83

3.7 Histograms for $U_{tot} \leq 2$, $M = 2$. 84

3.8 Histograms for $U_{tot} \leq 4$, $M = 4$. 85

3.9 Histograms for $U_{tot} \leq 8$, $M = 8$. 85

3.10 Iterations to compute load-bound for $U_{tot} \leq 4$. 86

3.11 Relation of lower bound for $\mathrm{DBF}(\tau_i, t)$ and $md(\tau_i, t)$ for τ_2 of Example 7. 86

3.12 Global comparison on feasible task sets: 2 processors, $D_i \leq T_i$, exponential distribution with mean 0.25. 87

4.1 Lock-step architecture 94

4.2 The hardware platform 95

4.3 Switching between modes 96

4.4 The supply function 99

4.5 Example of computation of schedP_i 101

4.6 Determining the feasible periods 104

Notation

Here we report the notations mostly used throughout this document.

M	the number of processors.
N	the number of tasks in the task set.
$\mathcal{T}$	the task set.
U_{tot}	the total utilization of the task set.
U_{max}	the maximum utilization of the task set.
λ_{tot}	the total density of the task set.
λ_{max}	the maximum density of the task set.
τ_i	the ith task.
C_i	the computation time of τ_i.
T_i	the period (or minimum interarrival time) of τ_i.
D_i	the relative deadline of τ_i.
Λ_i	the minimum between period and deadline, for task τ_i.
U_i	the utilization of τ_i.
λ_i	the density of τ_i.
S_i	the lower bound on the slack of τ_i.
$\overline{\Lambda}_i$	the same as Λ_i, but corrected with S_i.
τ_i^j	the jth job of τ_i.
r_i^j	the release time of τ_i^j.
d_i^j	the absolute deadline of τ_i^j
f_i^j	the finish time of τ_i^j
l_i^j	the laxity of τ_i^j.
$I_k(a,b)$	the interference suffered by τ_k in $[a,b)$.
$I_k^i(a,b)$	the interference suffered by τ_k due to τ_i in $[a,b)$.
$\beta_k^i(()b-a)$	an upper bound on $I_k^i(a,b)$
N_i	the number of jobs of τ_i in the part of $\beta_k^i(()b-a)$ called body.
ε_i	an upper bound on the part of $\beta_k^i(()b-a)$ called carry-in of τ_i.

$\mathrm{DBF}(\tau_i, t)$	the demand bound function of τ_i in an interval of length t.
δ_{sum}	the load bound function of τ_i.
$md(\tau_i, t)$	the maxmin demand of τ_i in an interval of length t.
$m\ell(\mathcal{T})$	the maxmin load of τ_i.
m_i	the integrity requirement of τ_i.
FT	the fault-tolerant mode.
FS	the fail-silent mode.
NF	the non-fault-tolerant mode.
$\mathcal{T}_m$	the subset of tasks requiring mode m.
O_m	the overhead to switch out of mode m.
P	the period of the switch among all the modes.
Q_m	the length of the time slot dedicated to mode m.
Qsm	the length of the time slot dedicated to mode m, decreased by the overhead O_m.
Δ_m	the maximum delay suffered by mode m.
α_m	the rate provided to mode m.
$Z_m(t)$	the supply function of mode m in an interval of length t (not in Appendix A).
$Z_m(t, t_0)$	the supply function of mode m in $[t_0, t_0 + t)$ (only in Appendix A).
$minZ_m(t)$	the minimum supply function of mode m in an interval of length t(only in Appendix A).
$maxZ_m(t)$	the maximum supply function of mode m in an interval of length t(only in Appendix A).

CHAPTER 1

Introduction

1. Technology invasion

Imagine the end of your working day. You turn off your laptop, while the desktop will continue working on heavy simulations through the whole night. After wearing your MP3 reader you walk to the bus stop. In the bus to the railway station, the display shows the news of the day. At the railway station the queue for the ticket is long, but you're lucky, there are several automatic ticket machines, so you can call home with your mobile, and inform you'll be there for dinner. The train departs, and after a while the voice of the speaker announces your stop. You get off, reach your car, and drive to your home listening the radio. Your GPS helps you in avoiding road works and the relative queue, and you reach home in time for dinner. Thanks to fridge, oven, and a bit of handiwork things are quickly on the table. Ready to eat... and you stop... what an amazing amount of computations brought you there?

The need for performance. In our everyday life, we are more and more surrounded by electronics, in every possible aspect of life, technology is breaking through. Moreover, electronic equipments are continually asked for new services and new capabilities. The example of automotive is clarifying: in only a few years, we assisted to the introduction of several new improved services, such as ABS (Anti-lock Braking System), EBD (Electronic Brake-force Distribution), ASR (Acceleration Slip Regulation), ESP (Electronic Stability Program) and AESP (Adaptive ESP), EPS (Electronic Power Steering) and other X-by-wire services.

How can we provide all the required computation? We could think to design new, improved, faster, powerful processors. But some problems arise:

- designing a new processor is a long and difficult task; we can clearly modify a previous architecture, which ease the work, but important problems remain, such as the testing phase;
- due to the time required to project, build and test the new processor, time-to-market increases; this is absolutely not acceptable, considering the speed at which in any technology related field new products are proposed;
- new and faster processors are more subjects to faults than older one; among the reasons we underline that an older processor has surely been tested not only in the classical testing phase, but also on the field, in months and years of use; as a consequence, problems of such processors are known, and the designer can take care of them; this is clearly not true for newer processors.

There is another solution, which allows to use older processors and provide nevertheless the huge amount of computational power required by the market: *multiprocessors*.

The need for integrity. Another important factor has to be taken in mind: since electronics influences all aspects of our life, we are every day more dependent to it. We particularly refer to "necessary" services, such as safety critical aspects in automotive, medical or space applications, but it is exactly the same with services usually considered less important: a faulty fridge or wash machine can create serious problems, even if it probably does not endanger the user. As a consequence, together with an increase in computational power, also an increase in integrity guarantees is required.

Unfortunately, in this regard the single processor approach has a major problem. Increases in speed and computational power are obtained essentially through technology scaling, which is based on the shrink of several physical characteristics of the processors. Considering, as example, the lower voltage levels, it is clearly easier an undesired switch from a level to another. The obvious consequence is that newer processors become more prone to faults and soft errors, which is exactly the contrary of what the market requires.

Again, using older processors would be much better, because of their lower probability to faults. And so, again, *multiprocessors* are an effective solution to provide higher performance and higher integrity. Moreover, multiprocessors can be used to provide additional integrity to the applications: we can easily implement the concept of space replication, often used as a base for fault-tolerance, by executing the same computation on more than one processors and comparing the results. This approach to fault-tolerance is clearly impossible on single processor platforms (although other techniques, such as time replication, can clearly be adopted).

The need for control. Multiprocessors can provide the necessary amount of computational power to any application. In this sense, in fact, it is sufficient to add processors to reach any desired amount of computational power. However, the problem we face is that controlling such a power is much more difficult than controlling an equivalent single processor. Liu [**Liu69**] observed that "*few of the results obtained for a single processor generalize directly to the multiple processor case: bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors*".

This is particularly true in safety critical fields, such as in automotive, medical and space applications. In these fields, in fact, together with classical requirements such as performance, integrity and correctness, we have another set of requirements to fulfill: time constraints. In such fields, one of the main duties of the application is the control subsystem, which clearly must react to changes in the environment within a certain time, otherwise the result is not only useless but potentially catastrophic. This is the key concept of a *real-time system.*

Unfortunately, if controlling a set of processors conflicting for bus access and shared resources is difficult, controlling them within certain prefixed time is even worse, also considering the fact that a complete theory in this field is still missing. For this reason, we believe research concerning the use of multiprocessor platforms in real-time systems is a necessary step to be done in the present and the near future. This thesis is completely dedicated to this goal.

1.1. Multiprocessors and the market. The interest for multiprocessors is not only theoretical. In the last decade, an increasing number of multi-core systems has been proposed in the embedded system domain as well as in the high level computing market. The major hardware providers are already developing the second generation of multi-processor chips, and are spending a considerable amount of resources in the research for next-generation parallel architectures.

The integration of multiple processors on a single chip constitutes one of the most important innovations in the design and development of embedded real-time systems. This is the solution adopted by, for instance, Texas Instruments's OMAP [**Ins**], NXP's Nexperia [**NXP**], STMicroelectronics's Nomadik [**STM**], ARM's MPCore [**ARM**], Sony-IBM-Toshiba's Cell [**Son**], and many others.

Moreover, FPGA based solutions, such as Altera's Stratix [**Alt**] allow to customize the hardware organization of the system. The developer can choose the number of CPU cores, the buses and their connections to the memory and peripherals. Some researchers have recently proposed hardware implementations of some parts of the operating system [**LSF, Fur00, AB04, WA02, KSM03**].

From the Real-Time community perspective, this kind of platforms represents an interesting workbench upon which validate the scheduling theory for multi-processor systems.

2. Real-time systems

Real-time systems are nowadays widespread and present in a wide variety of fields, such as control systems, avionic and automotive applications, environmental monitoring. Despite this fact, a clear and unified definition of real-time system does not exist. A good definition, proposed by Alan Burns and Andy Wellings in [**BW01**], is the following: "*A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period: the correctness depends not only on the logical result but also on the time it was delivered; the failure to respond is as bad as the wrong response*".

Whatever definition we consider, the key point of any real-time system is *predictability*. In fact, since the correctness of results strictly depends on the time they are delivered, it is essential to be able to guarantee that results are produced within a certain and known time instant. In other words, the system designer must be able to predict, at least in part, the evolution of the system.

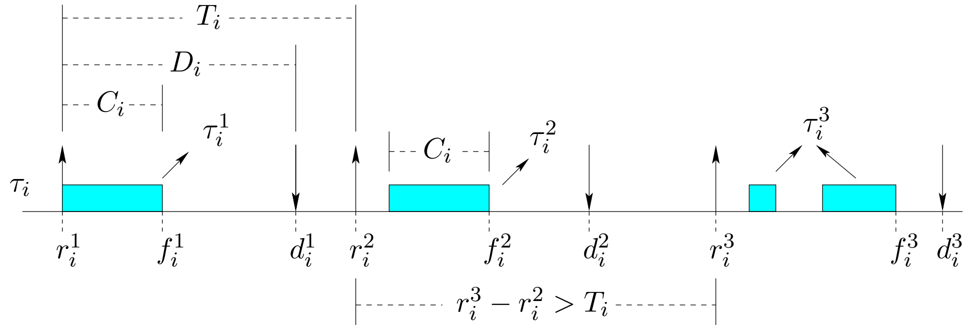

FIGURE 1.1. Example of task, job and related parameters.

Clearly guarantees can be given only within a certain range of precision, due to a series of events which are unpredictable by their nature. Examples are interrupts or cache and memory accesses. As a consequence, the approach to the analysis of such systems is usually extremely pessimistic, and guarantees are given in the *worst-case*, while the *average case* loses importance.

2.1. Model. In order to allow the designer to analyze the system, it is necessary to adopt a clear model for the application we want to describe. Since at this level what is important is the time analysis, the model must focalize essentially, if not solely, on timing aspects. As a consequence, functional aspects are not taken into account.

The model should be based on two opposite requirements:

- it should be rich enough to be able to describe correctly and with a sufficient degree of completeness the real world and the applications it is required to model;
- it should be easy enough to allow to fully analyze it and give reliable guarantees.

Meeting these two goals at the same time is quite hard. For this reason usually the second requirement is favored, while several aspects of real applications are not considered or hidden in gross overestimations and approximations.

In what follows, we introduce the main characteristics of the model we adopt throughout this thesis. To help the understanding, we graphically represent the parameters in Figure 1.1. However, in each chapter the model will be repeated and integrated with other useful concepts.

Tasks. A real-time application is considered to be composed of a set $\mathcal{T}$ of N real-time tasks τ_i, each of them composed of composed of base computations, called jobs, repeated from time to time. Each task is represented by a set of parameters, briefly introduced below.

The *computation time* C_i of a task (also called execution time) represents the time necessary to completely execute the task code. Since in general the code includes cycles, conditional structures and different paths to follow, the time necessary to complete is not fixed, and depends on a wide variety

of factors. As a consequence, usually C_i represents the *worst-case* computation time. If preemptions are allowed (see Section 3.5) the code can be interrupted and successively restarted, as is for job τ_i^3 in the figure.

Another important parameter is the *period*. In classical fields of application of real-time systems, several parts of the application are required to re-execute periodically. We could think, for example, to control systems, in which the sensor reading or the control algorithm have to be repeated at fixed time instants or with a fixed time interval between two consecutive executions. This concept is included in parameter T_i, which represents the period. In some cases, the period between two consecutive activations is not fixed, only bounded from below. In such a case T_i represents the *minimum interarrival time*, and the system is usually called *sporadic*. In the figure, τ_i^2 is released exactly T_i time instants after τ_i^1, while τ_i^3 is released later than T_i time instants after τ_i^2.

A third very important aspect is the time instant within which the task must be completed. Parameter D_i, called *relative deadline*, represents the maximum acceptable distance between the task activation and the end of the computation. The deadline D_i can be smaller or greater than T_i (as discussed in Section 3.4).

From the definitions, it is clear that $C_i \leq D_i$ and $C_i \leq T_i$, because in the other case, the system could never complete all its executions in time.

Finally, from the above parameters, we can extract the utilization U_i of the task, computed as $U_i = \frac{C_i}{T_i}$, that gives an idea of the fraction of processor that the task requires. The total utilization U_{tot}, obtained by summing the utilization of all the tasks, is a good estimation of the amount of computation that the processor(s) should execute in order to meet all the requirements.

Jobs. Since a task is usually activated several times, with a periodic or sporadic behavior, it does make sense to define separately the single activation and briefly introduce its parameters. We call each activation a *job* of the task, and we use the symbol τ_i^j to indicate job j of task i (or simply J_k, if the task is not important). Moreover we use r_i^j to identify the *release time*, that is the time instant at which the jth activation of task i happens. From this, and the relative deadline of the task, we obtain the absolute deadline $r_i^j + D_i = d_i^j$, which represents to time instant before which the execution of the job must be completed. In the interval $[r_i^j, d_i^j)$, the job must be guaranteed to execute for at least C_i, the computation time of the task. f_i^j, usually called *finish time*, represents the time instant at which the job actually completes its execution. It must be guaranteed that for each job τ_i^j of each task τ_i, $f_i^j \leq d_i^j$. A job is said to be ready in the whole interval $[r_i^j, f_i^j)$, and by extension, a task is ready (or backlogged) whenever it has a ready job.

2.2. Algorithms. In order to give any guarantees in a real-time system, it is mandatory to have an high degree of control over the order in which tasks are assigned to processor(s) for execution (in one word, *scheduled*). Classical scheduling algorithms for general purpose systems are First In First Out (FIFO, which executes tasks in order of their arrival time) and Round Robin (RR, which executes each task for a fixed amount of time

before switching to the next one in a "round way"). Unfortunately, these algorithms are not well-suited for real-time applications, meaning that they are often not able to meet the real-time constraints required by the application. This aspect of the problem has been deeply analyzed, and several different scheduling algorithms have been proposed. Here we give a short introduction on some of them that have a special importance in research and for this thesis.

Generalized Processor Sharing (GPS). The idea is to assign each ready job, in every time instant a share of the processor(s) equal to its utilization. This is only a theoretical algorithm, since it is clearly impossible to assign the same processor to more than one job in the same time instant. However, it is an interesting algorithm for comparison reasons, since it is able to obtain maximum utilization of the processors.

Fixed Priority (FP). Under FP, tasks are assigned offline a certain priority level, and each job of the task acquires the same priority. In every time instant the jobs executed are the ones, among the ready jobs, with the higher priority. The implementation of this algorithm is extremely simple, and it is easy to analyze it and so give real-time guarantees. Its drawback is that it is not able to guarantee full utilization, neither in single processors, nor in multiprocessors.

Earliest Deadline First (EDF). Under EDF, the order of execution does not depend on the tasks but directly on the jobs. In fact, once the absolute deadline of ready jobs is known, the scheduling algorithm selects for execution the jobs with the earliest deadline. The idea behind this algorithm is clear: if job J_1 has deadline before J_2, it is more urgent to execute J_1. This algorithm, maintaining easy implementation and analysis, has also a very good utilization bound on single processors (actually, under some hypothesis it is known to be optimal, meaning that under such hypothesis it is able to schedule every task set for which a correct schedule exists). Unfortunately, this good behavior is not maintained moving to multiprocessor platforms, due to anomalies that arise in the presence of more than one processor.

Earliest Deadline Zero Laxity (EDZL). Taking the lead from EDF, but trying to improve its behavior on multiprocessor platform, the EDZL algorithm is identical to EDF apart from one additional rule: whenever a job reaches a *critical instant*, it is scheduled for execution despite the fact that its deadline is not among the earliest ones. The formal definition is not important for now, and is delayed to Chapter 2, where the algorithm is analyzed. For now it is only important to underline that EDZL, at the price of a slightly more complex implementation, and some tricky aspects in the analysis, is able to provide great improvements in the behavior on multiprocessor platforms.

Hybrid algorithms. We group under this name a series of scheduling policies based on a mix of the characteristics of EDF and FP. The key idea is to give some tasks maximum priority (executing them them as in FP) and schedule the rest of the tasks as under EDF. The way the tasks to be executed at high priority are chosen, generates a variety of slightly different

algorithms. One example is EDF-US, where “US” means Utilization Separation, in which a utilization threshold is given, and tasks with utilization higher than the threshold are given higher priority. Another version uses the density $\lambda_i = \frac{C_i}{\min(D_i,T_i)}$, which represents a sort of upper bound on the amount of computation that a task can require in an interval. The algorithm takes the name of EDF-DS (Density Separation). A more interesting approach is EDF-UM (Utilization Monotonic), proposed in [**GFB03**] under the name PriD. The idea is easy: order tasks by utilization, and schedule the first k tasks with high priority, and the rest using EDF. What is interesting is the way k is chosen: $k \in [0, M-1]$ is the minimum value (if exists) such that the remaining $N-k$ lower utilization tasks can be scheduled on $M-k$ processors. In the worst-case, this algorithm supposes to assign to not more than $M-1$ tasks, one processor each, and schedules the rest of the tasks on the rest of the processors. Due to its high degree of freedom in choosing k, EDF-UM performs better than other similar algorithms. EDF-UM was proposed for implicit deadlines systems, where $T_i = D_i$, and so utilization and density are the same. For arbitrary deadlines systems, a better approach is EDF-DM (Density Monotonic), that uses density at the place of utilization. Since EDF-DM was shown to be best performer among several global scheduling algorithms [**Bak06b**], it must be taken into account as a term of comparison for other global scheduling algorithms.

Proportionate fair (Pfair). More than an algorithm, this is a class of algorithms which are all based on the same key idea: *fairness*. Other algorithms guarantee that in the long term each task occupies a share of the processor(s) equal to its utilization. However, if we consider the short period, the share of processor granted to one task can be extremely different from its utilization. It can happen that light tasks (i.e. tasks with low utilization) are granted 100% of the processor while heavy tasks (i.e. task with high utilization) cannot execute at all. The idea under Pfair is to try to grant a fair utilization of the processors in the short term as in the long term. This approach allows Pfair algorithm to reach utilization bound unreachable by other algorithms (and in particular the ones described above). However, they pay this good behavior with a very complex analysis, and extreme implementation problems (we are not aware of any working implementation, so far). Two well-known problems are processors synchronization and very high number of preemptions. It must be said, though, that while EDF, under certain hypothesis is optimal on single processors but loses optimality on multiprocessors, some Pfair algorithm is optimal on multiprocessors under the very same hypothesis. This algorithm will be not considered in the rest of this thesis: it has been shortly introduced here only for the sake of completeness, due to the theoretical importance of such an algorithm.

2.3. Schedulability and feasibility. In order to provide the required guarantees, it is necessary to develop analysis techniques, which, based on the task set and platform parameters, are able to answer to a simple question: *will the task set execute without missing any deadline?*.

If the answer refers to a specific scheduling algorithm, the problem under consideration is usually called *schedulability analysis*. Instead, if we don’t

focalize on any specific algorithm, but we refer to the simple fact that there is (or not) a way to execute all the tasks within their deadlines, the problem goes under the name of *feasibility analysis*. The following definitions capture the difference between the two problems.

Definition 2.1. A real-time task set $\mathcal{T}$ is said to be *schedulable* on a given hardware platform by a certain scheduling algorithm $\mathcal{A}$ if, for every possible pattern of releases of jobs, once decisions on tasks to be executed are taken basing on the rules of $\mathcal{A}$, each job of each task completes before its deadline.

Definition 2.2. A real-time task set $\mathcal{T}$ is said to be *feasible* on a given hardware platform, for every possible pattern of releases of jobs, there exists a scheduling algorithm $\mathcal{A}$ such that, once decisions on tasks to be executed are taken basing on the rules of $\mathcal{A}$, each job of each task completes before its deadline.

The reference to all the possible release patterns is necessary to take into account sporadic task systems, for which release times are not known in advance. In such a case, we are interested in the behavior of the system whatever happens at runtime, that is, for every possible release pattern.

Both problems are important for different reasons. It is clearly fundamental to study what a scheduling algorithm $\mathcal{A}$ is capable of, and in particular the guarantees it is able to provide. However, what is important is not what $\mathcal{A}$ can do among all possible task sets, but what it can do among all task sets for which a solution do exist. In this, feasibility analysis is essential.

For both problems, the best solution would be a necessary and sufficient test, that is a test which is able to give a clear answer in any case. Unfortunately, such analysis is usually very difficult or very long to complete. For this reason it is common to find either only sufficient or only necessary tests. In particular, the feasibility problem is usually solved by only necessary tests which answer to the question "Is there an algorithm $\mathcal{A}$ that correctly schedules the task set?", with "No" or "Maybe yes", but are not able to give any information about the characteristics of $\mathcal{A}$.

Instead, for the schedulability problem, what is generally proposed is an only sufficient schedulability test, whose response to the question "Is the task set schedulable by $\mathcal{A}$?" is either "Yes" or "Maybe not". Clearly, since real-time systems need to be predictable, "Maybe not" is considered perfectly equivalent to "No".

There is another approach to the schedulability problem which is worth to remember here: utilization and density bounds. The idea is to search, for a certain scheduling algorithm $\mathcal{A}$, what is the minimum value for the total utilization or (density, in some cases) of the system for which we can construct a non schedulable task set. If we are able to find such a bound, we can claim that every task set with lower utilization is schedulable by $\mathcal{A}$. Utilization bounds can be useful for comparison reason, because one can have an idea of what to expect from an algorithm of class of algorithms. However, it must be said that non schedulable task sets used to prove the bounds are often very special cases, which does not represent correctly the reality. For

this reason the comparison should be done with care. In Section 3.7 we consider again this problem and we present some known utilization bounds for some classes of algorithms.

The difference between schedulability tests and utilization bounds is that usually schedulability tests require much more information on the task set parameters, but are able to provide a positive answer for a larger set of task sets. However, schedulability tests and utilization bounds are in general incomparable, meaning that there exist task sets for which the first provides a positive answer while the second is negative, and vice versa. For this reason the best approach is probably to mix them in the analysis.

3. Taxonomy

Several distinctive aspects can be considered in order to classify real-time systems based on multiprocessor platforms. In what follow, we propose some key elements and a short taxonomy derived based on them. Note that some distinctions refer to base concepts of real-time systems, and are for both single and multiprocessors, while some others are strictly related to multiprocessor platforms.

We underline also that, together with the parameters proposed below, it would be possible to classify real-time systems and tasks based on several other aspects (jitter and/or offset are only examples). In what follows, we prefer to consider only the most important aspects, or the ones that actually influence this thesis.

3.1. Hardware. A first classification among real-time systems on multiprocessor platforms can be done considering the hardware selection, and in particular the processors composing the platform. We can so distinguish among

- *identical multiprocessors*: each processor is exactly identical to the others; as a consequence, there is no difference in capabilities, or in execution time required by tasks, so in principle each task can be scheduled on any processor without distinction; at the moment, this typology is by far the most considered in research, because of its simplicity with respect to the other solutions; for this reason, research related to identical multiprocessors is extremely vast (see for example [**Bak05, Bak06a, GFB03, BCL05a, BCL05b, LGDG03, LDG04, BCPV96**]);
- *uniform multiprocessors*: processors are identical, with the only exception of the speed, represented by a parameter s_i; tasks can execute on any processor, and the execution time is scaled depending on the processor speed; examples of research in this field are [**Bar01, FGB01, AT07**];
- *heterogeneous multiprocessors*: the set of processors includes cores with different capabilities and different speeds; as consequence, tasks can or cannot be executed on some cores; examples of such a platform could be one that integrates a general purpose processor and a DSP (digital signal processor) or a graphical coprocessor; due

to their complexity, these platform are not yet considered enough in research; we can cite for example [**Bar04b, Bar04a**].

In this thesis, we deeply focalize on identical multiprocessors.

3.2. Scheduling algorithms. Whenever more than one processor can physically execute the same task, it is important to establish the link between tasks and processors. This problem influences the characteristics of the scheduling algorithm, and allows to divide among three classes of algorithms: global, job-partitioned and task-partitioned algorithms.

Global algorithms are based on the idea that a task can migrate among the processors, and so execute on different processors in different time instants. For this reason, this class of algorithms is also known as of algorithms with migration. In practice, all the jobs are queued in a single global queue, and then extracted to be executed on any of the idle processors. The possibility to migrate from a processor to another becomes interesting when a job of a task is preempted by some higher priority job. In such a situation, whenever a processor becomes idle, the preempted job can restart executing on a different processor, instead of waiting for the original processor to complete the higher priority job. Works in this field consider for example classical single processor algorithms extended to multiprocessors, such as EDF-global [**Bak03, Bak05, GFB03, BCL05a**], and FP-global [**Bak03, Bak06a, BCL05b, ABJ01**], together with algorithms specifically designed for multiprocessors, such as EDZL [**CLAL02, CB07**] or the Pfair class of algorithms [**BCPV96, AS99, AS04b**]. The suffix "-global" is usually avoided, whenever it is clear that the algorithm we are referring is the global version of a relative classical single processor algorithm.

In *task-partitioned algorithms* (also called algorithm without migration), before system start time, the task set is divided (partitioned) among the processors, so that each task is strictly dedicated to only one processor, and only on this processor can execute (that is, it cannot migrate to another processor). For example, in [**LGDG03, LDG04**] the authors consider how to partition tasks on processor (in an identical multiprocessor platform), and what are the bounds on utilization for several partitioning policies, when FP and EDF are used to schedule tasks. In [**AT07**] a similar problem is considered, although on uniform, instead of identical, multiprocessors. Algorithms in this category are classical algorithms for single processor, integrated with some partitioning policy, such as First Fit or First Fit Decreasing. In the literature we find references to them as, e.g., EDF-FF and EDF-FFD, or generically EDF-partitioned. An overall description of some of the most known partitioning policies can be found in [**LDG04**]. Again, when the context is clear, the suffix "-partitioned" is omitted.

Finally, *job-partitioned algorithms* are a sort of mix of the two preceding classes, in that a task can migrate from one processor to another (as in global strategies), but a job cannot (as in task-partitioned algorithms). That is, a job can start executing on any idle processor, but once it selects one processor it can execute only on it. Examples of such algorithms are only modifications of global algorithms such as EDF-global or FP-global. Note that when preemptions are not allowed, under global strategies, once a job

starts executing, it will continue on the same processor up to completion. As a consequence, in such a case, global and job-partitioned algorithms coincide.

The three classes of algorithms have contrasting pros and cons. In global strategies the processors are apparently better utilized, and the load is automatically balanced. Moreover, it is well-known from the queuing theory that using a single queue scheduler results in lower average response times than having a queue for every single processor [**GH98**]. Further advantages of the global approach can be found in [**AJ00**], where a convincing argument shows that the average number of preemptions in a partitioned system is typically higher than in a globally scheduled system.

However, well-known anomalies arise due to the introduction of real competition among the tasks: an example is the Dhall's effect [**DL78**], in which light tasks can force a deadline miss on a heavy task, potentially reducing the average processor utilization to 0. It has been shown, in this sense, that algorithms known to be optimal on single processor systems (such as EDF and FP), lose their optimality on multiprocessors. Another important problem is related to cache and buses: whenever a task migrates from one processor to another, the whole cache must be invalidated on one processor and reloaded on the new one, possibly provoking loss in time and heavy load on the buses. Moreover they require a dedicated, complex, analysis, which discourages system designers to use them in real cases.

In contrast, task-partitioned algorithms are easier, since once tasks are partitioned, the analysis reduces to the well-known single processor case, for which research has proposed very good solutions. Unfortunately, difficulties are only moved elsewhere, and in particular in the partitioning phase: the problem of partitioning tasks among processors is equivalent to the classic bin-packing problem, which is known to be NP-hard in the strong sense. However, heuristics that propose good (although not optimal) solutions do exist [**Bar04a**]. Another problem relates to the load balancing, which is not automatic anymore. This means, for example, that if a task executes for more than assumed in the analysis, all tasks in the same processor are negatively affected. Instead, a global strategy can in general have a better answer to such a problem. Moreover, the missing automatic balance is a problem when tasks can join or leave the system at run-time. In such a case, it is possible to have at the same time processors with heavy load and almost idle processors.

Job-partitioned algorithms try to mix the two solutions, having the good aspects of global scheduling but trying to reduce problems with caches and bus traffic. This is based on the assumption that the most part of the memory space of a task relates to the single job, so avoiding its migration, while allowing migration of the task with its low memory footprint, could be a good compromise. To the best of our knowledge, research in this field is quite limited, and so we do not consider this solution in this thesis.

3.3. Task priorities. Once the class of the scheduling algorithm is selected, every scheduling algorithm has the same base behavior: fulfilling the constraints required by the algorithm class, execute the job(s) with higher priority. The difference among the algorithms is usually in how priority is

assigned to the jobs. This leads to another classification among real-time systems on multiprocessors.

In *task-fixed priority* algorithms, the priority is statically assigned to each task, and every job of the task has exactly the same priority; this leads to a sort of ordering among the tasks. The scheduling algorithm which represents this class is FP. Once FP is selected, another aspect needs attention, that is how the priorities are assigned. Examples are *Rate Monotonic* (RM) which assigns higher priority to tasks with shorter periods (i.e. rates), and *Deadline Monotonic* (DM) which is the same except that it is based on relative deadlines instead of periods. Both of these priority assignments are known to be optimal for single processors (under some constraints), but lose their optimality in multiprocessor systems. An interesting example of a different priority assignment is RM-US [**BCL05b**] (RM with Utilization Separation, where up to one task per processor with high utilization is assigned maximum priority, while the others are assigned as in RM). This class of algorithms has some important characteristics: they are easy to implement, and several working implementations do exist; their analysis is relatively easy, since a clear relationship among tasks is given; moreover, it is quite easy to tune them after a change in the application, since in most cases only modifications in the priority assignment rule are in order, while the scheduling algorithm (a much more critical component) remains untouched.

Job-fixed priority algorithms assign priority not to the tasks but to the jobs. An example of scheduling algorithm in this class is EDF, which assigns higher priorities to the jobs based on their absolute deadlines (that is, earlier deadline leads to higher priority). These algorithms maintain a quite easy implementation and analysis, but introduce some flexibility, which allows to increase the overall utilization of the processors.

Dynamic priority assignments allow to change the priority of tasks in every moment, and so give the maximum flexibility to the platform. This is the key aspect that allows these algorithms to reach extremely high utilization factors, whereas the other classes are limited to very low values. Unfortunately, due to this new flexibility, the analysis is usually very difficult. Moreover, the implementation tends to become more complex. Two examples of such algorithms are EDZL and the Pfair class of algorithms.

In single processors, we usually have only task-fixed and job-fixed priorities, because they are clearly easier to analyze and implement, but are able, at the same time, to guarantee very good behavior. In multiprocessors, dynamic priority algorithms are introduced to cope with the bad behavior of the previous two classes, and in particular the inadequate results of global strategies.

3.4. Periods and deadlines. Another factor to distinguish among different real-time systems on multiprocessors is the relation between period and deadline of a task. In particular, we distinguish among

- *implicit deadlines*, if for each task the deadline is exactly equal to the period;
- *constrained deadlines*, if deadlines are always less than or equal to periods;

- *unconstrained deadlines*, if no constraint is imposed on the values of periods and deadline, that is deadlines can be less than, equal to or greater than periods.

While this seems to be a minor aspect, it has at least two important effects, for which some attention is required. For the implicit deadlines case, analysis is relatively easy. When constrained deadlines are accepted, in the analysis of a job we need to consider that there is an interval in which, although a new job cannot arrive, the job under analysis cannot execute. This introduces some intricacy in the analysis. In the unconstrained deadlines case, things become even worse. In fact, if a task has deadline greater than the period, it could happen that one of its jobs is activated while the deadline of the previous one has not yet passed, so two jobs of the same task could be active at the same time. As a consequence, either the second job is delayed up to the time the previous one finishes (this is the classical approach) or two jobs of the same task execute at the same time, potentially conflicting to a great extent for critical sections. This has two results: the analysis is quite more difficult, and usually much more pessimistic, and the operating system must be written with these peculiarities in mind, in order to cope with possibly replicated state structures, buffers for values, and so on.

At the moment, progress in research is usually in steps: first of all consider implicit deadlines, then eventually extend to constrained deadlines (quite common) and then possibly extend to unconstrained deadlines.

This classification is clearly valid for both single and multiprocessor systems. In the rest of this thesis, sometimes we use unconstrained deadline tasks to identify only tasks with $D_i > T_i$, while arbitrary deadline refers to tasks with no constraint on deadlines. However, what we mean from time to time should be clear from the context.

3.5. Preemptability. Another aspect which can deeply modify the behavior of the system relates to the answer to this question: what happens when a job arrives with priority higher than some of the executing jobs, and no idle processors is present?

If the system allows preemptions, the lower priority job is descheduled, and the new job takes its place. Instead, if preemptions are not allowed, the higher priority job is forced to wait up to when one job finishes and a processor becomes idle. It is clear that preemptability improves the response time for higher priority tasks at the expenses of lower priority tasks. However, when a job is preempted in favor of another one, time is necessary to store the context of the first and load the context of the second, in the so called *context switch*. It must be said that, in the analysis, context switches are usually not taken into account, due to the small amount of time required. However, depending on the characteristics of the scheduling algorithm, the number of preemption can be very high, increasing the bandwidth lost in context switches. Moreover, since preemptions are usually not accounted for, when their number raises, the inaccuracy increases, making the analysis weaker.

Another problem that can show up in case of preemptions, is the fact that jobs could be preempted when they are executing in critical sections,

possibly forcing higher priority jobs to wait to enter the critical section. To avoid this phenomenon, it is often the case that *resource sharing protocols* are used to reduce the possibility to preempt lower priority jobs when they are in critical sections, by fact introducing a sort of restricted preemptability. Examples of these protocols for single processors are Priority Inheritance and Priority Ceiling Protocol [**SRL90**], or Stack Resource Policy [**Bak91**], while for multiprocessor we can remember some direct extensions of the last two: Multiprocessor Priority Ceiling Protocol [**RSL88**] and Multiprocessor Stack Resource Policy [**GLDN01**].

Despite the problems, the possibility to preempt lower priority tasks in favor of higher priority ones is extremely useful. For this reason, the approach is usually to allow preemptions and try to solve the problems it introduces, through resource sharing processors and scheduling algorithms with low number of preemptions (such as EDF and FP).

The model used in this thesis do not consider shared resources, and for this reason we have no doubts in allowing preemptions. As a consequence, in every time instant on an M-processors platform the M ready jobs with higher priority are under execution.

3.6. Periodicity. A final distinctive aspect is the meaning of the period of tasks. We can consider it as a real period, that is a new job of the task is activated once every period, or we can consider it as a minimum period (in this case it is usually called *minimum interarrival time*), which guarantees that a new job will be activated not less than a period after the previous one. In the two cases we talk about *periodic* and *sporadic* task models.

As for the case of the relation among periods and deadlines, this distinction is less important than the others but introduces some modifications in the analysis. The most important one is the fact that the schedulability analysis for sporadic tasks loses in precision, and in particular is always only sufficient. This is clear, since while a periodic task will be surely activated once every period, this is not true for a sporadic task, which in the limit could be never activated. That is, there exists always a pattern of releases, such that a given task set is correctly scheduled. However, it would not be correct to consider such a task set schedulable. This requires to slightly modify the definition of schedulability and feasibility for a sporadic system: a sporadic task set is schedulable (feasible) only if every possible pattern of activations is schedulable (feasible).

As the one before, this classification relates to real-time systems on both single and multiprocessor platforms.

3.7. Graphical classification. Based on the class of scheduling algorithm and the task priority assignment, the following Table 1 has been proposed in [**CFH+04**].

In the table, for each couple *priority-migration* some example of algorithms are proposed. In particular we report in the table some of the algorithms described in Section 2.2. In the same table, we report also the utilization bounds (as described in Section 2.3) known so far for each class. We remember that M is the number of processors in the platform, while

Migration / Priority	Task-partitioned	Job-partitioned	Global
Task-fixed	FP-partitioned $U \leq \frac{M+1}{1+2^{\frac{1}{M+1}}}$	FP-global $U \leq \frac{M+1}{2}$	FP-global $U \leq \frac{M+1}{2}$
Job-fixed	EDF-partitioned $U = \frac{M+1}{2}$	EDF-global $U \leq \frac{M+1}{2}$	EDF-global $U \leq \frac{M+1}{2}$
Dynamic	not useful	not useful	EDZL, Pfair $U = M$

TABLE 1. Classification of real-time systems on multiprocessors

U is the total utilization of the task set. As said above, the meaning of the bounds is that there exist task set with total utilization U arbitrarily greater than the bound that cannot be correctly scheduled by any algorithm belonging to that class. Whenever an equality is present (e.g., full migration and dynamic priority) it is also guaranteed that for each task set with utilization up to the bound, there exist at least one algorithm in the class able to schedule the task set. That is, the bound is "necessary and sufficient". Instead, in case of inequalities, the actual necessary and sufficient bound is not known, but it is surely not greater than the reported value. We underline that these bounds are valid for implicit deadline systems, while if constrained or constrained deadlines are allowed, things are much more complicated.

Analyzing the table, we can see that moving towards more freedom, we can improve the bounds. As an example, Pfair (which has the maximum freedom in both migration and task priority) is able to use up to 100% of the processors, while any other global algorithm with less than dynamic priority cannot reach a value higher than $\frac{M+1}{2}$. Unfortunately, the price of this good behavior is the extreme complexity in terms of implementation and analysis.

It is useful to repeat here that, despite the fact that theoretical bounds for some classes is very low, their actual behavior is much better. For example, while EDF-global has an extremely low upper bound on utilization (in Chapter 2 we show that this bound is 1, and it is independent of the number of processors), through schedulability analysis we can prove that EDF is able to schedule an high number of task sets with higher utilization than the bound.

4. Contributions and summary

Multiprocessors in real-time applications can really improve the behavior of systems, and have great benefits. However, research is still far from being able to provide enough guarantees. We believe this thesis is a good step in the direction of exploiting multiprocessors at their best in real-time systems. In particular, we propose and analyze the use of multiprocessors with two goals in mind:

- improve the performance of systems: more processors means more computational power that can be provided to the application, and so, potentially, a better behavior;

- improve the reliability of systems: more processors means the possibility to cross-check their work, verifying the correctness of the results they produce, and try to solve the problem in the event of faults and consequent errors.

In this thesis, we report results previously published in conference and journal papers, together with some new and improved analysis. In the next two chapters we consider the *Performance Problem*. In Chapter 2 we discuss the topic of how to improve performance of systems through the introduction of multiprocessors, and in particular we analyze the schedulability conditions for three algorithms: EDF, FP and EDZL. In Chapter 3 we consider the problem from the opposite side, that is the feasibility analysis: what are the conditions under which a task set is schedulable by some algorithm? This analysis helps in giving a better estimation of the algorithms we consider. It is possible, in fact, to better distinguish between task sets which fail due to scheduling algorithm and schedulability test, and task sets which fail due to their own characteristics. Then, in the following two chapters we consider the *Integrity Problem*, in which we want to tune a multiprocessor platform in order to provide some fault-tolerance to the applications. In Chapter 4 we suppose to use a task-partitioned algorithm, while in Appendix A we give some ideas on how to tackle the same problem using a global algorithm. Finally, in Chapter 5 we conclude and propose some future works.

CHAPTER 2

Performance Problem: schedulability analysis

1. Overview

Multiprocessor hardware platforms are becoming widespread and commonly used. One of the reason is that, given the current limits of hardware technology, new increases in computational power can be obtained more easily and in a cost-effective way by using more than one processor, rather than a more powerful single processor technology.

However, while real-time systems on single processors have been thoroughly studied, and solutions exist for a wide variety of problems, a complete theory of real-time scheduling for multiprocessor systems is still to come.

In this chapter, we will address the problem of scheduling real-time task sets on identical multiprocessor platforms consisting of M processors. This problem can be solved in two different ways: by partitioning tasks among processors, or with a global scheduler. The differences among the two approaches have been discussed in Chapter 1. We prefer to tackle the case of global scheduling algorithms, due to some advantages they have with respect to partitioned ones, e.g. the fact that they automatically balance the load on processors. Global scheduling algorithms maintain a global shared queue for all processors, and jobs in the queue are ordered by priority. From the queue, the M highest priority jobs are selected and executed on the M processors. This way, it can happen that a job starts executing on one processor and, after preemption and rescheduling, completes on a different one. In such a case, we say that the job *migrates* between two processors.

Among global scheduling algorithms, it is worth to cite the Pfair class of algorithms [**BCPV96, AS99, AS04b**], which is known to be optimal under certain hypothesis. Such algorithms are based on the concept of quantum: the time line is divided into equal-size intervals called quanta, and at each quantum the scheduler allocates tasks to processors. A disadvantage of this approach is that all processors need to synchronize at the quantum boundary, when the scheduling decision is taken. Moreover, if the quantum is small, the overhead in terms of number of context switches and migrations may be too high. To avoid such overheads, some different solutions have been proposed [**AS04b**].

We focus on different algorithms, and in particular EDF and FP, two well-known scheduling algorithms developed for single processors, and successively extended to multiprocessor platforms. Moreover, we will consider EDZL, a recently proposed scheduling algorithm mainly based on EDF, which has been developed specifically for multiprocessors.

The advantage of these algorithms is the relatively simple implementation and the minor overhead in terms of number of preemptions. However,

many negative results are known for such schedulers. For example, it is known that EDF is optimal on single processors, while it loses its optimality on multiprocessor platforms. Moreover, with respect to partitioned algorithms, the overhead of migrating a task from one processor to another needs to be taken into account. In fact, in modern architectures, processors have a local cache memory, and migrating a task may invalidate the content of the cache.

Although global scheduling approaches seem to be complex or require too much overhead, we think that in some case they can be a valid option. For example, in some embedded processor architecture, with no cache and with simpler structures, the overhead of migration has a lower impact on the performance. Furthermore, in FPGA-based architectures, implementing the scheduler in HW can further reduce the overhead.

From a theoretical point of view, we think that tackling the problem of global scheduling with EDF and FP algorithms can help understanding the general problem of scheduling in multiprocessors. In addition, it is also worth to consider EDZL, since it has the advantage of being specifically proposed for multiprocessor, and can so better exploit their benefits.

