Real Time Operating System design for Multiprocessor system-on-a-chip

Tesi per il conseguimento del diploma di perfezionamento

Anno Accademico 2004/2005

Paolo Gai (pj@sssup.it)

Collegio dei docenti

Prof. Paolo Ancilotti

Prof. Giorgio C. Buttazzo

Prof. Marco Di Natale

Ing. Giuseppe Lipari

RETIS Lab,

Scuola Superiore S. Anna,

Piazza Martiri della Libertà

56100 Pisa - Italy

to Giorgio and Assunta.

Acknowledgements

This thesis describes some research I have done during this PhD, that is a little thing compared to the many things I did in the same period of time (included many silly photos and dinners and ...). I always thought that this page of my thesis would be probably the most difficult to write (and in fact it has been the last one I wrote!), because many are the friends that I should thank, little the space I have in a single page, and still little my knowledge of the English language to cite them in the appropriate way. In any case, let me try:

First of all, *Beatrice*, that really supported me during these years.

Then, my supervisors *Giuseppe, Marco, Paolo*, *Giorgio and Luigi*: I really would like to thank them, for their friendship and for their advice. *Alberto*, that has been the light for many inspirations, tolerant in many occasions and of course a good fiend. *Lui*, that convinced me to start this journey, and *Sanjoy*, who always surprised me for his elegance in research and life.

Then, *Tomas*, good friend from the icy Sweden, *Peter*, *Gerhard*, *Radu*, *Damir*, *Paulo*, *Luis*, *Lucia*, *Giancarlo*, *Roberto*, *Rodrigo*, *Anton*, *Shelby*, *Alan*, *Guillem*, *Ian*, *Julio*, *Mario*, *Michael*, *Tullio*, *Nino*, *Clara*, *John*, *Joan, Bjorn, Jakob*, and many other friends I met jumping around universities and conferences.

After that, people I worked with since a while in Magneti Marelli, *Walter*, *Giovanni*, *Paolo*, *Claudio*, *Enrico*, *Giacomo*, and others, who gave me the opportunity to work on the Janus architecture.

And, last but not least, all the ReTiS Lab and the PSV (in random order): *Paolo* (together again after a long time), *Nicola*, *Francesco*, *Alessandro*, *Marko*, *Giacomo*, *Shiva*, *Davide*, *Igor*, *Luca*, *Gerardo*, *Tonino*, *Claudio*, *Marco*, *Gianluca*, *Bruno*, *Sachin*, *Simonetta*, *Mara*, *Michael*, *Rodolfo* (that highlighted some bugs in my proofs), *Enrico*, *Gabriele*, *Chiara*, *Antonio*, *Elisabetta*, *Fabrizio*, *Barbara*, *Giampiero*, *Mario*, *Michele*, *Bruno*, and of course all the other people that I should have cited but I just forgot...

PJ

Summary

The primary goal for real-time kernel software for single and multiple-processor on a chip is to support the design of timely and cost-effective systems. The kernel must provide time guarantees, in order to predict the timely behavior of the application, an extremely fast response time, in order not to waste computing power other than the application cycles and save as much RAM as possible in order to reduce the overall cost of the chip.

The research on real-time software systems has produced algorithms that allow to effectively schedule system resources while guaranteeing the deadlines of the application and to group tasks in a very short number of non-preemptive sets which require much less RAM memory for stack. Unfortunately, up to now, the research focus has been on time guarantees rather than the optimization of memory usage. Furthermore, these techniques do not apply to multiprocessor architectures which are likely to be widely used in future microcontrollers.

This thesis presents innovative scheduling and optimization algorithms, which solve the problem of guaranteeing schedulability with an extremely short operating system overhead and minimizing RAM usage.

I developed a fast and simple algorithm for sharing resources in homogeneous multiprocessor systems, together with an innovative procedure for assigning a preemption threshold to tasks. When used in conjunction with a preemption threshold assignment algorithm, the algorithm further reduces the RAM usage in multiprocessor systems.