Contribution. In this chapter we address the problem of schedulability analysis under EDF, FP and EDZL on multiprocessors. We propose two schedulability tests for each of the algorithms. The first solution is $\mathcal{O}\left(N^2\right)$ (where N is the number of tasks in the system), but does not provide an important improvement with respect to older tests. Instead, the second test, based on a recursive approach, has an higher worst-case complexity, but provides interesting benefits. Through simulation, we also verified that limiting the number of recursive steps does not strongly influences the results, while the complexity can be reduced to $\mathcal{O}\left(N^3\right)$ in the worst-case. In an extensive set of experiments, we compare the new proposed tests with older solutions, in order to validate our results. Moreover, we compare the couples algorithm-test, in order to provide an idea of the state of the art in global scheduling on multiprocessors.

2. System model

In order to analyze the behavior of real-time systems on multiprocessor platforms, it is necessary to provide an easy but complete system model. Part of the model we will use in this chapter has been already introduced in Chapter 1, but we repeat it here shortly for the sake of completeness.

We consider an application to be represented by a *sporadic task set* $\mathcal{T}$, that is a collection of N sporadic tasks τ_i. Each task τ_i is defined by *worst-case execution time* C_i, *relative deadline* D_i and *minimum interarrival time* T_i. It is often useful to define $\Lambda_i = \min(D_i, T_i)$, in order to unify the formulae for implicit, constrained and unconstrained deadline tasks. Note that for each task τ_i, $C_i \leq \Lambda_i$, since otherwise the task set would be trivially infeasible.

$U_i = \frac{C_i}{T_i}$ represents the *utilization* of a task τ_i, while $\lambda_i = \frac{C_i}{\Lambda_i}$ represents its *density*. By extension, U_{tot} and λ_{tot} represent total utilization and total density of the task set, respectively.

Each task τ_i is composed of a potentially infinite series of jobs τ_i^j, characterized by *release time* r_i^j, absolute deadline d_i^j and finish time f_i^j. A job is defined *ready* in the whole interval between its release time and its finish time, $[r_i^j, f_i^j)$. In the presence of unconstrained deadlines we consider that only one job of a task can execute in a certain time instant. That is, job τ_i^j cannot start executing until after f_i^{j-1}. The *release time sequence* r is the set of all release times of tasks in $\mathcal{T}$ (i.e., $r = \{r_i^j : \forall i, j\}$). A valid release times sequence r is a release time sequence where the minimum interarrival time between two consecutive releases of a task is respected.

In each time instant t in which a job τ_i^j is ready, we define the laxity $l_i^j(t)$ as the amount of blocking time that the job can suffer without missing its deadline. In other words, the laxity of a job is the difference between time to deadline and remaining computation time. In formulae, if $c_i^j(t)$ is the remaining computation time of job τ_i^j, the laxity at time t is defined as

$$l_i^j(t) = \left(d_i^j - t\right) - c_i^j(t). \tag{2.1}$$

A negative value of laxity means that the job will eventually miss its deadline, and a value equal to 0 means that the job must execute from t up to d_i^j, in order to be able to meet its deadline.

Based on the concept of laxity, we also define the slack S_i, as the distance between the finish time of one job of τ_i and its deadline, minimized over all the jobs of τ_i. In formulae

$$S_i = \min_j \left(d_i^j - f_i^j\right).$$

A negative value of S_i represents the fact that the task is not schedulable, while $S_i \geq 0$ guarantees the schedulability of the task. Goal of the schedulability test is so to verify if, under the chosen scheduling algorithm, $\forall i \;\; S_i \geq 0$.

We say that a sporadic task set is *feasible* on M processors if, for every valid release time sequence it is possible to schedule all the jobs so that no deadline is missed. Instead, we say that a sporadic task set is *schedulable* according to a given scheduling algorithm if, for every valid release time sequence the schedule produced by following the rules of the algorithm is such that no deadline is missed.

2.1. Blocking times. We consider only work conserving algorithms, that is algorithms that never leave a processor idle if there is some ready job to be executed. Due to this fact, if a job does not execute, it must be because it is blocked by some other job. We call *blocking time intervals* the time instants in which a job is ready but not executing.

Consider a schedulable task set with implicit and constrained deadlines. Since each job has $d_i^j \leq r_i^{j+1}$, in every time instant there can be only one ready job of each task. In such a case a ready job can be blocked only by the fact that all the M processors are busy executing higher priority jobs. In this case, we say that the job (and its task) is *priority-blocked*.

In the case of unconstrained deadlines tasks, the situation is different, because the finish time of a job could be after the release time of one (or

more) successive job(s) of the same task. Since in such a case the newer job can start executing only after the finish time of the previous job of the task, it can happen that a job is blocked by a preceding job that has not yet finished, despite the fact that some processor is idle. In this case, we say that the job (and its task) is *precedence-blocked*.

2.2. Interference. Based on the blocking times defined above, we can introduce the concept of *interference*. The interference on a task τ_k over an interval $[a, b]$ is the cumulative length of all intervals in which the task is priority-blocked. We will denote such interference with $I_k(a, b)$.

We also define the *interference* of a task τ_i on a task τ_k over an interval $[a, b)$, represented by $I_k^i(a, b)$, as the cumulative length of all intervals in which τ_k is priority-blocked, while τ_i is executing (which means that τ_i has higher priority than τ_k). Notice that by definition:

$$I_k^i(a, b) \leq I_k(a, b), \quad \forall i, k, a, b. \tag{2.2}$$

We underline here that neither $I_k(a, b)$ nor $I_k^i(a, b)$ include in their definition cases of precedence-blocking: they include only cases in which a job of τ_k could actually execute, given that one of the processor were idle. As a consequence, *priority-blocking* and *interference* could be used as synonyms.

2.3. Time division. To precisely define the notion of time, we follow the common idea, described for example in [**BC06a**], of considering only discrete time instants. That is, despite the fact that for mathematical convenience, points and durations in real time are modeled by real numbers, in an actual system time is not infinitely divisible. The times of event occurrences and durations between them cannot be determined more precisely than one tick of the system's most precise clock. Therefore, any time value t involved in scheduling is assumed to be a non-negative integer value and is viewed as representing the entire interval

$$[t, t+1) \stackrel{\text{def}}{=} \{x \in \mathbb{R} : \; t \leq x < t+1\}$$

The notation $[a, b)$ is used for time intervals as a reminder that the interval includes all of the clock tick starting at a but does not include the clock tick starting at b.

These conventions allow the use of mathematical induction on clock ticks for proofs, avoid potential confusion around end-points, and prevent impractical schedulability results that rely on being able to slice time at arbitrary time instants.

2.4. Scheduling algorithms. In this chapter, we analyze the behavior of three global scheduling algorithms, EDF, FP, and EDZL. It is useful to remind here their main characteristics.

EDF. Earliest Deadline First (EDF) schedules jobs according to their absolute deadline. That is, there is a global queue in which the jobs are ordered by absolute deadline, and in every time instant the jobs under execution are the first M in the queue, where M is the number of processors. EDF is known to be optimal on single processors, among job-fixed priority algorithms, in the sense that it is able to schedule every task set schedulable by any other job-fixed priority algorithm. Moreover, it has also a low number

of preemptions, and it is relatively easy to implement. For all these reasons, this is one the best algorithms for real-time systems on single processors.

Unfortunately, the optimality of EDF is lost moving to multiprocessors. It has been proven that the utilization bound of EDF on M processors is only $\frac{1}{M}$. In fact there exist task sets with utilization arbitrarily close to 1 that are not schedulable on M processors. To see this, it is sufficient to consider Example 1, in which a case of the so called Dhall's Effect [**DL78**] is shown.

Example 1. Consider the task set in the table below, composed of $M+1$ tasks τ_i, where ϵ is such that $\epsilon \ll K$.

i	C_i	D_i	T_i
1 to M	ϵ	$K-\epsilon$	$K-\epsilon$
$M+1$	K	K	K

This task set is not correctly scheduled by EDF, since if all the tasks are released together, the M jobs have higher priority and are scheduled first, so that the last job will miss its deadline. However, the utilization of this task set is only $M\frac{\epsilon}{K-\epsilon}+\frac{K}{K}\approx 1$, for ϵ sufficiently smaller than K. □

Despite this fact, even if in the worst-case EDF performs very badly on multiprocessors, it can be seen that the average behavior is much better than this, in the sense that EDF is able to schedule a high number of task sets with utilization greater than 1. For this reason, improvements in schedulability tests are fundamental. Due to the need for guarantees of real-time systems, a "good" scheduling algorithm with a poor schedulability test is as bad as a "bad" scheduling algorithm.

FP. Under Fixed Priority (FP), jobs in the global queue are ordered based on the priority statically assigned to the tasks at system startup. Then, as for EDF, in every time instant the jobs under execution are the first M in the queue. Among the possible priority assignment policies, Rate Monotonic (RM, which assigns higher priority to shorter period T_i) and Deadline Monotonic (DM, for which higher priority is assigned to shorter relative deadline D_i) are the best ones for single processors. In particular, it has been proved that on single processors, they are optimum assignments respectively for implicit deadlines and constrained deadlines. Moreover, as for EDF, the number of preemptions they require is quite low.

Unfortunately, also FP with DM assignment (DM, for short) loses its optimality on multiprocessors. The Dhall's Effect [**DL78**] is present again, as can be seen in Example 1, which perfectly applies to DM as well.

Without losing generality, from now on we consider that tasks are ordered by decreasing priority. That is, τ_i has higher priority than τ_j if $i<j$.

EDZL. To overcome the limits of EDF, Earliest Deadline Zero Laxity (EDZL) has been suggested by Cho, Lee, Ahn and Lin [**CLAL02**]. The idea is to mix the benefits of two well-known algorithms, EDF and Least Laxity First (LLF). Considering that the laxity of τ_i^j at a certain time t represents the blocking time that the job can accept in the interval $[t, d_i^j)$ without missing its deadline, LLF assumes that jobs with smaller laxity are more urgent, and at each time t it schedules jobs according to their actual

laxity at time t (in increasing order). Both EDF and LLF are optimal on single processors, but EDF is preferred due to the number of preemptions:

- very low for EDF, due to the fact that it assigns job-fixed priorities, which means that we can define a complete ordering among the jobs, and preemptions can be required only by higher priority job on lower priority ones;
- potentially infinite for LLF, because once two jobs have the same priority, they start switching in every time instant (because only one can execute at a time, and the laxity of the other decreases, requiring the preemption).

On multiprocessors, both of them lose their optimality, and maintain the number of preemptions, low for EDF, and high for LLF, but while the utilization of EDF goes down to $\frac{1}{M}$, that of LLF seems to remain high. To the best of our knowledge, no utilization bound has been proposed till now.

EDZL takes the lead from the two algorithms. In "normal" situation, EDZL follows the rules of EDF, i.e. it schedules jobs with earliest absolute deadline. However, at certain "critical instants", jobs are given maximum priority, in order to save them from deadline miss. The *critical instant rule* is based on LLF, in the sense that the critical instant is, for each job, the instant in which the laxity of the job reaches 0, meaning that the job cannot wait anymore to execute. The algorithm can be expressed as follows:

- order the jobs in the global queue as for EDF;
- whenever a job reaches zero laxity, move it to the head of the queue;
- schedule the first M jobs in the queue.

It is been shown [**CLAL02**] that this algorithm mixes the benefits of EDF and LLF on multiprocessor: low number of preemption and high utilization.

3. Summary of existing results

Recently, the problem of identifying schedulability conditions for real-time task sets on multiprocessors has been thoroughly considered. The problem has been addressed for both partitioned and global techniques. Due to the advantages discussed in Chapter 1, we prefer to focus on the global approach. For this reason, we report here some of the main results for the schedulability on multiprocessor platforms under global scheduling algorithms, focalizing on the algorithms we consider in the rest of this chapter: EDF, FP and EDZL.

We want to underline that all the tests below are usually incomparable, in the sense that none of them is strictly dominant with respect to the others, and it is possible to find task sets schedulable (by the selected algorithm) based on one test and not the other, for any pair of tests. For this reason, the best solution would be to use all the tests at the same time.

3.1. Tests for EDF. First of all, we briefly recall some of the schedulability tests proposed so far for multiprocessors when EDF-global is selected as scheduling algorithm.

DB-EDF. This test for EDF, also called GFB from the name of the authors, is due to Goossens, Funk and Baruah [**GFB03**]. The test can be classified among utilization and density bounds, since it is solely based on the utilization of the task set and the maximum task utilization. The original formulation, described in [**GFB03**], assumes an implicit deadline task model, and consists of a single simple inequality that compares the total utilization of the task set, U_{tot}, with a bound which depends on the number M of processors and the maximum task utilization U_{max}. Through a parallelism between the execution of a task set on a uniform and an identical multiprocessor, the authors are able to prove the following theorem.

THEOREM 3.1 (Original DB-EDF test). *A periodic task set $\mathcal{T}$ with implicit deadlines is schedulable by EDF upon an identical multiprocessors platform composed of M processors with unitary capacity, if*

$$\sum_{\tau_i \in \mathcal{T}} U_i \leq M\,(1 - U_{max}) + U_{max}. \tag{3.1}$$

The test was proven to be tight for periodic implicit deadline systems, in the sense that there exist task sets with total utilization exceeding the bound by an arbitrarily small amount ϵ, that cannot be scheduled by EDF.

The test was then extended twice. The first extension, due to Bertogna, Cirinei and Lipari [**BCL05a**], included cases of constrained deadlines systems, and allowed sporadic releases (instead of only strictly periodic ones). The second extension, due to an observation by Baruah, closed the circle, including also unconstrained deadline systems. The complete test is reported in the following theorem.

THEOREM 3.2 (Generalized DB-EDF test). *A sporadic task set $\mathcal{T}$ with arbitrary deadlines is schedulable by EDF upon an identical multiprocessors platform composed of M processors with unitary capacity, if*

$$\sum_{\tau_i \in \mathcal{T}} \lambda_i \leq M\,(1 - \lambda_{max}) + \lambda_{max}. \tag{3.2}$$

From now on we will refer with DB-EDF to the generalized test described in Theorem 3.2.

BAK-EDF. Another test for EDF was proposed [**Bak03**] and successively refined and generalized [**Bak05**] by Baker. As for DB-EDF the test was initially proved for constrained deadlines systems, while the extension included the case of unconstrained deadlines systems. Moreover, the second version contained an extension in the analysis that, at the price of an higher computational complexity, allowed to identify a larger set of schedulable task sets. Here we report only the extended version. The test is more complex than DB-EDF since it requires knowledge of all tasks parameters (i.e., C_i, D_i and T_i). The key idea is that a generic job τ_k^j, usually called the problem job, can miss its deadline only if all the jobs of other tasks with higher priority can require an amount of computation which is sufficient to force the problem job to wait until it is too late. The test is reported in the following theorem, based on the restatement proposed by Baker and Cirinei in [**BC06c**].

THEOREM 3.3 (BAK-EDF test). *A sporadic task set $\mathcal{T}$ with arbitrary deadlines is schedulable by EDF upon an identical multiprocessors platform composed of M processors with unitary capacity, if, for every task τ_k, there exists a positive value $\lambda \geq \lambda_k$ such that*

$$\sum_{i=1}^{N} \min\left(\beta_k^i(\lambda), 1\right) \leq M\left(1-\lambda\right)+\lambda \tag{3.3}$$

where

$$\beta_k^i(\lambda) \stackrel{def}{=} \begin{cases} U_i\left(1+\max\left(0, \frac{T_i - D_i}{D_k}\right)\right) & \text{if } U_i \leq \lambda \\ U_i\left(1+\max\left(0, \frac{T_i - \lambda D_i / U_i}{D_k}\right)\right) & \text{if } U_i > \lambda \text{ and } D_i \leq T_i \\ U_i\left(1+\max\left(0, \frac{T_i}{D_k}\right)\right) & \text{if } U_i > \lambda \text{ and } D_i > T_i \end{cases}$$

The test seems to be quite complex, due to the fact that apparently it requires to consider every $\lambda \geq \lambda_k$ for each task τ_k. However, Baker showed in [**Bak05**] that it is sufficient to consider $\lambda \in \{U_i : i = 1, \ldots, N\} \cup \{\lambda_k\}$, that is, not more than $N+1$ possible values of λ for each task.

From now on we will use BAK-EDF to refer to this test.

3.2. Tests for FP.

DB-FP. Following an idea similar to the one which lead to the DB-EDF test, Bertogna, Cirinei and Lipari proposed in [**BCL05b**] a density bound for constrained deadlines systems scheduled by FP.

THEOREM 3.4 (DB-FP test). *A sporadic task set $\mathcal{T}$ with constrained deadlines is schedulable by FP upon an identical multiprocessors platform composed of M processors with unitary capacity, if*

$$\sum_{\tau_i \in \mathcal{T}} \lambda_i \leq \frac{M}{2}\left(1-\lambda_{max}\right)+\lambda_{max}. \tag{3.4}$$

As we said, the density bound above as been proved only for constrained deadlines systems. We underline the similarities between this density bound for FP and the density bound expressed in DB-EDF. The only difference is in the factor 2 in the right hand side. Unfortunately, due to this factor, the behavior of this test for FP is extremely worse than those of DB-EDF. For this reason, although from a theoretical point of view this test is quite interesting, it is not really useful in practice.

From now on we will use DB-FP to refer to this test.

BAK-FP. Together with the EDF test described above, in [**Bak03**] Baker proposed also a test for FP and constrained deadlines systems, based on the very same analysis. As for the previous cases, the test was then extended, and a new test was proposed in [**Bak06a**], which improved the analysis and included the cases of unconstrained deadlines. The theorem below, based on the restatement proposed by Baker and Cirinei in [**BC06c**], reports the test.

THEOREM 3.5 (BAK-FP test). *A sporadic task set $\mathcal{T}$ with arbitrary deadlines is schedulable by FP upon an identical multiprocessors platform*

composed of M processors with unitary capacity, if, for every task τ_k, there exists a positive value $\lambda \geq \lambda_k$ such that

$$\sum_{i=1}^{k-1} \min\left(\beta_k^i(\lambda), 1\right) \leq M\left(1 - \lambda\right) \tag{3.5}$$

where

$$\beta_k^i(\lambda) \stackrel{def}{=} \begin{cases} U_i\left(1 + \max\left(0, \frac{T_i - C_i}{D_k}\right)\right) & \text{if } U_i \leq \lambda \frac{M}{M-1} \\ U_i\left(1 + \max\left(0, \frac{T_i - \lambda \frac{M}{M-1} D_i / U_i}{D_k}\right)\right) & \text{if } U_i > \lambda \frac{M}{M-1} \end{cases}$$

As for the BAK-EDF test, while the complexity appears high, Baker showed that it is sufficient to consider a subset of the possible values for λ, and in particular the values to be tested are only k values for each task τ_k, $\lambda \in \{U_i : \; i = 1, \ldots, k-1\} \cup \{\lambda_k\}$.

From now on we will use BAK-FP to refer to this test.

3.3. Unified test for EDF and FP. Based on the BAK-EDF and BAK-FP tests, and the analysis proposed by Bertogna, Cirinei and Lipari in [**BCL05a**] and [**BCL05b**] (and improved in the rest of this chapter), Baker and Cirinei proposed, in [**BC06c**], a unified analysis for both EDF and FP. The distinction between the two algorithms was completely included in one parameter, γ_i in the theorem below, which reports the proposed test.

From now on, BAK-FP will refer to this test.

THEOREM 3.6 (BC test). *A sporadic task set $\mathcal{T}$ with arbitrary deadlines is schedulable by EDF or FP upon an identical multiprocessors platform composed of M processors with unitary capacity, if, for every task τ_k, there exists a positive value $\lambda \geq \lambda_k$ such that one of the following criteria is satisfied:*

$$\begin{array}{c} \sum_{i=1}^{N} \min\left(\beta_k^i(\lambda), 1 - \lambda\right) < M\left(1 - \lambda\right) \\ \sum_{i=1}^{N} \min\left(\beta_k^i(\lambda), 1 - \lambda\right) = M\left(1 - \lambda\right) \quad \text{and} \quad \exists i | 0 < \beta_k^i(\lambda) < 1 - \lambda \end{array} \tag{3.6}$$

where

$$\beta_k^i(\lambda) \stackrel{def}{=} \begin{cases} 0 & \text{if } i \geq k \text{ and under FP} \\ U_i\left(1 + \max\left(0, \frac{\gamma_i}{D_k}\right)\right) & \text{if } U_i \leq \lambda \text{ and } i < k \text{ or under EDF} \\ U_i\left(1 + \max\left(0, \frac{d_i + \gamma_i - \lambda D_i / U_i}{D_k}\right)\right) & \text{if } U_i > \lambda \text{ and } i < k \text{ or under EDF} \end{cases}$$

and

$$\gamma_i \stackrel{def}{=} \begin{cases} -D_i & \text{if } i = k \text{ and under EDF} \\ T_i - C_i & \text{if } i < k \text{ and under FP} \\ T_i - C_i & \text{if } i < k \text{ and under EDF} \end{cases}$$

We believe this test is very interesting from a theoretical point of view, since it is able to unify, in a single analysis, the cases for two of the most used algorithms in real-time theory. Unfortunately its behavior is not as good as one could expect, particularly for EDF. When applied in conjunction with FP, the test is able to prove the schedulability of a large number of task sets, with respect to previous tests (and BAK-FP, in particular). Instead, when EDF is selected, the test does not work very well, sometimes being outdone by DB-EDF and sometimes by BAK-EDF. Probably due to this reason,

apparently FP is able to correctly schedule more task sets than EDF, when schedulability is verified with BC. The interested reader can find in [**BC06c**] an extended set of simulations.

From now on we will use BC to refer to this test.

3.4. Tests for EDZL. To the best of our knowledge, so far there is no schedulability test explicitly proposed for EDZL on multiprocessors. The only valid tests for EDZL are a consequence of an observation by Park, Han, Kim, Cho and Cho [**PHK+05**]. They proved (see Theorem 2 in [**PHK+05**]) that EDZL strictly dominates EDF, with the meaning that if a task set is schedulable by EDF on a platform composed of M processors, it is also schedulable by EDZL on the same platform, and there exist task sets schedulable by EDZL and not by EDF. Intuitively, as noted by Cho, Lee, Ahn and Lin [**CLAL02**], EDZL is actually an EDF algorithm with a "safety rule" (the zero laxity rule) to be applied in critical situations. It means that the scheduling of the two algorithm differs only in cases in which EDF fails scheduling some tasks.

The direct consequence of this, is that each schedulability test for EDF on an M-processors platform, is also a schedulability test for EDZL on an M-processors platform. However, simulation studies have shown that EDZL scheduling performs much better than EDF [**CLAL02**], so the high inefficiency of this approach is evident. It would be much better to exploit the differences between the two algorithms to propose an efficient test.

3.5. Summary of the tests. In Table 1 below we report the tests proposed so far for EDF, FP and EDZL, together with the year and a reference to the papers where they have been proposed in different extensions. Please remember that all these tests are in general incomparable, and the best solution, from the point of view of schedulability, is to use all the tests for a given algorithm together.

Deadlines / Test	Implicit		Constrained		Unconstrained	
DB-EDF	[**GFB03**]	2003	[**BCL05a**]	2005	Baruah	2005
BAK-EDF	[**Bak03**]	2003	[**Bak03**]	2003	[**Bak05**]	2005
BCL-EDF	[**BCL05a**]	2005	[**BCL05a**]	2005	This thesis	2007
BAK-FP	[**Bak03**]	2003	[**Bak03**]	2003	[**Bak06a**]	2005
BCL-FP	[**BCL05b**]	2005	[**BCL05b**]	2005	This thesis	2007
DB-FP	[**BCL05b**]	2005	[**BCL05b**]	2005	To Be Done	
BC	[**BC06c**]	2006	[**BC06c**]	2006	[**BC06c**]	2006
EDZL	[**CB07**]	2007	[**CB07**]	2006	[**CB07**]	2006

TABLE 1. Summary of existing schedulability tests for EDF, FP, and EDZL.

Note that BCL-EDF, BCL-FP and EDZL were not discussed in the summary because they are explained in details and extended in the rest of this chapter. However, they are reported in the table for completeness.

4. Predictability

An important subtlety in schedulability testing is that the so-called "worst-case" execution time C_i of each task is just an upper bound, while the actual computation times of different jobs of a task can vary. This leaves open the possibility that the upper bound, or even the actual maximum execution time of a task may not actually be the worst situation with respect to overall system schedulability. For multiprocessor scheduling, there are well-known anomalies, where a job set is schedulable by a given algorithm, but if the execution time of one or more jobs is *shortened*, the job set becomes unschedulable.

Ha and Liu [**HL94, Ha95**] studied this problem, and were able to identify certain families of scheduling algorithms that are *predictable* with respect to variations in job computation time. A scheduling algorithm is defined to be *completion-time predictable* if, for every pair of sets $\mathcal{J}$ and $\mathcal{J}'$ of jobs that differ only in the execution times of the jobs, and such that the execution times of jobs in $\mathcal{J}'$ are less than or equal to the execution times of the corresponding jobs in $\mathcal{J}$, then the completion time of each job in $\mathcal{J}'$ is no later than the completion time of the corresponding job in $\mathcal{J}$. That is, with a completion-time predictable scheduling algorithm it is sufficient, for the purpose of bounding the worst-case response time of a task or proving schedulability of a task set, to consider each job of each task as having actual execution times equal to the task's worst-case execution time.

An important class of scheduling algorithms for which Ha and Liu were able to prove completion-time predictability is that of *preemptive migratable fixed-job priority scheduling algorithms*. Since both EDF and FP pertain to this class of algorithms, we don't need to consider this problem for them. Unfortunately, while EDZL is preemptive and migratable, it does not have fixed job priorities. Therefore, while we might suspect that EDZL could be predictable with respect to computation time variations, a necessary first step in looking for an EDZL schedulability test is to verify that. Piao, Han, Kim, Park, Cho and Cho [**PHK$^+$06**] addressed this question and showed that EDZL is completion-time predictable on the domain of integer time values. The result clearly also applies to any other discrete time domain. We give a somewhat more self-contained and direct proof below.

THEOREM 4.1 (Predictability of EDZL). *The EDZL scheduling algorithm is completion-time predictable, with respect to variations in execution time.*

PROOF. We actually prove a stronger thesis; that is, if the only difference between $\mathcal{J}$ and $\mathcal{J}'$ is that some of the actual job computation times are shorter in $\mathcal{J}'$ than in $\mathcal{J}$, then the accumulated computation time of every uncompleted job in the EDZL schedule for $\mathcal{J}'$ is greater than or equal to the accumulated computation time of the same job in the EDZL schedule for $\mathcal{J}$ at every instant in time. It will follow that no job can have an earlier completion time in $\mathcal{J}$ than in $\mathcal{J}'$, since the actual computation times in $\mathcal{J}$ are at least as long as in $\mathcal{J}'$.

Suppose the above hypothesis is false. That is, there exist job sets $\mathcal{J}$ and $\mathcal{J}'$ whose only difference is that some of the actual job computation times

are shorter in $\mathcal{J}'$ than in $\mathcal{J}$, and such that at some time t the accumulated computation time of some uncompleted job is less with $\mathcal{J}'$ than with $\mathcal{J}$. We will show that this leads to a contradiction, and the theorem will follow.

Without loss of generality, we can restrict attention to the case where $\mathcal{J}$ and $\mathcal{J}'$ differ only in the actual computation time of one job. To see this, observe that between $\mathcal{J}$ and $\mathcal{J}'$ there is a finite sequence of sets of jobs such that the only difference between one set and the next is that the actual computation time of one job is decreased. Let $\mathcal{J}$ and $\mathcal{J}'$ be the first pair of successive jobs in such a sequence such that at some time t the accumulated computation time of some uncompleted job J is less with $\mathcal{J}'$ than with $\mathcal{J}$.

Let t be the earliest instant in time after which the accumulated computation time of some uncompleted job is less with $\mathcal{J}'$ than with $\mathcal{J}$, and let J be such a job. That is, up to t the accumulated computation time of each uncompleted job in the schedule for $\mathcal{J}$ is less than or equal to the accumulated computation time of the same job in the schedule for $\mathcal{J}'$, and after time t the accumulated computation time of job J is greater with $\mathcal{J}$ than with $\mathcal{J}'$.

Job J must be scheduled to execute starting at time t with $\mathcal{J}$ and not with $\mathcal{J}'$. This means some other job J' is scheduled to execute in place of J with $\mathcal{J}'$. That choice cannot be based on deadline, since the deadlines of corresponding jobs are the same with $\mathcal{J}$ and $\mathcal{J}'$, so it must be based on the zero-laxity rule. That is, J' has zero laxity at time t with $\mathcal{J}'$ but not with $\mathcal{J}$. However, that would require that J' has greater accumulated computation time at time t with $\mathcal{J}$ than it does with $\mathcal{J}'$. This is a contradiction of the choice of t. Therefore, the theorem must be true.

□

The property depicted above goes under several names, and with slightly different definitions: predictability (Ha and Liu [**HL94, Ha95**]), robustness (Mok and Poon [**MP05**]), sustainability (Baruah and Burns [**BB06**]). The latter is actually a more general property that considers also the case of variations in other system parameters.

5. Schedulability analysis

In this and the following sections, we will analyze the scheduling of real-time task sets on multiprocessor platforms composed of M processors, taking into account three different scheduling algorithms: EDF, FP and EDZL. The analysis is similar for the three algorithms, as is the final schedulability test proposed for each of them, so it will be conducted for all of them in parallel, in order to better focus on the similarities and differences.

The analysis is mostly based on the line of reasoning used in [**Bak03**]. In order to clarify the methodology, we briefly describe the main steps that will be followed to derive the schedulability tests:

(1) as in [**Bak03**], we start by assuming that a generic job τ_k^j, called the *problem job*, misses its deadline d_k^j; moreover, we assume that this is the first missed deadline in the system;

(2) we identify a specific interval preceding the missed deadline d_k^j, called the *overload window* of τ_k^j, in which the schedulability problem can be better tackled (Section 5.1);
(3) if we were able to precisely compute the interference suffered by the problem job in the overload window, the schedulability test would simply consist in verifying that it is not sufficient to force a deadline miss, leading to a contradiction; unfortunately, computing such interference is very difficult;
(4) therefore, we give an upper bound to the interference in the overload window and derive a sufficient schedulability condition for the problem job, which is clearly valid for each job of τ_k (Sections 5.2 to 5.7); the difference among the scheduling algorithms will be mainly exploited at this step;
(5) a complete, only sufficient, schedulability test is obtained by simply repeating the same computation for each task: if no job of any task can miss its deadline, then the system is clearly schedulable (Section 6);

In Sections 5 and 6 we follow such line of reasoning. The test has complexity $\mathcal{O}\left(N^2\right)$, and consists, for each task, in a sum over all the other tasks, and a comparison. This test is based on a static estimation of the upper bound of the interference $I_k^i\left(a, b\right)$ of a task τ_i in the overload window of task τ_k. In Sections 7 and 8, we will improve the test by introducing a recursive approach that allows to give a better estimation of the upper bound of the interference at the price of an higher complexity.

5.1. Overload window. The first step in the schedulability analysis is to identify the interval we called overload window in which it is relatively easy to analyze the schedulability of a task. Since it is very difficult to estimate the precedence-blocking suffered by a task (apart for the case of constrained deadlines), we decide to define the overload window as the interval preceding the deadline d_k^j of the problem job, in which the problem job does not suffer precedence-blocking.

Once the overload window is identified, we need to find what is the necessary condition in such an interval for a job τ_k^j to miss its deadline d_k^j. Consider the example on three processors in Figure 2.1.

It is clear that τ_k^j can miss its deadline only if the other jobs occupy all the processors for a sufficient amount of time, which depends on C_k and the interval we consider. This is at the base of an observation reported by Baker in [**Bak03**], which is also the basis of [**PSTW97**]. We report it here, generalized to be more useful for our case.

Observation 5.1 (Lower bound on blocking-time). Consider a job τ_k^j and a generic interval $[d_k^j - t, d_k^j)$ of length t preceding its deadline, where $C_k \leq t \leq D_k$. If τ_k^j misses its deadline d_k^j, the sum of the lengths of all intervals in which τ_k^j does not execute in $[d_k^j - t, d_k^j)$ must exceed $t - C_k$.

The reason of the observation is clear. Consider an interval of length greater than C_k, ($t \geq C_k$) preceding the deadline, where the job is always ready ($t \leq D_k \Rightarrow d_k^j - t \geq r_k^j$). If in such an interval the job has sufficient

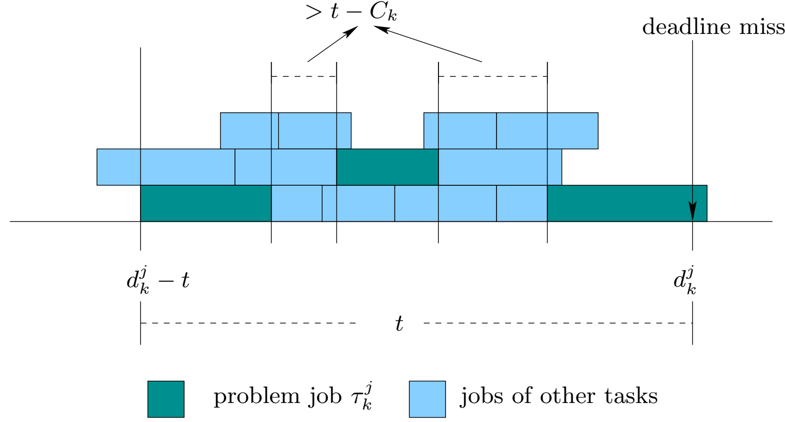

FIGURE 2.1. Necessary condition for a deadline miss.

time to execute (blocking time less than $t - C_k$), then τ_k^j it is able to complete before the deadline. We underline that in general the condition expressed in the observation above is necessary but not sufficient for a deadline miss, since the problem job could execute, in part or entirely, in the interval $[r_k^j, t)$.

Now we can map the above observation in our case, distinguishing between constrained deadlines (which includes also the case of implicit deadlines) and unconstrained deadlines.

Constrained deadlines. If τ_k has constrained deadline, as we observed in Section 2.1, it can never be precedence-blocked in the whole interval $[r_k^j, d_k^j)$. Since in this interval τ_k^j requires to execute for C_k time instants, it is clear that it can reach zero laxity only if it suffers priority-blocking for at least $D_k - C_k$. By extension, it can reach negative laxity (and so eventually miss its deadline) only if it suffers priority-blocking for more than $D_k - C_k$. Since we consider discrete time, an equivalent condition for negative laxity is that the priority-blocking is at least $D_k - C_k + 1$. The interval can be rewritten as $[d_k^j - D_k, d_k^j)$, since by definition $d_k^j = r_k^j + D_k$. Note that, for implicit and constrained deadlines systems, the condition expressed above is necessary and sufficient for a job to reach zero (or negative) laxity.

Unconstrained deadlines. If τ_k has unconstrained deadline (i.e. $T_k \leq D_k$), the above condition is not necessary anymore. In fact, in the interval $[r_k^j, d_k^j)$ more than one job of τ_k can be ready, which means that the problem job τ_k^j can be precedence-blocked. It is very difficult to take into account precedence-blocking, because it strictly depends on the finish times of the jobs, which are usually not known in advance (and very difficult to compute without actually simulating the system). However, since the problem job is the first job missing a deadline, if we consider the interval $[d_k^{j-1}, d_k^j)$, we know that no jobs of τ_k preceding the problem job can be ready (as all the previous deadlines expired). As a consequence, in this interval there can only be priority-blocking, as in the preceding case. The length of this interval is T_k, so the smaller interval in which τ_k^j cannot suffer priority-blocking is $[d_k^j - T_k, d_k^j)$. In order for the problem job to reach zero or negative laxity, it

is necessary, although not sufficient, that the problem job cannot execute for C_k time instants in this interval. In other words, the interval $[d_k^j - T_k, d_k^j)$ is suitable as overload window for unconstrained deadline tasks, and the problem job can reach zero laxity only if it suffers priority-blocking (i.e. interference) for at least $T_k - C_k$ in such an interval. Again, by extension, the problem job must suffer priority-blocking for more than $T_k - C_k$ (or at least $T_k - C_k + 1$) in order to reach negative laxity and so eventually miss its deadline. This condition is necessary but clearly not sufficient, because the problem job could have started executing before this interval, and so it could be able to complete in time despite the suffered priority-blocking.

Unification. We can unify the definition of overload window using parameter Λ_k, as below.

DEFINITION 5.2 (Overload window). The *overload window* of a problem job τ_k^j is the interval of length Λ_k preceding its deadline d_k^j. In formulae, it is $[d_k^j - \Lambda_k, d_k^j)$.

The necessary condition for a job to reach, and surpass, zero laxity is stated in the following lemma and corollary. It is valid indifferently for EDF, FP or EDZL, but can also be easily extended for other algorithms or classes of algorithms.

LEMMA 5.3 (Necessary condition for negative laxity). *A job τ_k^j of a task τ_k can reach negative laxity only if it suffers interference for at least $\Lambda_k - C_k + 1$ in the overload window $[d_k^j - \Lambda_k, d_k^j)$. In formulae, the condition is*

$$I_k(d_k^j - \Lambda_k, d_k^j) \geq \Lambda_k - C_k + 1. \tag{5.1}$$

PROOF. Follows from the above discussion, and the definitions of $I_k(a, b)$ and Λ_k. □

We underline again that the Lemma would be clearly correct if we use, instead of Equation (5.1),

$$I_k(d_k^j - \Lambda_k, d_k^j) > \Lambda_k - C_k. \tag{5.2}$$

In fact, due to the time division explained in Section 2.3, $I_k(d_k^j - \Lambda_k, d_k^j)$ can assume only integer values, so the two are equivalent. We prefer to use Equation (5.1), because it will be more useful in what follows.

COROLLARY 5.4 (Necessary condition for zero laxity). *A job τ_k^j of a task τ_k can reach zero laxity only if it suffers interference for at least $\Lambda_k - C_k$ in the overload window $[d_k^j - \Lambda_k, d_k^j)$. In formulate, the condition is*

$$I_k(d_k^j - \Lambda_k, d_k^j) \geq \Lambda_k - C_k. \tag{5.3}$$

PROOF. Follows from the above discussion, and the definitions of $I_k(a, b)$ and Λ_k. □

Note that both conditions are necessary for arbitrary deadlines systems, but become necessary and sufficient if we limit our attention to constrained deadlines systems.

Further notes on EDZL. The necessary conditions stated in Lemma 5.3 and Corollary 5.4 are clearly valid for any of the three scheduling algorithms under consideration (EDF, FP and EDZL). However, for the EDZL case we can improve the analysis.

For EDF, and FP, when a job reaches zero laxity, nothing is done to prevent the deadline miss. For the EDZL case, instead, the priority of a job with zero laxity is raised to the maximum and the job is scheduled up to its deadline, completing in time. The only case in which it can be preempted and miss its deadline is when there are more than M jobs in the same situation. As a consequence, while it remains true that a deadline miss can happen only when there exists a job for which Lemma 5.3 is true, we also know that for the lemma to be verified, it must be the case that at the end of the overload window of the problem job there are at least M other jobs with zero laxity, for which then Corollary 5.4 must hold. If we were able to exactly compute the interference, the presence of M other jobs with zero laxity would be included in the interference. In fact, when the problem job reaches zero laxity, it can suffer priority-blocking only from other jobs with zero laxity, so the computation of the interference would take this fact into account. Since we are forced to use upper bounds on interference, it is necessary to verify separately the presence of at least $M+1$ tasks that can reach zero or negative laxity. This observation will be extremely useful to improve the EDZL schedulability test.

5.2. Worst-case release times. The interference $I_k(d_k^j - \Lambda_k, d_k^j)$ in the overload window defined above cannot be easily computed. Moreover, it depends on the specific job under analysis, which means that we should compute it job by job. A possible solution is to use, instead of the actual value of interference, an upper bound to the interference. From now on, we call this upper bound $\beta_k^i(\Lambda_k)$. By definition, the interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$ of τ_i on τ_k^j in the overload window cannot be greater than the amount of computation that τ_i requires in the same interval. More specifically, the interference is composed only of computation of τ_i actually executed at a priority higher than that of τ_k^j.

The first step in computing $\beta_k^i(\Lambda_k)$ is to estimate what is the worst-case situation for the release times of τ_i. As shown by Baker [**Bak03**], we can assume that jobs of τ_i are released as soon as possible, one exactly T_i time instants after the other. This situation, in fact, is clearly the one that maximizes the amount of computation required by any task in a given interval. Then, in order to compute the worst-case execution time in the overload window, we need to determine which is the worst-case release time sequence of the jobs of τ_i: in practice what is the alignment between the jobs of τ_i and the problem job τ_k^j that maximizes the computation time required by τ_i in $[d_k^j - \Lambda_k, d_k^j)$.

The worst-case alignment depends on the scheduling algorithm selected, since this changes the relation between the priorities, so we now split the analysis, and consider the algorithms one by one. In all the three cases, we propose the worst-case release sequence, and we show that shifting the

releases forward or backward cannot increase $\beta_k^i(\Lambda_k)$. Due to the periodicity, the maximum amount of shift we need to consider is T_i.

Worst-case for EDF. In the case of EDF, as noted by Baker [**Bak03**], the worst-case release times sequence of τ_i is when one of the jobs of τ_i is released at $d_k^j - D_i$, so that its deadline is aligned with the deadline of the problem job. An example is represented in Figure 2.2. The execution of each job is supposed to be exactly before its deadline: once the deadlines are aligned as said above, this assumption increases to the maximum the amount of computation requested in any interval ending at d_k^j.

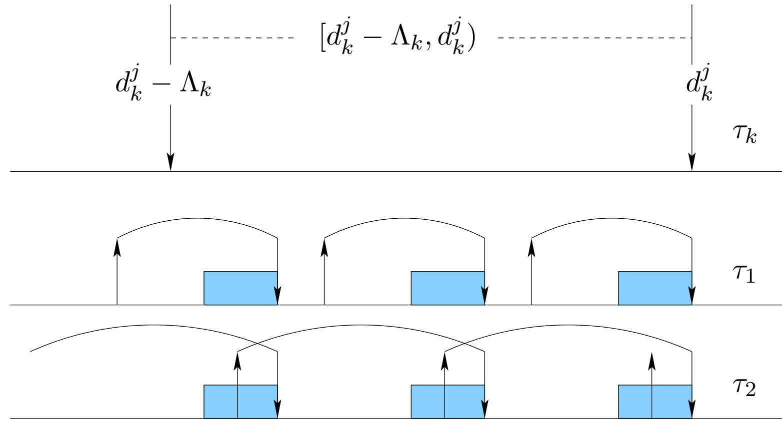

FIGURE 2.2. Worst-case workload under EDF

In the figure, τ_1 has constrained deadline, while τ_2 has unconstrained deadline. In [**Bak03**], Baker considered only the case of constrained deadlines, but in the figure we can see that there is no difference between the two cases. Moving forward by an amount $x \le T_i$ all the releases and deadlines of τ_i, can increase $\beta_k^i(\Lambda_k)$ on the left side of the interval by a maximum of $\min(x, C_i)$. However, on the right side we have a decrease of exactly one job (whose deadline moves after d_k^j, decreasing its priority). Instead, moving backward all the releases can only decrease on the left, while there is no increase on the right. As a consequence, the worst-case is the one depicted in the figure: periodic releases, one deadline aligned with d_k^j and each job executing exactly before its deadline.

Worst-case for FP. In [**Bak03**] Baker considered also the case of FP (again, only for constrained deadlines systems), and showed that the worst-case release sequence of τ_i is when one of the jobs of τ_i is released at $d_k^j - C_i$, so that its finish time f_i^j can be aligned with the deadline of the problem job. We claim that this is the worst-case release time sequence also for unconstrained deadlines systems. The situation is reported in Figure 2.3 for one constrained deadline task (τ_1) and two unconstrained deadline tasks (τ_2 and τ_3).

Only tasks with priority higher than the problem job must be considered. That is, in the analysis of τ_k^j we consider only tasks τ_1 to τ_{k-1}.

Comparing Figure 2.3 with Figures 2.2 and 2.4, it is evident that while in EDF and EDZL we have a periodic situation in the worst-case executions, in

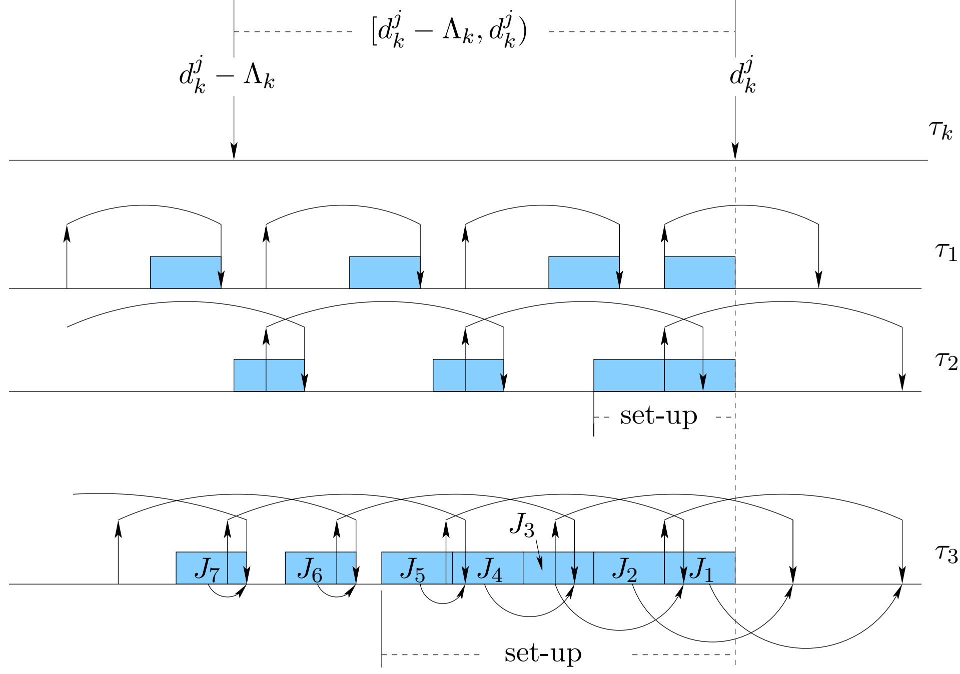

FIGURE 2.3. Worst-case workload under FP

the case of FP things are more irregular: in a first interval from d_k^j backward, we have an initial *set-up phase* in which the execution of more jobs can be packed one after the other, and only after a while the situation comes back to be periodic as it is in EDF and EDZL. The total length of the set-up clearly depends on the parameters of the tasks. The set-up involves the execution of only one job, in the case of constrained deadlines (see τ_1 in the figure). For the case of unconstrained deadlines tasks, note that for τ_3 job J_2 finishes $D_i - T_i$ before its deadline, J_3 finishes $D_i - 2T_i + C_i$ before its deadline and so on. The decreasing behavior is guaranteed by the fact that $C_i \leq T_i$ (clearly if $C_i = T_i$, the task can fill with executions any interval). We can continue packing executions up to when each added job finishes before its deadline: the jth job can be added only if $D_i - (j+1)\,T_i + jC_i \geq 0$. From this we obtain that the maximum number of jobs involved in the set-up is $\left\lfloor \frac{D_i - C_i}{T_i - C_i} \right\rfloor$ plus the last job J_1, for a total length of $\left(\left\lfloor \frac{D_i - C_i}{T_i - C_i} \right\rfloor + 1 \right) C_i$. This formula can be verified for the tasks in the figure (see Example 2 below). From the previous deadline backward, the behavior is periodic as is for EDF and EDZL.

EXAMPLE 2. Consider tasks τ_1, τ_2 and τ_3 in Figure 2.3. The parameters of the tasks are reported below.

i	C_i	D_i	T_i
1	7	15	19
2	7	24	19
2	7	24	11

Task τ_1 has constrained deadline and no set-up. Tasks τ_2 and τ_3 have unconstrained deadline, and have set-up phase involving respectively 2 and 5 jobs. □

This behavior complicates the analysis, both here and in the computation of the upper bound of the interference.

If the overload window is shorter than the length of the set-up phase, task τ_i is able to completely fill one processor in the interval of interest, so no shift (backward or forward) is needed to increase the amount of computation required by the task in the overload window. Let's consider an overload window larger than the length of the set-up phase.

Suppose to shift forward all the releases by an amount $x \leq C_i$. On the left side we obtain a maximum increase of x, but we have exactly the same decrease on the right side. If the shift is $x > C_i$, we surely lose the last job, since its release time is after d_k^j. We can maintain all the other jobs, by anticipating their execution. In fact, since the shift is assumed to be not more than T_i, all the release times of job previously included in the overload window (with the exclusion of the last job) remain at least C_i time instants before d_k^j. As an example, in Figure 2.3, see what happens to the release time of J_2 in the event of a forward shift of less than T_3. We could increase on the left side, because one job could enter the overload window. However, since we are supposing that the overload window is longer than the set-up phase, and $x \leq T_i$, the increase cannot involve more than one job. That is, the increase on the left side is surely counteracted by the decrease on the right side. As a consequence, the shift forward does not increase the maximum execution of τ_i in the overload window.

Consider now shifting backward. A shift of $x \geq T_i - C_i$ is necessary to have an increase of at most $x - (T_i - C_i)$ on the right side of the interval. However, such a shift is guaranteed to impose a decrease of at least the same amount on the left side depending on the position of the release times of τ_i with respect to the starting point of the overload window. Again, the shift does not increase the maximum execution of τ_i in the overload window.

As a consequence, the worst-case is the one depicted in Figure 2.3: strictly periodic releases such that the last job can execute exactly between its release time and d_k^j, and all the other jobs executed as late as possible.

Worst-case for EDZL. The case of EDZL has been analyzed in [**CB07**]. The worst-case release time sequence of τ_i is the same as for EDF, that is when one of the jobs of τ_i is released at $d_k^j - D_i$, so that its deadline is aligned with the deadline of the problem job. For completeness, we repeat here the analysis, referring to Figure 2.4.

Under EDZL, together with the jobs of τ_i with deadline in $[d_k^j - \Lambda_k, d_k^j)$, we must consider the contribution of jobs with deadline after d_k^j but with zero-laxity. However, these jobs can contribute only after having reached zero-laxity, which cannot happen before C_i time instants before their deadline (point A in Figure 2.4). This is equivalent to consider these jobs to execute exactly before their deadline. In such a situation, only one job of τ_i with deadline after d_k^j could contribute to $\beta_k^i(\Lambda_k)$.

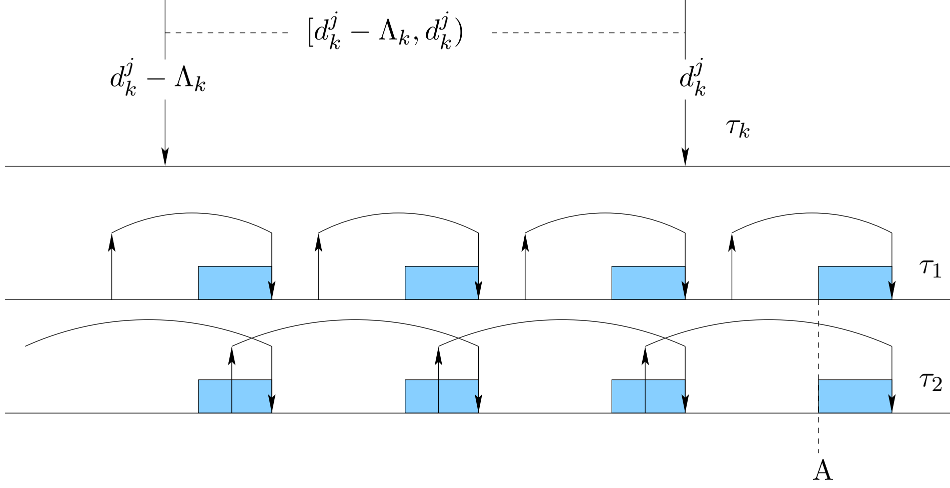

FIGURE 2.4. Worst-case workload under EDZL

Consider now what happens when shifting the release times from the situation depicted in the figure. Moving forward the releases by $x \leq T_i$ means to decrease $\beta_k^i(\Lambda_k)$ on the right side. If the job does not execute under zero-laxity, postponing its deadline means to reduce its priority, which means that it cannot contribute anymore to $\beta_k^i(\Lambda_k)$. The loss would be C_i. Instead, supposing the last job of T_i is executing at the end of its interval, and so with high priority, the minimum loss is $\min(x, C_i)$. At the same time, on the left side of the interval there is an increase of at most $\min(x, C_i)$. Complexively, the shift forward does not increase the maximum execution of τ_i in the overload window.

Even shifting backward does not increase the maximum execution of τ_i in the overload window. An increase on the right side can be obtained only if the shift is at least $x \geq T_i - C_i$, and in such a case the increase cannot be more than $x - (T_i - C_i)$. In contrast, it is simple to see that on the left side there is a decrease of at least the same quantity.

In the end, the worst-case release times sequence for EDZL is the same as for EDF: strictly periodic releases such that one deadline is aligned with d_k^j, and jobs executing exactly before their deadline.

5.3. Interference estimation. From the worst-case release times sequences proposed in the previous section for the three algorithms, we can now derive an upper bound of the interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$. Assuming that the jobs execute as in the figures, the upper bound will be composed of two distinct contributions:

- the *body*: a number N_i of jobs whose contribution is the complete execution C_i
- the *carry-in*: at most one job, called *carried-in job*, whose contribution could be less than the whole computation time C_i; we use ε_i to refer to the upper bound of such contribution.

There are several possibilities to account for the two contributions. We could consider one job to be part of the body whenever the overload window

is large enough to include its release time. This solution is good for constrained deadline tasks, while it does not fit well for unconstrained deadlines tasks, since we account for a job while a new one could already be accounted for (see τ_2 in Figures 2.2, 2.3 and 2.4). Another possibility is to consider the deadline of the previous job, supposing that each job suffers the worst-case precedence-blocking and so it is forced to execute only after the deadline of the previous job). This has been the approach in [**CB07**]. We could also suppose that each job executes exactly before its deadline, and so account for a new job in the body if the overload window includes the whole execution (that is, it reaches $d_i^j - C_i$). This is justified by the fact that, as we saw, the situation in which each task execute as late as possible is the one that maximizes the upper bound of the interference.

For what concerns the carry-in, we always suppose the worst-case situation: the carry-in job executes exactly before its deadline. As a consequence, the carry-in contribution is limited by the length of the job, C_i, and the length of the fraction of overload window in which the carry-in job can execute.

We will not deepen this analysis, but we underline that, if the formulae are well-defined, they account for the same base situation (the situation in which all the jobs are packed in order to maximize the execution in the overload window), so there is actually no difference among the three approaches: what does change is only what is accounted for in the body and in the carry-in, while the sum remains the same.

Due to mathematical simplicity, we prefer the second approach, and we account for a new job in the body when the overload window includes the deadline of the previous job. In Section 5.6 we report the formulae obtained with the other approaches, and we show, with an example, that the final result is the same.

EDF and EDZL have the same worst-case release time sequence, and can be treated together. Instead, FP needs a dedicated analysis.

5.4. Upper bound for EDF and EDZL. Consider the release times sequence in Figures 2.2 and 2.4. Since we want to account for a new job in the body if the overload window is large enough to include the deadline of the previous job, and the first deadline coincides with the end of the overload window, the value of N_i can be computed as

$$N_i = \left\lfloor \frac{\Lambda_k}{T_i} \right\rfloor . \tag{5.4}$$

With this choice for the body, the formula of N_i remains the same for both constrained and unconstrained deadlines.

The carry-in contribution is bounded by C_i and the interval between the start time of the overload window and the first deadline of τ_i inside the overload window. In formula,

$$\varepsilon_i = \min\left(C_i, \Lambda_k - N_i T_i\right) . \tag{5.5}$$

For EDF and EDZL, we obtain the following upper bound of the interference provoked by τ_i in the overload window $[d_k^j - \Lambda_k, d_k^j)$ of the problem job τ_k^j:

(5.6) $I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = N_iC_i + \varepsilon_i = N_iC_i + \min(C_i, \Lambda_k - N_iT_i)$.

As a consequence, the upper bound on interference when EDF of EDZL are used, is expressed in the following system of equations:

$$\begin{cases} N_i = \left\lfloor \dfrac{\Lambda_k}{T_i} \right\rfloor \\ \varepsilon_i = \min(C_i, \Lambda_k - N_iT_i) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = N_iC_i + \varepsilon_i \end{cases} \tag{5.7}$$

5.5. Upper bound for FP. The case of FP is complicated by the presence of the set-up phase in which τ_i can completely fill one processor. If the overload window is shorter than the set-up phase, we could easily consider this fact in order to give an upper bound on the interference. However, we prefer to cope with the computation of the upper bound exactly as in the case of EDF and EDZL. Thanks to this fact, the math is easier, we can unify the cases of constrained and unconstrained deadlines, and we maintain the similarities with the case of EDF and EDZL. In the set-up phase we overestimate the interference, but this overestimation can be corrected by considering that the interference is always bounded by the length of the overload window. However, we will see that in the schedulability test this bound is not necessary, since a more strict one is used in all the tests.

Consider the release times sequence in Figure 2.3, and suppose there are no jobs of τ_i^j prior to the ones in the figure. As for EDF and EDZL, we want to account for a job in the body if the overload window is large enough to include the deadline of the previous job. In this case, however, we must include in the body also the jobs whose deadlines are after d_k^j, but that are anyway able to execute in the overload window. That is, we include also the jobs with release time at least C_i time instants earlier than d_k^j.

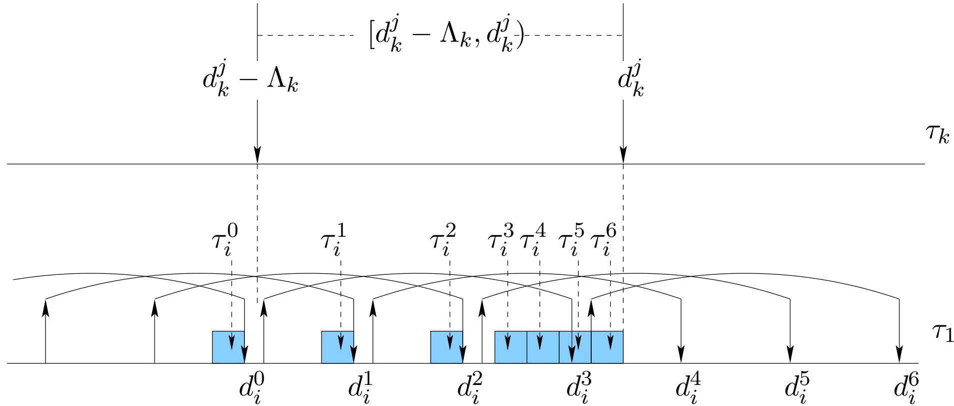

FIGURE 2.5. Deadline positions under FP

As an example, consider Figure 2.5. τ_i^2 and τ_i^3 are included in the body because, in the worst-case, they are supposed to completely execute respectively in $[d_i^1, d_i^2)$ and $[d_i^2, d_i^3)$, and such intervals are completely included in

the overload window. Instead, τ_i^4 to τ_i^6 are included in the body because, even if they have deadline after d_k^j, there is enough time between their release and d_k^j for them to execute. The two subsets of jobs have in common the fact that their deadlines fall in the interval $(d_k^j - \Lambda_k, d_k^j + D_i - C_i]$. The term $D_i - C_i$ in the right extreme of the interval descends from the fact that the last job is released at $r_i^j = d_k^j - C_i$ and so its deadline is at $r_i^j + D_i = d_k^j - C_i + D_i$. As a consequence, the value of N_i can be computed as

$$N_i = \left\lfloor \frac{\Lambda_k + D_i - C_i}{T_i} \right\rfloor . \tag{5.8}$$

As for EDF and EDZL, the value of N_i remains correct for both constrained and unconstrained deadlines.

The carry-in is bounded by the same values, although this time the length of the interval in which the carried-in job executes is computed differently. In particular, as in Equation (5.8), we add $D_i - C_i$ to the formula, obtaining

$$\varepsilon_i = \min\left(C_i, (\Lambda_k + D_i - C_i) - N_i T_i\right) \tag{5.9}$$

The larger positive term is offset by the larger value of N_i, so that the final upper bound represents exactly the same concept: the length of the interval between $d_k^j - \Lambda_k$ and the first deadline of τ_i inside the overload window.

Recall that if the overload window is shorter than the set-up phase of τ_i, we have to limit the value of $\beta_k^i(\Lambda_k)$ with the length of the overload window. The upper bound of the interference provoked by τ_i in the overload window $[d_k^j - \Lambda_k, d_k^j)$ of the problem job τ_k^j, when FP is selected, is

$$\begin{aligned} I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) &= \min\left(N_i C_i + \varepsilon_i, \Lambda_k\right) = \\ &= \min\left(N_i C_i + \min\left(C_i, (\Lambda_k + D_i - C_i) - N_i T_i\right), \Lambda_k\right). \end{aligned} \tag{5.10}$$

Summarizing, the upper bound of the interference for FP can be expressed by the following system:

$$\begin{cases} N_i = \left\lfloor \dfrac{\Lambda_k + D_i - C_i}{T_i} \right\rfloor \\ \varepsilon_i = \min\left(C_i, (\Lambda_k + D_i - C_i) - N_i T_i\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = \min\left(N_i C_i + \varepsilon_i, \Lambda_k\right). \end{cases} \tag{5.11}$$

5.6. Alternative approaches. For the sake of completeness, in this section we briefly report the formulae we obtain using other possible approaches to the computation of body and carry-in. Computing them is not difficult, and requires to follow exactly the same steps as done in Sections 5.4 and 5.5.

Above, we supposed to account for a new job in the body if the overload window includes the deadline of the previous job. Suppose now to use, as a discriminant, the interval in which the job can surely execute: from r_i^j to d_i^j for constrained deadlines, and from d_i^{j-1} to d_i^j for unconstrained deadlines.

In such a case we obtain the following formulae for EDF and EDZL:

$$\begin{cases} N_i = \left\lfloor \dfrac{\Lambda_k - \Lambda_i}{T_i} \right\rfloor + 1 \\ \varepsilon_i = \min\left(C_i, \max\left(0, \Lambda_k - N_i T_i\right)\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = N_i C_i + \varepsilon_i \end{cases} \tag{5.12}$$

For FP, the formulae are as below:

$$\begin{cases} N_i = \left\lfloor \dfrac{(\Lambda_k + D_i - C_i) - \Lambda_i}{T_i} \right\rfloor + 1 \\ \varepsilon_i = \min\left(C_i, \max\left(0, (\Lambda_k + D_i - C_i) - N_i T_i\right)\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = \min\left(N_i C_i + \varepsilon_i, \Lambda_k\right). \end{cases} \tag{5.13}$$

The above equations reduce exactly to Equations (5.11) and (5.7) in the case of unconstrained deadlines, since in such a case in both approaches we consider a job in the body if the overload window includes the previous deadline.

Instead, since Figures 2.2, 2.3 and 2.4 represent not only the worst-case release time sequence but also the worst-case execution, we could think to account for a new job in the body whenever the overload window includes its entire execution in the worst-case. Following this approach, the formulae for EDF and EDZL are

$$\begin{cases} N_i = \left\lfloor \dfrac{\Lambda_k - C_i}{T_i} \right\rfloor + 1 \\ \varepsilon_i = \max\left(0, \Lambda_k - N_i T_i\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = N_i C_i + \varepsilon_i. \end{cases} \tag{5.14}$$

Under FP, we obtain

$$\begin{cases} N_i = \left\lfloor \dfrac{(\Lambda_k + D_i - C_i) - C_i}{T_i} \right\rfloor + 1 = \left\lfloor \dfrac{\Lambda_k + D_i - 2C_i}{T_i} \right\rfloor + 1 \\ \varepsilon_i = \max\left(0, (\Lambda_k + D_i - C_i) - N_i T_i\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = \min\left(N_i C_i + \varepsilon_i, \Lambda_k\right). \end{cases} \tag{5.15}$$

Note that, as above, for FP the set-up phase is quite difficult to consider. So we use a generic upper bound, valid in any case, and then we further limit the obtained upper bound of the interference with the length of the overload window.

In order to identify the three approaches, we call them respectively D_i-based, Λ_i-based and C_i-based upper bounds.

Remember that all the three approaches described above to compute an upper bound on the interference bring to *exactly the same upper bound*, as can be seen in the case of EDF and EDZL in Example 3 below. The difference among the three approaches is only in the mathematical complexity of the formulae, and the fact that in the same time instant, depending on the chosen approach, the job can be accounted for in the body or in the carry-in.

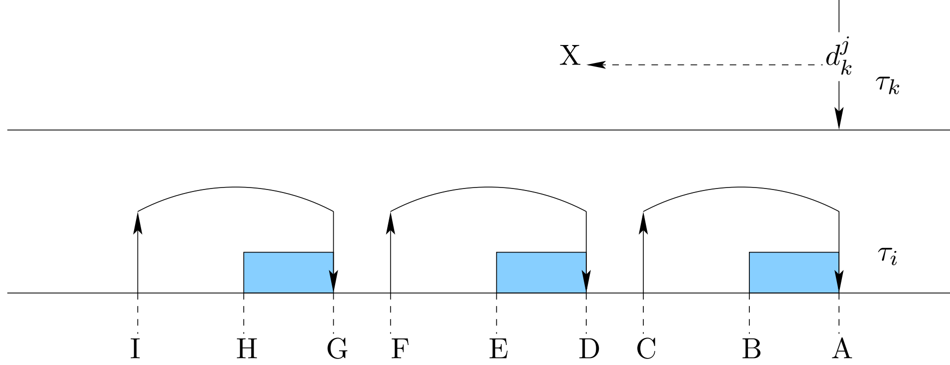

FIGURE 2.6. Worst-case execution of τ_i under EDF and EDZL

EXAMPLE 3. Consider the worst-case release times sequence under EDF and EDZL, for task τ_i in Figure 2.6 below.

In Table 2 we report the values of N_i, ε_i and $\beta_k^i(\Lambda_k)$ computed with the three approaches described above, as functions of the length X of the overload window $[d_k^j - \Lambda_k, d_k^j)$. We considered only boundary time instants, at which the values are discontinuous. Clearly when X increases between A and B, ε_i increases linearly as well in all the approaches, maintaining always the same value, as it does between D and E and between G and H.

Approach \ Point		A	B	C	D	E	F	G	H	I
D_i-based	N_i	0	0	0	1	1	1	2	2	2
	ε_i	0	C_i	C_i	0	C_i	C_i	0	C_i	C_i
	β_k^i	0	C_i	C_i	C_i	$2C_i$	$2C_i$	$2C_i$	$3C_i$	$3C_i$
Λ_i-based	N_i	0	0	1	1	1	2	2	2	3
	ε_i	0	C_i	0	0	C_i	0	0	C_i	0
	β_k^i	0	C_i	C_i	C_i	$2C_i$	$2C_i$	$2C_i$	$3C_i$	$3C_i$
C_i-based	N_i	0	1	1	1	2	2	2	3	3
	ε_i	0	0	0	0	0	0	0	0	0
	β_k^i	0	C_i	C_i	C_i	$2C_i$	$2C_i$	$2C_i$	$3C_i$	$3C_i$

TABLE 2. Computation of $\beta_k^i(\Lambda_k)$ with the three approaches.

We can see that despite the differences in N_i and ε_i in different approaches, the final result for $\beta_k^i(\Lambda_k)$ is always the same. If we consider, instead of task τ_i in the figure, a task with the same C_i and D_i, but $T_i^{'} > D_i$ (that is, an unconstrained deadline task), the worst-case execution does not change with respect to the figure, and only the releases are anticipated. In such a case, it is easy to see that D_i-based and C_i-based approaches remain exactly the same, because they do not involve in any way the release times. For what relates to the Λ_i-based approach, we said above that this is equivalent to the D_i-based approach for unconstrained deadlines, and so it is equivalent to the previous ones.

□

An example for the case of FP would be slightly more complex, due to the presence of the set-up phase. However, as for EDF and EDZL, we could verify that N_i and ε_i can change in different approaches, but their changes are balanced, so that the final result for $\beta_k^i(\Lambda_k)$ is not affected by the selected approach.

5.7. Upper bound refinement. In the previous section, we proposed upper bounds on the interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$ given by the amount of computation that a task τ_i can require in the overload window. However, this is quite pessimistic, and further analysis are in order to decrease such an estimation. We can improve the upper bound of the interference as follows. We underline that the results below are valid for all the algorithms under analysis (EDF, FP or EDZL). Moreover, they can be extended to other algorithms or classes of algorithms.

Let's focus on the overload window of the problem job. Please recall that in such an interval the problem job cannot suffer precedence-blocking. Thus, every time it is blocked, it is blocked by priority. As a consequence, in every time instant in which τ_k^j is priority-blocked, the M processors must be occupied by exactly M jobs of tasks other than τ_k, and each of these jobs has priority higher than the problem job. Consequently, the respective M values of interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$ increase. From this descends that

$$I_k(d_k^j - \Lambda_k, d_k^j) = \frac{\sum_{i \neq k} I_k^i(d_k^j - \Lambda_k, d_k^j)}{M}. \tag{5.16}$$

Using this result we can prove the following lemma, which will be useful in improving the estimation of interference.

LEMMA 5.5 (Bound on considered interference). $I_k(d_k^j - \Lambda_k, d_k^j) \geq x \iff \sum_{i \neq k} \min\left(I_k^i(d_k^j - \Lambda_k, d_k^j), x\right) \geq Mx$

PROOF. *If.* From the hypothesis, it follows that

$$\begin{aligned} I_k(d_k^j - \Lambda_k, d_k^j) &= \sum_{i \neq k} \frac{I_k^i(d_k^j - \Lambda_k, d_k^j)}{M} \geq \\ &\geq \sum_{i \neq k} \frac{\min\left(I_k^i(d_k^j - \Lambda_k, d_k^j), x\right)}{M} \geq \\ &\geq \frac{Mx}{M} = x. \end{aligned}$$

Only If. Let $\mathcal{T}' \subseteq \mathcal{T}$ be the set of tasks τ_i for which $I_k^i(d_k^j - \Lambda_k, d_k^j) \geq x$, and ξ the cardinality of $\mathcal{T}'$. If $\xi \geq M$ the lemma directly follows, so we consider only $\xi < M$.

$$\sum_{i \neq k} \min\left(I_k^i(d_k^j - \Lambda_k, d_k^j), x\right) = \xi x + \sum_{\tau_i \notin \mathcal{T}'} I_k^i(d_k^j - \Lambda_k, d_k^j) =$$
$$= \xi x + M I_k(d_k^j - \Lambda_k, d_k^j) - \sum_{\tau_i \in \mathcal{T}'} I_k^i(d_k^j - \Lambda_k, d_k^j) \geq$$
$$\geq \xi x + M I_k(d_k^j - \Lambda_k, d_k^j) - \xi I_k(d_k^j - \Lambda_k, d_k^j) =$$
$$= \xi x + (M - \xi)\, I_k(d_k^j - \Lambda_k, d_k^j) \geq \xi x + (M - \xi)\, x = Mx.$$

□

The idea behind this Lemma is the following: if the interference suffered by the problem job τ_k^j is x, each one of the other tasks τ_i cannot contribute to the sum of Equation (5.16) with more than x. Hence, in order to verify the thesis ($I_k(d_k^j - \Lambda_k, d_k^j) = x$) we can consider each single interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$ as to be limited by x. In other words, it is sufficient to consider each contribution $I_k^i(d_k^j - \Lambda_k, d_k^j)$ up to x, if we want to show that the interference $I_k(d_k^j - \Lambda_k, d_k^j)$ is at least x.

Note that if we use of $>$ on both sides of the statement, the *Only if* part of the lemma is not true anymore. Suppose that exactly M tasks contribute to the interference suffered by τ_k, with a contribution $y > x$. Limiting each single contribution to x, the strict equality holds in the sum on the right side, even if the interference suffered by τ_k is $I_k(d_k^j - \Lambda_k, d_k^j) > x$. On the contrary, it is sufficient to repeat the proof to verify that the *If* part of the lemma remains correct if we substitute $\geq$ with $>$.

6. Schedulability tests

From the upper bounds on interference proved for EDF, EDZL and FP, we can now propose a schedulability test for each scheduling algorithm. In this section, we use the D_i-based upper bound (i.e. Equations 5.7 and 5.11), but it is easy to propose equivalent schedulability tests for the other solutions described in Section 5.6.

6.1. EDF schedulability test. In order to propose a computable schedulability test for EDF, it is sufficient to verify, using the upper bound proposed in Section 5.4, if the interference can be sufficient to force a deadline miss. We obtain the following theorem.

THEOREM 6.1 (BCL-EDF Test). *A task set $\mathcal{T}$ is schedulable on M processors by EDF if, for every task τ_k*

$$\sum_{i \neq k} \min\left(\beta_k^i(\Lambda_k), \Lambda_k - C_k + 1\right) < M\left(\Lambda_k - C_k + 1\right). \tag{6.1}$$

where

$$\beta_k^i(\Lambda_k) = N_i C_i + \min\left(C_i, \Lambda_k - N_i T_i\right)$$

and

$$N_i = \left\lfloor \frac{\Lambda_k}{T_i} \right\rfloor.$$

Proof. Suppose the inequality holds for τ_k. From Lemma 5.5 and the upper bound on interference under EDF, given in Equation (5.7), we have that $I_k(d_k^j - \Lambda_k, d_k^j) < (\Lambda_k - C_k + 1)$ for each job of τ_k. As a consequence, by Lemma 5.3 no job of τ_k can reach negative laxity and miss its deadline. If that is true for every task, no deadlines can be missed in the system and so the task set is schedulable. □

6.2. FP schedulability test. For FP, we can follow the same approach as for EDF. However, we can take into account the fact that the M tasks with higher priority can never miss their deadline, since they will always have a processor assigned at the moment they are released (by preempting a lower priority task, if necessary). As a consequence, we can limit our attention to the other tasks. The obtained test is expressed in the theorem below.

Theorem 6.2 (BCL-FP Test). *A task set $\mathcal{T}$ is schedulable on M processors by FP if, for every task τ_k (where tasks are ordered by priority and $k > M$)*

$$\sum_{i<k} \min\left(\beta_k^i(\Lambda_k), \Lambda_k - C_k + 1\right) < M\left(\Lambda_k - C_k + 1\right). \tag{6.2}$$

where

$$\beta_k^i(\Lambda_k) = N_i C_i + \min\left(C_i, (\Lambda_k + D_i - C_i) - N_i T_i\right)$$

and

$$N_i = \left\lfloor \frac{\Lambda_k + D_i - C_i}{T_i} \right\rfloor.$$

Proof. Suppose the inequality holds for τ_k. From Lemma 5.5 and the upper bound on interference under FP, given in Equation (5.11), we have that $I_k(d_k^j - \Lambda_k, d_k^j) < (\Lambda_k - C_k + 1)$ for each job of τ_k. As a consequence, by Lemma 5.3 no job of τ_k can reach negative laxity and miss its deadline. If that is true for every task, no deadlines can be missed in the system and so the task set is schedulable. □

Note that the upper bound to the interference represented by $\beta_k^i(\Lambda_k)$ does not include the minimum with Λ_k introduced in Equation (5.11). Such bound was conceptually useful to avoid overestimations of the interference in the set-up phase. However, as anticipated in Section 5.5, it becomes useless, since the minimization in Equation (6.2) is more strict.

Note again that the sum in Equation (6.2) involves only tasks with priority higher than the task τ_k under analysis. This seems to be a great advantage of this test with respect to the equivalent test for EDF.

6.3. EDZL schedulability test. When EDZL is selected, the test proposed for EDF in Section 6.1 could be used (since each test for EDF is also a test for EDZL). However, such test can be greatly improved by a fact underlined at the end of Section 5.1. As we said, when the system is scheduled with EDZL, a deadline can be missed only if there can be at least $M + 1$ jobs with zero laxity at the same time. This concept would be hidden in the definition of interference, if we were able to compute it, but it is lost in moving to the upper bound. We can re-introduce it by requiring

the presence of at least $M+1$ tasks for which the zero laxity condition is reachable. The test can be expressed as in the theorem below.

THEOREM 6.3 (EDZL Test). *A task set $\mathcal{T}$ is schedulable on M processors by EDZL unless there exist at least $M+1$ tasks τ_k for which*

$$\sum_{i \neq k} \min\left(\beta_k^i(\Lambda_k), \Lambda_k - C_k + 1\right) \geq M\left(\Lambda_k - C_k + 1\right). \tag{6.3}$$

where

$$\beta_k^i(\Lambda_k) = N_i C_i + \min\left(C_i, \Lambda_k - N_i T_i\right)$$

and

$$N_i = \left\lfloor \frac{\Lambda_k}{T_i} \right\rfloor,$$

and for at least one of them the $>$ strictly holds in Equation (6.3).

PROOF. Suppose Equation (6.3) holds for less than $M+1$ tasks. From Lemma 5.5 and the upper bound on interference under EDZL, given in Equation (5.7), we have that $I_k(d_k^j - \Lambda_k, d_k^j) \geq (\Lambda_k - C_k + 1)$ for less than $M+1$ tasks. As a consequence, by Corollary 5.4, less than $M+1$ tasks can reach zero laxity. Whenever one job of these tasks reaches zero laxity, it is scheduled for execution and cannot be preempted, so it will not miss its deadline. Hence, no deadline can be missed in the system.

Now suppose Equation (6.3) holds for at least $M+1$ tasks, but for none of them it holds with the strict $>$. In such a case, by Lemma 5.3 no job can reach negative laxity, and so there cannot be deadline miss. □

We have to note that this theorem is extremely pessimistic: what is really important is not if $M+1$ tasks can reach zero or negative laxity, but if they can do that *at the same time*, which is a more stringent requirement. Unfortunately, this condition is quite complicated to verify, and we are not aware of any feasible approach to this problem, apart from the simulation of the system.

6.4. Comments on the tests. To better understand the key idea behind the schedulability tests proposed above, consider that we are not interested in computing the actual worst-case interference for a job, but only in verifying if it is greater or equal than $\Lambda_k - C_k + 1$ (which is the necessary condition for a deadline miss, see Lemma 5.3). In order to verify this, it is sufficient to consider, for each task τ_i a contribution to the interference not greater than $\Lambda_k - C_k + 1$. If the sum of these bounded contributions is greater than or equal to $M(\Lambda_k - C_k + 1)$, then the total interference $I_k(d_k^j - \Lambda_k, d_k^j)$ is at least $\Lambda_k - C_k + 1$ (by Lemma 5.5). In the other case, the total interference cannot be greater than $\Lambda_k - C_k$, and so the task is surely schedulable. The term $(\Lambda_k - C_k + 1)$ in the minimum is one of the main differences between our work published in [**BCL05a**] and [**BCL05b**] (in a slightly different form) and the results presented in [**Bak05**] and [**Bak06a**]. Through experiments, it is easy to verify that this brings a substantial improvement on the schedulability test. In [**BC06c**], instead, the analysis by Baker [**Bak05, Bak06a**] is integrated with the minimization of the interference described above, and the results are quite interesting, especially for FP.

It could appear that the use of the +1 in the formulae is superfluous, and one could equivalently use

$$\sum \min\left(\beta_k^i(\Lambda_k), \Lambda_k - C_k\right) \le M\,(\Lambda_k - C_k)$$

Indeed, this is not true. Suppose that the interference to τ_k^j is due to exactly M tasks, and each of them contributes with $\Lambda_k - C_k + x$, with $x \ge 1$ (remember that x is surely an integer, due to the time division). The total interference $I_k(d_k^j - \Lambda_k, d_k^j)$ is so $\Lambda_k - C_k + x$, and τ_k^j is not schedulable, due to Lemma 5.3. However, limiting to $\Lambda_k - C_k$ the single contribution, we would obtain that the sum on the left of the equation is exactly $M\,(\Lambda_k - C_k)$ and the test would be passed. Using the strict inequality and the +1 term helps in avoiding this counter effect of the interference limitation, and gives a further explanation of the discussion related to Lemma 5.3.

7. Improving the analysis through the slack

The theorems of the previous section suffer from the gross overestimation of the carry-in done in Section 5.3. Through experiments, it is easy to see that this almost counterbalances the improvements introduced by bounding the interference with the term $\Lambda_k - C_k + 1$, with respect, for example, to the BAK-EDF test of Section 3.1. For this reason, it is of primary importance to find a way to give a better estimation of the carry-in. With this goal in mind, a further analysis of the meaning of Theorems 6.1, 6.2 and 6.3 is useful. Consider for example the case of EDF, and, in applying the test of Theorem 6.1, suppose that for a certain task τ_k we find that the inequality holds, and in particular

$$(\Lambda_k - C_k + 1) - \frac{\sum_{i \ne k} \min\left(\beta_k^i(\Lambda_k), \Lambda_k - C_k + 1\right)}{M} = x > 0.$$

This means not only that no job of τ_k will miss its deadline, but also that, in the worst-case, each job τ_k^j will finish at least $\lceil x - 1 \rceil$ time instants before its deadline. That is,

$$S_k \ge \left\lceil (\Lambda_k - C_k) - \frac{\sum_{i \ne k} \min\left(\beta_k^i(\Lambda_k), \Lambda_k - C_k + 1\right)}{M} \right\rceil. \tag{7.1}$$

To see this, remember that the sum represents an upper bound on the interference $I_k(d_k^j - \Lambda_k, d_k^j)$ that all the tasks other than τ_k can impose on τ_k^j, while the first term is the interference necessary to force a deadline miss. If $x \in (0, 1]$, considering the time division discussed in 2.3, we have that in the overload window, in the worst-case, we have M processors occupied at the same time for at most $\Lambda_k - C_k$, and only $M - 1$ processors occupied for an additional interval of length 1. That is, in the overload window one of the M processor is idle (and so will execute τ_k^j) for at least $\Lambda_k - (\Lambda_k - C_k) = C_i$, guaranteeing to complete τ_k^j in time. It is straightforward to replicate the reasoning for different values of x and different scheduling algorithms (using the respective tests).

As an example, in Figure 2.7, we suppose to have 3 processors, and a job of $\tau_k = (C_k = 5, D_k = 10, T_k > D_k)$ to be executed. Suppose the upper bound on the interference computed is $M\,(\Lambda_k - C_k + 1) = 17$. This amount

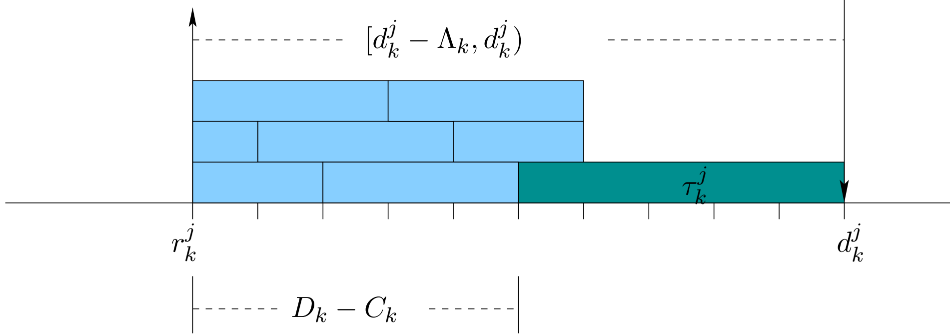

FIGURE 2.7. Example of slack

of computation, in the worst-case, is distributed among the processors as in the figure, with 6 units of computation on 2 processors, and only 5 on the third processor. In such a situation, τ_k^j can complete its computation in time by executing on the third processor.

The value computed with Equation (7.1) represents a lower bound on the slack S_k of a task τ_k, as defined in Section 2. This lower bound can be used when the task is the one under analysis, but also when the task is one of the tasks contributing to the interference of another task under analysis. From now on we refer to both the slack of τ_i and its lower bound with the same symbol S_i, specifying what we mean only when it is necessary.

We can use the proposed bound on the slack to improve the analysis proposed in Theorems 6.1, 6.2 and 6.3, in three different aspects: not only the estimation of the carry-in, but also the length of the overload window and the number of jobs included in the body.

Due to the easier formulae we obtain after the improvements described below, we prefer to start from the Λ_i-based upper bounds on interference proposed in Section 5.6 (instead of the D_i-based upper bound of Sections 5.4 and 5.5. We report them here to help the understanding. For EDF and EDZL, we found (see Equation (5.12))

$$\begin{cases} N_i = \left\lfloor \dfrac{\Lambda_k - \Lambda_i}{T_i} \right\rfloor + 1 \\ \varepsilon_i = \min\left(C_i, \max\left(0, \Lambda_k - N_i T_i\right)\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = N_i C_i + \varepsilon_i \end{cases} \tag{7.2}$$

Instead, for FP, we had (as in Equation (5.13))

$$\begin{cases} N_i = \left\lfloor \dfrac{(\Lambda_k + D_i - C_i) - \Lambda_i}{T_i} \right\rfloor + 1 \\ \varepsilon_i = \min\left(C_i, \max\left(0, (\Lambda_k + D_i - C_i) - N_i T_i\right)\right) \\ I_k^i(d_k^j - \Lambda_k, d_k^j) \leq \beta_k^i(\Lambda_k) = \min\left(N_i C_i + \varepsilon_i, \Lambda_k\right). \end{cases} \tag{7.3}$$

7.1. Improving the overload window. In Section 5.1, we identified the overload window as the interval in which the problem job does not suffer precedence-blocking. Thanks to the lower bound on the slack, the overload window can be re-analyzed and possibly enlarged.

For implicit and constrained deadline tasks, the slack analysis does not help. In such a case, the start time of the overload window is fixed to the release time of the job. This cannot be improved following the slack analysis, since clearly τ_k^j cannot execute before r_k^j.

For unconstrained deadline tasks, we fixed to $d_k^j - T_i = d_k^{j-1}$ the start time of the overload window, basing on the fact that this is the first time instant after which we are sure that τ_k^j does not suffer precedence-blocking. However, due to the reasoning above, we know that

$$f_k^{j-1} \leq d_k^{j-1} - S_k = d_k^j - (T_k + S_k) \, .$$

That is, we can enlarge the overload window by S_k. Clearly the problem job cannot start executing before its release time, so the overload window must be defined as

$$[d_k^j - (T_k + S_k) \, , d_k^j) \cap [r_k^j, d_k^j),$$

for a total length of $\min(D_k, T_k + S_k)$. This formula remains valid also for implicit and constrained deadline tasks, considering that for such tasks D_k will be always selected in the minimum.

Since from now on $\Lambda_k = \min(D_k, T_k)$ is substituted in several of the formulae by the value $\min(D_k, T_k + S_k)$, from now on we define $\overline{\Lambda}_k = \min(D_k, T_k + S_k)$.

The different definition of the overload window allows to modify the analysis, since Lemma 5.3 and the derived corollary can be modified to take into account the new value. We obtain that a job can reach negative laxity only if $I_k(d_k^j - \overline{\Lambda}_k, d_k^j) > \overline{\Lambda}_k - C_k + 1$.

From this, two modifications are derived in Theorems 6.1, 6.2 and 6.3:

- the right hand side of the inequalities in the schedulability tests is increased of MS_k, making it easier for a task τ_k to pass the test and result to be schedulable;
- the upper bound on the interference of tasks other than τ_k has to be computed in a different interval; since each task has an interval S_k time instants larger, and could completely fill it, the left hand side of the inequalities in the schedulability tests could increase by at most NS_k, making it more difficult for a task τ_k to pass the test.