Finally, I discuss the problem of multiprocessor scheduling for asymmetric architectures composed by a general purpose CPU and a DSP. The challenging issue addressed in this part is to verify whether the use of a dedicated processor can effectively enhance the performance of an embedded system still maintaining some kind of real-time guarantee. In particular, I provide a method for increasing the schedulability bound both for fixed and dynamic scheduling, allowing a more efficient use of the computational resources.

Yes, as usual this document has been made using LaTeX and LyX.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Digital Revolution that is happening in the last twenty years is slowly changing the way products are designed and used. We live in a world of increasingly intrusive information technology, requiring informations to be always available in the right places, asking more and more features to the final products.

Tight requirements and new features are now common also in mechanical systems, where different objectives (ecological reasons for pollution reduction, integration of different subsystems, higher requirements for control performance) constantly adds complexity to the embedded controllers used to control the physical systems.

All these electronic systems needs to have some kind of relationship with the environment they are working in, because in some way they have to communicate informations (e.g., the temperature of a room, the weight of the people that are on a lift) or to directly control some mechanical parts (e.g., they have to set the spark timing in a car engine). This interaction is related to giving the result of the computation *in time*, that intuitively means the results have to be somehow synchronized with the response times of the external environment. For example, if you press the brake pedal of a car while driving, you would like to reduce the speed of the car *as soon as possible*; if you watch a movie using a DVD player, you probably want to *see* the movie and to ear the sound in a *synchronized* way.

All these implicit requirements are translated at design time in temporal constraints. For example, in the case of the brake pedal, the time elapsed between the press of the pedal by the the driver and the start of the brake action on the wheel should be in the order of the response time of the human brain that is driving the car, that is, probably, a few milliseconds (ms). That time sets an end-to-end requirement, that have to be met by properly choosing the mechanics and the electronics composing the braking system. In the example of the movie, once the sound of the movie started the DVD player has to display one frame every 40 ms, to give the idea to the human brain that is watching the TV set that he is seeing a movie and not a set of still frames.

All these computing environments that have to interact with the outside environment are called *real-time* systems. There are different kind of real-time systems, depending on the criticality of the synchronization that have to be reached with the environment. It is out of the scope of this thesis to make a comprehensive enumeration of real-time systems;

however, an important distinction between the so-called *hard* real-time systems (where the violation of a time requirement may be catastrophic) and other kind of systems, often called *soft* real-time-systems (where the consequence of a bad timing is just a performance degradation) is needed.

Furthermore, it is worth noting that real-time features only become a problem when the available resources are limited. For example, in a DVD player, cost and thermical reasons may force the use of less powerful computing architectures that are cheap and do not require a fan. These architectures may need an adequate resource scheduling policy to make efficient use of the (little) computing power available.

Real-time systems are often *concurrent* systems. The notion of concurrency (that means the ability of doing many things at the same time) is again linked to the lack of physical resources. For example, a microwave oven have to perform different concurrent tasks while heating up your hamburger: it has to produce the microwaves that cooks your food, rotate the dish to perform a proper uniform cooking, countdown and update the LCD with the remaining cooking time, and check if the user opens the oven's door, to eventually stop producing the microwaves.

Such a small embedded device like a microwave oven is probably controlled by a single CPU embedded behind the buttons and the LCD display of the man-machine interface. The software running on that CPU has to *schedule* properly different activities, and make them behave as they each run on a dedicated CPU.

The software that runs on the CPU is often divided in different layers. A common distinction is made between the *application* (the cooking algorithms, that are specific for each microwave oven) and the *firmware* (that is the infrastructure on which the application runs, that typically can be reused to design other embedded systems, such as coffee machines).