The net result could be a loss, and computing a new lower bound on the slack S_k would bring to a worse estimation. In such a case, it is not convenient to take into account the new overload window. It is better to maintain the length of the overload window that originated the best estimation for the slack, in the hope that new bounds on the slack of other tasks allow for improvements in the overload window of τ_k. However, the previous, higher, lower bound on the slack remains valid, and can be used in the analysis of other tasks. In other words, changing the length of the overload window brings to a new estimation of the lower bound of the slack S_k: the lower bound to be used in the formulae is the highest value computed so far, and not necessarily the last one.

Note that this reasoning is valid also if at the first step we find a negative bound for the slack. In such a case, as said above, it is not convenient to take into account it to shrink the overload window, and we maintain the original interval.

7.2. Improving the body. In order to estimate the upper bound on the interference, it is necessary to compute the number of job completely included in the overload window. As we noticed above, increasing S_k modifies the length of the overload window, forcing to recompute the value of N_i for each task τ_i. However, if τ_i has unconstrained deadline, another interesting modification is introduced by the increase in the lower bound on S_i. In fact, since a job τ_i^j can execute in an interval whose length is increased (as for τ_k^j) to $\overline{\Lambda}_i$, we can repeat the analysis of Section 5.3. As above, we split the analysis for EDF and EDZL from the analysis for FP.

EDF and EDZL. The number of jobs included in the body, in the case of constrained deadlines tasks, is not influenced by the introduction of the slack. Instead, for unconstrained deadlines tasks, we want to account for a new job in the body only if the overload window is large enough to include the latest possible finish time of the previous job, that is the deadline minus the lower bound on the slack. The formula for N_i, valid for arbitrary deadlines, becomes

$$N_i = \left\lfloor \frac{\overline{\Lambda}_k - \overline{\Lambda}_i}{T_i} \right\rfloor + 1 \tag{7.4}$$

where the introduction of S_i is hidden in the term $\overline{\Lambda}_i$. Note that, as expected, S_i does not influence the value of N_i, if τ_i has constrained deadline. Note also that in the formula we include also the new length of the overload window, represented by $\overline{\Lambda}_k$.

It must be noted that, if $S_i \geq \overline{\Lambda}_k$, we could find $N_i \leq -1$, which clearly does not make sense. In effect, under EDF and EDZL if a task τ_i has minimum slack greater than the overload window, it means that it cannot contribute to the interference of the problem job. In fact, $S_i > 0$ means that the job can never reach zero laxity, so its priority depends only on the position of its deadline. However, if $S_i > \overline{\Lambda}_k$, either the job has lower priority (due to its deadline being after d_k^j) or it finishes before $d_k^j - \overline{\Lambda}_k$ (that is, before the start time of the overload window. In such a case, $I_k^i(d_k^j - \Lambda_k, d_k^j) = 0$, and the contribution of τ_i to the upper bound on interference must be considered null.

FP. For the case of FP, the modification is similar. For constrained deadline tasks, there is no difference, since we should continue to consider a job in the body if its release time is included in the overload window. Instead, for unconstrained deadlines tasks, the job must be accounted for only if the overload window includes the maximum finish time of the previous job. This change requires, as above, the introduction of the factor S_i in the formula of N_i. The final formula, for arbitrary deadlines tasks, is based on the previous defined value $\overline{\Lambda}_i$, and is the following:

$$N_i = \left\lfloor \frac{(\overline{\Lambda}_k + D_i - C_i) - \overline{\Lambda}_i}{T_i} \right\rfloor + 1. \tag{7.5}$$

As for EDF and EDZL, the presence of S_i is hidden in $\overline{\Lambda}_i$, and is such that it does not introduce modifications in the case of constrained deadlines. Moreover, we introduced also $\overline{\Lambda}_k$ to represent the new overload window.

In this case, the problem of negative values for N_i described for EDF and EDZL cannot show up, because the term $D_i - C_i$ added to the formula avoid this. In fact, by their meaning, $S_i \leq D_i - C_i$, and so the formula is always strictly non negative. In effect, we can expect this behavior: under FP, a task with higher priority than τ_k can always produce some interference independently on slack and deadline positions.

7.3. Improving the carry-in. The final difference introduced by the lower bound on the slack relates to the original problem: better estimating the carry-in. Since the carried-in job of τ_i finishes at least S_i time instants before its deadline, the interval in which it can execute inside the overload window is reduced by the same amount. As a consequence, in bounding the carry-in, the formula can become

$$\varepsilon_i = \min\left(C_i, \max\left(0, \Lambda_k - N_i T_i - S_i\right)\right) \tag{7.6}$$

for EDF and EDZL, and

$$\varepsilon_i = \min\left(C_i, \max\left(0, \left(\overline{\Lambda}_k + D_i - C_i\right) - N_i T_i - S_i\right)\right) \tag{7.7}$$

for FP.

8. Schedulability tests: recursive approach

Using the results discussed above, we propose an improved sufficient schedulability test, based on the fact that the value of any of the lower bounds on the slacks directly depends on, and influences, the values of all the others. The idea is to search, recursively, for the best possible estimation of all the S_i: at each step compute all the S_i, and verify if the new values can help in improving them in the next step. When no further improvement is possible, the behavior depends on the scheduling algorithm selected, as in the tests of Section 6:

- for EDF and FP, if all the lower bounds on the slacks are positive, then no task can miss its deadline, and so the task set is schedulable; instead if at least one task exists for which the lower bound on the slack is non-positive, nothing can be said;
- for EDZL, if not more than M tasks can have non-positive slack, then the task set is schedulable, while if at least $M + 1$ tasks can have non-positive slack, then nothing can be said.

We can now express new schedulability tests based on the above discussion. The *core* of the test, reported in Figure 2.8, takes the lead from the systems of inequalities (7.2) for EDF and EDZL, and (7.3) for FP, in which we include the improvements due to the slack.

In the core, for a task $\mathcal{T}_i$ we always take into account the higher estimation of S_i found so far. In the analysis of tasks other than T_i, the value of S_i is always used directly in the formulae. However, when T_i is under analysis, the best value of S_i can be taken into account in two different ways. We use parameter OW to chose between the two possibilities. If the last estimation of the slack produced an increase, it is convenient to search for new improvements in the largest possible overload window. Instead, if the last recursive step resolved in a decrease, it is better to take into account not the largest possible interval, but the one in which the best estimation

---

CoreED $(\mathcal{T}, M, \tau_k, OW)$

1 **for** each $\tau_i \neq \tau_k \in \mathcal{T}$, **do**

2 $\quad N_i \leftarrow \max\left(0, \left\lfloor \frac{OW - \overline{\Lambda}_i}{T_i} \right\rfloor + 1\right)$

3 $\quad \beta_k^i \leftarrow N_i C_i + \min\left(C_i, \max\left(0, OW - N_i T_i - S_i\right)\right)$

4 $bound \leftarrow \left\lceil (OW - C_k) - \frac{\sum_{i=1}^{k} \min\left(\beta_k^i, OW - C_k + 1\right)}{M} \right\rceil$

5 **return** *bound*;

CoreFP $(\mathcal{T}, M, \tau_k, OW)$

1 **for** each $\tau_i \neq \tau_k \in \mathcal{T}$, **do**

2 $\quad N_i \leftarrow \left\lfloor \frac{(OW + D_i - C_i) - \overline{\Lambda}_i}{T_i} \right\rfloor + 1$

3 $\quad \beta_k^i \leftarrow N_i C_i + \min\left(C_i, \max\left(0, (OW + D_i - C_i) - N_i T_i - S_i\right)\right)$

4 $bound \leftarrow \left\lceil (OW - C_k) - \frac{\sum_{i=1}^{k} \min\left(\beta_k^i, OW - C_k + 1\right)}{M} \right\rceil$

5 **return** *bound*;

Figure 2.8. Core of the schedulability test for EDF and EDZL, above, and FP, below.

of S_i was computed, trying to improve the analysis only exploiting the improvements on the slack computed for the other tasks. See Section 7.1 for further explanations on this topic.

Note also that the maximum taken at line 2 is necessary to cope with the problem described in Section 7.2, of N_i possibly negative if the slack S_i is greater than the length of the overload window. Forcing N_i to 0 guarantees also that the total contribution of τ_i is 0 (as we expect), since it forces to 0 also the carry-in.

8.1. Recursive Test. We propose Algorithm *Recursive Test* in Figure 2.9 as a generic schedulability test, that can be customized for the three algorithms EDF, FP and EDZL. The test will be called respectively REDF, RFP, or REDZL. The specialization is necessary in the following points:

- at lines 9 and 11 the *CoreX* procedure is CoreED for REDF and REDZL, or CoreFP for RFP(see Figure 2.8);
- the termination test at line 19 depends on the algorithm: for REDF and RFP the test is passed if all the tasks have strictly non negative lower bound on the slack S_k; instead, for REDZL, the test is passed if no more than M tasks have negative S_k;
- under FP, the M highest priority tasks can always find a processor to execute, preempting some lower priority task if no idle processor exists, and so they will always complete in time; as a consequence, the tasks to be verified (line 7) are all the tasks for REDF and REDZL, but only the tasks from $M+1$ to N, for RFP;
- under FP the M highest priority tasks always finish C_i time instants after the release, so the bound to their slack is always maximized to $D_i - C_i$. As a consequence, the initialization step at line 2 requires to fix all the slacks S_i to -1 (assume non schedulability, and try to

---

Recursive Test $(\mathcal{T}, M)$

```
 1 for each i do
 2    initialize S_i
 3    OW_i^New ← min(D_k, T_k)
 4    OW_i^Old ← min(D_k, T_k)
 5    flag_i ← TRUE
 6 while some improvement is possible do
 7    for each τ_k in T that requires verification do
 8        if (flag_k = TRUE) then
 9            s_new = CoreX(T, M, τ_k, OW_k^New)
10        else
11            s_new = CoreX(T, M, τ_k, OW_k^Old)
12        if (s_new ≥ S_k) then
13            S_k ← s_New
14            flag_k ← TRUE
15            OW_k^Old = OW_k^New
16            OW_k^New = min(D_k, T_k + S_k)
17        else
18            flag_k = FALSE
19    if (test is passed) do
20        return YES
21 return NO
```

Figure 2.9. Pseudo-code for the Recursive Test.

improve), for REDF and REDZL. For RFP the first M slacks are maximized, while the others are fixed to -1.

In line 6, *some improvement is possible* if at the previous step at least one of the lower bounds on the slack has been improved. In such a situation, in fact, the improved value potentially allows for further improvements.

In the Recursive Test, at lines 9 and 11 we use alternatively two different values for the overload window, depending on the previous result (parameter *flag*, set at line 14 or at line 18). As explained above, this helps in considering always the length of the overload window which, most probably, can give the better results. Whenever we improve the estimation of the slack, we use the largest possible overload window, while if we find an estimation which is worse than the previous one, it is convenient, at the next step, to start again with the overload window which provided the highest value for S_k.

Comparing Theorems 6.1, 6.2 and 6.3 with the Recursive Test in Figure 2.9, we note that the core part of the test is essentially the same, although the formulae are mainly based on $\overline{\Lambda}_k$ and $\overline{\Lambda}_i$, instead of Λ_k and Λ_i. The main difference is in line 6 of the algorithm, which allows to use the acquired knowledge about the lower bound on the slack of the tasks to improve the analysis.

Another difference between the Recursive Test and the tests in Section 6 is the following. In the Recursive Test, at each step, we compute s_1, then we compute s_2 using the improved estimation of s_1 and the latest estimation of

all the other S_i, for s_3 we use the new knowledge of s_1 and s_2, together with the old values for the rest of the S_i, and so on. As a consequence, even after one single step, the Recursive Test improves with respect to the previous tests. It should be clear, then, that the Recursive Test, in its three versions, strictly dominates respectively BCL-EDF, BCL-FP, and EDZL, in the sense that if a task set is declared schedulable by one of the tests in Section 6, then it is declared schedulable by the Recursive Test, and there are task set declared schedulable only by the Recursive Test.

Clearly we pay this increase in performance of the test with an increased complexity, due to the recursive step. However, these tests are usually implemented off-line, i.e. before the start time of the system. In such a case, we can accept to spend some time in a more complex test, to gain in precision. We will discuss this issue more in depth in Section 9.3, basing on the results of the experiments.

8.2. The Recursive Test and EDF-DM. The Recursive Test, specialized for EDF, has another interesting use in the schedulability analysis of the hybrid algorithms described in Chapter 1, Section 2.2. As we said, the idea of these hybrid algorithms is to mix EDF and FP in a single algorithm, where a subset of tasks is given maximum priority, while the others are scheduled by EDF. Algorithms of this class have their distinction in the number and policy in which the high priority tasks are chosen.

We consider EDF-DM, one of the most known variants, but the same reasoning is valid for several other possible algorithms based on the same concept. The idea is to assign maximum priority to the k highest density tasks, where k is chosen so that the rest of the tasks can be scheduled by EDF. The test for this algorithm is based on a pessimistic assumption: the k maximum priority tasks cannot occupy more than one processor each, so in the worst-case the remaining $N-k$ tasks are assigned the remaining $M-k$ processors. The theorem below reports the best schedulability test proposed so far.

THEOREM 8.1 (EDF-DM schedulability test). *A task set $\mathcal{T}$ is schedulable on M processors by EDF-DM if there exists a value $k \leq M-1$ such that the $N-k$ lower density tasks are schedulable by EDF on $M-k$ processors.*

PROOF. Follows from the discussion above, by assigning maximum priority to the k highest density tasks. □

The best solution, so far, was to use DB-EDF to test the schedulability under EDF of the lower priority tasks. Using our REDF, we are able to prove schedulable with EDF-DM a larger amount of task sets.

9. Experimental results

In order to validate the proposed tests and compare their behavior with the best existing tests, cited in Section 3, we ran a long series of simulations, using different combinations of task parameters. We analyzed as well the behavior of the tests varying the number of processors, the number of tasks and the total system utilization. We report here only some of the experiments, representative of the general behavior.

The experiments reported in the figures were generated based on the following characteristics of the tasks:

- 2, 4 and 8 processors;
- task utilization U_i extracted according to exponential distribution with mean 0.25, 0.50 and 0.75, re-extracting tasks with $U_i > 1$;
- period T_i (and, implicitly, execution time C_i) extracted uniformly in $[1/U_i, 10000]$;
- deadline D_i uniformly extracted between C_i and T_i, whenever we focus on constrained deadlines;
- deadline D_i uniformly extracted between C_i and $2T_i$ or between C_i and $4T_i$, whenever we accept arbitrary deadlines.

According to the time division explained in Section 2.3, we forced C_i, D_i and T_i of each extracted task to the nearest integer. Considering that the largest period is $10,000$, we claim this does not really influences the results.

Initially, we extracted $M + 1$ tasks, verifying that the total utilization was lower than M, and we applied all the tests to the task set. Then we added a new task, and we applied again the tests to the new obtained task set. This process was repeated as long as the total utilization was lower than M. Then, a new initial set of $M + 1$ tasks was extracted. This procedure was repeated until $1,000,000$ task sets were extracted and tested.

Each graph represents the results of simulations on $1,000,000$ task sets: the X axis corresponds to the total utilization U_{tot} of the task set, and the Y axis corresponds to the number of task sets with U_{tot} in the range $[X - 0.01, X + 0.01)$ that satisfy the tests. Each line represents the number of task sets proved schedulable by one specific test. In all the graphs, the thick line marked with "Total" represents the number of task sets extracted with a given total utilization. Note that due to the procedure of extraction of the tasks, it more probable to have high utilization task sets.

We compared our tests of Section 6 and the Recursive Test with previous tests explained in Section 3, and in particular

- for EDF, DB-EDF and BAK-EDF;
- for FP, DB-FP and BC;
- for EDZL, none.

For EDF, we did not consider the BC test, since its results are quite poor. For FP, in [**BC06c**] BAK-FP was shown to be outperformed by BC, and so it was not considered. Moreover, since DB-FP is valid only for constrained deadlines systems, it was included in the simulation only in such cases.

In all the simulations, we verified that the mean value of the exponential distribution had a minor effect on the results. For this reason, the graphs below report only the results of experiments where the mean value is 0.25.

9.1. Experiments. Through the experiments, we verified that while BCL-EDF and BCL-FP do not provide interesting improvements, their recursive equivalents offer an interesting enhancement in testing the schedulability of task sets under EDF and FP. As expected, the three versions of the Recursive Test performed always strictly better then BCL-EDF, BCL-FP and EDZL, recognizing every task set recognized by the non recursive tests.

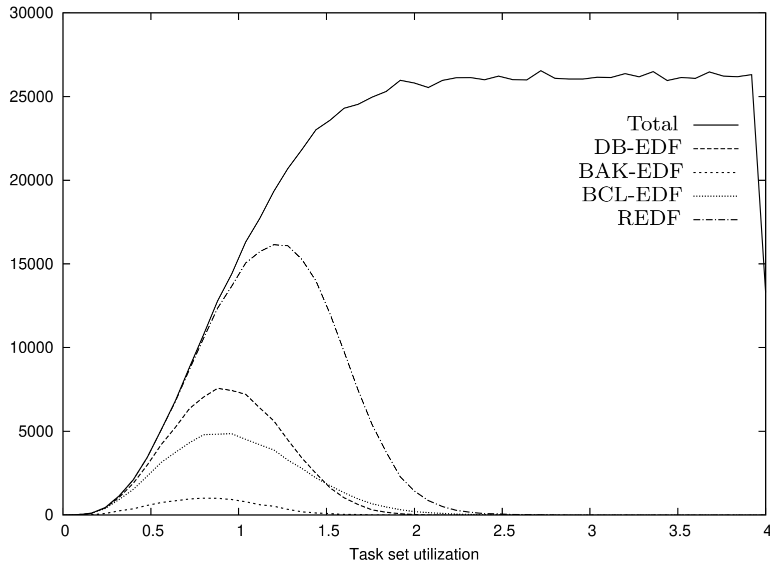

FIGURE 2.10. EDF: 4 processors, $D_i \leq T_i$.

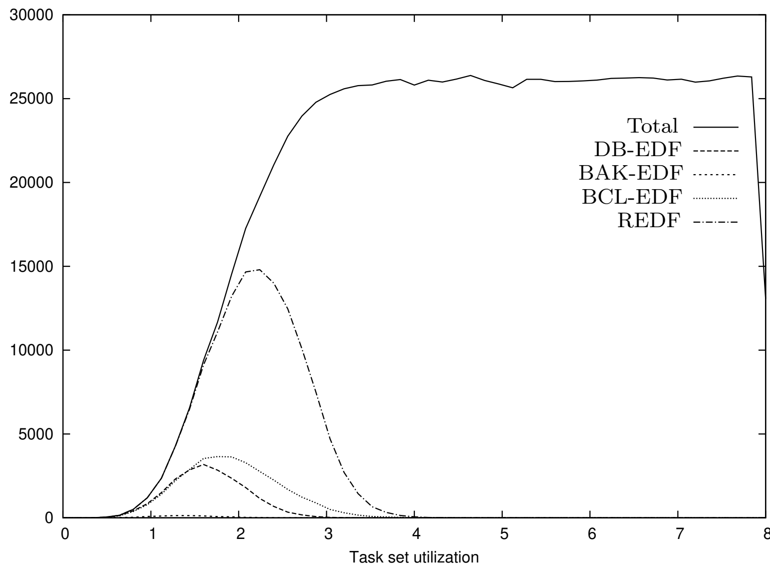

FIGURE 2.11. EDF: 8 processors, $D_i \leq T_i$.

The good behavior of REDF and RFP is particularly evident when used for constrained deadlines systems, where they are always the best tests for both EDF and FP.

The advantages in using the Recursive Test are maximized for EDF. Moreover, the gain of REDF with respect to other tests for EDF is emphasized as the number of processors increases. As an example, in Figures 2.10 and 2.11 we show the results for EDF, obtained for 4 and 8 processors.

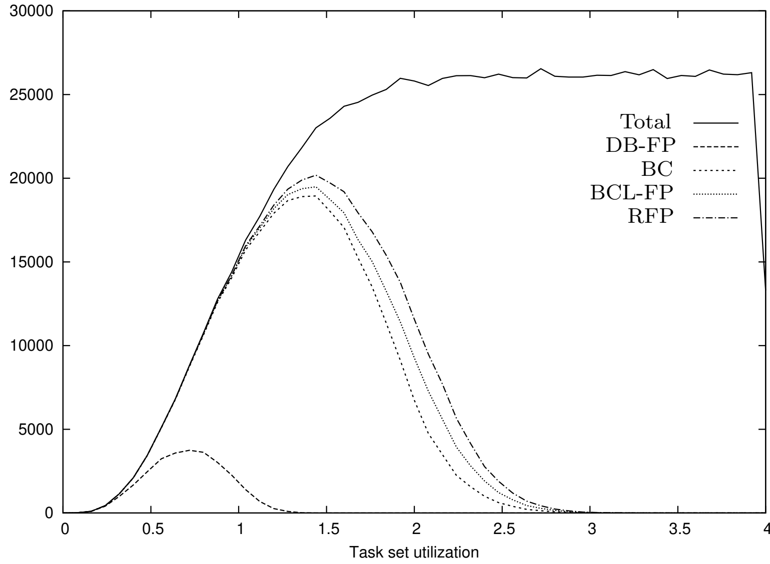

FIGURE 2.12. FP: 4 processors, $D_i \leq T_i$.

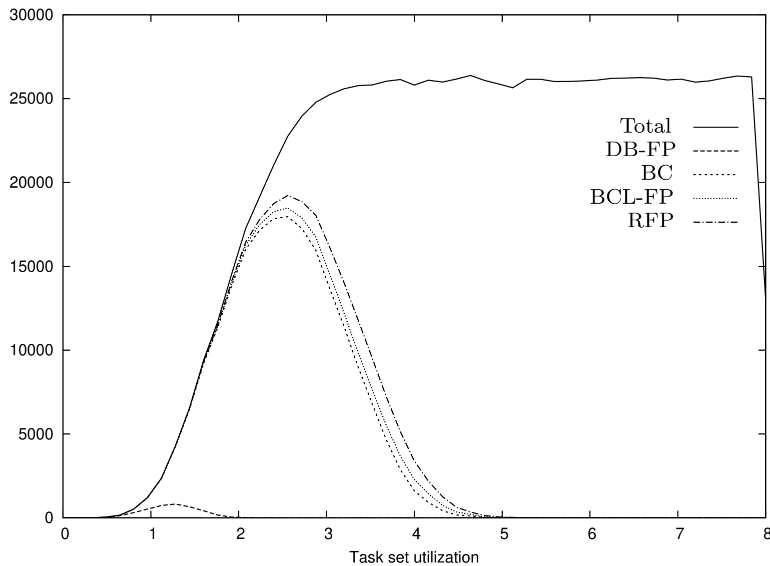

FIGURE 2.13. FP: 8 processors, $D_i \leq T_i$.

The same situation for FP is reported in Figures 2.12 and 2.13. From these experiments, it is clear that DB-FP, even if interesting from a theoretical point of view, is not a good test to recognize task sets schedulable under FP. Considering the difference between DB-FP and all the other tests, it is confirmed the fact that while utilization and density bounds can be useful for a fast comparison among the capabilities of different algorithms, they can also lead to wrong conclusions: an high bound is a guarantee that the

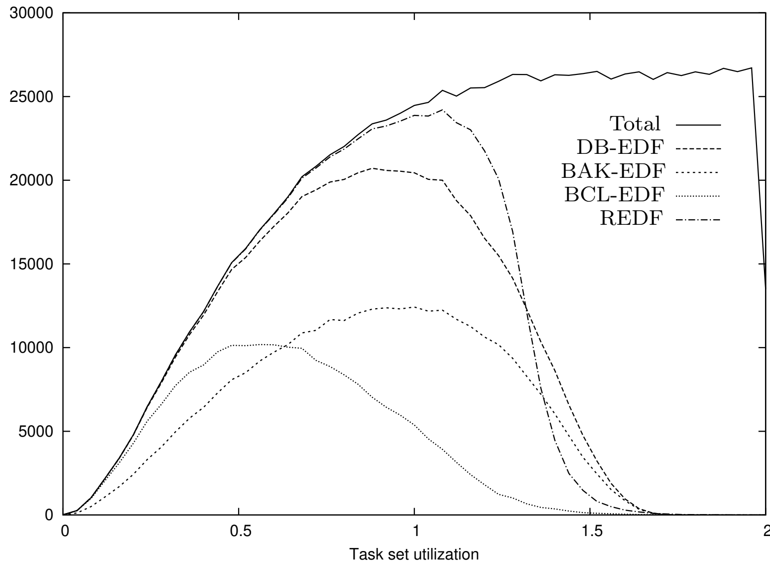

FIGURE 2.14. EDF: 2 processors, $D_i \leq 4T_i$.

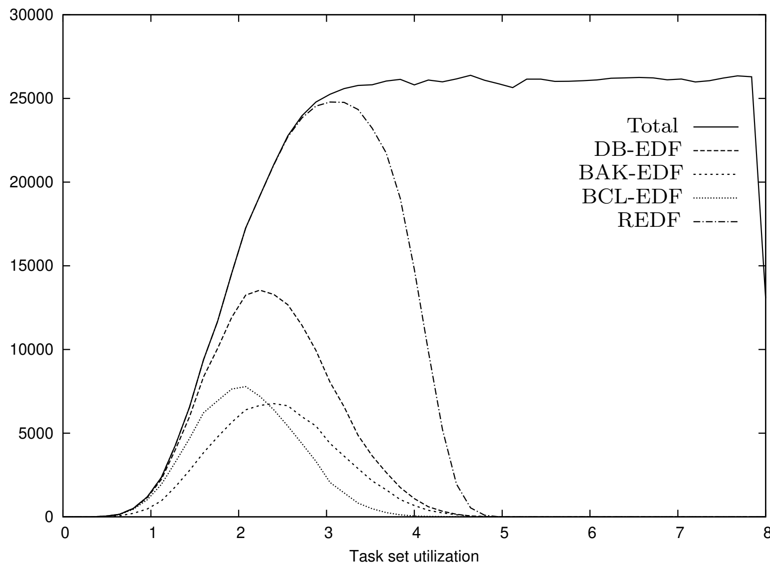

FIGURE 2.15. EDF: 8 processors, $D_i \leq 4T_i$.

algorithm performs generally well, while a low bound only guarantees that the algorithm performs bad in some cases.

Moving to unconstrained deadlines systems, the good behavior of REDF and RFP is maintained, especially for low utilizations, but the situation is in general more complex.

Consider first of all EDF. Together with cases in which REDF can be surpassed by DB-EDF and, in a few cases, BAK-EDF, we find also cases in which the benefit from the use of REDF seems to be maximized. In

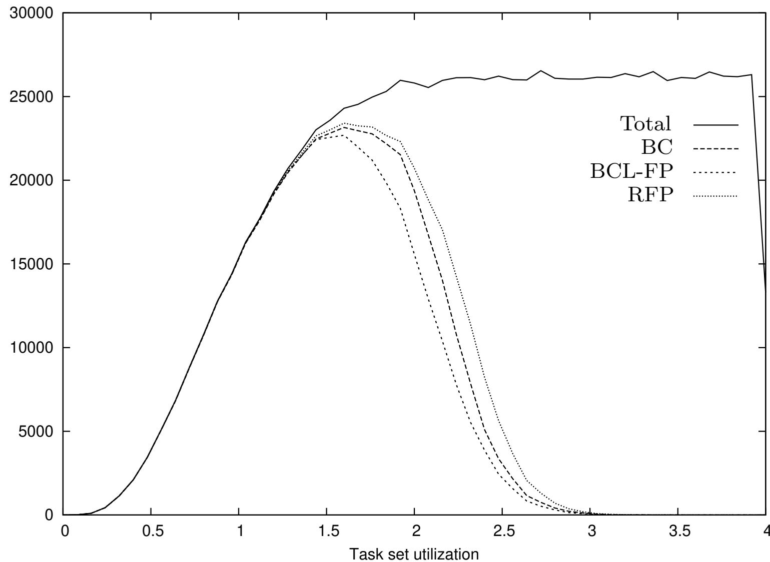

FIGURE 2.16. FP: 4 processors, $D_i \leq 2T_i$.

Figure 2.14 we see that for 2 processors and $D_i \leq 4T_i$, DB-EDF and BAK-EDF can identify more schedulable high utilization task sets than REDF. At the same time, if we move to 8 processors (as shown in Figure 2.15), the gain due to REDF is impressive (although due not only to its merit, but also to the bad behavior of previous test).

As said above, we verified that REDF does not strictly dominate DB-EDF in the sense that we found some schedulable task sets that DB-EDF reveals while REDF does not. This is expected, considering the completely different approach which lead to the proposition of the two tests. Moreover, it was already shown in [**BCL05a**] that the BCL-like approach performs very well in the presence of heavy tasks, while DB-EDF suffers from this, and vice-versa. This trend is maintained in moving to REDF, but the extreme improvement given by the recursive step allows to almost overrule the advantages of DB-EDF. As a consequence, even if there are task sets for which DB-EDF performs better than REDF, in our tests they were limited to few cases in few experiments (less than an overall 4% of the number of extracted task sets in the worst-case, shown in Figure 2.14).

For the case of FP, we verified that with unconstrained deadlines the BC test performed better than our BCL-FP and RFP, and this effect was more evident for $D_i \leq 4T_i$ than for $D_i \leq 2T_i$. The reason of this seems to be the fact that both BCL-FP and RFP must assume a very short overload window (based on T_k), in which it is easier to find a job that can miss a deadline. The length of the overload window was then increased in RFP but apparently not enough. Instead, formulae for BC can use the deadline D_k. Increasing the difference between deadline and period can increase the performances of BC with respect to those of RFP. Related experiments are shown in Figures 2.16 and Figures 2.17 (compare them with Figure 2.12 above, where constrained deadlines were assumed).

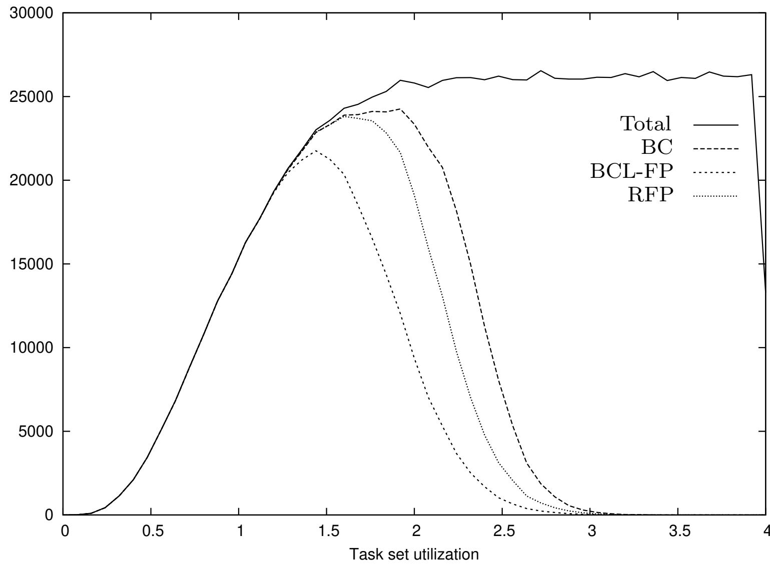

FIGURE 2.17. FP: 4 processors, $D_i \leq 4T_i$.

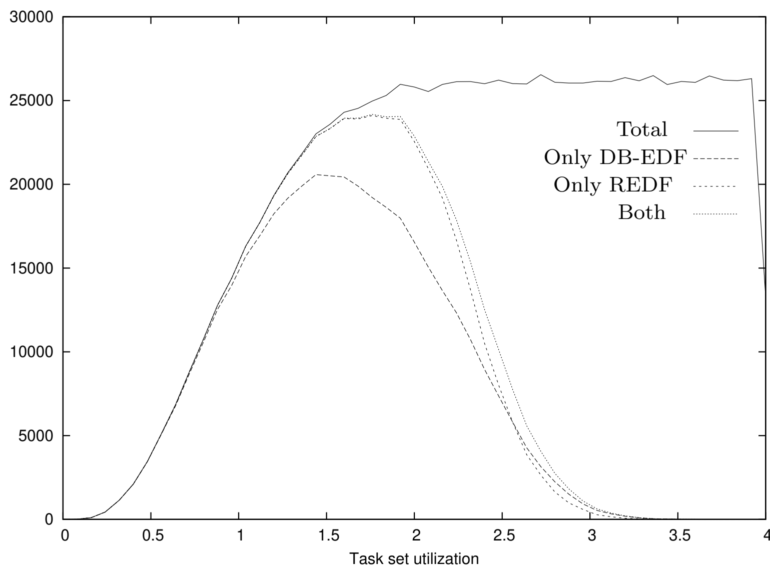

FIGURE 2.18. Improvements for EDF-DM: 4 processors, $D_i \leq 2T_i$.

As said in Section 8.2, REDF can be used to improve the schedulability test for EDF-DM and similar hybrid algorithms (see Chapter 1). We verified such improvement through simulations. In Figure 2.18, we consider the case of 4 processors and unconstrained deadlines ($D_i \leq 2T_i$). The figure reports task sets schedulable by EDF-DM, using, as EDF test, DB-EDF, REDF, and their union (EDF-DM in the figure).

In some cases with high utilization, the use of DB-EDF allows to recognize more task sets than using REDF. This behavior shows up only with

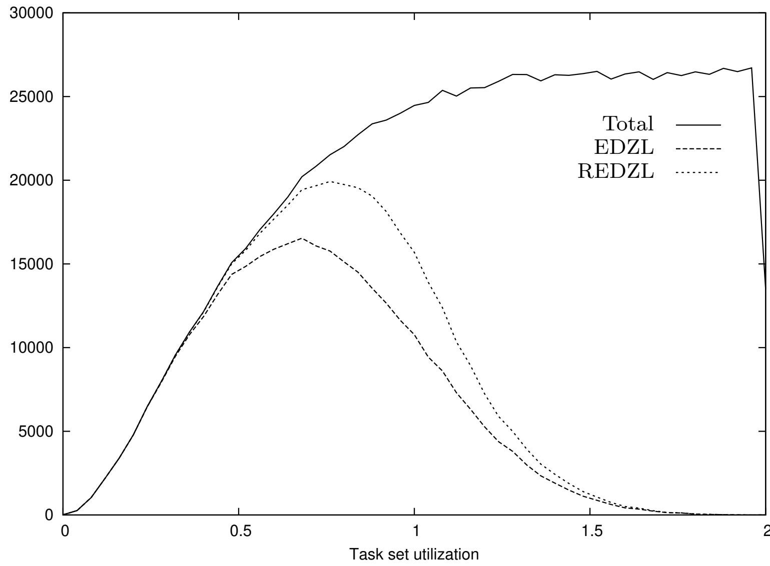

FIGURE 2.19. EDZL: 2 processors, $D_i \leq T_i$.

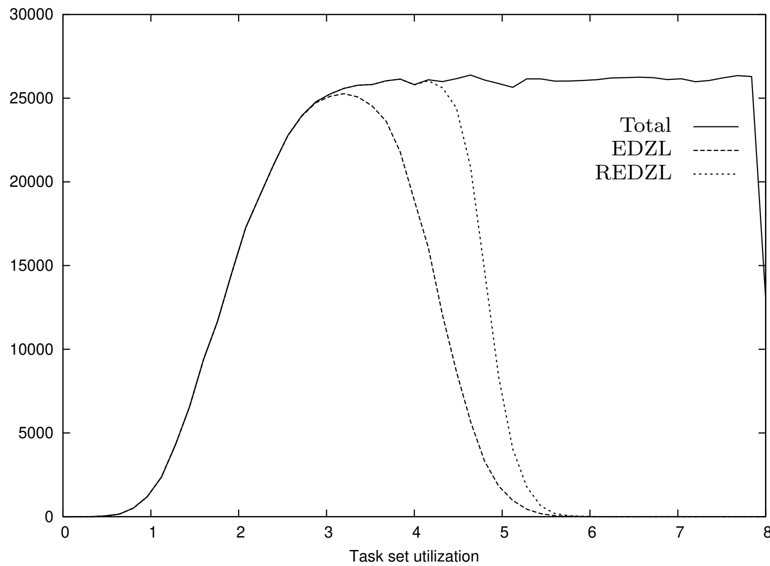

FIGURE 2.20. EDZL: 8 processors, $D_i \leq 4T_i$.

unconstrained deadlines, and increases as the difference between deadlines and periods increases. However, the best solution is clearly to use the EDF tests together, most of all considering the very low complexity of DB-EDF.

For EDZL, we cannot compare the tests with any previous test. We only studied the overall behavior of EDZL and REDZL in different situations. The generally good behavior was clear in every single graph. Figures 2.19 and 2.20 are only two examples.

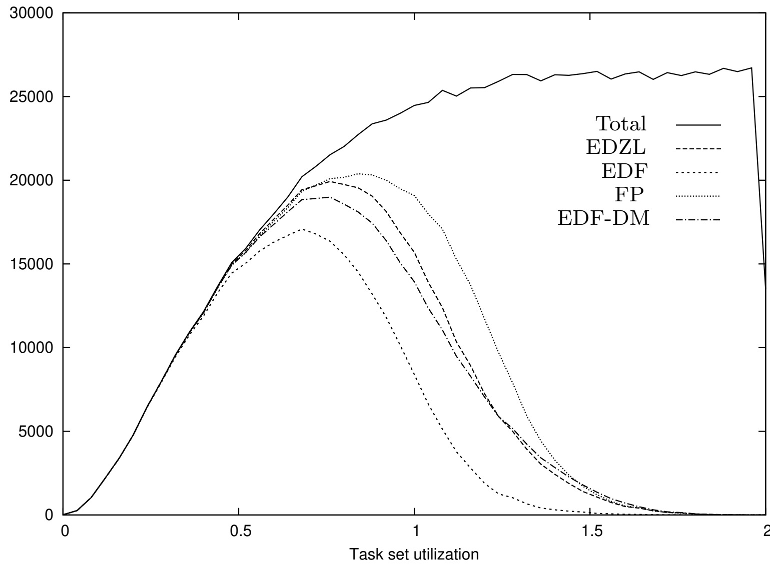

FIGURE 2.21. Global comparison: 2 processors, $D_i \leq T_i$.

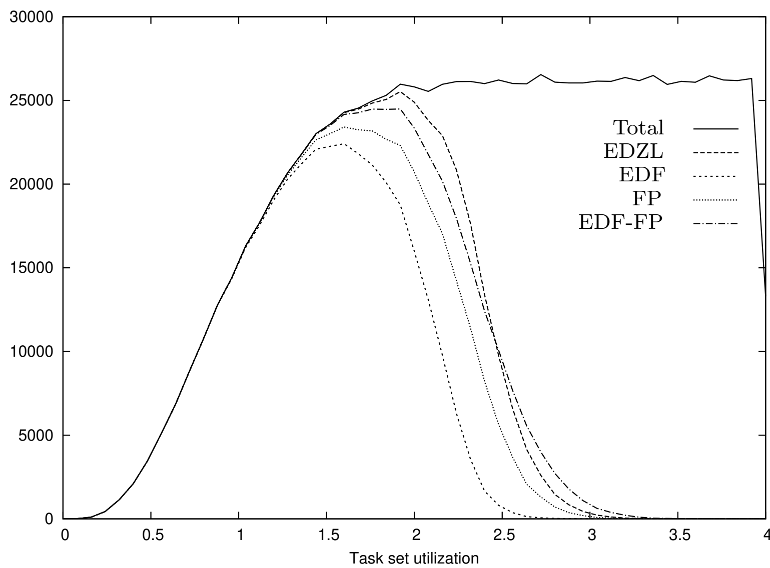

FIGURE 2.22. Global comparison: 4 processors, $D_i \leq 2T_i$.

9.2. Global comparison. It is worth to make a comparison among different pairs algorithm-test, in order to give an idea of the overall situation. We considered EDF with REDF, FP with RFP, EDF-DM with the union of DB-EDF and REDF, and EDZL with REDZL.

In Figures 2.21, 2.22 and 2.23 we report three configurations, which differ in the number of processors and the relation between deadlines and periods.

From the overall analysis (over $27,000,000$ task sets divided in 27 different configurations) we verified that

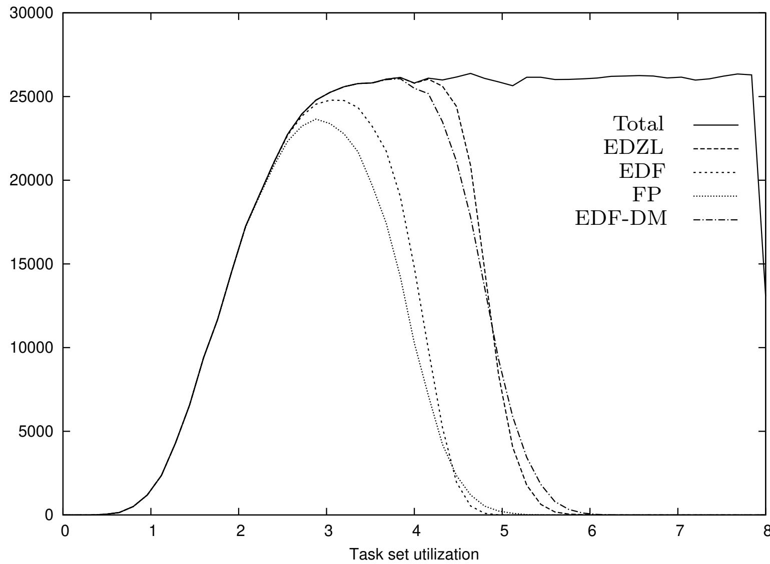

FIGURE 2.23. Global comparison: 8 processors, $D_i \leq 4T_i$.

- EDF can compete with FP only when $D_i \leq 4T_i$; since it is believed that EDF performs better than FP (it has been shown in single processors, but to the best of our knowledge no one proved it for multiprocessors), the reason of the problem seems to lie in the tests; the reason of this is probably related to the fact that for FP the behavior of the M higher priority tasks is known, and the tests for FP can take advantage of this knowledge.
- EDF and FP can compete with EDZL and EDF-DM only when considering small number of processors; this is a clear advantage of the two algorithms, more than the tests, which derives from the fact that EDZL and EDF-DM are specifically proposed for multiprocessors;
- it must be underlined the impressive result of FP in Figure 2.21; in such a case (low number of processors, and constrained deadlines), FP behaves better than any other test; this is a problem of the tests, which for EDZL and EDF-DM cannot gain much from the multiprocessor (only two processors);
- for unconstrained deadlines, and high U_{tot}, EDF-DM seems to be, the best solution, although EDZL behaves very well;
- both EDF and FP have performances far beyond their utilization bounds of $\frac{1}{M}$;
- both EDZL and EDF-DM have high performance despite the fact that both their schedulability tests are based on extremely pessimistic assumptions; relaxing these assumptions could lead to impressive results.

We want to underline that, in the general case, all proposed tests provide results that are far from the necessary and sufficient test (i.e. the simulation). To see this, we refer to [**CB07**], where a comparison is done between

the results of the test and of the brute-force simulation, for both EDF and EDZL. It is evident that a great distance remains, even after our improvement proposed in this thesis, and we believe that there is still lot of space for improvement.

9.3. Complexity. As we said in Section 8.1, the complexity of the tests is not a real problem when the tests are performed off-line. Consider that even in the worst case (8 processors and $D_i \leq 4T_i$), each test took less than 4ms to execute, and a mean of $500\,\mu$s was achieved, which means that the complexity is a problem only in simulations, where each test is executed to test some millions of task sets. However, it is important to consider how long does each test takes, particularly in the case of the recursive tests, in order to verify if the increase in complexity is counterbalanced by a sufficient increase in performance.

Consider for the moment tests other than the recursive tests. It is easy to see that they all have a very low computational complexity: DB-EDF and DB-FP have complexity equal to $\mathcal{O}(N)$ (where N is the number of tasks), BCL-EDF, BCL-FP, and EDZL are $\mathcal{O}(N^2)$, while BAK-EDF and BC are $\mathcal{O}(N^3)$.

The complexity of the recursive test, by its nature, is not easily computable. While the core of the test is $\mathcal{O}(N^2)$, the problem is represented by the recursive step. For this reason we chosed to evaluate the average and maximum number of steps through experiments. We found a great difference in complexity among the three tests, and between constrained and unconstrained deadline systems.

For constrained deadlines systems, during our experiments, over several millions of task sets (each single experiment analyzed $1,000,000$ task sets), we verified that the maximum number of steps necessary to find an answer (either a positive answer or a not improvable solution) was high only for REDF, while for RFP and REDZL it was never more than $2N$, and actually near $1N$. This offers some ideas of the actual average complexity of the test when applied to constrained deadlines systems, which seems to be close to $\mathcal{O}(N^3)$ for RFP and REDZL. The reason of the difference between REDF and the others is probably due to the following facts:

- RFP has a perfect estimation of the slack of the first M tasks even before the first step; this gives an extra boost to the test, which is able to find very good estimations of the slacks in only a few steps;
- REDZL can accept up to M tasks with non-positive slack, which means that the test can rapidly converge to a positive answer;
- REDF has none of the previous mechanisms, and as a consequence it can happen that the test suffers for a "slow start" (due to the worst possible estimation of the slacks) and a "slow end", continuing to search for a positive solution, which in some case does not even exist.

Further analyzing the case of FP, we noted the following behavior. Before the first step M tasks have perfect estimation of the slack. In the first step, the $(M+1)^{\text{th}}$ task has the best possible estimation for its slack. This is due to the fact that the slack of the $(M+1)^{\text{th}}$ task is influenced only by

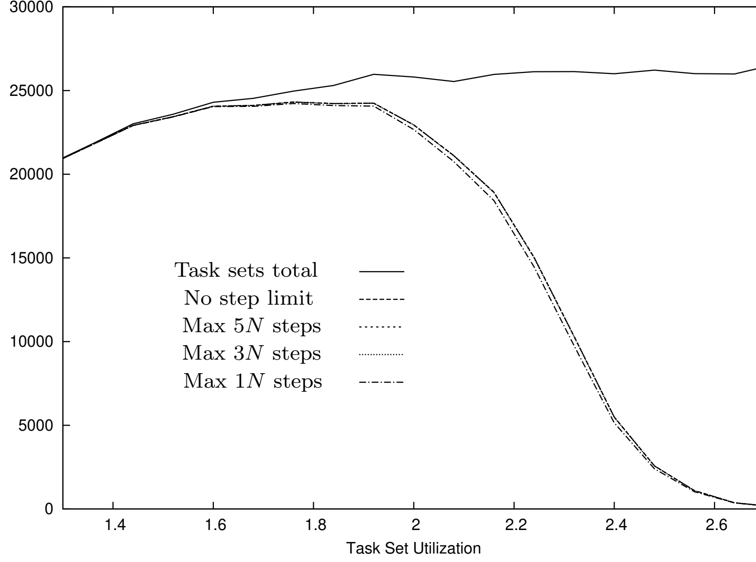

FIGURE 2.24. Comparison of results for REDF when the number of step is limited.

previous tasks, for which the slack cannot be improved. So, in any case, the estimation of S_{M+1} will never change. This line of reasoning can be repeat for every task, which already in the first step is analyzed under the best possible conditions. As a consequence, at the end of the first step, we already have the best possible estimation for all the slack. That is, for constrained deadlines systems under FP the answer at the end of the first step is definitive. The test is so only $\mathcal{O}(N^2)$. Unfortunately, this good behavior is not maintained for unconstrained deadlines systems, due to the fact that in such a case the estimation of the slack for tasks other than the first M can change, mainly due to the change in the length of the overload window.

The pessimistic estimation of the overload window for unconstrained deadlines systems affect the Recursive Test for all the three algorithyms. The maximum number of steps required to reach an answer explodes for all the three cases, reaching values over $1000N$. It must be said that REDF continues to be the worst, while REDZL is the best of the three, proving that the early termination condition explained above is extremely useful also from this point of view.

However, it must be said that while the maximum number of steps was excessively high, this happened in very few cases, and the average was always between $0.8N$ and $2.3N$. As a consequence, we believe that we could solve the complexity problem by limiting the number of steps, with a loss in the number of verified task sets of only some decimal. In order to verify this hypothesis, we repeated some of the experiments, limiting the number of steps to only $5N$ (that is, after $5N$ steps, we considered not schedulable the task set), $3N$ and $1N$, and compared the new and old results.

In Figure 2.24 we report the result of one comparison for REDF in the case of 4 processors, $D_i \leq 4T_i$, and $1,000,000$ task sets. The figure is magnified for task set utilizations between 1.3 and 2.7, the only region where

a difference was perceivable. Only a small difference can be seen, and only when the step number is limited to N. We conducted the same comparison for RFP and REDZL and, as expected, the difference was even smaller.

10. Conclusions

In this chapter we tackled the problem of schedulability analysis of global scheduling systems, where the scheduling algorithm is chosen among EDF, FP and EDZL.

We proposed two different sets of schedulability tests for all the three scheduling algorithms. For EDF and FP, the first tests (BCL-EDF and BCL-FP), at the price of the same computational complexity, do not really improve the results of previously known tests proposed by Goossens, Funk and Baruah [**GFB03**] and Baker and Cirinei [**BC06c**]. However, they are the base for the proposition of the second test, the Recursive Test of Section 8. This new test has an higher complexity, but has a good improvement with respect to previous tests.

For the particular case of REDF, its good performance helps in improving not only for EDF, but also for EDF-DM, an hybrid between EDF and FP proven to perform very well in global scheduling of real-time tasks on multiprocessors.

The case of EDZL is more interesting, because, while it is known that the scheduling algorithm performs very well (and in particular it strictly dominates EDF), we are not aware of any other schedulability test presented so far for such an algorithm. Also for EDZL, we presented two tests, one $\mathcal{O}(N^2)$ test equivalent to BCL-EDF and BCL-FP and one based on the same Recursive Test as above. Even with the first test, we are able to recognize more task sets schedulable with EDZL then task set schedulable with EDF or FP verified with the Recursive Test. When using the Recursive Test, EDZL overrules EDF and FP, and compares very well also with EDF-DM.

Future improvements for these two algorithms should relate to the verification and correction of some gross overestimations in the analysis. For EDZL, an important goal would be to verify not only that $M+1$ tasks can reach zero laxity, but also that they can have zero laxity "at the same time". For EDF-DM, we could improve the analysis in two directions. First of all by considering that in general the k high priority tasks don't require k full processors, but only fractions of them, which leaves more space for the rest of the tasks. Second improvement could be the research for a better choice of the k tasks: instead of only considering the k tasks with highest density, searching for the tasks that actually suffer more from the parallel execution (by fact, considering different hybrid algorithms).

An important improvement, which could allow to improve at the same time the tests for all the algorithms, should be done in the estimation of the overload window. In fact, it can be noted that in some sense assuming each task to suffer the worst-case precedence-blocking is equivalent to consider each task to have deadline D_i shortened to the period T_i, and to be released $D_i - T_i$ time instants later, therefore losing the benefits that can derive from the enlargement of the deadline. The recursive computation introduced in the Recursive Test of Section 8 is a first step in the direction of improving

such estimation. Another possibility could be, for example, to enlarge the research for the lower bound on the slack not only on the overload window, but in the whole interval between release and deadline of a task.

Considering the limits of the tests proposed, due to the pessimistic hypothesis we were forced to do, it is clear that EDZL and EDF-DM are two of the best known algorithms for global scheduling of real-time tasks upon multiprocessor platforms, and our tests are an important step forward in their analysis.

CHAPTER 3

Performance Problem: feasibility analysis

1. Overview

In the previous chapter we considered the problem of how to guarantee the schedulability of sets of sporadic real-time tasks under three different scheduling algorithms: EDF, FP and EDZL. In this chapter we consider the opposite problem: what is the necessary condition for the feasibility of scheduling a set of independent sporadic hard-deadline tasks?

Recall that *schedulability analysis* considers if a specific scheduling algorithm $\mathcal{A}$ can correctly schedule a task set on a given platform, while *feasibility analysis* verifies if such a scheduling algorithm $\mathcal{A}$ can exist.

Several sufficient tests have been derived for the schedulability of a sporadic task set on a multiprocessor using a given scheduling policy, such as FP-global or EDF-global [**ABJ01, Bak05, Bak06a, BFB05, BCL05a, BCL05b, GFB03, SB02**]. For example, it can be shown that a set of independent periodic tasks with implicit deadlines (that is, $\forall i \; D_i = T_i$) will not miss any deadlines if it is scheduled by EDF-global policy on M processors, provided that the total utilization U_{tot} not exceed $M\,(1 - U_{max}) + U_{max}$, where U_{max} is the maximum single-task processor utilization. In Chapter 2 we provide some new tests for EDF-global, FP-global and EDZL that help in improving the number of task sets recognized as schedulable.

One difficulty in evaluating and comparing the efficacy of such schedulability tests has been distinguishing the causes of failure. That is, when one of these schedulability tests is unable to verify that a particular task set is schedulable there are three possible explanations:

(1) the problem is with the task set, which is *not feasible*, (i.e., not able to be scheduled by any policy);
(2) the problem is with the scheduling policy, in the sense that the task set is *not schedulable* by the given policy, even though the task set is feasible;
(3) the problem is with the test, which is *not able to verify* the fact that the task set is schedulable by the given policy.

To the best of our knowledge, there are no known algorithms other than "brute force" enumeration that can distinguish the above three cases.

The following facts are fairly well known:

- A sporadic task set $\mathcal{T}$ is feasible on M processors (with the GPS algorithm described in Chapter 1) if $\lambda_{tot} \leq M$;
- A sporadic task set $\mathcal{T}$ is not feasible on M processors if $U_{tot} > M$;

Baruah and Fisher [**BF05**] showed that $\delta_{sum} \leq M$ is a necessary condition for the feasibility of the task set $\mathcal{T}$ composed of N tasks on a platform

with M unit-capacity processors, where

$$\delta_{sum} \stackrel{\text{def}}{=} \sup_{t>0} \frac{\sum_{i=1}^{N} \text{DBF}(\tau_i, t)}{t}$$

and

$$\text{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right)$$

Fisher, Baker, and Baruah [**BFB05, FBB06**] showed how to approximate the load bound function δ_{sum} efficiently, in polynomial time, and that the criterion $\delta_{sum} > M$ was significantly more effective than the criterion $U_{tot} > M$ as a test of infeasibility when tested on several large collections of pseudo-randomly generated task sets.

In this chapter we derive an improvement on the above load bound function, which allows the detection of a strictly larger range of infeasible task sets, including the example below.

EXAMPLE 4. Consider the task set below.

i	C_i	D_i	T_i	$\text{DBF}(\tau_i, 1)$
1	2	2	4	0
2	1	1	2	1
3	1	1	2	1

Since $\delta_{sum} = 2 \leq M = 2$, the task set cannot be declared infeasible on 2 processors. □

The task set in the example above is clearly infeasible on $M = 2$ processors (since all three tasks should execute in $[0, 1)$), but $\delta_{sum} = 2$. The problem is that $\text{DBF}(\tau_1, 1) = 0$ under-estimates the real demand of task τ_1 in the interval $[0, 1)$. Task τ_1 must execute for one unit of time in the interval $[0, 1)$ in order to meet its deadline at 2. The effective combined demand of the three tasks over this interval should be 3, not 2. The phenomenon observed above was recognized previously by Johnson and Maddison [**JM74**]), who used the term "throwforward" to describe the amount of execution time that a task with later deadline, like τ_1, must complete before an earlier deadline of another task, like τ_2 and τ_3.

The new load bound function $m\ell(\mathcal{T})$ defined here is similar to δ_{sum}, but differs by using $\text{DBF}(\tau_i, t) + \max(0, t - (jp_i + d_i - e_i))$ instead of $\text{DBF}(\tau_i, t)$. The additional term corrects for cases of under-estimation of the actual worst-case computational load of an interval like the example above, by taking into account the throwforward of jobs whose deadlines may occur past the end of the interval. That way, in the Example 4 above, $m\ell = 3 > M = 2$ and so the task set is correctly declared infeasible.

Contributions. The work described in this chapter has been previously published in [**BC06a**], while an extended version can be found as a Technical Report at [**BC06b**]. Here we report the following contributions:

(1) we show how to recognize a significant number of infeasible task sets, by computing a new load-bound function and determining whether the load bound exceeds the available number of processors;
(2) we show that the new load bound retains the property of the δ_{sum} load bound of Baruah, Mok, and Rosier [**BMR90**] that $m\ell \leq 1$ is

a necessary and sufficient condition for single-processor feasibility, and a necessary and sufficient test of single-processor EDF schedulability;

(3) we provide empirical evidence of the degree of improvement in ability to detect infeasible task sets using the new load-bound function, as compared to the previously-defined load bound function δ_{sum};
(4) we provide an algorithm for computing the new load-bound function to any specified degree of accuracy within polynomial time;
(5) we provide empirical evidence that the new algorithm can be computed at least as efficiently as the best previously known algorithm for computing δ_{sum};
(6) we verify on some examples taken from the previous chapter the benefits obtained in using the load-bound function during the estimation of the behavior of scheduling algorithms on multiprocessor platforms.

2. System model

In order to make this chapter self-contained, we report here briefly the system model under consideration. A *sporadic task set* $\mathcal{T}$ is a collection of sporadic tasks $\{\tau_1, \tau_2, \dots, \tau_N\}$, where each task τ_i is characterized by a *worst-case execution time* C_i, a *relative deadline* D_i and a *minimum interarrival time* T_i. To cope with implicit, constrained and unconstrained deadline tasks, we also define $\Lambda_i = \min(D_i, T_i)$. We assume that for each task τ_i, $C_i \leq \Lambda_i$, since otherwise the task set would be trivially infeasible. Tasks are also characterized by their utilization $U_i = \frac{C_i}{T_i}$ and their density $\lambda_i = \frac{C_i}{\Lambda_i}$, while U_{tot} and λ_{tot} represent respectively total utilization and total density of the task set.

Each task τ_i generates a potentially infinite sequence of jobs τ_i^j, each of them with *release time* r_i^j, absolute deadline d_i^j and finish time f_i^j. We call *release time sequence* r the set of all release times of tasks in $\mathcal{T}$ (i.e., $r = \{r_i^j : \forall i, j\}$). Moreover, we consider a release time sequence to be valid, if the minimum interarrival time between two consecutive releases of a task is respected. Whenever not specified, we assume the release time sequence to be valid.

A sporadic task set is *feasible* on M processors if, for each possible release time sequence there is a schedule for M processors that meets all the task deadlines. A task system is *schedulable* according to a given scheduling algorithm if, for every possible release time sequence the algorithm produces a schedule that meets all the task deadlines.

Moreover, we remember that, as in Chapter 2, we consider to be able to schedule only at non-negative integer clock ticks t (despite the fact that real numbers are usually used to model the system). In this chapter, t represents the whole interval

$$[t, t+1) \stackrel{\text{def}}{=} \{x \in \mathbb{R} | t \leq x < t+1\}$$

Consequently, $[a, b)$ represents the set of clock ticks from a up to b, and b is not included. As we said in the previous chapter, due to this choice, we

can use mathematical induction on clock ticks for proofs, avoiding potential confusion around end-points, and preventing impractical schedulability results that rely on being able to slice time at arbitrary points.

3. The maxmin load

Given a sporadic task τ_i and a release time sequence, the *minimum demand* of τ_i in any specific time interval is defined to be the minimum amount of time that τ_i must execute within that interval in order to meet all its deadlines.

Note that this definition of the minimum demand of a task does not presume any specific scheduling policy, and it takes into account release times and deadlines both inside and outside the interval. In the latter respect this definition of minimum demand is different from the definition of demand on which the definition of δ_{sum} above is based; in δ_{sum} only tasks with deadlines and release times inside the interval are considered.

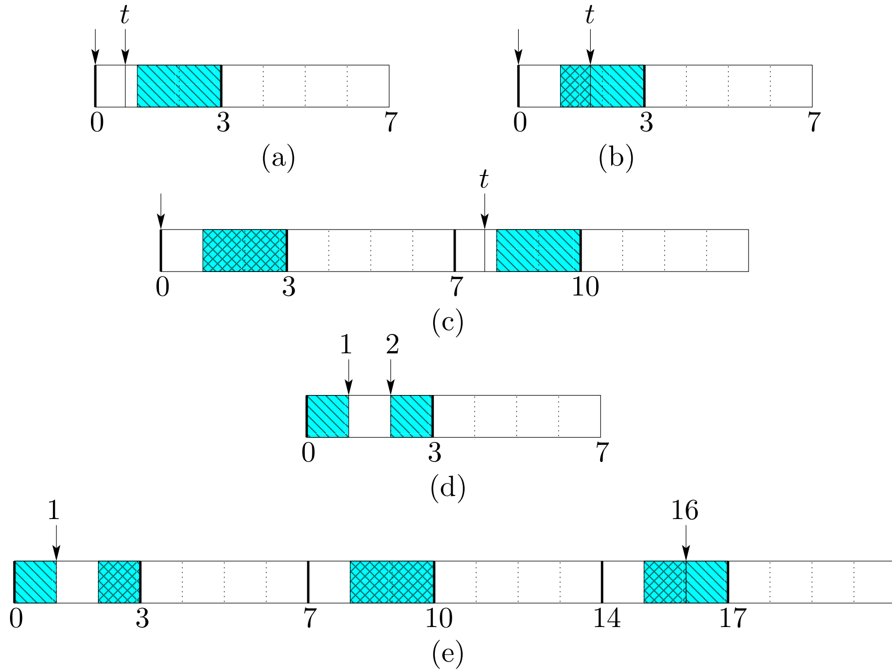

FIGURE 3.1. Minimum demand examples.

EXAMPLE 5. Consider a task $\tau_i = (C_i = 2, D_i = 3, T_i = 7)$ and the release time sequence $0, 7, 14$. Figure 3.1 shows the minimum demand of τ_i for several intervals. The diagonally shaded areas show the most favorable position of τ_i's execution in the schedule for minimizing the work done in the given interval while still meeting deadlines. The cross-hatched areas indicate the portion of that work that cannot be moved outside the given interval without missing a deadline.

(a) The minimum demand of the interval $[0, t)$ is zero for $t \leq 1$, since it is possible to meet all deadlines and not start any jobs of τ_i before time 1.

(b) The minimum demand of the interval $[0, t)$ is $2 - (3 - t)$ for $1 < t \leq 3$, since the first job of τ_i can execute for at most $3 - t$ time instants between t and the deadline 3.
(c) The minimum demand of the interval $[0, t)$ is 2 for $3 < t \leq 8$, since execution of the second job does not need to start until time 8 in order to meet the deadline at time 10.
(c) The minimum demand of the interval $[1, 2)$ is zero, since half the execution of the first job can be done before the start of the interval and the other half can be postponed until after the interval.
(e) The minimum demand of the interval $[1, 16)$ is 4, since the first job cannot do more than one unit of execution before time 1, the second job cannot be postponed past 10, and the third job cannot postpone more than one unit of execution past 16.

□

Given a sporadic task τ_i and a time duration t, the *maxmin demand* $md(\tau_i, t)$ of τ_i for intervals of length t is defined to be the maximum of the minimum demand of $[a, a+t)$, taken over all the valid release time sequences and all interval start times $a \geq 0$.

The *maxmin load* of a set $\mathcal{T}$ of N sporadic tasks is

$$m\ell \stackrel{\text{def}}{=} \sup_{t \geq 0} \sum_{i=1}^{N} md(\tau_i, t)/t$$

From the requirement that $C_i \leq \Lambda_i$, it is clear that $md(\tau_i, t)/t \leq 1$, and so the above least upper bound sup is well defined.

For purposes of analysis, it is helpful to think in terms of intervals that start at time zero.

Given a sporadic task τ_i and a time duration t, the *canonical demand* $cd(\tau_i, t)$ of τ_i for intervals of length t is defined to be the minimum demand of the interval $[0, t)$ with periodic releases starting at time zero (that is $r_i^j = jT_i$ for each $j = 0, 1, 2, \ldots$).

THEOREM 3.1 (Critical zone). *For any set $\mathcal{T}$ of sporadic tasks and any $t \geq 0$, $md(\tau_i, t) = cd(\tau_i, t)$.*

PROOF. Let $\hat{r}$ be any release time sequence and $[a, a+t)$ be any interval of length t. Consider any single task τ_i. It is enough to show that the minimum demand of τ_i in $[a, a+t)$ under $\hat{r}$ is no greater than the minimum demand of τ_i in $[0, t)$ under the canonical release time sequence (where $r_i^j = jT_i$ for each $j = 0, 1, 2, \ldots$). This is done by a series of modifications to the release time sequence $\hat{r}$, each of which does not reduce the minimum demand.

Step 1. Without loss of generality, delete from $\hat{r}$ all the release times that do not contribute to the minimum demand of τ_i in $[a, a+t)$, and let $b \stackrel{\text{def}}{=} \hat{r}_i^1$ be the release time of the first job of τ_i that contributes to that minimum demand.

Step 2. The next step is to show that the minimum demand of τ_i in $[a, a+t)$ under $\hat{r}$ is no greater than the minimum demand with strictly periodic releases starting at b. If there are any release times in $\hat{r}$ that are farther apart

than T_i, shifting those releases closer to the start of the interval $[a, a+t)$ (maintaining the release time sequence valid) cannot decrease the minimum demand in the interval. Therefore, it is sufficient to limit consideration to cases where $\hat{r}_i^j = b + jp_i$.

Step 3. The next step is to show that the minimum demand in $[a, a+t)$ will not be decreased by shifting all the release times so that the first release occurs at a. If $b \geq a$, it is clear that the minimum demand in $[a, a+t)$ will not be decreased by next shifting all the release times down by $b - a$. Therefore, it only remains to show that if $b < a$ the minimum demand in $[a, a+t)$ will not be decreased by shifting all the release times up by $a - b$, so that the first release occurs at a.

Since this is the most complex part of the proof we provide two different ways to cope with it: a more intuitive description (Step 3a), and a more formal proof (Step 3b).

Step 3a. Note that the shift has different effects near the beginning and the end of the interval, and in particular the minimum demand of the task

- near a tends to be increased by the shift;
- near $a+t$ tends to be decreased by the shift.

In order to show that the overall minimum demand in $[a, a+t)$ is not decreased, we consider separately the consequences of the shift at the beginning and at the end of the interval $[a, a+t)$, and we compare the two results.

Since the job of τ_i released at b contributes to the minimum demand of τ_i in $[a, a+t)$, it must be that $b + C_i > a$. The next release of τ_i is at $b + T_i \geq b + C_i > a$ (since $C_i \leq \Lambda_i$). Moreover, due to the minimum interarrival time constraint, there is no release in $[b - T_i, b)$, so no job released before b can be impacted by the shift (that is, after the shift they cannot contribute to the minimum demand, so it was safe to delete them from $\hat{r}$ in Step 1). As a consequence, if we consider what happens near a after the shift, only the job released at b changes its contribution to the minimum demand, which means that the minimum demand near a is increased by exactly the shift amount, $a - b$.

Consider now what happens near the end of the interval, where the overall minimum demand tends to be decreased. The minimum demand of jobs near $a+t$ is the amount of execution that cannot be postponed until after $a+t$. The latest that each job can be postponed is the interval of length C_i immediately preceding its deadline. These intervals are non overlapping, since the job deadlines are all separated by T_i, and $C_i \leq \Lambda_i \leq T_i$. Due to this fact, the shift cannot decrease the minimum demand near the end of the interval more than the shift amount, $a - b$.

So, any decrease in minimum demand near $a+t$ is offset by the increase near a. As a further note, consider that in the particular case that the same job is influenced by both the increase near a and the decrease near $a+t$, the overall reasoning remains valid, and so even in this case shifting all the release times up by $a - b$ does not decrease the minimum demand of τ_i in $[a, a+t)$.

Step $3b$**.** The first job released before a that makes a nonzero contribution to the minimum demand is released at b, so $b + C_i > a$. Since $C_i \leq \Lambda_i \leq T_i$, the next release of τ_i cannot occur earlier than $b + C_i$, so the one and only job of τ_i released before a that contributes to the minimum demand is released at b. Note also that jobs released before b cannot increase their contribution to the minimum demand due to the shift, since even after the shift they are released earlier than $a - T_i$ (due to the minimum interarrival time constraint), which means that they can complete their execution before a (again, $C_i \leq \Lambda_i \leq T_i$). Since their contribution remains 0 even after the shift, it was safe to delete them from $\hat{r}$ in Step 1.

Consider now the following cases for the value of the relative deadline D_i of τ_i.

(1) $D_i \geq a - b + t$: in this case, since $b + D_i \geq a + t$, the deadline of the job of τ_i released at b is after the end of the interval $[a, a + t)$. This case is illustrated by Figure 3.1(d). Every successive job of τ_i, even if released inside $[a, a+t)$, can completely execute between $b + D_i$ and its deadline (which, for the minimum interarrival time constraint, cannot be before $b + D_i + T_i$), so they do not contribute to the minimum demand. It means that the minimum demand of task τ_i in $[a, a + t)$ before and after the shift is formed by only the contribution of the job released at b. The minimum demand of the job before the shift is the execution time of the task, C_i, minus what can be completed before the interval, $a - b$, minus what can be completed between the end of the interval and the deadline, $(b + D_i) - (a + t)$. The total is $C_i - (a - b) - ((b + D_i) - (a + t)) = C_i - D_i + t$. After the shift, the minimum demand is the execution time of the task C_i, minus what can be completed after the interval, $((a + D_i) - (a + t))$. The total is again $C_i - ((a + D_i) - (a + t)) = C_i - D_i + t$. So in this case the shift does not change the minimum demand.

(2) $t \leq D_i < a - b + t$: before the shift, the job of τ_i released at b has its deadline inside $[a, a + t)$ (since $b + D_i < a + t$), so its contribution to the minimum demand is the execution time of the task, C_i, minus what can be completed before the interval, $a - b$, for a total of $C_i - (a - b)$. After the shift, the job is released at a and has its deadline at $a + D_i$, after the end of the interval $a + t$. The contribution changes to its execution time, C_i, minus what can be executed after $a + t$, $D_i - t$, for a total of $C_i - (D_i - t)$. The net result is an increase in the minimum demand equal to $(a - b) - (D_i - t)$ (which is always positive, since $D_i < a - b + t$).

Consider now the contribution of the successive jobs. Since, as said above, these jobs cannot be released before a, their minimum demand is the amount of execution that cannot be postponed until after the end of the interval. The latest that each job can be postponed is the interval of length C_i immediately preceding its deadline. These intervals are non-overlapping, since the job deadlines are all separated by T_i, and $C_i \leq \Lambda_i$. For the same reason, they are surely after the deadline $b + D_i$ of the job of τ_i released at

b. So, their contribution is at most equal to $(a+t)-(b+D_i)$, i.e. the length of the interval $[b+D_i, a+t)$. Note that, for what we said above, the interval is well-defined, and its length is a positive number. After the shift, the whole contribution of these jobs is postponed after the interval $[a, a+t)$, so their contribution goes to 0, for a maximum net decrease of $(a+t)-(b+D_i)$.

The decrease in the contribution of the jobs released after a, is offset by the increase computed above for the job released at b. Again, in this case the shift does not decrease the minimum demand. Note also that while the increase is sure, the decrease is only the worst-case, so the shift not only cannot decrease the whole contribution, but can potentially increase it.

(3) $D_i < t$: before the shift, exactly as above, the contribution of the job of τ_i released at b is $C_i - (a-b)$. After the shift, the job of τ_i released at b is completely executed inside the interval $[a, a+t)$ (since its deadline is at $a + D_i < a + t$) so its contribution to the minimum demand is incremented by exactly the shift amount $a-b$.

Consider now the contribution of the successive jobs. Exactly as we say above, the contribution of these jobs is the amount of execution not postponed until after $a+t$, they are postponed up to the intervals of length C_i exactly before their deadlines, and these intervals are non-overlapping. After the shift, the above sequence of C_i-length intervals has been shifted by $a-b$, so the maximum amount of minimum demand that is shifted from inside to outside the interval is $a-b$.

Again, after the shift, any decrease in minimum demand of the jobs released after a is offset by the increase of the job released at b. So, even in this last case the shift does not decrease the minimum demand.

Since in no case the shift can decrease the minimum demand, we can shift all the release times so that the first job contributing to the minimum demand is released exactly at a.

Step 4. The last step is to observe that the minimum demand in $[a, a+t)$ by periodic releases starting at a is the same as the minimum demand in $[0, t)$ with periodic releases starting at zero.

□

The following necessary condition for feasibility of a sporadic task set follows very directly from the above definitions and theorem.

THEOREM 3.2 (Infeasibility test). *If a set $\mathcal{T}$ of sporadic tasks is feasible on M processors for every valid release time sequence, then $m\ell \leq M$.*

PROOF. Suppose $m\ell > M$. By the definition of $m\ell$, there is some time duration t for which $\sum_{i=1}^{N} md(\tau_i, t)/t > M$. By the definition of $m\ell$ and Theorem 3.1 above, given the canonical release time sequence of each task (that is, strictly periodic releases starting at 0), the total execution time in $[0, t)$ must exceed Mt, which is a contradiction. □

By Theorem 3.2, the condition $m\ell \leq M$ is necessary for the feasibility of a sporadic task set $\mathcal{T}$ over an M-processors platform. While this condition is not sufficient for feasibility in general it is sufficient as well as necessary for the case $M = 1$.

COROLLARY 3.3 (Feasibility test for one processor). *If $m\ell \leq 1$ then $\mathcal{T}$ is schedulable on a single processor.*

PROOF. The proof is by contradiction. Suppose there is a task set $\mathcal{T}$ for which $m\ell \leq 1$, and a valid release time sequence such that with EDF τ_k is the first task missing a deadline, and the missed deadline is at time t.

Let $[t-\Delta, t)$ be the maximal busy interval with endpoint t, that is, the longest interval ending at t for which there are continually one or more uncompleted jobs with deadlines in $[t-\Delta, t)$. It follows that $[t-\Delta, t)$ is the release time of a job with deadline in $[t-\Delta, t)$ and the only jobs that execute in this busy interval are jobs that are released in $[t-\Delta, t)$ and have deadlines in $(t-\Delta, t]$. Since t is a missed deadline of τ_k, $\Delta \geq D_k$.

Let $md\,(\tau_i, \Delta, t)$ denote the total minimum demand (defined in Section 3) of the jobs of τ_i in the time interval of length Δ starting at t (i.e., $[t-\Delta, t)$), and let $W_i(t-\Delta, t)$ denote the sum of the execution times of all the jobs of τ_i that have both release time in $[t-\Delta, t)$ and deadline in $(t-\Delta, t]$. It follows from the definitions that $md\,(\tau_i, \Delta, t) \geq W_i(t-\Delta, t)$.

Since τ_k misses a deadline at t,

$$\sum_{i=1}^{N} md\,(\tau_i, \Delta, t) \geq \sum_{i=1}^{N} W_i(t-\Delta, t) > \Delta$$

By the definition of maxmin demand, $md\,(\tau_i, \Delta) \geq md\,(\tau_i, \Delta, t)$, and so

$$\sum_{i=1}^{N} md\,(\tau_i, \Delta)\,/\Delta \geq \sum_{i=1}^{N} md\,(\tau_i, \Delta, t)\,/\Delta > 1$$

By the definition of $m\ell$,

$$m\ell = \sup_{t \geq 0} \sum_{i=1}^{N} md(\tau_i, t)/t \geq \sum_{i=1}^{N} md\,(\tau_i, \Delta)\,/\Delta > 1$$

which contradicts the assumption that $m\ell \leq 1$.

□

Unfortunately, $m\ell \leq m$ is not a sufficient condition for the feasibility of task sets on multiprocessors, as can be seen in the example below.

EXAMPLE 6. Consider the task set below.

i	C_i	D_i	T_i
1	1	1	2
2	1	1	2
3	2	3	3

The task set illustrates why the result of Corollary 3.3 does not generalize to multiple processors. For this task set, $m\ell = 2 = M$ (computed for $t = 1$ or $t = 3$), which means that it cannot be declared infeasible. However, the task set is clearly not feasible on $M = 2$ processors. The problem is that τ_3

needs to execute for two time units in the interval $[0, 3)$ and there are two time units of execution time available, but the only two units of execution time available run in parallel on two different processors. □

4. How to compute maxmin demand

In order to be able to use Theorem 3.2 as a test of infeasibility, it is necessary to compute the function $m\ell$. The first step is to compute $md(\tau_i, t)$. We can do that for any t as follows:

THEOREM 4.1 (Maxmin demand). *For any sporadic task τ_i and time duration t,*

$$md(\tau_i, t) = j_t C_i + \max\left(0, t - (j_t T_i + D_i - C_i)\right) \tag{4.1}$$

where

$$j_t \stackrel{def}{=} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right)$$

PROOF. By Theorem 3.1, computing $md(\tau_i, t)$ is the same as computing $cd(\tau_i, t)$. Let j_t be the number of jobs of τ_i that must execute to completion entirely within $[0, t)$. It follows that $j_t = 0$ if-and-only-if $t < D_i$. For $j_t \geq 1$ the deadline of the j_tth job falls on or before t and the deadline of the next job falls after t, that is

$$\begin{array}{rcl} (j_t - 1)\, T_i + D_i & \leq t < & j_t T_i + D_i \\ \dfrac{t - D_i}{T_i} & < j_t \leq & \dfrac{t - D_i}{T_i} + 1 \end{array}$$

Since j_t is an integer,

$$j_t = \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1$$

Whether some portion of the execution of the j_{t+1}th job must complete in $[0, t)$ depends on whether $t - (j_t T_i + D_i - C_i) > 0$ (i.e., whether the j_{t+1}th job has "throwforward" on t [**JM74**]).

Case 1: If $t - (j_t T_i + D_i - C_i) \leq 0$ then the j_{t+1}th job can complete by the deadline without executing at all in the interval $[0, t)$. This case is shown in Figure 3.2(a).

Case 2: If $t - (j_t T_i + D_i - C_i) > 0$ then the j_{t+1}th job cannot complete by the deadline $j_t T_i + D_i$ unless it has already completed at least $t - (j_t T_i + D_i - C_i)$ execution by time t. This is case is shown in Figure 3.2(b).

Taking the maximum of these two cases, and adding $j_t C_i$ for the execution times of the first j_t jobs, we obtain Equation (4.1).

□

Note that the function $md(\tau_i, t)$ defined here is similar to $\mathrm{DBF}(\tau_i, t)$. It differs only in being larger by $\max\left(0, t - (jT_i + D_i - C_i)\right)$ (the throwforward term). Therefore, prior techniques for computing rapid approximations to $\mathrm{DBF}(\tau_i, t)$ [**AS04a, BFB05, FBB06**] can be modified to fit $md(\tau_i, t)$. In particular, the function $md(\tau_i, t)$ can be approximated within a tolerance of $U_i\,(T_i - C_i)$ for sufficiently large t, as in Lemma 4.2 below (consider Figure 3.3 for a graphical explanation of the lemma).

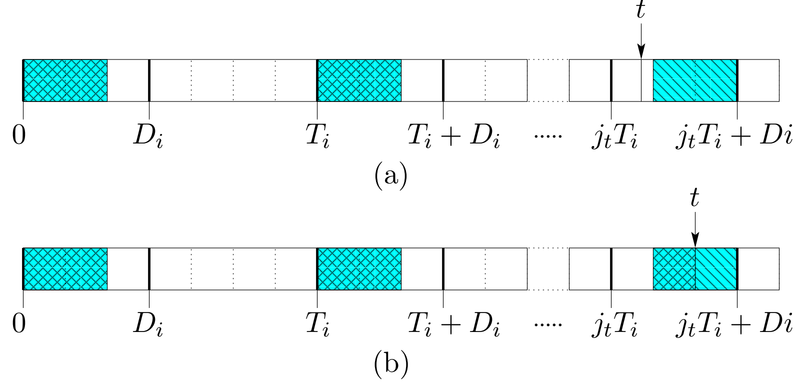

FIGURE 3.2. The two cases of the proof of Theorem 4.1.

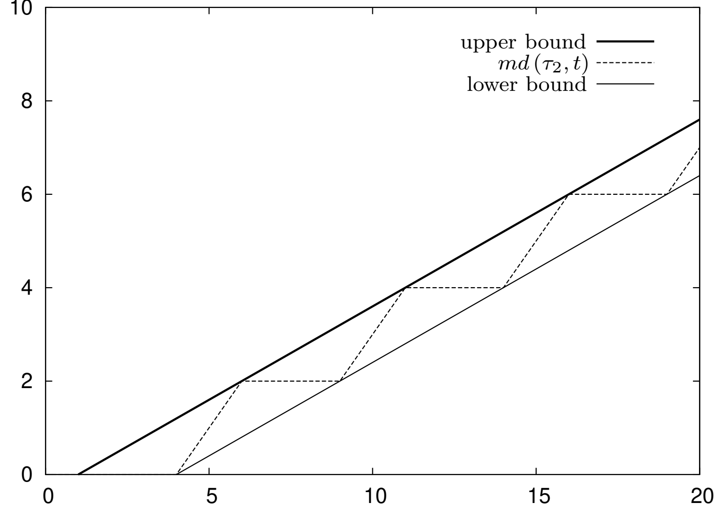

FIGURE 3.3. $md\,(\tau_2, t)$ with lower and upper bounds.

LEMMA 4.2 (Bounds on the maxmin demand). *For any sporadic task τ_i, if $t \geq D_i$ then*

$$U_i\,(t - D_i + C_i) \leq md(\tau_i, t) < U_i\,(t - D_i + T_i) \tag{4.2}$$

PROOF. Since $t \geq D_i$, by Theorem 4.1, $j_t = \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1 \geq 1$ and $md(\tau_i, t)$ is the maximum of two functions, given by j_tC_i and $j_tC_i + t - j_tT_i - D_i + C_i$, which coincide at all the points t such that $t = j_tT_i + D_i - C_i$ for some integer j_t.

The value of $md(\tau_i, t)$ is constant with respect to t and has the value j_tC_i when

$$(j_t - 1)\,T_i + D_i \leq t < j_tT_i + D_i - C_i$$

It increases linearly with t and has the value $t - D_i - j_t\,(T_i - C_i) + C_i$ when

$$j_tT_i + D_i - C_i \leq t < j_tT_i + D_i$$

Therefore, $md(\tau_i, t)$ is bounded above by the linear interpolation of the points where $md(\tau_i, t)$ changes from increasing to constant and bounded

below by the linear interpolation of the points where $md(\tau_i, t)$ changes from constant to increasing. The upper bound can be obtained by interpolating the values of $md(\tau_i, t)$ at the points $t = (j_t - 1)\,T_i + D_i$.

$$md(\tau_i, t) = j_t C_i = \frac{t - D_i + T_i}{T_i} C_i = U_i\,(t - D_i + T_i)$$

and the lower bound can be obtained by interpolating the values of $md(\tau_i, t)$ at the points $t = j_t T_i + D_i - C_i$.

$$md(\tau_i, t) = j_t C_i = \frac{t - D_i + C_i}{T_i} C_i = U_i\,(t - D_i + C_i)$$

□

Note that the upper bound for $\mathrm{DBF}(\tau_i, t)$ (see [**BFB05, FBB06**]) is the same as for $md(\tau_i, t)$, but the lower bound for $\mathrm{DBF}(\tau_i, t)$ is smaller by $U_i C_i$.

EXAMPLE 7. Consider the task set below.

i	C_i	D_i	T_i
1	2	3	7
2	2	6	5

The zig-zag dotted line in Figure 3.3 shows the function $md(\tau_i, t)$ for τ_2. The solid lines are the upper and lower bounds of Equation (4.2). □

5. How to approximate maxmin load

Calculating $m\ell$ requires the research of the maximum of the function $\sum_{i=1}^{N} md(\tau_i, t)/t$ over an unbounded range of real numbers t. This can be done efficiently because it is possible to show that there is a finite range of values of t at which the maximum of $\sum_{i=1}^{N} md(\tau_i, t)/t$ can occur.

Observe that $md(\tau_i, t)$ alternates between constant intervals and intervals of linear increase. The changes from increasing to constant occur at the points $jT_i + D_i$, for $j = 1, 2, \ldots$. This behavior is maintained summing all the contributes from different tasks, so the function $\sum_{i=1}^{N} md(\tau_i, t)$ is made up of constant segments and linearly increasing segments. Each constant segment begins at one of the points $jT_i + D_i$, for $j = 1, 2, \ldots$ and $i = 1, 2, \ldots, N$. After the sum is divided by t, there are peaks at those same points, as shown in Example 8.

EXAMPLE 8. Consider the task set given in Example 7. Figure 3.4 shows $md\,(\tau_1, t)$ (thin solid line), $md\,(\tau_2, t)$ (dotted line), and $md\,(\tau_1, t) + md\,(\tau_2, t)$ (heavier solid line). Changes from increasing to constant are at $3, 10, 17, \ldots$ for τ_1, and $6, 11, 16, \ldots$ for τ_2. At the same points the sum decreases its slope. In Figure 3.5 the same values are divided by t, and the peaks are at the same points. □

Lemma 4.2 provides the following bounds for $md(\tau_i, t)/t$, for $t \geq D_i$, within a tolerance of $U_i\,(T_i - C_i)\,/t$:

$$U_i\left(1 + \frac{C_i - D_i}{t}\right) \leq \frac{md(\tau_i, t)}{t} \leq U_i\left(1 + \frac{T_i - D_i}{t}\right) \tag{5.1}$$

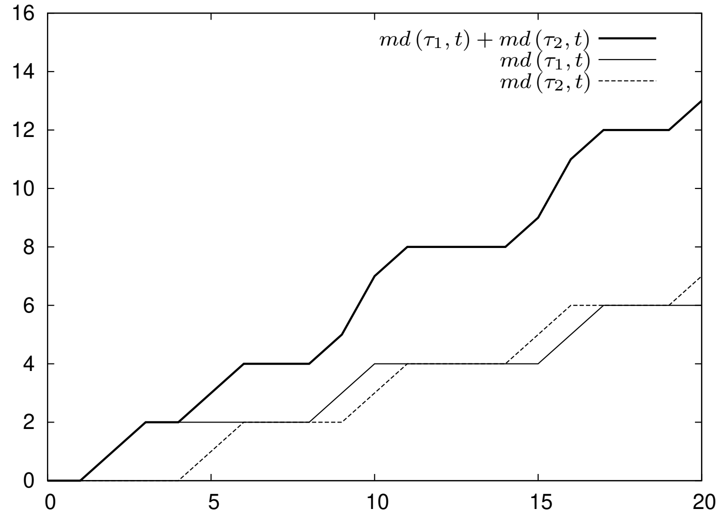

FIGURE 3.4. $md(\tau_1, t)$, $md(\tau_2, t)$, and their sum.