Most of the real-time systems academic research deals with the basic question of "what is the best way to design the firmware of a real-time system?". Many answers have been found for different kind of systems, related to different design parameters like:

- the number of entities (CPUs) available in the system;
- the kind of *scheduling algorithm* that is chosen for running the different *tasks* of the application;
- the kind of *synchronization mechanisms* of the different activities;
- the kind of *cooperation* required to accomplish a task (are these tasks all independent or do they need to share information?);
- the kind of *communication* between different tasks in the system (we consider only shared memory architectures or also distributed systems connected through a network?);
- the amount of resources available (can advanced features be implemented or should we save as much resources as possible to have a cheaper final product).

This thesis addresses some solutions for the design of a firmware for real time systems that takes in account the design parameters just listed.

In particular, the thesis proposes some techniques for the design of the firmware of embedded system-on-a-chip (SoC) that in general are composed by more than one CPU, and that have to be designed taking in account the amount of available resources (in my case, the RAM space used for stacks).

1.1 Why multiprocessors?

The first question which needs an answer is why there is a real need for multiprocessor SoC. If we analyze the trend for the integration of future applications in the embedded market, and especially in the automotive market [28], it is clear that a standard uniprocessor microcontroller architecture will not be able to support the needed computing power even taking into account the IC technology advances.

To increase computational power in embedded real-time systems there are then two possible ways:

- to increase the processor speed;
- to increase the parallelism of the architecture.

The first option requires the use of caching, deep pipelining or other advanced architectures. This solution suffers from serious drawbacks in the context of real-time embedded systems: caching makes very hard or impossible to determine the worst case execution times of programs; deep pipelining is not effective because of the large number of stalls caused by reactions to asynchronous events. Also, parallelism at the instruction level (VLIW architectures) requires large silicon areas and drastically increases code size.

Therefore, the best option and the future of many embedded applications seems to rely on the adoption of multiple-processor-on-a-chip architectures.

This idea, that was a seminal idea only four years ago, now has reached some kind of maturity; in fact, many multiprocessor SoC are now available on the marketplace.

The Janus microcontroller (see the scheme of Figure 1.1), developed by PARADES, ST Microelectronics and Magneti Marelli in the context of the MADESS[45] project, is an example of a dual-processor platform for power-train applications. Two 32-bit ARM7TDMI processors connected by a crossbar switch to 4 memory banks and two peripheral buses for I/O processing (low and high bandwidth) provide twofold computational power, compared to a single (ARM7TDMI) processor architecture, at very low increment of the silicon area, i.e. at comparable system costs. Both CPUs share the same address space. The main memory is organized in different modules and types: SRAM and FLASH. In architectures with multiple processors, memory access is the most important bottleneck of the system. Almost any communication flow is between the memory and other system components. To allow a correct synchronization and communication among tasks allocated to different processors, the architecture provides hardware support for inter-processor communication by interrupt inter-processor mechanisms and for shared memory by atomic test-and-set.

Although Janus implements a symmetric multiprocessor, there are other promising symmetric architectures composed by a RISC processor (or a microcontroller) and one