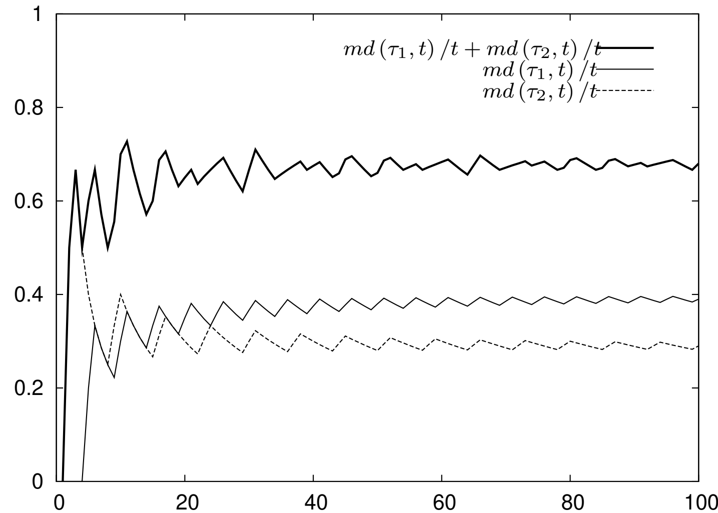

FIGURE 3.5. $\frac{md(\tau_1,t)}{t}$, $\frac{md(\tau_2,t)}{t}$, and their sum.

Since the value of the upper bound expression (on the right above) tends to U_i for sufficiently large t, the global maximum is at one of the early peaks. The question is how far to look.

By computing $md(\tau_i, t)$ exactly for small values of t and using the above linear approximations for large values of t, the search space for the maximum can be limited to a size that is polynomial in the length of the task set. The technique is analogous to that used to compute δ_{sum} in [**FBB06**], which is based on an earlier work by Albers and Slomka [**AS04a**] for uniprocessor feasibility analysis.

For an approximation to $md(\tau_i, t)/t$ from below, suppose to define the function $g_\epsilon(\tau_i, t)$ as follows

$$g_\epsilon(\tau_i, t) \stackrel{\text{def}}{=} \begin{cases} md(\tau_i, t)/t, & \text{if } t < \max\left(D_i, N\frac{U_i(T_i - C_i)}{\epsilon}\right) \\ U_i\left(1 + \frac{C_i - D_i}{t}\right) & \text{otherwise} \end{cases} \tag{5.2}$$

Note that the approximation can be done from either above or below, depending on whether one is interested in proving infeasibility (for M processors) or feasibility (for a single processor). However, the latter is not very interesting, since the δ_{sum} test is already sufficient for the single-processor case. Therefore, we only describe in full the approximation from below. For the approximation to $md(\tau_i, t)$ from above one just needs to replace $U_i\left(1 + \frac{C_i - D_i}{t}\right)$ by $U_i\left(1 + \frac{T_i - D_i}{t}\right)$.

If $t < \max\left(D_i, N\frac{U_i(T_i - C_i)}{\epsilon}\right)$, $md(\tau_i, t)/t - g_\epsilon(\tau_i, t) = 0 \le \frac{\epsilon}{N}$. Otherwise, from Equation (5.1) it follows that, for every $t \ge 0$,

$$\begin{aligned} 0 &\le md(\tau_i, t)/t - g_\epsilon(\tau_i, t) \\ &\le \frac{U_i\,(T_i - C_i)}{t} \le \frac{\epsilon}{N} \end{aligned}$$

Summing, we obtain

$$0 \le \sum_{i=1}^{N} md(\tau_i, t)/t - \sum_{i=1}^{N} g_\epsilon(\tau_i, t) \le \epsilon \tag{5.3}$$

Let

$$\hat{g}_\epsilon(\mathcal{T}) \stackrel{\text{def}}{=} \sup_{t \ge 0} \sum_{i=1}^{N} g_\epsilon(\tau_i, t)$$

It follows from Equation (5.3) that

$$\hat{g}_\epsilon(\mathcal{T}) \le m\ell \le \hat{g}_\epsilon(\mathcal{T}) + \epsilon$$

Observe that $g_\epsilon(\tau_i, t)$ is monotonic with respect to t except at the points where $\left\lfloor \frac{t - D_i}{T_i} \right\rfloor$ makes a jump, that is, only at values $t = kT_i + D_i$ such that $t < N\frac{U_i(T_i - C_i)}{\epsilon}$, for $k = 0, 1, 2, \ldots$ and $i = 1, 2, \ldots, N$. Therefore, local maxima of $\sum_{i=1}^{N} g_\epsilon(\tau_i, t)$ can occur only at such points. The set $\mathcal{S}(\mathcal{T}, \epsilon)$ of such points can be described as follows:

$$\left\{ kp_i + d_i \mid 0 < k < \frac{NU_i\,(T_i - C_i)}{\epsilon T_i} - \frac{D_i}{T_i}, 1 \le i \le N \right\} \tag{5.4}$$

Therefore,

$$\hat{g}_\epsilon(\mathcal{T}) = \max_{t \in \mathcal{S}(\mathcal{T}, \epsilon)} \sum_{i=1}^{N} g_\epsilon(\tau_i, t)$$

and since $g_\epsilon(\tau_i, t) \le md(\tau_i, t)/t$,

$$\hat{g}_\epsilon(\mathcal{T}) \le \max_{t \in \mathcal{S}(\mathcal{T}, \epsilon)} \sum_{i=1}^{N} md(\tau_i, t)/t \le m\ell$$

The value $\hat{g}_\epsilon(\mathcal{T})$ can be computed by evaluating $\sum_{i=1}^{N} g_\epsilon(\tau_i, t)$ at each of the points in $\mathcal{S}(\mathcal{T}, \epsilon)$. Given any fixed tolerance $\epsilon > 0$, and assuming that each for each task τ_i, $U_i \leq 1$, Equation (5.4) provides the following bound on the number of points $t \in \mathcal{S}(\mathcal{T}, \epsilon)$.

$$\sum_{i=1}^{N} \frac{NU_i (T_i - C_i)}{\epsilon T_i} - \frac{D_i}{T_i} = \sum_{i=1}^{N} \frac{NU_i}{\epsilon} - \sum_{i=1}^{N} \left(\frac{NU_i^2}{\epsilon} + \frac{D_i}{T_i} \right) \leq \frac{N}{\epsilon} U_{tot}$$

A straightforward implementation of $\sum_{i=1}^{N} g_\epsilon(\tau_i, t)$ has $\mathcal{O}(N)$ complexity, so the total complexity is $\mathcal{O}(N^2 U_{tot}/\epsilon)$. Note that since $U_{tot} \leq N$, $\mathcal{O}(N^2 U_{tot}/\epsilon) \in \mathcal{O}(N^3/\epsilon)$. As a consequence, the runtime of the algorithm Approximate-$m\ell$ is polynomial in the number N of tasks in $\mathcal{T}$ and $1/\epsilon$, and is independent of the task parameters.

Note further that in an application the number of processors M is known and one can assume that $U_{tot} \leq M$, so $\mathcal{O}(N^2 U_{tot}/\epsilon) \in \mathcal{O}(N^2 M/\epsilon)$.

Additional heuristics, based on the following two lemmas, can often reduce the actual running time below the $\mathcal{O}(N^2 M/\epsilon)$ worst-case bound.

Lemma 5.1 (Relations among feasibility functions). *For any sporadic task set $\mathcal{T}$,*

$$U_{tot} \leq \delta_{sum} \leq m\ell \leq \lambda_{tot}$$

Proof. It was shown in [**FBB06**] that $U_{tot} \leq \delta_{sum} \leq \lambda_{tot}$. The function $m\ell$ defined here is similar to δ_{sum}, but differs by using $md(\tau_i, t) = \mathrm{DBF}(\tau_i, t) + \max(0, t - (jT_i + D_i - C_i))$ instead of $\mathrm{DBF}(\tau_i, t)$. Therefore, it is clear that $\delta_{sum} \leq m\ell$, and so only the upper bound needs to be proved.

If $t < D_i$ then, from the definition of $md(\tau_i, t)$,

$$md(\tau_i, t) = \max(0, t - D_i + C_i)$$

This is non-decreasing with respect to t, and so the maximum value of $md(\tau_i, t)/t$ for $t \leq D_i$ is $C_i/D_i \leq \lambda_i$.

If $t \geq D_i$, then, from Equation (5.1),

$$\frac{md(\tau_i, t)}{t} \leq U_i \left(1 + \frac{T_i - D_i}{t} \right)$$

If $T_i \geq D_i$ the term $\frac{T_i - D_i}{t}$ is non-increasing with respect to t, and so for $t \geq D_i$,

$$\frac{md(\tau_i, t)}{t} \leq U_i \left(1 + \frac{T_i - D_i}{D_i} \right) = \frac{C_i}{D_i} = \lambda_i$$

If $T_i < D_i$ the term $\frac{T_i - D_i}{t}$ is increasing with respect to t, so the least upper bound is the limit for large t.

$$\frac{md(\tau_i, t)}{t} \leq \lim_{t \to \infty} \frac{md(\tau_i, t)}{t} = U_i = \lambda_i$$

It follows that the least upper bound of $\sum_{i=1}^{N} md(\tau_i, t)/t$ is bounded by λ_{tot}. □

We underline that Lemma 5.1 above simplifies the proof of Corollary 3.3. In fact, based on the fact that $\delta_{sum} \leq m\ell$, and considering the well-known fact that $\mathcal{T}$ is schedulable by EDF on one processor if $\delta_{sum} \leq 1$ [**BHR90**], we

clearly obtain that if $m\ell \leq 1$, then $\delta_{sum} \leq 1$ and the task set is schedulable by EDF on a single processor.

Another condition for early termination of the search for $m\ell$ is based on the fact that $md(\tau_i, t)/t$ tends to U_i for sufficiently large t (cfr. Equation (5.1)), and so the difference between U_{tot} and $\sum_{i=1}^{N} md(\tau_i, t)/t$ gives an upper bound on t. This is expressed in Lemma 5.2 below.

LEMMA 5.2 (Upper bound on maxmin load research). *If*

$$U_{tot} + \gamma \leq \sum_{i=1}^{N} \frac{md(\tau_i, t)}{t} \tag{5.5}$$

for some $\gamma > 0$, then

$$t \leq \sum_{i=1}^{N} \frac{U_i \max(0, T_i - D_i)}{\gamma} \tag{5.6}$$

PROOF. It follows from Equation (5.1) above that if $t \geq D_i$ then

$$\frac{md(\tau_i, t)}{t} \leq U_i \left(1 + \frac{T_i - D_i}{t}\right) \leq U_i \left(1 + \frac{\max(0, T_i - D_i)}{t}\right)$$

There are two other cases:

(1) If $t < D_i - C_i$ then

$$\frac{md(\tau_i, t)}{t} = 0 \leq U_i \left(1 + \frac{\max(0, T_i - D_i)}{t}\right)$$

(2) If $D_i - C_i \leq t < D_i$ then, since $U_i \leq 1$, $U_i (t - D_i) \geq t - D_i$, and so

$$\begin{aligned}
\frac{md(\tau_i, t)}{t} &= \frac{t - D_i + C_i}{t} \\
&\leq \frac{U_i (t - D_i) + C_i}{t} = \frac{U_i (t - D_i) + U_i T_i}{t} \\
&= U_i \left(1 + \frac{T_i - D_i}{t}\right) \\
&\leq U_i \left(1 + \frac{\max(0, T_i - D_i)}{t}\right)
\end{aligned}$$

Therefore,

$$\sum_{i=1}^{N} \frac{md(\tau_i, t)}{t} \leq U_{tot} + \frac{\sum_{i=1}^{N} U_i \max(0, T_i - D_i)}{t} \tag{5.7}$$

Composing Equations (5.5) and (5.7) yields

$$U_{tot} + \gamma \leq U_{tot} + \frac{\sum_{i=1}^{N} U_i \max(0, T_i - D_i)}{t}$$

from which Equation (5.6) follows.

□

Pseudo-code for the algorithm APPROXIMATE-$m\ell$ is given in Figure 3.6. The initial value given to $m\ell$ in line 1 is based on the relation $m\ell \geq U_{tot}$ from Lemma 5.1. The algorithm incorporates two heuristics, both applied in line 8, that permit the computation of $m\ell$ to terminate without looking

---

APPROXIMATE-$m\ell\,(\mathcal{T},\epsilon)$

1 $m\ell \leftarrow U_{tot}$
2 $D_{\max} \leftarrow \max_{i=1}^{n} D_i$
3 $limit \leftarrow \max\left(\mathcal{S}\left(\mathcal{T},\epsilon\right)\right)$
4 **for** each $t \in \mathcal{S}\left(\mathcal{T},\epsilon\right)$, in increasing order **do**
5 $\quad m\ell \leftarrow \max\left(m\ell, \sum_{i=1}^{N} md(\tau_i,t)/t\right)$
6 $\quad$ **if** $(t > D_{\max})$ **then**
7 $\quad\quad limit \leftarrow \min\left(limit, \sum_{i=1}^{N} U_i \max\left(0, T_i - D_i\right) / \left(m\ell - U_{tot}\right)\right)$
8 $\quad$ **if** $((t \geq limit)$ or $(m\ell > \lambda_{tot}(\mathcal{T}) - \epsilon))$ **then**
9 $\quad\quad$ **return** $m\ell$
10 $\quad$ **return** $m\ell$;

FIGURE 3.6. Pseudo-code for approximate computation of $m\ell$ from below.

at all the elements of $\mathcal{S}\left(\mathcal{T},\epsilon\right)$. One of these is the relationship $m\ell \leq \lambda_{tot}$, from Lemma 5.1, which allows to return immediately if the maximum of $md(\tau_i,t)$ over the range of values examined exceeds $\lambda_{tot} - \epsilon$. The other heuristic allows to return immediately if the range of values examined so far exceeds the value of *limit* computed at line 7, based on Lemma 5.2.

Note again that the above computation approximates $m\ell$ from below, for use in proving a task set is not feasible on M processors. Similar reasoning can be used to approximate $m\ell$ from above, with similar runtime complexity. The technique of approximation from above is explained in more detail in [**FBB06**], in the context of computing δ_{sum}.

6. Empirical performance

It is clear that the new load-bound function $m\ell$ is an improvement over δ_{sum} for screening out infeasible task sets, since we have shown in Lemma 5.1 that $\delta_{sum} \leq m\ell$ and we have shown by Example 4 that in some cases $m\ell > M$ while $\delta_{sum} \leq M$.

To get a sense of how often the difference between these two functions is enough to matter, we ran experiments on a large number of pseudo-randomly chosen sets of tasks. The experiments compared the number of task sets eliminated by the new load-bound function against the number eliminated by the old load-bound function δ_{sum}. We report here only the more significant results of such experiments.

Figures 3.7, 3.8, and 3.9 show the result of experiments on $1,000,000$ pseudo-randomly generated task sets with uniformly distributed utilizations and uniformly distributed constrained deadlines, with total utilizations limited to $U_{tot} \leq M$ for $M = 2$ and 8. Each graph is a histogram in which the X axis corresponds to values of U_{tot} and the Y axis corresponds to the number of task sets with U_{tot} in the range $[X, X + 0.01)$ that satisfy a given criterion.

- For the top line, which is unadorned, there is no additional criterion. That is, the Y value is simply the number of task sets with

$X \leq U_{tot} < X + 0.01$. The experiments did not include any task sets with $U_{tot} > M$, since they are clearly not feasible.

- For the second line, decorated with large boxes, the criterion is $\delta_{sum} \leq M$. The Y value is the number of task sets that *might* be feasible according to this criterion. The space between this line and the first indicates how many task sets are detected as infeasible.
- The third line, decorated with small solid squares, corresponds to the criterion $m\ell \leq M$. The Y value is the number of task sets that *might* be feasible according to this criterion. The region between this line and the one above it indicates the improvement in recognition of infeasible task sets due to using $m\ell$, as compared to δ_{sum}. It can be seen that the condition $m\ell \leq M$ identifies significantly more infeasible task sets than the condition $\delta_{sum} \leq M$, especially for systems with a large number of tasks and processors.
- The bottom line, decorated with small circles, corresponds to the criterion $\lambda_{tot} \leq M$. This indicates how many task sets at each utilization level are *definitely* feasible according to this criterion. The region between this line and the one above it indicates the number of task sets whose feasibility remains indeterminate using the simple tests described here.

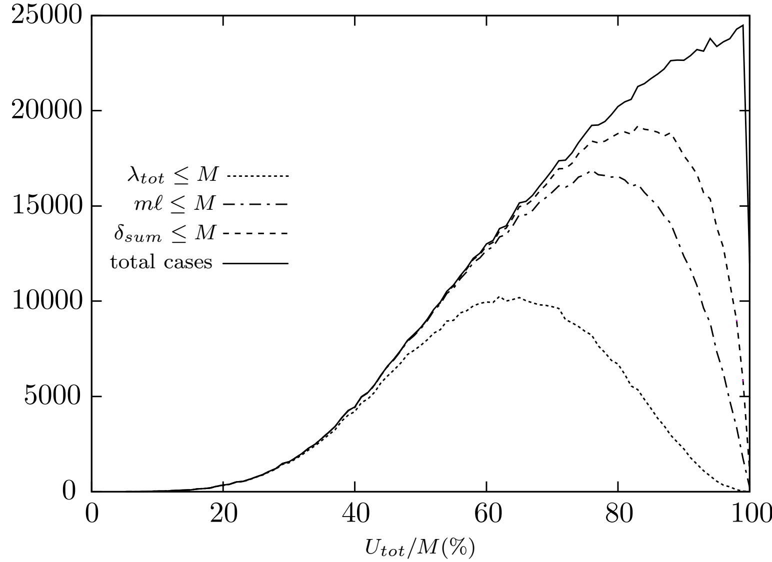

FIGURE 3.7. Histograms for $U_{tot} \leq 2$, $M = 2$.

However, we must say that the major benefits from the introduction of the load-bound function (both δ_{sum} and $m\ell$ relate to constrained deadline systems. To see this, suppose to have a task set in which all the deadlines are greater than the periods. In such a case, the only working screening method is $U_{tot} \leq M$, since a deadline larger than the period only give more freedom to the tasks. As a consequence, the feasibility is limited only by the utilization. This means that the load-bound function improves the analysis of only constrained deadlines tasks.

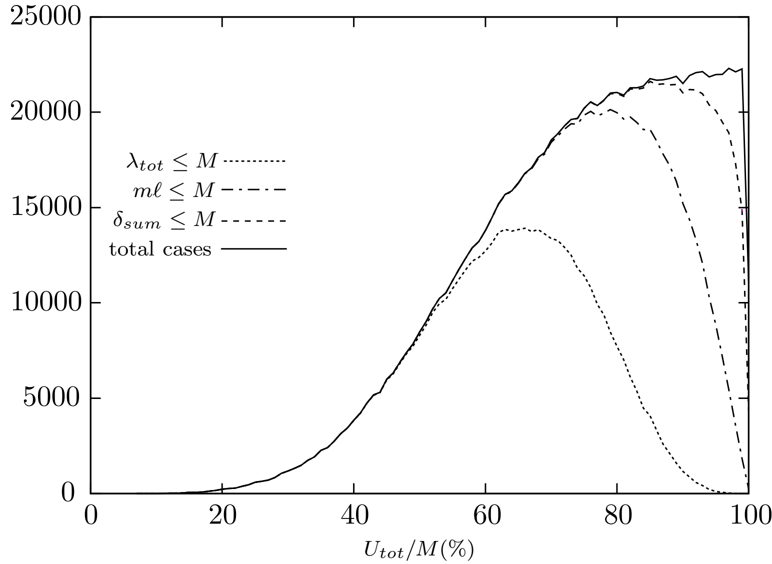

FIGURE 3.8. Histograms for $U_{tot} \leq 4$, $M = 4$.

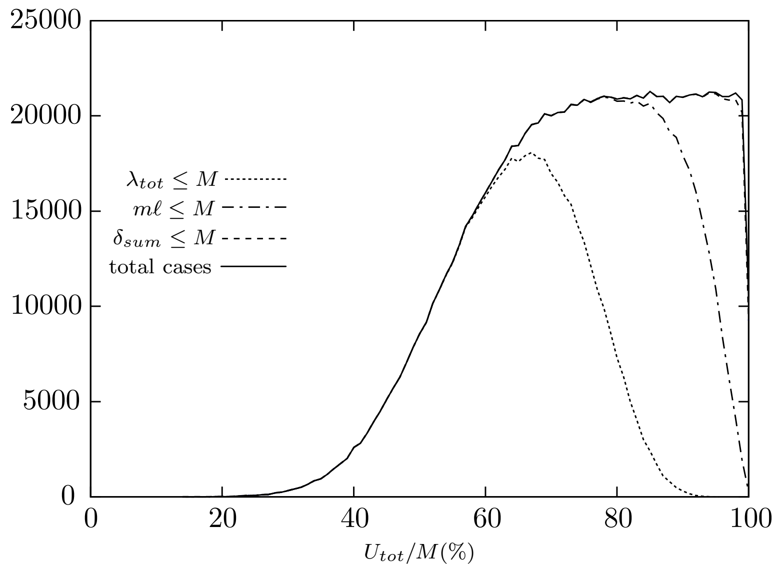

FIGURE 3.9. Histograms for $U_{tot} \leq 8$, $M = 8$.

Another important aspect is the computational complexity of our algorithm. Figure 3.10 shows the number of iterations taken by algorithm APPROXIMATE-$m\ell$ to compute $m\ell$ and the number of iterations taken to compute δ_{sum}. For fairness in comparison, δ_{sum} was approximated from below, using an algorithm similar to APPROXIMATE-$m\ell$. Note that this is distinct from the algorithms for approximating δ_{sum} reported in [**FBB06**] because the primary subject of interest was feasibility, and we are approximating from below because the current subject of interest is infeasibility.

Observe that the computation of $m\ell$ converges faster than the computation of δ_{sum}. Since the algorithms are virtually the same, the reason for the faster convergence is that the linear function used to estimate the error

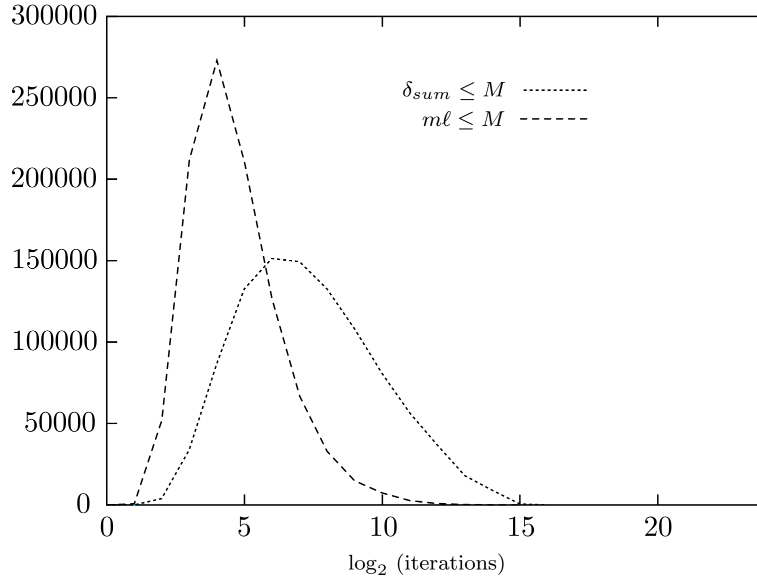

FIGURE 3.10. Iterations to compute load-bound for $U_{tot} \leq 4$.

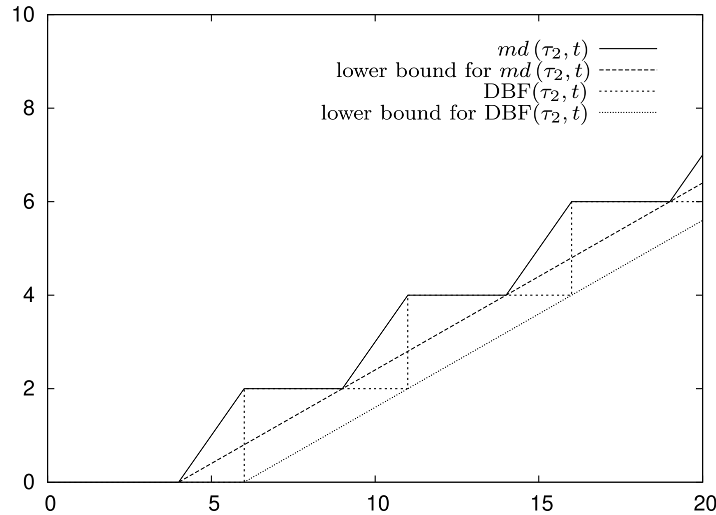

FIGURE 3.11. Relation of lower bound for DBF(τ_i, t) and $md(\tau_i, t)$ for τ_2 of Example 7.

for $md(\tau_i, t)$ is larger than that for DBF(τ_i, t). This is illustrated for the task τ_2 of Example 7 in Figure 3.11. The thick solid zigzag line is the function $md\,(\tau_2, t)$ and the thick dashed straight line is the corresponding lower bound. The thinner solid stepped line is the function DBF(τ_2, t) and the thin dashed straight line is the corresponding lower bound.

7. Improvements in simulations

The possibility to recognize a task set as infeasible (that is, not schedulable whatever scheduling algorithm we decide to use), is particularly useful

in the estimation and comparison of the performances of the scheduling algorithms. Consider for example the global comparison among algorithms done in the previous chapter. In the analysis of Figures 2.21, 2.22 and 2.23 of Chapter 2, we were forced to conclude that, despite the interesting improvement we have done in detecting schedulable task sets with different algorithms and tests, the situation remained quite bad for high utilizations. While this is true (and partly expected, considering the pessimistic hypothesis at the base of the tests), the situation for constrained deadline systems is better than presented in such experiments. As an example, we report below the same experiment of Figure 3.12, with the only difference that we implemented also the feasibility test described in this chapter (see Chapter 2 for a complete description of the test generation method).

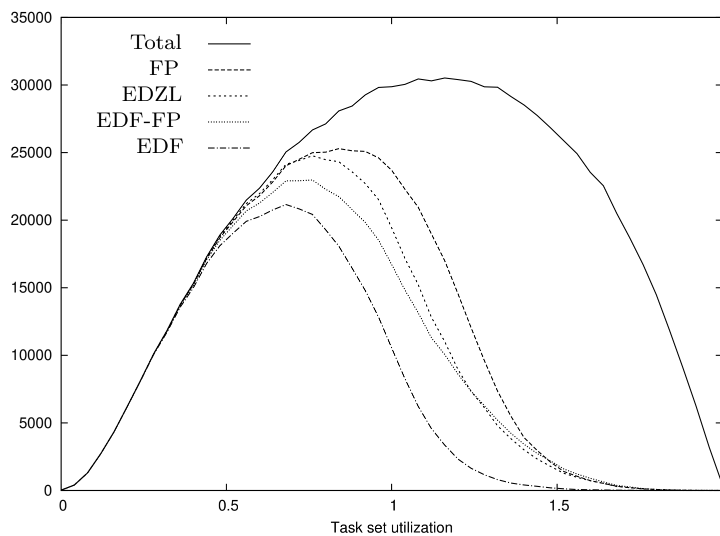

FIGURE 3.12. Global comparison on feasible task sets: 2 processors, $D_i \leq T_i$, exponential distribution with mean 0.25.

The results clearly show that while we are still far from the optimality, the gap is less than previously shown.

8. Conclusion

The new load-bound function $m\ell$ provides a more accurate and more rapidly computable method than has been known before for detecting that a sporadic task set is infeasible. The value of this result is in narrowing the area of uncertainty, between the task sets that can be verified as feasible (e.g., by showing they are schedulable according to some specific algorithms such as EDF, Pfair, or others) and those that are not schedulable.

Although the method for computing $m\ell$ is only approximate, an approximation is adequate for all practical purposes. In any real application of a schedulability test one needs to allow a margin of safety, to account for such factors as errors in predicting the worst-case task execution times, inaccuracy of the timer used to schedule the tasks, and preemption delays due to critical sections.

Incidental to the main objective of this research, it turned out that the condition $m\ell \leq 1$ is also a necessary and sufficient test for feasibility and EDF schedulability of sporadic task sets on single processors systems. This confirms that $m\ell$ is a consistent and natural refinement of δ_{sum}. Moreover, since the approximate computation of $m\ell$ (from below) converges faster than δ_{sum}, it would seem to have a practical advantage.

The problem of finding an efficient feasibility test that is both necessary and sufficient for multiprocessor systems, or to show that this problem is NP-hard, appears to remain open.

CHAPTER 4

Integrity Problem: the partitioned approach

1. Overview

Multiprocessors are usually dedicated to improving performances of the systems through parallel execution of different activities at the same time instant. However, there is another interesting possibility to exploit multiprocessor platforms: providing fault-tolerance to safety-critical applications. In such applications, the event of deadline misses or wrong results produced is not only undesirable, but potentially dangerous for life. As a consequence, it is necessary to guarantee both timeliness and reliability. Examples are easily found in space or medical applications, in which a fault can endanger the life of the involved people, and lead to extreme waste of money for the loss of expensive equipments.

This is not a new problem, but its importance is increasing in recent years, due to technology scaling. It is well-known [**Bau01, SKK^{+}02**], in fact, that technology scaling sensitizes electronic devices to external disturbs. Reasons of this fact relate to several aspects of scaling, such as lower voltage levels, lower capacitances, higher working frequency. The overall effect is the growth of the probability that alpha particles, atmospheric neutrons and similar low energy particles cause temporary bit-flipping in memory and logic circuits (so called *soft errors*). For this reason, tolerance to soft errors is bound to become a major aspect in system design. We underline that while this is particularly true for safety-critical systems, the same requirement for fault-tolerance applies to almost every field. In fact, as the probability to faults raises, non-critical applications must demonstrate a certain degree of fault robustness, as well.

The classical way of providing fault-tolerance on multiprocessor platforms is to use time and/or space redundancy. In time redundancy, the same software is executed two or more times on the same CPU, and the produced results are compared. However, final results could be influenced by the effective instants of execution, possibly leading to several different, although correct, results. In such a case, the comparison is difficult. If, for example, the computation depends on external values read from sensors, clearly the time of computation is essential. Note that in general it is not even safe to store the read values, because usually data samples have a predefined, and possibly short, time validity. In space redundancy, on the contrary, the same software is executed at the same time on different CPUs. One solution based on this idea is to execute replicas in *lock-step*: each involved processor executes the *same* code at the *same* time (i.e., step by step), and every result is compared, instantly revealing differences in each single instruction. To simplify the circuitry one idea is to reduce the

comparison points, for example considering only I/O operations. This way, we also obtain that comparisons can be completely implemented outside the processor (e.g., at the interconnection between processor and bus). This is very useful, because there is no need to re-design the processor, with several positive effects:

- processor design is a very difficult, long and expensive task; avoiding it allows to save money and time;
- a new processor requires deep testing procedures, in particular when employed in safety-critical applications; instead, processors already on the market are deeply tested by both specific testing procedures (before market) and use (after market); in such a case, a new testing phase can be avoided;
- the time-to-market is reduced, since the design and test phase is avoided;
- platform upgrades are easier, since a new, more powerful processor can be integrated in the platform easily.

By using these techniques, it is possible to implement a *fault-tolerant* system (with more than 2 CPUs and by using an appropriate voting or fault-masking mechanism), or a *fail-silent* system (with two CPUs and a comparator) where the fault is simply detected but not corrected. In both cases, the fault-tolerant behavior is achieved at the cost of a reduced computational power.

This approach is usually static, in the sense that the configuration does not vary in time (i.e., processors are coupled or independent for the whole platform life). Hence the fault-tolerance characteristics and the performance are not adjusted on the application requirements. However, such a limitation may be too restrictive because not all software tasks are fault-critical. In general, applications consist of a mixture of fault-tolerant, fail-silent and non-fault-tolerant tasks. It would be desirable to use the multiprocessor platform at its best: as a replicated hardware platform for fault-tolerant and fail-silent tasks; as a parallel processor platform for the non-fault-tolerant tasks. Unfortunately, in classical fault-tolerant systems, such a degree of flexibility is not allowed.

Contributions. In this chapter we describe a technique, first published in [**CBLF07**], based on dynamic on-line reconfiguration of a 4-processors multicore hardware platform, to achieve a trade-off between performance (through parallelism) and fault-tolerance (through hardware replication). Our technique consists in dividing the time-line into time slots, each one dedicated to the service of a different class of tasks: fault-tolerant tasks that require hardware replication, fail-silent tasks, and non-fault-tolerant tasks that require parallelism. At each slot, we dynamically reconfigure the hardware platform to support a certain degree of replication or parallelism. To guarantee the schedulability of each class of tasks in its slot, we apply the theory of hierarchical scheduling [**LB04, SL03, FM02**].

Based on that, we propose a technique to configure the platform to achieve different goals. We then propose two examples: first we show how to tune the platform to minimize the processor bandwidth wasted in overhead; then we consider how to maximize its flexibility at run-time.

In this chapter we apply this technique to partitioned scheduling algorithms for multiprocessor platforms, such as EDF and FP. However, it would be desirable to extend such techniques to use also global scheduling algorithms such as EDF-global, FP-global or EDZL. In Appendix A we consider some of the problems we face to achieve this extension.

1.1. Related works. The problem of fault-tolerance in multiprocessor systems has been thoroughly addressed. Two main branches are usually proposed: on one side, building stronger hardware platforms, capable to resist to faults and continue their work at the same or at a degraded level; on the other side, making the application able to recover from faults, either by time or space redundancy.

The literature about hardware fault-tolerance techniques is extremely vast, and an exhaustive analysis is almost impossible. To protect the application from faults, a general approach is *redundancy*. The idea is to introduce redundant copies of the elements to be protected (processors or other components), and exploit them in the event of a fault.

We focus our research on the so called *lock-step configuration*, in which the redundancy is used for both fault detection and recovery. In a lock-step configuration (explained in more detail in Section 2.4), two or more processors execute the same code cycle by cycle, and a dedicated circuitry compares the results. When a fault is detected, recovery can be performed via SW (e.g., checkpoints and re-execution) and/or HW (e.g., reconfiguration of the application on the remaining processors).

Far from being only a theoretical technique for providing fault-tolerance in multiprocessor platforms, the lock-step approach has been applied in several commercial systems. The core of the Sequoia computer [**Ber88**] was a large set of *Processor Elements*, each one composed of two Motorola MC68020 operating in lock-step and a comparator testing the identicalness of the results. In more recent years, the lock-step idea is exploited, e.g., as part of the *Continuous Processing Technology* implemented in the Stratus ftServer family of products. Similarly, in the HP NonStop Advanced Architecture it is possible to configure two Intel Itanium processors to work in lock-step [**NSA**].

Similar solutions exist also for embedded systems. Xilinx produces the Virtex-II Pro FPGA, based on two IBM PowerPC 405, and uses the FPGA as the core of its ML310 Embedded Development Platform. One of the proposed applications for the ML310 Platform is a processor lock-step for fault-tolerant applications [**NG**]. The IBM PowerPC 750GX processor includes a *lock-step facility* [**Els, PPC**], which integrates all the circuitry, from comparator to data steering, necessary for the connection of two identical processors in lock-step.

A major disadvantage in the way this technique is implemented in commercial systems is the lack of flexibility: despite the fact that every system is composed of tasks with different integrity requirements, the fault-tolerance techniques do not vary in time. In this sense, the architecture we propose is extremely more flexible, since its fault-tolerance techniques can be tuned on the specific application, limiting the necessary loss in computational power.

In the field of software fault-tolerance techniques, scheduling algorithms with safety characteristics are particularly interesting. A long series of works from the Real-Time Systems Research Group at University of York, and summarized in [**Pun97, Lim03**], propose different solutions for the schedulability of real-time task sets under fixed priority, considering various assumptions on faults and fault-tolerance techniques. Another interesting approach is based on the well-known concept of primary/backup [**CB98, MMG94**], in which, for each critical task in the system, a backup copy (which provides a degraded service) is activated when a fault impairs the primary one. Analysis is conducted to guarantee that either the primary or the backup copy is able to complete before the deadline of the primary copy. That way, results are anyway produced in time.

2. System model

The goal of our analysis is to propose a methodology to configure a multiprocessor platform so that it is able to resist to faults, and in particular to soft errors. In what follows, the considered model is explained, taking into account the characteristics of faults, the software architecture and the hardware platform.

2.1. Fault model. We limit our attention to soft errors in multiprocessor systems. As explained earlier, soft errors are transient errors in memory or logic due to, e.g., alpha particles, neutrons or similar low energy particles. Due to the nature of soft errors, we consider each fault to be transient, which means that the faulty condition lasts for a limited and short time interval, after which traces of the fault remain only in possible wrong values. Moreover, since soft error rates statistically guarantee that time between two failures is sufficient to perform simple recovery operations, we assume that only one fault can affect the system at a time. This allows us to rely on the *single transient fault* assumption.

In a multicore environment all cores are integrated in the same chip. Hence one could think that a single faulty event could bring to simultaneous or correlated errors in different processors. Under the hypothesis that only soft errors influence the system, this problem does not show up, since a single particle can strike only one core. This means that even in a multicore environment the single transient fault assumption does make sense.

It is outside the scope of this work to discuss how a fault can be recovered. Informally, we can say that since the fault is transient, the recovery involves three steps:

(1) waiting for the end of the fault;
(2) correcting data errors due to tasks aborted in an inconsistent state or wrong results written by non protected tasks;
(3) restarting some of the non-protected tasks that were influenced by the fault.

Note also that if permanent faults are taken into account, the approach becomes much more difficult. The first step, in fact, must be to distinguish between permanent and transient faults; this is usually done through integrity routines, that deeply test the state of the system, or the part of the

system involved in the fault. Once the nature of the fault has been detected, the system needs either recovery (for transient faults) or reconfiguration (for permanent faults).

2.2. Operating modes. An environment prone to faults must be protected to avoid potentially catastrophic situations. It is clear, though, that not all the parts of the application have the same criticality, and different aspects can require different levels of fault-robustness. Consider an application which controls a car engine and shows its activity on a screen. While we could accept the visualization to be degraded, the control algorithm must produce the correct result despite the presence of faults. This idea is expressed by the concept of *operating mode* (or *mode*, for short). Intuitively, different parts of the application require to execute in distinct modes, every one characterized by a specific degree of tolerance to faults. In particular, under the single transient fault assumption, the following 3 modes can be required:

- in **fault-tolerant mode** (FT) the system is not damaged by the presence of a fault, i.e. wrong results can never be produced; additional resources are necessary in order to identify the fault and either correct it, or isolate it; in the second case, extra resources are also necessary to produce the correct result;
- in **fail-silent mode** (FS), in case of a fault, the system (or the faulty part of it) is made silent, in order to avoid errors and wrong results to propagate; resources must be dedicated to both identification and isolation of the fault;
- in **non-fault-tolerant mode** (NF) no fault-tolerance guarantee is given, i.e. the behavior of the system in the presence of a fault is unpredictable; since nothing is done to protect the system, the resources requirement is the minimum possible, that is all the resources can be dedicated to increase performance.

Note that this classification is strictly dependent on the hypothesis, and may change if, for example, we relax the single transient fault assumption. As a short example, if we suppose that up to 2 faults can affect the platform at the same time, there could be an operating mode in which tasks are guaranteed to tolerate the first fault, and become silent after the second.

2.3. Application model. The model we consider here is the sporadic task model with constrained deadlines introduced in Chapter 1. That is, an application consists of a set of N independent real-time tasks to be executed on a multiprocessor platform. A real-time task τ_i is characterized by worst-case computation time C_i, minimum interarrival time T_i and relative deadline D_i (such that $D_i \leq T_i$). U_i represents the utilization $\frac{C_i}{T_i}$ of task τ_i. For any given subset of tasks $\mathcal{T}$ we denote its utilization by $U(\mathcal{T})$ and we naturally set it equal to $\sum_{\tau_i \in \mathcal{T}} U_i$. Finally, it is important to point out that we consider all the tasks to be independent, i.e. they do not share data with critical sections.

Depending on its desired robustness to faults, every task requires to execute in one of the modes described in Section 2.2, and is defined to be a fault-tolerant (FT), fail-silent (FS) or non-fault-tolerant (NF) task. Based on

the required mode represented by parameter m_i, tasks are then partitioned into three sets: $\mathcal{T}_{\mathsf{FT}}$ for FT tasks, $\mathcal{T}_{\mathsf{FS}}$ for FS tasks, and $\mathcal{T}_{\mathsf{NF}}$ for NF tasks. In formulas

$$\begin{cases} \mathcal{T}_{\mathsf{FT}} = \{\tau_i | m_i = \mathsf{FT}\} \\ \mathcal{T}_{\mathsf{FS}} = \{\tau_i | m_i = \mathsf{FS}\} \\ \mathcal{T}_{\mathsf{NF}} = \{\tau_i | m_i = \mathsf{NF}\} \end{cases}$$

We assume that the task set is fixed and known before run-time, i.e. no task is dynamically created or deleted from the system.

Given an hardware platform able to provide the modes described above, the goal of this work is to schedule the application such that for every task no deadline is missed and the required mode is guaranteed.

2.4. Hardware architecture. The hardware architecture is based on the concept of *lock-step*. The generic lock-step configuration is composed of two identical processors and a checker. The two processors execute the same instructions cycle by cycle, so that in a fault free situation their local contextes remain synchronized. Meanwhile, the checker compares all the outputs from the processors: if the outputs are the same, it is assumed that there are no faults, so the access to the bus is granted, and memory is read or written; otherwise, the access is blocked and an error signal is raised. One possible implementation of such an architecture is represented in Figure 4.1. We have a master, CPU 1, which controls the bus activity, while CPU 2 acts as a slave: it receives instructions and data from the bus only as a consequence of the master's requests, executes them and sends its output to the checker. The checker can so compare data on the bus due to the master's activity, and data received from the slave. In the occasion of a difference, the checker can block the bus.

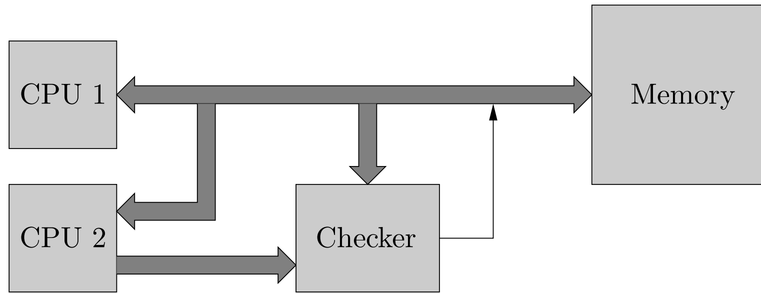

FIGURE 4.1. Lock-step architecture

The lock-step architecture guarantees that a fault in one processor is intercepted before it can propagate to the main memory (or to the input/output subsystem), hence the memory integrity is preserved. Of course, it is possible to force more than two processors to execute the same code, building the so called *redundant lock-step*, and obtaining a more robust channel. In such a channel, the checker can not only compare the outputs and reveal a fault, but it can, through voting, identify the correct value, and grant the access to the bus to the right processor(s). In practice, at the price of an

additional processor and a more complex checker, we obtain a fault-tolerant channel.

Exploiting the lock-step concept, and slightly modifying the Shared Memory Dual Lock-Step Architecture proposed in [**BFM+03**] by Baleani, Ferrari, Mangeruca, Sangiovanni-Vincentelli, Peri and Pezzini, we consider an hardware platform composed of 4 processors and a shared memory architecture, depicted in Figure 4.2.

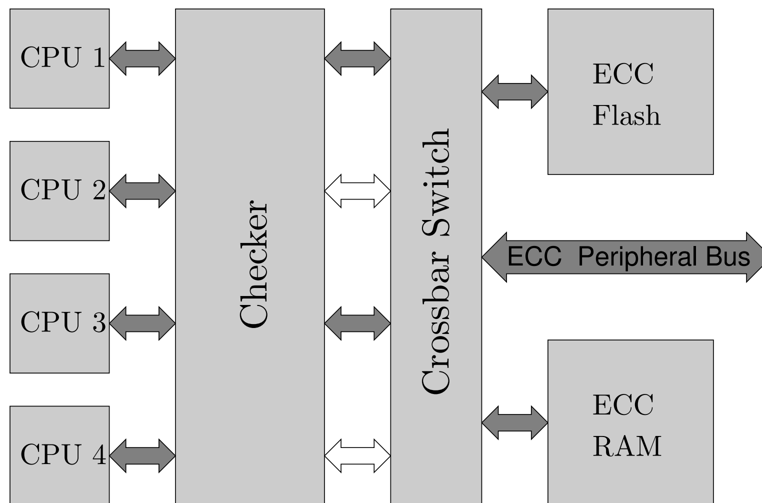

FIGURE 4.2. The hardware platform

The key element in the platform is the *checker*, which conceptually integrates three aspects: the comparison of the results provided by the processors, the control of the bus and memory access, and the reconfiguration of the platform.

The checker can change on-line its internal configuration, in order to provide to the application the three operating modes described in Section 2.2. At each instant, the 4 processors can work in one of the following modes:

- all 4 in one redundant lock-step channel, to provide the FT mode: due to the single fault assumption, only one processor can fail, so the correct result can be guessed by majority; the 4 processors build a single fault-tolerant channel;
- coupled in 2 lock-step channels, providing the FS mode: again only one processor in a couple can fail, so after a fault a mismatch between the two outputs is immediately revealed and the channel is blocked; the two couples realize two independent fail-silent channels;
- all 4 in parallel, implementing the NF mode: the 4 processors work independently, and no fault-tolerance guarantees are given, whereas the highest computational power is delivered.

From now on, we use *processor* and *channel* to identify both the physical entity and the virtual entity composed of one or more processors, and able to provide a certain degree of integrity. The meaning should always be clear from the context.

The modes are provided to the application by periodically switching from a mode to another, maintaining a sort of temporal separation between different modes. We call *mode switch* the on-line change of the system configuration between two modes. So, in every time interval, one of the modes is selected, and only tasks requiring that mode can execute. How the reconfiguration of the platform is actually performed is not investigated.

In practice, the time line is divided into intervals of length P, each one composed of three slots of length Q_{FT}, Q_{FS} and Q_{NF}, one for each mode respectively. Accordingly, the subset $\mathcal{T}_{\mathsf{FT}}$ will be executed during the first slot of length Q_{FT} (when a redundant lock-step channel is built), the subset $\mathcal{T}_{\mathsf{FS}}$ during the second slot of length Q_{FS} (on the two lock-step channels implemented), and the subset $\mathcal{T}_{\mathsf{NF}}$ during the last slot of length Q_{NF} (when 4 independent processors are present).

Another aspect we have to consider is that the mode switch requires some operations, such as task state synchronization and data storing. Hence O_{FT}, O_{FS} and O_{NF} represent respectively the overheads when switching out of modes FT, FS and NF. Moreover, we define the sum of the three overheads $O_{\mathrm{tot}} = O_{\mathsf{FT}} + O_{\mathsf{FS}} + O_{\mathsf{NF}}$. Note that the overhead consumes part of the time available to the respective subset, so the generic overhead O_m is included in the slot Q_m. This leads to the definition of $\tilde{Q}_m = Q_m - O_m$ as the amount of time available to the tasks in the generic mode m. The notation is depicted in Figure 4.3.

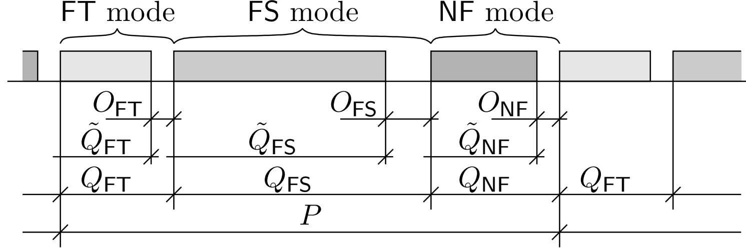

FIGURE 4.3. Switching between modes

However, it must be noted that whenever the platform switches from a mode to another, it is necessary to synchronize the processors. This is particularly expensive whenever we switch from a configuration with more channels and less fault-tolerance guarantees to a configuration with less channels and more fault-tolerance guarantees (i.e. when we switch from NF to any other mode or from any mode to FT). In fact it can happen that one channel finishes its activity later than the others, due for example to atomic operations, in which case it is required that the faster channel(s) wait for the slower one to be ready. In our analysis, this waiting time can be taken into account through the overheads. As said above, the whole synchronization and reconfiguration work is performed inside the checker.

We point out that in FT mode, 3 processors are sufficient to provide a fault-tolerant channel. Thus, it might be possible to use the 4th processor for running NF tasks. However, this additional degree of flexibility would complicate both the hardware management circuitry and the operating system. Thus, to simplify the platform management, we prefer to maintain the

modes separated in time. Please note that from this point of view, there is no difference between considering all the 4 processors together in a fault-tolerant channel, and using only 3 processors for the fault-tolerant channel and shutting down the 4th processor.

We underline that, while the platform under consideration has only 4 processors, the analysis below is easy to extend to any number of processors.

Fault protection. In order to provide any level of fault-robustness to the application, it is necessary to protect the whole platform from faults, including memory, bus, interrupt controllers and every other hardware component.

The redundancy of processors allows to detect and/or mask faults in standard commercial processors, without requiring any new, very expensive, processor design.

It must be noted, though, that if the fault happens on a processor currently executing independently (i.e. the platform is working in NF mode), the fault is surely not detected until the next mode switch. If we also consider that the fault is transient by assumption, it is possible that the system never reveals the fault. Therefore, an NF task executing on the faulty processor may generate incorrect results. We accept this, since it is consistent with the requirements described in Section 2.3.

The checker and the crossbar switch are special purpose circuits, so they can be protected via dedicated fault-tolerance techniques, for example self-checking or fault-free design. The same reasoning is valid for other smaller circuits, such as the interrupt controllers, which might be protected via replication or (due to the relatively low gate count of their logic) dedicated fault-tolerance techniques. Finally, memories and buses are protected using Error Correction Codes (ECCs) in order to retain error masking capabilities on these components when operating in lock-step mode.

3. Design methodology

The proposed architecture opens a new wide range of possibilities for exploiting the trade-off between fault-tolerance, implemented by hardware replication, and computational power, provided by parallel execution. When tuning the fault-tolerant platform, we assume to know in advance the set of tasks to be executed and their desired robustness to faults. The final design must guarantee that each task completes within the assigned deadline, and executes in the required mode (NF, FS or FT).

When the platform provides some degree of parallelism (i.e. during modes FS and NF) it is necessary to decide how to schedule the tasks on the multiprocessor. As described in Chapter 1, two main classes of algorithms are known in the literature: the *partitioning* [**LDG04, Bar04a**] and the *global* [**SA02**] strategies. In this chapter, we focus on the partitioned scheme (in which each task is statically assigned to one processor), whereas the analysis of global strategies is considered in Appendix A. Moreover, since the goal of this paper is the tuning of the platform once the characteristics of the application are known, we consider the tasks to be manually partitioned. In Section 5 we briefly analyze this problem.

During NF mode, four processors are available. Hence the tasks in $\mathcal{T}_{\mathsf{NF}}$ are partitioned into the 4 subsets $\mathcal{T}^1_{\mathsf{NF}}$, $\mathcal{T}^2_{\mathsf{NF}}$, $\mathcal{T}^3_{\mathsf{NF}}$ and $\mathcal{T}^4_{\mathsf{NF}}$, depending on the

processor they will run on. Similarly the tasks in $\mathcal{T}_{\mathsf{FS}}$ are partitioned into the 2 subsets $\mathcal{T}_{\mathsf{FS}}^1$ and $\mathcal{T}_{\mathsf{FS}}^2$. Finally, during FT mode only one channel is available hence no further partition is required.

Once tasks are partitioned, the schedulability problem leads to the well studied case of one single processor. However each subset of tasks can only run during the dedicated mode, which is allocated only a fraction of the total available time. A considerable amount of efforts have been recently dedicated to the study of such a problem. This scheduling problem is generally referred to as *hierarchical scheduling*. In the next section, we recall the most significant results which will be borrowed from the literature.

Before this, let us formally define the problem: given a set of real-time tasks $\tau_i = (C_i, T_i, D_i, m_i)$ defined as in Section 2.3, and a 4-processors hardware platform as described in Section 2.4, and assuming that the switching overhead for mode m is O_m, and a partitioning scheduling strategy is adopted, what are the parameters (P and Q_m for each mode m) of the operational modes that guarantee the schedulability of each task τ_i in the required mode m_i?

3.1. Hierarchical Scheduling. As shown in Figure 4.3, $\tilde{Q}_{\mathsf{FT}}$, $\tilde{Q}_{\mathsf{FS}}$, and $\tilde{Q}_{\mathsf{NF}}$ represent respectively the amount of time actually available to $\mathcal{T}_{\mathsf{FT}}$, $\mathcal{T}_{\mathsf{FS}}$, and $\mathcal{T}_{\mathsf{NF}}$ tasks in each period P.

We remark that the tolerance to faults of the system is higher when $\tilde{Q}_{\mathsf{FT}}$ dominates the other values. On the other hand, greater values of $\tilde{Q}_{\mathsf{NF}}$ maximize the delivered computational power, since four processors are available in NF mode.

The overall goal of this design methodology is to find the best balance between these two opposite needs. More formally, the admissible values of $(Q_{\mathsf{FT}}, Q_{\mathsf{FS}}, Q_{\mathsf{NF}})$ must be correctly tuned to guarantee that all the tasks τ_i complete within their respective deadlines D_i.

We can estimate the *computational power* provided during each mode by a supply function, which is defined as follows.

DEFINITION 3.1. Given a mode $m \in \{\mathsf{FT}, \mathsf{FS}, \mathsf{NF}\}$, the supply function $Z_m(t)$ of the mode m is the *minimum amount of time* provided during the mode m in any interval whose length is t. Formally,

$$Z_m(t) = \min_{t_0}\{\text{time provided in } [t_0, t_0 + t] \text{ during mode } m\}.$$

The introduction of the supply function is very useful for verifying the schedulability of real-time tasks, because it provides a minimum time guarantee which is granted under any circumstances. The idea of the supply function has been already exploited both in the field of real-time [**LB04, SL03, AP04**] and in networking [**LBT01**]. In Figure 4.4 we show the supply function for a generic mode m.

LEMMA 3.2 (Supply function, from [**LB04**]). *The supply function of a generic mode $m \in \{\mathsf{FT}, \mathsf{FS}, \mathsf{NF}\}$ is*

$$Z_m(t) = \begin{cases} j\,\tilde{Q}_m & \text{if } t \in [j\,P, (j+1)\;P - \tilde{Q}_m) \\ t - (j+1)\left(P - \tilde{Q}_m\right) & \text{otherwise} \end{cases} \qquad (3.1)$$

where $j = \left\lfloor \frac{t}{P} \right\rfloor$.

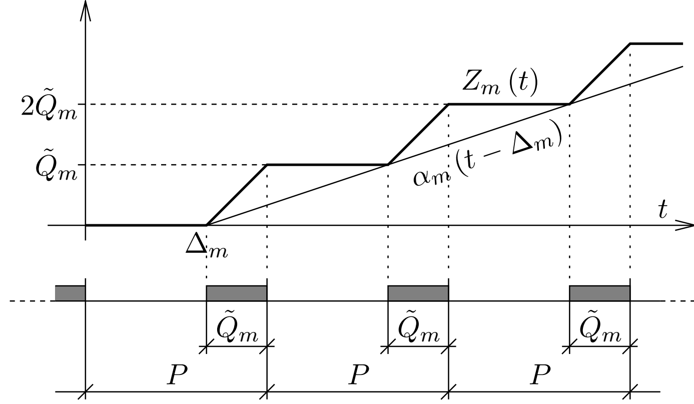

FIGURE 4.4. The supply function

From the supply function $Z_m(t)$ of each mode, we derive two important features: the *rate* α_m, which roughly denotes the fraction of processor available for mode m, and the *delay* Δ_m, which is the maximum amount of time a task executing during mode m could be forced to wait without being served. See [**LB04, FM02**] for a more formal definition of these two quantities.

It is possible to prove [**LB04**] that the relationship between (α_m, Δ_m) and the mode parameters is:

$$\alpha_m = \frac{\tilde{Q}_m}{P} \qquad \Delta_m = P - \tilde{Q}_m \tag{3.2}$$

The values of α_m and Δ_m are useful, since they introduce the following simple lower bound of the supply function

$$Z'_m(t) = \max(0, \alpha_m(t - \Delta_m)), \tag{3.3}$$

as can be noticed in Figure 4.4. Since we always have $Z'_m(t) \leq Z_m(t)$, assuming $Z'_m(t)$ as supply function is safe, meaning that every solution feasible with $Z'_m(t)$ is always feasible with $Z_m(t)$. However, for simplicity, we consider the supply function equal to $Z'_m(t)$, and from now on $Z_m(t)$ will denote the supply function of Equation (3.3). The full consideration of the exact $Z_m(t)$ does not present any conceptual difficulty, but it is only tedious to develop the math properly.

Through $Z_m(t)$, we have characterized the amount of computation that can be provided during the modes FT, FS and NF. Next we focus on the schedulability condition of the task subsets.

3.2. Schedulability analysis. The schedulability condition of sets of tasks running under a server characterized by some supply function has been already investigated in the literature. See for example [**FM02, SL03, SRLK02**]. In this work, we consider the opposite problem: given a task set, what are all the possible supply functions which guarantee the task deadlines?

In what follows, we propose solutions to the problem for two different scheduling algorithm: FP and EDF. However, we could extend the following analysis to every scheduling algorithm for which it is possible to express the schedulability condition on a server characterized by $Z_m(t)$.

FP scheduler. If tasks are scheduled by fixed priorities the feasibility condition of a task set $\mathcal{T}$ allocated to a mode m, characterized by a rate α_m and a delay Δ_m is provided by the following theorem.

THEOREM 3.3 (FP schedulability under $Z_m(t)$, Theorem 3 in [**LB04**]). *A task set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$ is schedulable by Fixed Priorities (FP) in the mode m, described by (α_m, Δ_m), if:*

$$\forall \tau_i \in \mathcal{T} \quad \exists t \in \mathsf{schedP}_i \quad \Delta_m \le t - \frac{W_i(t)}{\alpha_m} \tag{3.4}$$

where

$$W_i(t) = C_i + \sum_{\tau_j \in \mathsf{hp}(\mathcal{T}, \tau_i)} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{3.5}$$

and (i) schedP_i *is the set of* scheduling points *of task* τ_i *as defined in* [**BB04**] *and (ii)* $\mathsf{hp}(\mathcal{T}, \tau_i)$ *is the set of tasks in* $\mathcal{T}$ *with a higher priority than* τ_i*.*

The set of scheduling points schedP_i is defined as follows.

DEFINITION 3.4. Set schedP_i is defined as $\mathcal{P}_{i-1}(D_i)$, and the latter is recursively defined as:

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t) \end{cases}$$

where tasks are assumed to be ordered by decreasing priority.

schedP_i is the smallest set of points where Equation (3.4) must be checked for the task τ_i to be schedulable. That is, if we don't find a point in schedP_i for which Equation (3.4) is fulfilled, there is no possibility to find such a point elsewhere, so the task is not schedulable in mode m. Clearly, since we search for the existence of at least one value of t which fulfills the equation, whenever we find one such point t, there is no need to continue the research. A complete detailed explanation can be found in [**BB04**]. Here we propose only an example on how the set schedP_i is practically computed.

Suppose to compute schedP_4 for task τ_4 in Figure 4.5. We start from the first deadline of τ_4, d_4 and for each higher priority task (all the other tasks, in this case) we select the last release before d_4. That is, r_1^4, r_2^3 and r_3^3 join d_4 in the set. Then, for each of the new points we repeat the procedure. That is, due to r_3^3 we add to the set r_2^3 and r_1^3, and due to r_2^3 we add r_1^2 (although it was already present). No new point is added from r_1^4, since τ_1 is the highest priority task. Finally, repeating for r_2^3 we would add again r_1^3 to the set $\mathsf{schedP}_4 = \{d_4, r_3^3, r_2^3, r_1^3, r_1^4\}$.

Substituting in Equation (3.4) the expressions of α_m and Δ_m from Equation (3.2), we obtain

$$\bigwedge_{\tau_i \in \mathcal{T}} \bigvee_{t \in \mathsf{schedP}_i} P - \tilde{Q}_m \le t - \frac{W_i(t)}{\tilde{Q}_m} P$$

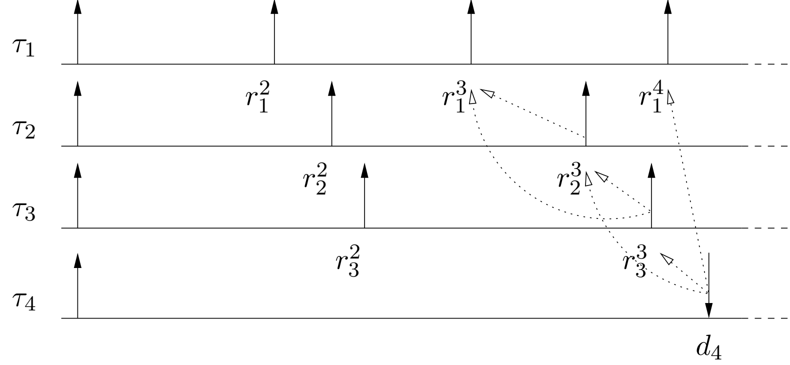

FIGURE 4.5. Example of computation of schedP_i

By performing a sequence of algebraic manipulations we find that

$$\bigwedge_{\tau_i \in \mathcal{T}} \bigvee_{t \in \mathsf{schedP}_i} P\,\tilde{Q}_m - \tilde{Q}_m^2 \le t\,\tilde{Q}_m - P\,W_i(t)$$

$$\bigwedge_{\tau_i \in \mathcal{T}} \bigvee_{t \in \mathsf{schedP}_i} \tilde{Q}_m^2 + (t-P)\,\tilde{Q}_m - P\,W_i(t) \ge 0$$

$$\bigwedge_{\tau_i \in \mathcal{T}} \bigvee_{t \in \mathsf{schedP}_i} \tilde{Q}_m \ge \frac{\sqrt{(t-P)^2 + 4\,P\,W_i(t)} - (t-P)}{2}$$

and then, by recalling the meaning of logic operators, we find the following explicit relationship between the lengths of the time slots $\tilde{Q}_m$ and the period P:

$$\tilde{Q}_m \ge \max_{\tau_i \in \mathcal{T}} \min_{t \in \mathsf{schedP}_i} \frac{\sqrt{(t-P)^2 + 4\,P\,W_i(t)} - (t-P)}{2} \tag{3.6}$$

Notice that the right hand side of Equation (3.6) depends on the period P, the parameters of the task set (through $W_i(t)$), and the adopted scheduling policy, which is fixed priorities. We can then compact Equation (3.6) in a very convenient way as follows

$$\tilde{Q}_m \ge \mathsf{minQ}(\mathcal{T}, \mathsf{FP}, P) \tag{3.7}$$

where all the complex dependencies of Equation (3.6) are hidden in the function minQ.

EDF scheduler. If tasks are scheduled by EDF then we can use the results of hierarchical scheduling when the local scheduler is EDF [**SL03, Bin04**]. If the supply function has a rate α_m and a delay Δ_m, then Theorem 3.5 ensures the feasibility of a task set under EDF.

THEOREM 3.5 (EDF schedulability under $Z_m(t)$, from [**Bin04**]). *A task set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$ is schedulable by Earliest Deadline First (EDF) in the mode m, described by (α_m, Δ_m), if:*

$$\forall t \in \mathsf{dlSet}(\mathcal{T}) \quad \Delta_m \le t - \frac{W(t)}{\alpha_m} \tag{3.8}$$

where

$$W(t) = \sum_{i=1}^{N} \max\left\{ \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor, 0 \right\} C_i \tag{3.9}$$

and $\mathsf{dlSet}(\mathcal{T})$ *is the set of all deadlines up to the minimum common multiple of all* T_i *of the tasks in* $\mathcal{T}$*.*

We underline that, on single processors, the classical approach to analyze sporadic systems is to consider their *periodic equivalent* (that is, a task set in which all the tasks release jobs as soon as possible, fulfilling the minimum interarrival requirement). In fact, on single processors it is easy to verify that this is the worst-case for the *demand bound function* expressed in Equation (3.9). Under this hypothesis, the schedule is periodic with period of length equal to the minimum common multiple of all T_i, which justifies the choice of $\mathsf{dlSet}(\mathcal{T})$ in the theorem. It is easy to verify that the proposed schedulability test is the so called *Processor Demand Criterion* [**BHR90**] applied to a fraction of the processor instead of the whole processor.

Notice that the previous formulation also applies to task sets with static offset and jitter. However, we develop all our results in the simpler case of no offset/jitter, because the full treatment does not present theoretical problem but the math is heavier (see [**Bin04**] for further details).

Using the same arguments which led to Equation (3.7), we can equivalently find the conditions on the time quantum $\tilde{Q}_m$ such that the mode m can feasibly schedule all the tasks allocated to it if they are scheduled by EDF. In fact we have

$$\tilde{Q}_m \geq \mathsf{minQ}(\mathcal{T}, \mathsf{EDF}, P) \tag{3.10}$$

where in this case

$$\mathsf{minQ}(\mathcal{T}, \mathsf{EDF}, P) = \max_{t \in \mathsf{dlSet}(\mathcal{T})} \frac{\sqrt{(t-P)^2 + 4\,P\,W(t)} - (t-P)}{2} \tag{3.11}$$

Finally note again that by the same arguments we could consider all the possible scheduling algorithms for which a schedulability condition can be expressed over a server described by a supply function $Z_m(t)$.

3.3. Integration between modes. Until now, we have only considered a generic task set $\mathcal{T}$ in isolation and the relationship between the time quantum Q_m of different slots is neglected. Now we want to combine the equations which describe the feasible time quanta to find all the feasible $(Q_{\mathsf{NF}}, Q_{\mathsf{FS}}, Q_{\mathsf{FT}})$. Let us generically denote by alg the scheduling algorithm adopted to schedule the tasks.

During FT mode, the quantum Q_{FT} must be large enough to schedule within the deadlines all the tasks in $\mathcal{T}_{\mathsf{FT}}$. Recalling that $\tilde{Q}_{\mathsf{FT}} = Q_{\mathsf{FT}} - O_{\mathsf{FT}}$, the condition on Q_{FT} is

$$Q_{\mathsf{FT}} - \mathsf{minQ}(\mathcal{T}_{\mathsf{FT}}, \mathsf{alg}, P) \geq O_{\mathsf{FT}} \tag{3.12}$$

During FS mode the platform is configured to offer computational resources equivalent to two processors (i.e., it has two channels). Hence the

time quantum Q_{FS} must accommodate the tasks of both the sets $\mathcal{T}^1_{\mathsf{FS}}$ and $\mathcal{T}^2_{\mathsf{FS}}$. The feasibility of both the sets is ensured by selecting Q_{FS} as follows

$$Q_{\mathsf{FS}} - \max_{i \in \{1,2\}} \mathsf{minQ}\left(\mathcal{T}^i_{\mathsf{FS}}, \mathsf{alg}, P\right) \geq O_{\mathsf{FS}} \tag{3.13}$$

By doing so, in fact, the allocated quantum satisfies the feasibility condition of both $\mathcal{T}^1_{\mathsf{FS}}$ and $\mathcal{T}^2_{\mathsf{FS}}$.

Finally, following the same previous arguments, the tasks allocated in NF mode do not miss any deadline if the following condition is verified

$$Q_{\mathsf{NF}} - \max_{i \in \{1,2,3,4\}} \mathsf{minQ}\left(\mathcal{T}^i_{\mathsf{NF}}, \mathsf{alg}, P\right) \geq O_{\mathsf{NF}} \tag{3.14}$$

All the three Equations (3.12), (3.13) and (3.14) define a lower bound to the admissible values for the time quanta as a function of the period P. A larger time quantum could be assigned to a mode, but this clearly influences the parameters of the other modes and/or the total period P. The relationship between the time quanta of different modes and the period can be expressed by summing side by side the three inequalities above. We obtain the following interesting condition on the period P:

$$P - \sum_{m \in \{\mathsf{FT},\mathsf{FS},\mathsf{NF}\}} \max_{i=1,\dots,\mathsf{numP}_m} \mathsf{minQ}\left(\mathcal{T}^i_m, \mathsf{alg}, P\right) \geq O_{\mathrm{tot}} \tag{3.15}$$

where numP_m is the number of available channels in mode m.

Note that a value of P fulfilling Equation (3.15) does not necessarily determine a feasible solution, unless the lengths of the time quanta satisfy also the three Equations (3.12), (3.13), and (3.14). However, this problem shows up only in case of a wrong selection of Q_{FT}, Q_{FS} and Q_{NF}, while it is guaranteed that once P is chosen to satisfy Equation (3.15), a solution for the values of the time quanta does exist. The 4 relationships must be considered as a set of instruments that support the designer during the selection of the parameters. The final choice depends on the specific goal the designer wants to achieve.

In order to clarify the application of the proposed technique, in the next section we show how two different design goals bring to two different solutions.

4. Example of Application

We consider an application composed of 13 real-time tasks requiring different operating modes. Table 1 reports the operating mode required by each task, the task index, the computation times, and the periods.

Mode	NF					FS				FT			
i	1	2	3	4	5	6	7	8	9	10	11	12	13
C_i	1	1	1	2	6	1	1	2	1	1	1	1	2
T_i	6	8	12	10	24	10	15	20	4	12	15	20	30

Table 1. The task set data

For simplicity, we assume the task set to have implicit deadlines although our proposed method well applies also to constrained deadlines system. We

are also considering the problem of extending the analysis to unconstrained deadlines systems.

NF tasks are partitioned to the four processors available in NF mode as $\mathcal{T}_{\mathsf{NF}}^1 = \{\tau_1\}$, $\mathcal{T}_{\mathsf{NF}}^2 = \{\tau_2, \tau_3\}$, $\mathcal{T}_{\mathsf{NF}}^3 = \{\tau_4\}$ and $\mathcal{T}_{\mathsf{NF}}^4 = \{\tau_5\}$. Similarly, FS tasks need to be partitioned in two groups, one for each fail-silent channel. The two subsets are $\mathcal{T}_{\mathsf{FS}}^1 = \{\tau_6, \tau_7, \tau_8\}$ and $\mathcal{T}_{\mathsf{FS}}^2 = \{\tau_9\}$. All FT tasks run on a unique fault-tolerant channel and are not partitioned.

Once the tasks are partitioned on the processors, a scheduling algorithm must be selected. In this example we explore both FP and EDF as scheduling algorithms. For the case of FP, we assign Rate Monotonic (RM) priorities, i.e. tasks with shorter period have higher priority.

We highlight that these choices can be also mixed together, in the sense that the scheduling algorithm for any of the subsets can be either EDF or FP with RM, or FP with some other priority assignment, or even some other scheduling algorithm, provided that the schedulability analysis in the framework of the hierarchical scheduling is developed also for this third algorithm. In fact, from the analysis of each task we obtain bounds on the acceptable values: mixing different algorithms requires only to fulfill all the bounds.

After the task partitioning and a suitable selection of the scheduling algorithms, Equation (3.15) describes all the feasible values of periods P. A Matlab routine to compute all the feasible periods resulting from Equation (3.15) has been developed by Enrico Bini, and can be found on the Web at `http://feanor.sssup.it/~bini/faultMP/`. Figure 4.6 shows the region of feasible periods for both EDF and FP. Notice that, as expected, the EDF region is larger than the FP one, because it is well known that every FP schedulable task set is also schedulable under EDF.

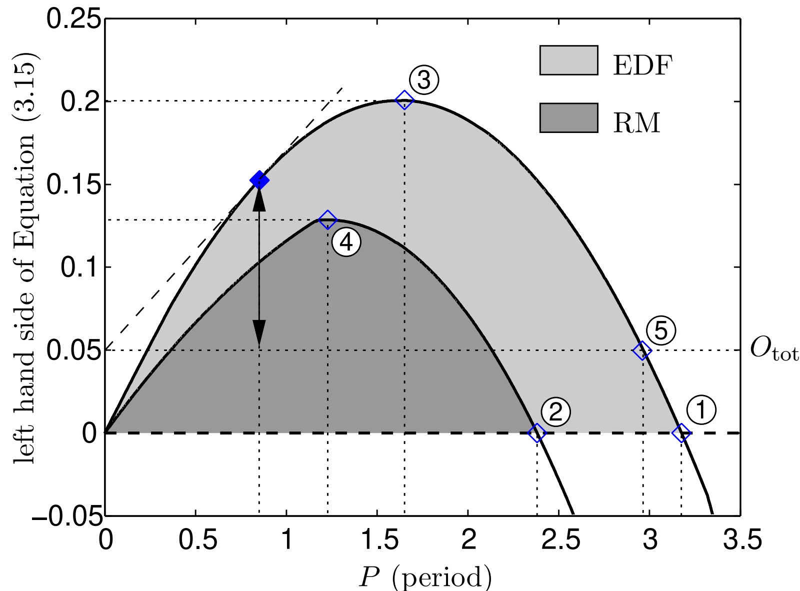

FIGURE 4.6. Determining the feasible periods

The feasible regions are above the zero, because $O_{\mathrm{tot}} \geq 0$ and hence, below the zero it wouldn't exist any feasible period P. If the overhead is 0 then the maximum feasible period P is 3.176 if using EDF and 2.381 if FP (points ① and ② in the figure). From the figure we can also find the maximum admissible total overhead to have a feasible solution which is 0.201 when using EDF and 0.129 when FP (points ③ and ④).

Let us suppose a realistic example where the total overhead is an intermediate value, e.g. 0.05 (also depicted in Figure 4.6). A first possible design goal may be to **minimize the bandwidth wasted in overhead** $\frac{O_{\mathrm{tot}}}{P}$. This goal is achieved selecting the maximum feasible period. If the adopted scheduler is EDF, the maximum period is 2.966 (point ⑤). In correspondence to this period we can now find the admissible values of $\tilde{Q}_{\mathsf{FT}}$, $\tilde{Q}_{\mathsf{FS}}$ and $\tilde{Q}_{\mathsf{NF}}$ from Equation (3.12), (3.13) and (3.14) respectively (see Table 2(b)).

$$\begin{cases} \tilde{Q}_{\mathsf{FT}} \geq 0.820 \\ \tilde{Q}_{\mathsf{FS}} \geq 1.281 \\ \tilde{Q}_{\mathsf{NF}} \geq 0.815 \\ \tilde{Q}_{\mathsf{FT}} + \tilde{Q}_{\mathsf{FS}} + \tilde{Q}_{\mathsf{NF}} = 2.966 - 0.05 = 2.916 \end{cases} \tag{4.1}$$

In order to make the reader confident of the correctness of these results we remind that a **necessary** condition for the schedulability of a task set $\mathcal{T}$ is that the allocated bandwidth is not smaller than the task set utilization $U(\mathcal{T})$ (Table 2(a)). Let us verify that this is true for all the subsets of tasks. For $\mathcal{T}_{\mathsf{FT}}$ we have that

$$\frac{\tilde{Q}_{\mathsf{FT}}}{P} = 0.276 \geq U(\mathcal{T}_{\mathsf{FT}}) = \frac{1}{12} + \frac{1}{15} + \frac{1}{20} + \frac{2}{30} = 0.267$$

so the bandwidth provided in $\mathcal{T}_{\mathsf{FT}}$ is sufficient to execute all the tasks requiring fault-tolerance capabilities. For $\mathcal{T}_{\mathsf{FS}}$ and $\mathcal{T}_{\mathsf{NF}}$, we have to consider that since the bandwidth provided must be sufficient to serve every subset of $\mathcal{T}_{\mathsf{FS}}$ and $\mathcal{T}_{\mathsf{NF}}$, we obtain the following inequalities:

$$\frac{\tilde{Q}_{\mathsf{FS}}}{P} = 0.432 \geq \max_{i=1,2} U\left(\mathcal{T}^{i}_{\mathsf{FS}}\right) = 0.267$$

$$\frac{\tilde{Q}_{\mathsf{NF}}}{P} = 0.275 \geq \max_{i=1,\ldots,4} U\left(\mathcal{T}^{i}_{\mathsf{NF}}\right) = 0.250$$

Since this design choice does minimize the bandwidth wasted in the mode switches, it provides the higher amount of unused bandwidth in each mode, because the time quanta are allocated at their maximum.

By following this design goal, Equations (3.12), (3.13) and (3.14) must hold with the equal sign and selecting any larger time quantum would bring to violations in one or more constraints. This phenomenon happens because we have selected a period value on the boundary of the feasible region, which means that there is only one acceptable selection of values for the time quanta. This solution, however, may present a critical aspect: if we allow tasks to dynamically arrive and leave the system, we can only take advantage of the unused bandwidth on each mode, but the length of the time quanta cannot be modified at all.

Instead there may be design scenarios where some tasks arrive dynamically and it would be very convenient to shrink or enlarge the time quanta. How do we proceed in this hypothesis? What does the goal of our design become?

Informally speaking, we would like to select the period such that the **time quanta can vary as much as possible at run-time**. Basically we would like to redistribute, if necessary, the most possible bandwidth among the modes. Let us explain graphically how to do it. In Figure 4.6, for every value of P, the value on the curve represents the minimum sum of the lengths of the slots, as obtained from Equations (3.12), (3.13) and (3.14). Instead, the point (P, O_{tot}) for the same value of P represents the maximum sum, as obtained from Equation (3.15). So, the vertical distance from any point (P, O_{tot}) to the boundary represented by the curve measures the slack amount in Equation (3.15), i.e. the acceptable variability in the slot lengths. The slack bandwidth is maximum when the ratio between the slack amount and the period P is maximum (indicated in Figure 4.6 by the maximum slope of the dashed line). The solution found in this case is reported in Table 2(c).

		P	O_{tot}	$\tilde{Q}_{\mathsf{FT}}$	$\tilde{Q}_{\mathsf{FS}}$	$\tilde{Q}_{\mathsf{NF}}$	slack
(a)	required utilization			0.267	0.267	0.250	
(b)	length	2.966	0.050	0.820	1.281	0.815	0.000
	allocated utilization	1.000	0.017	0.276	0.432	0.275	0.000
(c)	length	0.855	0.050	0.230	0.252	0.220	0.103
	allocated utilization	1.000	0.059	0.269	0.294	0.257	0.121

TABLE 2. Possible design solutions

Notice that 12.1% of the bandwidth can be redistributed dynamically, although in this case the allocated bandwidth for each slot is much tighter to the required amount. Note also that, as expectable, while the overhead remains constant, its bandwidth increases.

As a final remark, please notice that all the computations are developed in the case of an EDF scheduler. The same reasoning applies to the FP scheduling algorithm as well.

5. Partitioning tasks among processors

Whenever there is more than one channel at disposal (that is, in FS and NF modes), an important aspect to consider in system design is how tasks can be partitioned . As we noted in Section 3, this problem is very delicate, since it can strongly influence the overall result.

To notice the criticality of the partitioning, let us consider again the example of Section 4, and suppose that task τ_2 is moved to $\mathcal{T}^4_{\mathsf{NF}}$ and τ_8 and τ_9 are interchanged. The resulting partitioning is the following:

- $\mathcal{T}^1_{\mathsf{NF}} = \{\tau_1\}$, $\mathcal{T}^2_{\mathsf{NF}} = \{\tau_3\}$, $\mathcal{T}^3_{\mathsf{NF}} = \{\tau_4\}$, and $\mathcal{T}^4_{\mathsf{NF}} = \{\tau_2, \tau_5\}$, in NF;
- $\mathcal{T}^1_{\mathsf{FS}} = \{\tau_6, \tau_7, \tau_9\}$ and $\mathcal{T}^2_{\mathsf{FS}} = \{\tau_8\}$, in FS;
- $\mathcal{T}_{\mathsf{FT}} = \{\tau_{10}, \tau_{11}, \tau_{12}, \tau_{13}\}$, in FT.

It is clear that the rate α_m of every mode must be sufficient to serve all the subsets requiring that mode. That is, it must be that $\forall i, m\ \alpha_m \geq U\left(\mathcal{T}_m^i\right)$. However, it is easy to verify that under the partitioning proposed above, we should have $\alpha_{NF} > 0.375$, $\alpha_{FS} > 0.417$, and $\alpha_{FT} > 0.267$. The sum is greater than 1, so there is no choice of parameters which can guarantee the application under such a partitioning, and the reason is the wrong partitioning.

The manual solution is feasible only in simple cases (as our example was), since when the number of tasks to be partitioned raises it is very difficult to find a good solution. Here we propose and shortly discuss some of the solutions proposed so far. We underline that neither of them is optimal with respect to any of our specific goals. As we said in Chapter 1, the simple partitioning problem is already known to be NP-hard in the strong sense [**Pap81**], the mix of partitioning problem and mode parameters research is almost impracticable.

Classical approaches to the problem of partitioning are based on policies such as First Fit (FF, where a task is assigned to the first processor that can schedule it), Best Fit (BF, assignment to the processor where there is more space), Worst Fit (WF, assignment to the processor where there is less space after the assignment), or the associated *Ordered* versions, where tasks are ordered, usually by decreasing utilization, prior to searching for the right processor. An overall description and analysis of these techniques can be found in [**LDG04**].

Unfortunately, these approaches do not always fit our problem. In particular FF and WF tend to fill a processor before starting with a new one, which means that they maximize the utilization of some processor (requiring high rate for the modes) leaving some other almost free.

BF could be a suitable policy, especially if we consider the Decreasing version which tends to balance the utilization of all the involved processors. However, due to the delicateness of the problem, we believe it is better to consider a different technique, based on combinatorial optimization, proposed in [**Bar04a**]. The key idea of this technique, based on previous works by Potts [**Pot85**] and Lenstra, Shmoys and Tardos [**LST90**], is to represent the partitioning problem as the optimization problem reported below, where the goal is to minimize the maximum per-processor utilization.

$$\text{Minimize}\, \mathcal{U} \text{ subject to} \begin{cases} \sum_{j=1}^{M} x_i^j = 1 & \forall i = 1 \text{ to } N \\ \sum_{i=1}^{N} x_i^j U_i \leq \mathcal{U} & \forall j = 1 \text{ to } M \\ x_i^j \geq 0 \text{ is an integer} & \forall i = 1 \text{ to } N, j = 1 \text{ to } M \end{cases}$$

In the equations above, x_i^j represents the fact that task τ_i is assigned ($x_i^j = 1$) or not ($x_i^j = 0$) to processor j. Considering that x_i^j is a non-negative integer (constraint 3), the first equation guarantees that each task is assigned to 1 and only 1 processor, while the second equation computes the utilization of each processor. Using this technique, we tend to overthrow the constraint on the rate due to the task subsets utilization, and so we can guarantee the maximum freedom in the selection of the α_m parameters. A

complete description of this technique is out of the scope of this work, the interested reader can find it well-described in the original paper [**Bar04a**].

We want to underline again that our methodology is influenced by the partitioning technique only in the solutions it can find, while the steps to be followed are exactly the same for every possible partitioning technique.

6. Conclusions

In this chapter, we propose a flexible management scheme of an identical multiprocessor platform for scheduling periodic real-time tasks with integrity requirements. We are not aware of any previous attempt to dynamically reconfigure a hardware architecture in different operating modes (lock-step or parallel) to support at the same time integrity requirements by means of hardware replication, and parallel execution for high performance. Thanks to an hardware platform capable of exploiting a high parallelism as well as a high replication, we propose a methodology which improves these capabilities through flexibility, allowing to achieve the best trade-off between the two opposite possibilities. The proposed analysis allows to face and solve the configuration problem with different goals in mind, such as minimizing the overhead or maximizing on-line flexibility.

Our algorithm is a first step towards a complete methodology for designing real-time fault-tolerant systems based on a multiprocessor. As explained in Section 5, the *partitioning problem* needs to be approached as integral part of the whole system design, instead of a small problem solved in advance with known techniques. This would enable to search for the optimum in the whole design space instead of on the reduced space obtained after the allocation of tasks to processors. Also, a future direction of investigation will be to explore the possibility of providing different fault-tolerance services during the same time quantum per period, as well as the same fault-tolerance service during more than one time quantum per period, improving flexibility of the platform at the cost of an higher complexity in the analysis.

Other modeling and analysis problems needs to be addressed. The most urgent is the need to model *interacting* tasks, i.e. tasks that share resources through mutex semaphores and tasks interacting through remote procedure calls (RPC). We also need to better address the fault-recovery phase. We plan to combine our methodology with existing software techniques for fault-recovery (as checkpointing and primary-backup). Moreover, we are investigating on-line reconfiguration algorithms to recover as many tasks as it is possible after a fault.

Finally, in Appendix A we explore the possibility of using global scheduling algorithms in substitution of the partitioned one considered so far.

CHAPTER 5

Conclusions

In this thesis we coped with the problem of how to exploit the power of symmetric multiprocessor platforms (SMP) to improve the behavior of real-time systems. The analysis has been conducted in two different directions. From one side we considered how to improve the performances of the systems from the point of view of computational power provided. From the other side, we tackled the goal of providing to the applications some integrity guarantees.

1. Performance

In an attempt to guarantee higher performances to real-time application by using multiprocessors, we considered two different problems that require attention: schedulability and feasibility (more precisely, infeasibility).

Schedulability. We first analyzed the behavior of real-time systems when scheduled by three well-known global scheduling algorithms (EDF, FP and EDZL). After discussing previous results in the field, we proposed tests to verify the schedulability of a given task set when one of the three algorithms is used. Between the two tests proposed for each scheduling algorithm, the first has lower complexity but worse results, while the second test, at the price of an average complexity of approximately $O(N^3)$, is able to recognize more schedulable task sets than with previous tests.

We have validated our results through an extensive set of simulations, which took into account possible differences in the performances of the schedulability tests when several parameters of the system change. In particular, we verified how the tests perform when

- the number of processors changes from 2 to 8;
- the mean utilization of the single tasks changes from 0.25 to 0.75;
- the deadlines of the tasks are constrained or unconstrained.

Analyzing the simulation, we are convinced that the proposed schedulability tests are a step forward in the analysis of real-time systems on multiprocessors. However, it was also evident that in general our tests are not strictly better than previous tests. Although our tests have usually better results, the tests are in general incomparable: we found not only cases in which one task set was recognized by one test and not the others and vice-versa, but also cases in which the percentage of schedulable task sets was higher for one test than for the others and vice-versa. For this reason, apparently the best solution would be to mix the tests and use all of them to maximize the percentage of task set declared schedulable.

Feasibility. The second analysis relates to the opposite problem: the research for conditions that guarantee that a task set will eventually miss

some deadline, whatever scheduling algorithm we decide to use. Thanks to this analysis, it is possible to distinguish between task sets that cannot be correctly scheduled, and task sets for which it is possible to find a schedule such that all the deadlines are meet.

This kind of analysis is based on two motivations. From one side, such analysis helps in speeding up simulations, since it allows to discard infeasible task sets without even considering them. Moreover, it allows to verify the overall behavior of the schedulability tests evaluating them only on feasible task sets. From the other side, this analysis could help in the verification of applications: if we find that a given application is not correctly scheduled, through a feasibility test, we can (partly) discriminate between unfeasible task sets and poor-performing pairs *scheduling algorithm/schedulability test*. It remains open the problem of how to discriminate between a poor algorithm and a poor test. For this goal, the only solution seems to be to further improve the schedulability analysis.

In the field of feasibility analysis, in this thesis we discuss a new necessary feasibility condition which allows to recognize as infeasible strictly more task sets than with previous tests, narrowing the region of uncertainty. We have to underline that the test is only necessary, in the sense that a task set that passes the test *could* be feasible, but it is not guaranteed to be so.

2. Integrity

The second goal of this thesis was to consider how to exploit multiprocessors to provide some degree of integrity guarantees to real-time applications. The key idea was to use multiprocessors to provide space redundancy to the applications. In practice, we suppose to force more than one processor to execute the same code at the same time, in a lock-step configuration, and we compare their results, instantly revealing any single fault. That way at the price of a reduced computational power, we can provide different integrity guarantees, depending on the requirements of the application. The problem usually faced with such solutions is the excessive rigidity of the platform, that offers the same level of integrity to all its parts. We addressed the problem by supposing to be able to reconfigure the platform at predefined time instants, switching from a configuration to another in a periodic way, in order to provide in different time instants different trade-offs between fault-tolerance and performance.

We took into account a platform that, in different time slots, can be configured to be equivalent to

- 4 non-fault-tolerant processors;
- 2 fail-silent processors;
- 1 fault-tolerant processor.

Based on this reconfigurable platform, in the thesis we provided a methodology that allows to tune the platform on the specific application. The goal is to find the set of values for the lengths of the slots that guarantee that each task can meet all its deadlines and execute in the required slots (that is, with the required integrity guarantees). In particular, through our design technique, it is possible to find a mathematical description of the space of the solutions (the acceptable lengths of the slot), supposed that FP-partitioned

or EDF-partitioned are used to schedule the tasks. Moreover, using the described technique, it is possible to select the best values for different goals in mind, such as minimizing the overhead lost in reconfiguration, or maximizing the flexibility of the platform.

To the best of our knowledge, no previous work allowed to exploit reconfigurable multiprocessor platforms to provide at the same time real-time and fault-tolerance guarantees, in a trade-off between performance and integrity.

3. Future works

The work of this thesis provided answers to several open questions in multiprocessors research. However, it also opened the doors to an incredible amount of possible improvements, and problems to solve.

We improved the number of task sets recognized schedulable, and the number of task sets recognized infeasible. However, there is still a wide gap in between. One future step in research is surely to shrink (and hopefully fill) this gap. The most promising path, in this sense, is probably to take into account scheduling algorithms specifically developed for multiprocessors (instead of extensions of algorithms for single processor), such as EDZL or EDF-FP. However, despite the common idea that EDF and FP are poor scheduling algorithms for multiprocessor, it seems that they can indeed have quite good results, so another goal is to continue the improvement of their schedulability tests.

Progress in schedulability tests for global scheduling algorithms is a prerequisite to use them in reconfigurable multiprocessor platforms. We plan to consider this step in order to take advantage from the characteristics of global scheduling in such platforms. Appendix A considers this problem and suggests some aspects to investigate.

Last but not least goal is unification. Throughout the thesis we used different hypothesis on task sets and scheduling algorithms. For example, we supposed unconstrained deadlines to cope with schedulability analysis, whereas we accepted only constrained deadlines for integrity. Another example is the fact, already discussed, that we used in one case global algorithms and in the other case partitioned algorithms. The final result should be a unification and relaxation of all the assumptions, in order to give the maximum flexibility in all possible applications.

4. Conclusion

As we have shown, through multiprocessors it is possible to provide any necessary improvement in computational power provided to the requiring applications, but we can also offer different degrees of safety and protection from faults. Moreover, it is also possible to mix the two previous requirements, offering at the same time both real-time and fault-tolerance guarantees. However, several problems need yet to be addressed and solved, and several aspects require improvements to be really useful.

We believe that multiprocessors can really be the answer for the dramatically increasing request for integrity and performance. There is only one need: **research must go on**.

APPENDIX A

Integrity Problem: hints on the global approach

1. Overview

In Chapter 4 we described a methodology to exploit multiprocessors to provide integrity guarantees to the applications, taking into account the fact that different parts of the application can have different requirements. For this reason, we considered the application divided into three subsets, FT for tasks requiring fault-tolerance, FS for tasks requiring fail-silence, and NF for tasks without any particular integrity requirement. The hardware platform is then configured differently for each subset: as a single fault-tolerant channel for FT tasks, as two fail-silent channels for FS tasks, and as four non-fault-tolerant channels for NF tasks. That way the platform is able to provide, in every moment, the best trade off between performance and integrity.

One important limitation in the proposed methodology is the fact that whenever more than one channel is present (i.e. in FS and NF modes), we use a partitioned approach to schedule tasks on the processors (see Chapter 1). This approach has its best advantage in the fact that once the tasks are partitioned among the processors, the problem reduces to a series of single processor schedulability problems, for which well-known solutions exist. However, there are also several drawbacks, overcame by the other possible approach: *global scheduling*.

Among the drawbacks of partitioned algorithms with respect to global ones, we specially underline the fact that while the former requires to take into account system balance, the latter automatically balances the load among the processors. This is particularly useful for applications in which tasks can join and leave the system at runtime. This characteristic assumes an even more significant importance in the reconfigurable platforms we consider. Since the parameters that guide the reconfiguration, i.e. P and Q_m for each mode (see Chapter 4), are tuned on the application at a certain time instant, whenever some task joins or leaves the system, a retuning could be necessary. In such a case, the automatic load balance offered by global scheduling algorithms improves the flexibility of the system, simplifying this aspect and possibly avoiding the need for platform reconfiguration.

Another aspect relates to the moment in which one fault is revealed. The work described in our thesis is based on the *single transient fault* assumption described in Chapter 4: whenever a fault appears, we suppose to know that it is only a temporary problem, which will eventually be solved without the need for our intervention. In the real world, however we cannot completely rely on this assumption, so after a fault is detected, some inquiry procedure should be activated, to verify the nature of the fault. In the

meanwhile, for safety reasons, we should assume that the channel involved in the fault is broken and lost, and so reconfigure the load on the remaining processors. This procedure is clearly much easier under the presence of an automatic load balance system such as the one provided by global scheduling algorithms.

The model we took into account in the previous chapter is simple. It is based on only 3 modes, and the system periodically switches from one mode to the other. In such a case, each task can execute only in a well-defined time interval, which depends on the operational mode required by the task and the parameters of the system, so the partitioning phase is affordable, and very good heuristics exist to solve the problem. If we suppose to have a more complex system model, for example because there are more than 4 processors, the number and characteristics of the modes can be different and possibly more diversified. In this situation, partitioning the tasks among the processors can be much more difficult. Global scheduling could, again, simplify the system configuration.

For all these reasons, it is of primary importance to extend our platform reconfiguration technique to the case of global scheduling algorithms, in order to allow future extensions to different typologies of faults, task set models, and hardware configurations. This research is long and difficult, most of all because of the delay in global scheduling research. In this appendix, we consider some of the problems we have to face before proposing a complete methodology (as done in Chapter 4), and we give some hints on possible solutions to these problems.

We underline that this appendix represents part of our work-in-progress in the field of multiprocessors for integrity. As a consequence, we are forced not to deepen details, and remain at a higher level of description. Goal of this appendix is more to raise questions than to offer complete answers.

1.1. System Model. In this appendix, we take into account the same model considered in Chapter 4. We briefly list here the main characteristics.

- fault model: single transient fault assumption;
- operating modes: fault-tolerant mode FT, fail-silent mode FS, non-fault-tolerant mode NF;
- application model: task set $\mathcal{T}$ comprised of N tasks τ_i, each one characterized by computation time C_i, constrained deadline D_i, period T_i and required integrity mode m_i, and composed of potentially infinite jobs τ_k^j;
- hardware architecture: 4-processors platform, reconfigurable in 3 different configurations to provide 4 NF processors, 2 FS processors or 1 FT processor.

We call *slot* the interval between two reconfigurations in which a certain mode is provided to the application. Moreover, we define *pattern of the slots* the infinite series of periodic reconfigurations from one mode to another, performed by the hardware platform. Note that, due to the periodicity, the pattern of the slots is easily represented by little information: the base sequence of the slots, the length of each slot, and the starting slot.

We also redefine the concept of supply function expressed in Definition 3.1 of Chapter 4. In this appendix, we prefer to use three different definitions to name the minimum, maximum and actual computational power provided in a given interval. The definition is based on the knowledge of the pattern of the slots, introduced above.

DEFINITION 1.1 (Supply Function). Given a mode $m \in \{\mathsf{FT}, \mathsf{FS}, \mathsf{NF}\}$, the supply function $Z_m(t, t_0)$ of the mode m is the *amount of time* provided during the mode m in the interval of length t starting at time s. Formally,

$$Z_m(t, t_0) = \{\text{time provided in } [t_0, t_0 + t] \text{ during mode } m\}.$$

From this the definitions of minimum supply function $minZ_m(t)$ and maximum supply function $maxZ_m(t)$ follow:

$$minZ_m(t) = \min_{t_0}\{Z_m(t, t_0)\}.$$

$$maxZ_m(t) = \max_{t_0}\{Z_m(t, t_0)\}.$$

Note that what in Chapter 4 was called supply function, here becomes the minimum supply function. Note also that, by definition, in every time instant t

$$minZ_m(t) \le Z_m(t, t_0) \le maxZ_m(t).$$

We define the remaining computation time $c_i^j(t)$ of a job τ_i^j as the amount of computation that the job has still to execute at time t. This quantity clearly depends on the schedule.

Finally we define the laxity $l_k^j(t)$ of a job τ_i^j as the difference between time to deadline and remaining computation time. The laxity represents the time that a job can waste not executing, and still be able to meet its deadline. Usually, the laxity of τ_k^j is defined

$$l_k^j(t) = \left(d_k^j - t\right) - c_k^j(t). \tag{1.1}$$

For any other missing concept that we use in this appendix we refer to the rest of the thesis, were these concepts have been thoroughly used. Moreover, we underline again the fact that here we want only to discuss problems and possible solutions: we prefer to sacrifice some mathematical precision and severity for the benefit of clearness.

2. Global scheduling algorithms for fault-tolerance

The first step in the analysis is to understand the behavior of a global scheduling algorithm in the particular situation we consider, that is when only a fraction of the processor is dedicated to a task subset. As in Chapter 2, we consider three global scheduling algorithms: EDF, FP and EDZL.

For the case of EDF and FP, there is no need to modify the scheduling algorithms due to the fact that only a fraction of processor is used. Both of the algorithms, in each time instant, schedule for execution the tasks with higher priority, and the priority is assigned as usual (absolute deadline for EDF, or static assignment for FP). The only difference is the fact that, since each task requires a specific fault-tolerance mode m_i, the algorithms can chose tasks only in the subset of tasks requiring the mode under execution.

While there is no need to modify the rules of EDF of FP to use them on a fault-tolerant multiprocessor, EDZL requires some further discussion.

The key idea which allows EDZL to strictly dominate over EDF, and have its good behavior, is the fact that when a job reaches a *critical instant*, its priority is raised to the maximum, in order to be able to complete before its deadline. The *critical instant* for τ_k^j is an instant such that if τ_k^j does not start executing immediately, it will eventually miss its deadline: the instant $\hat{t}$ such that the remaining execution time of the job, $c_k^j\left(\hat{t}\right)$ is equal to the maximum computation time the multiprocessor can provide to τ_k^j before its deadline d_k^j.

Suppose to use EDZL in a dedicated multiprocessor system. Since one of the processors can be completely dedicated to the execution of τ_k^j, the computation time provided to the job in any interval starting at t is exactly the length of the interval, $d_k^j - t$. For this reason the critical instant $\hat{t}$ is such that $d_k^j - t = c_k\left(\hat{t}\right)$, which is equivalent to say that the laxity $l_k^j\left(t\right)$ of the job is 0. From this the name usually given to the Critical Instant Rule of *Zero Laxity Rule*.

However, in a reconfigurable multiprocessor platform such as the ones we consider, this is not a sufficient condition. In fact, the system is forced to switch from one configuration to another in a periodic way, so that it is not true anymore that one of the processors can be completely dedicated to the execution of one specific job. The processor can be dedicated to one specific job only when the system is configured to execute in the mode m required by the task of the job under consideration. This forces to redefine the Critical Instant Rule or, more precisely, to modify the definition of laxity. In what follows, we propose two different approaches.

2.1. Critical Instant Rule: supply function. Whenever a job τ_k^j with $m_k = m$ is released, it is easy to compute the jitter between its deadline d_k^j and the periodic pattern of the slots. In particular, it is possible to compute the interval between d_k^j and the next reconfiguration of the platform to mode m. Based on that, it is also possible to compute, in every time instant t, the supply function $Z_m\left(t, r_k^j\right)$, representing the actual computational power provided to mode m before d_k^j.

One possibility is so to consider, as Critical Instant, the instant $\hat{t}$ in which the remaining execution time $c_k^j\left(\hat{t}\right)$ is equal to the value of the supply function $Z_m\left(d_k^j - \hat{t}, \hat{t}\right)$ computed at the same time $\hat{t}$. This is equivalent to redefine the laxity as

$$l_k^j\left(t\right) = Z_m\left(d_k^j - t, t\right) - c_k^j\left(t\right), \tag{2.1}$$

while the Critical Instant Rule remains the same: the priority of a job is raised whenever the laxity of the job reaches 0.

Note that this definition remains coherent to the original definition of laxity given in Equation (1.1) for dedicated multiprocessors, since clearly in such platforms $Z_m\left(d_k^j - t, t\right) = d_k^j - t$.

This approach as the advantage that the rule is exact, as it was in the original formulation. In fact it remains true that a job τ_k^j reaching the Critical Instant at $\hat{t}$ can meet its deadline if and only if it executes in every time instant in $[\hat{t}, d_k^j)$ in which the platform is configured in mode m. The problem with this approach is that, since $Z_m(t, s)$ depends on the release time of the job, the rule is actually different for each job of each task, and should be computed for each job at the moment of its release.

2.2. Critical Instant Rule: minimum supply function. It would be interesting to find a definition of the Critical Instant Rule which does not change for every job of a task. For this goal, we could base the definition on the minimum supply function $minZ_m(t)$, instead of the supply function $Z_m(t, s)$. The use of $minZ_m(t)$ allows to unify the definition of critical instant for each job of a task, since the value of $minZ_m(t)$ does not depend on the phase between release time of a job and mode switch of the platform. Note that the minimum supply function does not even depend on the specific task, but only on the mode m we consider. So, for two different jobs executing in the same mode, the only difference is the value of the remaining execution time $c_k^j(t)$.

Again, we maintain the same Critical Instant Rule, while the definition of critical instant changes due to a different definition of laxity. In particular, the laxity becomes

$$l_k^j\left(\hat{t}\right) = minZ_m(t) - c_k^j(t) \tag{2.2}$$

ant the critical instant is a time instant $\hat{t}$ such that $l_k^j\left(\hat{t}\right) = 0$.

Remember that in Chapter 4 we considered two different definitions of the minimum supply function in the parameters P and Q_m of the mode m.

We first gave an exact definition:

$$minZ_m(t) = \begin{cases} j\,\tilde{Q}_m & \text{if } t \in [j\,P, (j+1)\;P - \tilde{Q}_m) \\ t - (j+1)\left(P - \tilde{Q}_m\right) & \text{otherwise} \end{cases} \tag{2.3}$$

where $j = \left\lfloor \frac{t}{P} \right\rfloor$.

Then we proposed the following simplified version:

$$minZ'_m(t) = \max\left(0, \frac{\tilde{Q}_m}{P}\left(t - \left(P - \tilde{Q}_m\right)\right)\right), \tag{2.4}$$

We underline that the Critical Instant Rule remains valid for both definitions of the minimum supply function. We also underline that in both cases, while we gain in consistency (among different jobs and tasks) we lose in precision. In fact, since the minimum supply function $minZ_m(t)$ represents only the minimum amount of time provided to a job, the actual time provided to the job, represented by the supply function $Z_m(t, s)$ can be more than the minimum. This means that in general the job could start executing later than the critical instant and still meet its deadline. This fact will be considered in the next section to verify the correctness of the rule.

2.3. Correctness of the Critical Instant Rule. Another problem must be considered when we want to use EDZL to schedule tasks on a flexible multiprocessor platform. While in a dedicated multiprocessor system we are sure to be able to raise the priority whenever the Critical Instant Rule requires so, this could be not true on a flexible multiprocessor platform. The problem lies in the fact that the critical instant could be reached while a different mode is running. In order to show that the rule is anyway correct we have to verify that one of the following is true:

- the critical instant $\hat{t}$ cannot be reached for τ_k^j, unless the system is running in mode m_k;
- although τ_k^j can reach the critical instant $\hat{t}$ even if the system is not running in mode m_k, it is safe to raise the priority of the job later, and in particular when the platform is reconfigured to mode m_k.

We can show that this is true for both possible definitions of the Critical Instant Rule and the laxity.

$Z_m(t, t_0)$ **approach.** Consider a job τ_k^j with $m_k = m$, and assume that the scheduling is based on the Critical Instant Rule given in Section 2.1. The laxity, defined in Equation (2.1), depends on two elements: the remaining computation time, and the actual computation time provided, represented by the supply function $Z_m\left(d_k^j - t, t\right)$. When mode m is not running, the job under analysis cannot execute, so $c_k^j(t)$ is constant. However, since mode m is not running, $Z_m\left(d_k^j - t, t\right)$ does not change, since no computational power is provided to mode m. As a consequence, the laxity remains constant, and the critical instant $\hat{t}$ cannot be reached. So, the Critical Instant Rule can apply only during mode m, when the priority can be correctly raised, guaranteeing for a correct behavior.

$minZ_m(t)$ **approach.** Suppose, instead, to take advantage of the definition of laxity given in Equation (2.2), which involves the minimum supply function $minZ_m(t)$ at the place of the supply function. As in the previous case, $c_k^j(t)$ cannot decrease when m is not running, so it remains constant. However, this could be not true for $minZ_m(t)$. In fact the value of the minimum supply function depends on the position of the mode switches under the worst-case alignment between the pattern of the slots and the release of τ_k^j. As a consequence, the value of $minZ_m(t)$ depends not on the actual pattern of the slots but on the worst-case, and so can decrease even if mode m is not running. This leads to the fact that the critical instant $\hat{t}$ could be reached while mode m is not running, forcing the rule to be delayed up to the start time of next slot of mode m (from now on $t^* \geq \hat{t}$).

Despite this fact, the Critical Instant Rule can be delayed up to the next slot of mode m, without endangering the timeliness of the system. In fact consider the relation between the values of $Z_m(t, s)$ and $minZ_m(t)$. By their definitions, $minZ_m(t) \leq Z_m(t, s)$. This relation is true at time $\hat{t}$, when the Critical Instant Rule should be applied. This means that, if one takes into account the time actually provided in mode m (instead of the minimum one represented by $minZ_m(t)$), the job does not need to start immediately to execute, in order to meet its deadline. Moreover, since when mode m is

not running $Z_m(t,s)$ remains constant, it comes out that at time t^*, when a new slot of mode m starts

$$minZ_m(t^*) \leq Z_m\left(t^*, d_k^j - t^*\right) = Z_m\left(\hat{t}, d_k^j - \hat{t}\right).$$

This means that, even if the Critical Instant Rule applies at time $\hat{t} \leq t^*$, it is safe to start executing τ_k^j with maximum priority only at time t^*. In fact,

$$c_k^j\left(\hat{t}\right) = minZ_m\left(\hat{t}\right) \leq Z_m\left(t^*, d_k^j - t^*\right),$$

and so the time actually provided in mode m before d_k^j is sufficient for the task to meet its deadline.

3. Schedulability analysis

The next step in using global scheduling algorithms of Chapter 2 on flexible multiprocessor platforms such as the ones described in Chapter 4, is to study schedulability conditions under which task sets are guaranteed to meet all their deadlines. At this stage, we suppose the platform to be statically tuned, and we only want to verify if the task set is correctly schedulable on such a tuned platform. The following step should be to guide the tuning as we did in Chapter 4.

The analysis must be necessarily based on some schedulability test for dedicated multiprocessors, such as the ones we proposed in Chapter 2. Unfortunately those tests, while improving the past situation in schedulability analysis, remain quite far from the necessary and sufficient condition. As a consequence, their use inevitably introduces some pessimism in the analysis, and forces a loss in our platform tuning. We prefer to focalize on the non-recursive version of the tests, due to the fact that at this stage the improvement obtained through the recursive step is counterbalanced by an extreme intricacy in the analysis. We prefer to avoid it, here, considering that the goal of this appendix is not to give a full, working, procedure, but only to provide some ideas on the path to follow, and underline some problems we have to face.

Basing on the schedulability tests of Chapter 2, Section 6, we want to:

- compute the idle time that a problem job τ_k^j has in its overload window, considering the pattern of the slots under which it executes;
- compute the upper bound on interference that jobs of tasks other than τ_k can produce in the overload window of τ_k^j, again taking into account the pattern of the slots imposed by the tuning;
- verify if the interference is sufficient to force a deadline miss for the problem job.

It is necessary to verify what is the worst-case situation for the release times of the jobs. That is we need to understand what is the worst-case alignment among release time of the problem job, release time of all the other jobs, and pattern of the slots.

3.1. Worst-case release times. This 3-elements alignment is in general quite complex to study. For the case of EDF, the situation is made easier by an observation: whatever the positions of the slots are, a task τ_i has maximum upper bound on the interference I_k^i provoked on τ_k^j when d_k^j is aligned with the deadline of one job of τ_i. This can be verified by the usual release shifting analysis conducted in Chapter 2. Shifting forward all the releases leads to the loss of a complete job near the end of the interval without a sufficient increase near the start time of the interval, while shifting backward cannot increase near the end of the interval but could decrease near the start time. The proof is completed by the observation that the described behavior does not depend on the pattern of the slots (although the amount of the increases and decreases clearly does).

The main consequence of this fact is that in the case of EDF the research for the worst-case alignment does not involve anymore three elements, but only two: the releases of *all* the jobs and the pattern of the slots.

Unfortunately, the above reasoning does not directly map on FP or EDZL. The problem lies in the fact that while the worst-case for EDF depends only on the deadlines, which are not influenced by the pattern of the slots, the worst-case for either FP of EDZL is defined taking into account also the intervals in which the jobs actually execute, which depends on the pattern of the slot.

The situation for EDZL is even clearer if we consider to adopt the Critical Instant Rule based on the real laxity (Section 2.1), since in such a case the priority of the tasks directly depends on the position of the slots. It means that in the research for the worst-case alignment, all the three elements must be considered together.

For this reason, here we focalize only on EDF, for which an easier solution exists, while we leave for the future a detailed analysis of the cases of FP and EDZL.

3.2. Worst-case slot positioning. Once the worst-case alignment between problem job and interfering tasks is established, we need to study the relation between such an alignment and the pattern of the slots.

This study must be guided by two opposite needs: from one side minimizing the computational time provided to the problem job τ_k^j in the overload window; from the other side maximize those provided to the interfering jobs, that way maximizing the upper bound on their interference.

Taking the lead from these two needs, the solution we propose here is to consider the two problems separately. With this approach, the solution is quite simple:

- the minimum computational power provided to τ_k^j in any interval of length t is given by the value of the minimum supply function $minZ_m(t)$;
- the maximum computational power provided to tasks interfering with τ_k^j in any interval of length t is bounded by the value of the maximum supply function $maxZ_m(t)$.

3.3. Sketch of schedulability test. Using the results above, we can propose here a sketch of a possible test to verify the schedulability of a

task set under EDF, when only a fraction of an M-processor can be used. We underline that what we report below is only a possible direction of investigation, and not yet a clear result to be used in practice.

The first bound above helps us in re-analyzing Lemma 5.3 of Chapter 2, which gives a necessary condition for a job to reach negative laxity. Informally, in a reconfigurable platform as the one we consider, it is true that τ_k^j can reach negative laxity only if it suffers interference for at least $minZ_m(\Lambda_k) - C_k + 1$ in the overload window.

For what relates the second bound above, it can be used to limit the interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$ provoked by task τ_i to the problem job in its overload window.

We can mix the above results. We obtain that, when EDF is used to schedule a task set on an M-processor, a job of τ_k can miss its deadline only if

$$\sum_{i \neq k} \min\left(\beta_k^i(\Lambda_k), maxZ_m(\Lambda_k) - C_k + 1\right) \geq M\left(minZ_m(\Lambda_k) - C_k + 1\right) \tag{3.1}$$

In this formula, modified from Equation (6.1) (Theorem 6.1 in Chapter 2), $\beta_k^i(\Lambda_k)$ is an upper bound on the interference $I_k^i(d_k^j - \Lambda_k, d_k^j)$, given by

$$\beta_k^i(\Lambda_k) = N_i C_i + \min\left(C_i, \Lambda_k - N_i T_i\right),$$

and N_i is the number of jobs completely executed in the overload window, and is computed as

$$N_i = \left\lfloor \frac{\Lambda_k}{T_i} \right\rfloor.$$

From this, a schedulability test follows by simply verifying that the above condition is false for any task τ_i in the task set.

4. Comments and future research

The proposed test is quite similar to the test proposed for EDF when an M-processor is at full disposal of the application. However, it is only a starting point, that needs improvements in several aspects.

It is evident that the assumptions on the position of the slots are extremely pessimistic, since we assume separately the worst-case for the right and the left side of Equation (3.1). That is, we consider the situation that maximizes the sum on the left side, and the situation that minimizes the value on the right side. However, these two cases are not likely to happen together. As a consequence, an important direction to study is the research for the alignment of the pattern of the slots that actually minimizes the difference between the two sides of the equation. During this research, there is another aspect that requires to be taken into account. A job of τ_i included in the body of $\beta_k^i(\Lambda_k)$ is guaranteed to be able to execute completely in the overload window, despite the position of the pattern of the slots. This is done in the schedulability analysis of τ_i (that is, when we consider a job of τ_i to be the problem job). Instead, if we consider the carry-in, this is not true anymore. In fact it could happen that, in the whole interval between start time of the overload window and deadline of the carried-in job, the system never run in mode m. In such a situation, the carry-in should be considered

equal to 0, which at the moment is not done. Since considering this aspect allows to decrease the interference estimation of job τ_i, it can offer another way to improve the test and the analysis.

The improvements proposed above are only minor ones. The real goal in our future research is much more ambitious: extend the analysis to the best tests for all the three, and possibly other, global algorithms, and use the analysis not only to test the schedulability, but also to guide the tuning of the platform.

Bibliography

[AB04] Neil C. Audsley and Konstantinos Bletsas, *Fixed priority timing analysis of real-time systems with limited parallelism*, Proceedings of the 16th Euromicro Conference on Real-Time Systems, ECRTS 2004 (Catania, Italy), July 2004, pp. 231–238.

[ABJ01] Björn Andersson, Sanjoy K. Baruah, and Jan Jonsson, *Static-priority scheduling on multiprocessors*, Proceedings of the 22nd IEEE Real-Time Systems Symposium, RTSS 2001 (London, UK), December 2001, pp. 193–202.

[AJ00] Björn Andersson and Jan Jonsson, *Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition*, Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications, RTCSA 2000 (Cheju Island, South Korea), December 2000, pp. 337–346.

[Alt] Altera, *Stratix III Device Family*, Web page: `http://www.altera.com/products/devices/stratix3/st3-index.jsp`.

[AP04] Luís Almeida and Paulo Pedreiras, *Scheduling within temporal partitions: response-time analysis and server design*, Proceedings of the 4th ACM International Conference on Embedded software, EMSOFT 2004 (Pisa, Italy), September 2004, pp. 95–103.

[ARM] ARM, *ARM11 MPCore*, Web page: `http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html`.

[AS99] James H. Anderson and Anand Srinivasan, *A new look at pfair priorities*, 1999, available at `http://www.cs.unc.edu/~anderson/papers/pfair.ps`.

[AS04a] Karsten Albers and Frank Slomka, *An event stream driven approximation for the analysis of real-time systems*, Proceedings of the 16th Euromicro Conference on Real-Time Systems, ECRTS 2004 (Catania, Sicily), July 2004, pp. 187–195.

[AS04b] James H. Anderson and Anand Srinivasan, *Mixed Pfair/ERfair scheduling of asynchronous periodic tasks*, Journal of Computer and System Sciences **68** (2004), no. 1, 157–204.

[AT07] Björn Andersson and Eduardo Tovar, *Competitive analysis of partitioned scheduling on uniform multiprocessors*, Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium, IPDPS 2007 (Long Beach, CA, USA), March 2007, p. 147.

[Bak91] Theodore P. Baker, *Stack-based scheduling of real-time processes*, Real-Time Systems **3** (1991), no. 1, 67–99.

[Bak03] ———, *Multiprocessor EDF and deadline monotonic schedulability analysis*, Proceedings of the 24th IEEE Real-Time Systems Symposium, RTSS 2003 (Cancun, Mexico), December 2003, pp. 120–129.

[Bak05] ———, *An analysis of EDF schedulability on a multiprocessor*, IEEE Transactions on Parallel and Distributed Systems **16** (2005), no. 8, 760–768.

[Bak06a] ———, *An analysis of fixed-priority schedulability on a multiprocessor*, Real Time Systems **32** (2006), no. 1–2, 49–71.

[Bak06b] ———, *A comparison of global and partitioned EDF schedulability tests for multiprocessors*, Proceedings of the 14th International Conference on Real-Time and Network Systems, RTNS 2006 (Poitiers, France), May 2006, pp. 119–127.

[Bar01] Sanjoy K. Baruah, *Scheduling periodic tasks on uniform multiprocessors*, Information Processing Letters **80** (2001), no. 2, 97–104.

[Bar04a] ———, *Partitioning real-time tasks among heterogeneous multiprocessors*, Proceedings of the 33rd International Conference on Parallel Processing, ICPP 2004 (Montréal, Québec, Canada), August 2004, pp. 467–474.

[Bar04b] Sanjoy K.r Baruah, *Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms*, Proceedings of the 25th IEEE Real-Time Systems Symposium, RTSS 2004 (Lisbon, Portugal), December 2004, pp. 37–46.

[Bau01] Robert C. Baumann, *Soft errors in advanced semiconductor devices - part 1: The three radiation sources*, IEEE Transaction on Device and Materials Reliability **1** (2001), no. 1, 17–22.

[BB04] Enrico Bini and Giorgio C. Buttazzo, *Schedulability analysis of periodic fixed priority systems*, IEEE Transactions on Computers **53** (2004), no. 11, 1462–1473.

[BB06] Sanjoy K. Baruah and Alan Burns, *Sustainable scheduling analysis*, Proceedings of the 27th IEEE Real-Time Systems Symposium, RTSS 2006 (Rio de Janeiro, Brasil), December 2006, pp. 159–168.

[BC06a] Theodore P. Baker and Michele Cirinei, *A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks*, Proceedings of the 27th IEEE Real-Time Systems Symposium, RTSS 2006 (Rio de Janeiro, Brasil), December 2006, pp. 178–187.

[BC06b] ———, *A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks*, Tech. Report TR-060501, Florida State University Department of Computer Science, Tallahassee, FL, USA, May 2006, available at `http://www.cs.fsu.edu/research/reports`.

[BC06c] ———, *A unified analysis of global EDF and fixed-task-priority schedulability of sporadic task systems on multiprocessors*, Tech. Report TR-060401, Florida State University Department of Computer Science, Tallahassee, FL, USA, November 2006, available at `http://www.cs.fsu.edu/research/reports`. Accepted for a special issue of the Journal of Embedded Computing. To appear.

[BCL05a] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari, *Improved schedulability analysis of EDF on multiprocessor platforms*, Proceedings of the 17th Euromicro Conference on Real-Time Systems, ECRTS 2005 (Palma de Mallorca, Spain), July 2005, pp. 209–218.

[BCL05b] ———, *New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors*, Proceedings of the 9th International Conference on Principles of Distributed Systems, OPODIS 2005 (Pisa, Italy), December 2005, pp. 306–321.

[BCPV96] Sanjoy K. Baruah, Neil K. Cohen, Greg Plaxton, and Donald A. Varvel, *Proportionate progress: a notion of fairness in resource allocation*, Algorithmica **15** (1996), no. 6, 600–625.

[Ber88] Philip A. Bernstein, *Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing*, Computers **21** (1988), no. 2, 37–45.

[BF05] Sanjoy K. Baruah and Nathan Fisher, *The partitioned multiprocessor scheduling of sporadic task systems*, Proceedings of the 26th IEEE Real-Time Systems Symposium, RTSS 2005 (Miami, FL, USA), December 2005, pp. 321–329.

[BFB05] Theodore P. Baker, Nathan Fisher, and Sanjoy K. Baruah, *Algorithms for determining the load of a sporadic task system*, Tech. Report TR-051201, Florida State University Department of Computer Science, Tallahassee, FL, USA, December 2005, available at `http://www.cs.fsu.edu/research/reports`.

[BFM+03] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini, *Fault-tolerant platforms for automotive safety-critical applications*, Proceedings of the 6th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2003 (San José, CA, USA), 2003, pp. 170–177.

[BHR90] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier, *Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor*, Real-Time Systems **2** (1990), no. 4, 301–324.

[Bin04] Enrico Bini, *The design domain of real-time systems*, Ph.D. thesis, Scuola Superiore Sant'Anna, Pisa, Italy, October 2004, available at `http://feanor.sssup.it/~bini/thesis/`.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier, *Preemptively scheduling hard-real-time sporadic tasks on one processor*, December 1990, pp. 182–190.

[BW01] Alan Burns and Andy Wellings, *Real-time systems and programming languages. Ada 95, Real-Time Java and real-time POSIX*, 3rd ed., Addison Wesley Longmain, March 2001.

[CB98] Marco Caccamo and Giorgio C. Buttazzo, *Optimal scheduling for fault-tolerant and firm real-time systems*, Proceedings of the 5th International Workshop on Real-Time Computing Systems and Applications, RTCSA 1998 (Hiroshima, Japan), October 1998, pp. 223–231.

[CB07] Michele Cirinei and Theodore P. Baker, *EDZL schedulability analysis*, Proceedings of the 19th Euromicro Conference on Real-Time Systems, ECRTS 2007 (Pisa, Italy), July 2007, To appear.

[CBLF07] Michele Cirinei, Enrico Bini, Giuseppe Lipari, and Alberto Ferrari, *A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors*, Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium, IPDPS 2007 (14th WPDRTS) (Long Beach, CA, USA), March 2007, p. 149.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James H. Anderson, and Sanjoy K. Baruah, *Handbook of scheduling: Algorithms, models, and performance analysis*, ch. 31: A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms, CRC Press, 2004.

[CLAL02] Seongje Cho, Suk-Kyoon Lee, Sang Ahn, and Kwei-Jay Lin, *Efficient real-time scheduling algorithms for multiprocessor systems*, IEICE Transactions on Communications **E85-B** (2002), no. 12, 2859–2867.

[DL78] Sudarshan K. Dhall and Chung L. Liu, *On a real-time scheduling problem*, Operations Research **26** (1978), no. 1, 127–140.

[Els] Dale Elson, *PowerPC Processors Tips: Two PowerPC 750GXs are better than one*, Web page: `http://www.ibm.com/developerworks/power/library/pa-nl18-tip.html`.

[FBB06] Nathan Fisher, Theodore P. Baker, and Sanjoy K. Baruah, *Algorithms for determining the demand-based load of a sporadic task system*, Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2006 (Sydney, Australia), December 2006, pp. 135–146.

[FGB01] Shelby Funk, Joël Goossens, and Sanjoy K. Baruah, *On-line scheduling on uniform multiprocessors*, Proceedings of the 22nd IEEE Real-Time Systems Symposium, RTSS 2001 (London, UK), December 2001, pp. 183–192.

[FM02] Xiang Feng and Aloysius K. Mok, *A model of hierarchical real-time virtual resources*, Proceedings of the 23rd IEEE Real-Time Systems Symposium, RTSS 2002 (Austin, TX, USA), December 2002, pp. 26–35.

[Fur00] Johan F. Furunäs, *Benchmarking of a real-time system that utilises a booster*, Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000 (Las Vegas, NV, USA), vol. 4, June 2000.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy K. Baruah, *Priority-driven scheduling of periodic task systems on multiprocessors*, Real Time Systems **25** (2003), no. 2–3, 187–205.

[GH98] Donald Gross and Carl M. Harris, *Fundamentals of queueing theory*, 3rd ed., Wiley and Sons, January 1998.

[GLDN01] Paolo Gai, Giuseppe Lipari, and Marco Di Natale, *Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip*, Proceedings of the 22nd IEEE Real-Time Systems Symposium, RTSS 2001 (London, UK), December 2001, pp. 73–83.

[Ha95] Rhan Ha, *Validating timing constraints in multiprocessor and distributed systems*, Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, USA, 1995, available as Technical Report UIUCDCS-R-95-1907.

[HL94] Rhan Ha and Jane W. S. Liu, *Validating timing constraints in multiprocessor and distributed real-time systems*, Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, ICDCS 1994 (Poznan, Poland), June 1994, pp. 162–171.

[Ins] Texas Instruments, *OMAP*, Web page: `http://www.omap.com`.

[JM74] H. H. Johnson and M. S. Maddison, *Deadline scheduling for a real-time multiprocessor*, Proceedings of the European Computer Conference, EUROCOMP 1974 (London, UK), 1974, pp. 139–153.

[KSM03] Pramote Kuacharoen, Mohamed Shalan, and Vincent J. Mooney, *A configurable hardware scheduler for real-time systems*, Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2003 (Las Vegas, NV, USA), June 2003, pp. 95–101.

[LB04] Giuseppe Lipari and Enrico Bini, *A methodology for designing hierarchical scheduling systems*, Journal of Embedded Computing **1** (2004), no. 2, 257–269.

[LBT01] Jean-Yves Le Boudec and Patrick Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*, Lecture Notes in Computer Science, LNCS, vol. 2050, Springer-Verlag, 2001.

[LDG04] José M. López, José L. Díaz, and Daniel F. García, *Utilization bounds for EDF scheduling on real-time multiprocessor systems*, Real-Time Systems **28** (2004), no. 1, 39–68.

[LGDG03] José M. López, Manuel García, José L. Díaz, and Daniel F. García, *Utilization bounds for multiprocessor rate-monotonic scheduling*, Real-Time Systems **24** (2003), no. 1, 5–28.

[Lim03] George M. de Araùjo Lima, *Fault tolerance in fixed-priority hard real-time distributed systems*, Ph.D. thesis, University of York, York, UK, May 2003, available at `http://www.cs.york.ac.uk/ftpdir/reports/index.php`.

[Liu69] Chung L. Liu, *Scheduling algorithms for multiprocessors in a hard real-time environment*, JPL Space Programs Summary **37-60** (1969), no. 2, 28–37.

[LSF] Lennart Lindh, Johan Starner, and Johan F. Furunäs, *From single to multiprocessor real-time kernels in hardware*, Proceedings of the 1st Real-Time Technology and Applications Symposium, RTAS 1995 (Washington, DC, USA).

[LST90] Jan K. Lenstra, David B. Shmoys, and Éva Tardos, *Approximation algorithms for scheduling unrelated parallel machines*, Mathematical Programming: Series A and B **46** (1990), 259–271.

[MMG94] Daniel Mossé, Rami G. Melhem, and Sunondo Ghosh, *Analysis of a fault-tolerant multiprocessor scheduling algorithm*, Proceedings of the 24th International Symposium on Fault-Tolerant Computing, FTCS 1994 (Austin, TX, USA), June 1994, pp. 16–25.

[MP05] Aloysius K. Mok and Wing-Chi Poon, *Non-preemptive robustness under reduced system load*, Proceedings of the 26th IEEE Real-Time Systems Symposium, RTSS 2005 (Miami, FL, USA), December 2005, pp. 200–209.

[NG] Harn-Hua NG, *PPC405 Lockstep system on ML310 - Application Note*, available at `http://www.xilinx.com/bvdocs/appnotes/xapp564.pdf`.

[NSA] *HP NonStop Advanced Architecture - A business white paper*, available at `http://h71028.www7.hp.com/ERC/downloads/NSAABusinessWP.pdf`.

[NXP] Philips NXP, *Nexperia*, Web page: `http://www.nxp.com`.

[Pap81] Christos H. Papadimitriou, *On the complexity of integer programming*, Journal of the ACM **28** (1981), no. 4, 765–768.

[PHK+05] Minkyu Park, Sangchul Han, Heeheon Kim, Seongje Cho, and Yookun Cho, *Comparison of deadline-based scheduling algorithms for periodic real-time tasks on multiprocessor*, IEICE Transactions on Information and Systems **E88-D** (2005), no. 3, 658–661.

[PHK+06] Xuefeng Piao, Sangchul Han, Heeheon Kim, Minkyu Park, Yookun Cho, and Seongje Cho, *Predictability of Earliest Deadline Zero Laxity algorithm for multiprocessor real-time systems*, Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC 2006 (Gjeongju, Korea), April 2006, pp. 359–364.

[Pot85] Chris N. Potts, *Analysis of a linear programming heuristic for scheduling unrelated parallel machines*, Discrete Applied Mathematics **10** (1985), no. 2, 155–164.

[PPC] *PowerPC750 Lockstep Facility - Application Note*, available at http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/292763D22B80DFEF872570C1006DF928/$file/750GX_Lockstep11-17-05.pdf.

[PSTW97] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein, *Optimal time-critical scheduling via resource augmentation (extended abstract)*, Proceedings of the 29th annual ACM Symposium on Theory of computing, STOC 1997 (El Paso, TX, USA), May 1997, pp. 140–149.

[Pun97] Sasikumar Punnekkat, *Schedulability analysis for fault tolerant real-time systems*, Ph.D. thesis, University of York, York, UK, June 1997, available at http://www.cs.york.ac.uk/ftpdir/reports/index.php.

[RSL88] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky, *Real-time synchronization protocols for multiprocessors*, Proceedings of the 9th IEEE Real-Time Systems Symposium, RTSS 1988 (Huntsville, AL, USA), December 1988, pp. 259–269.

[SA02] Anand Srinivasan and James H. Anderson, *Optimal rate-based scheduling on multiprocessors*, Proceedings of the 34th ACM Symposium on Theory of Computing, STOC 2002 (Montréal, Québec, Canada), May 2002, pp. 189–198.

[SB02] Anand Srinivasan and Sanjoy K. Baruah, *Deadline-based scheduling of periodic task systems on multiprocessors*, Information Processing Letters **84** (2002), no. 2, 93–98.

[SKK+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi, *Modeling the effect of technology trends on the soft error rate of combinational logic*, Proceedings of the 32nd International Conference on Dependable Systems and Networks, DSN 2002 (Bethesda, MD, USA), June 2002, pp. 389–398.

[SL03] Insik Shin and Insup Lee, *Periodic resource model for compositional real-time guarantees*, Proceedings of the 24th IEEE Real-Time Systems Symposium, RTSS 2003 (Cancun, Mexico), December 2003, pp. 2–13.

[Son] Toshiba Sony, IBM, *Cell Processor*, Web page: http://cell.scei.co.jp/.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky, *Priority inheritance protocols: an approach to real-time synchronization*, IEEE Transactions on Computers **39** (1990), no. 9, 1175–1185.

[SRLK02] Saowanee Saewong, Ragunathan Rajkumar, John P. Lehoczky, and Mark H. Klein, *Analysis of hierarchical fixed-priority scheduling*, Proceedings of the 14th Euromicro Conference on Real-Time Systems, ECRTS 2002 (Wien, Austria), June 2002, pp. 152–160.

[STM] STMicroelectronics, *Nomadik*, Web page: http://www.st.com/stonline/products/families/mobile/processors/processorsprod.htm.

[WA02] Michael Ward and Neil C. Audsley, *Hardware implementation of programming languages for real-time*, Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2002 (San José, CA, USA), September 2002, pp. 276–285.