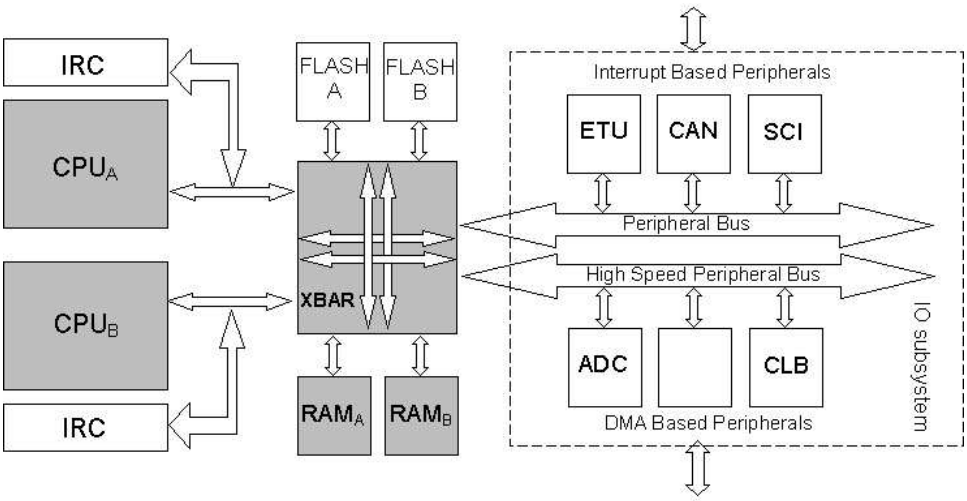

Figure 1.1: The Janus Dual Processor system.

or more DSPs [55, 35]. For example, the Texas Instruments SMJ320C80 is a single-chip MIMD[1] parallel processor that consists of a 32-bit RISC master processor, four 32-bit parallel DSPs, a transfer controller, and a video controller. All the processors are tightly coupled through an on-chip crossbar switch that provides access to a shared on-chip RAM.

Recently, Altera Corporation released a new version of its Nios II embedded processor [22], that is a 32 bit platform designed to be mapped on FPGA devices. In that way, the user can build a system with several processors, each one doing its specific job. Just to give an idea of the level of parallelism that can be implemented with those systems, a NIOS II processor can use from 700 to 2000 logic elements (depending on its configuration), whereas Altera's FPGAs for NIOS systems range from 3 to 40K logic elements, meaning that these FPGAs may potentially have tens of processors on a single chip.

Many other designs will be also available on the market soon, bringing the possibility to really use and take advantage of the increased power provided by multiprocessor.

The applications running on these new single-chip platforms require predictable (and fast) scheduling algorithms. In addition, kernels must fit in a few kilobytes of memory, and, together with the application, they must use the smallest possible amount of RAM memory. Resource sharing must be carefully handled and all communication primitives on shared memory must be designed in order to allow only a limited blocking time.

Moreover, the main problem with these systems is that there there is not a common way of exploiting these multiprocessor architectures in an efficient and scalable way. The purpose of this thesis is to find ways to use these systems without wasting the computing power available.

[1] Multiple Instruction Multiple Data.

Chip name	Description	ROM	RAM
AT91M40800	Atmel 32 bit ARM7 SoC	-	4 Kb
AT91F40816	Atmel 32 bit ARM7 Soc	3 Mb	8 Kb
H8/3297	Hitachi 8 bit microcontroller	60 Kb	2 Kb
ST10F269	Siemens 16 bit microcontroller	256 Kb	12 Kb
Janus	Double ARM7 by ST, PARADES, MM	256 Kb	64 Kb

Table 1.1: Typical memory sizes for system-on-a-chip.

1.2 Saving RAM space

Many embedded applications are designed on SoC architectures to reduce the cost of the final product. In these systems, RAM is extremely expensive in terms of chip space, and it heavily impacts on the cost of the final product (it is often necessary to re-design part of the application just to save a few RAM bytes).

Table 1.1 shows the memory space on a set of microcontrollers available on the market. As it can be seen, ROM memory (typically flash memory) is ten times bigger than the available RAM. That fact is justified because each cell of RAM can be implemented using 6 transistors, whereas a cell of Flash memory can be implemented with only one, allowing higher density.

Since, in general, there is a limited memory (both ROM and RAM) on a SoC, the application and the firmware must use the smallest possible amount of RAM memory.

In the design of the kernel mechanisms for such small embedded systems, it has been clear from the beginning that the choice of the real-time scheduling discipline influences both the memory utilization and the system overhead. For example, selecting a non preemptive scheduling algorithm can greatly reduce the overall requirement of stack memory whereas using a preemptive algorithm could increase the processor utilization.

For this reason, it has been very important to exploit different combinations and configurations of scheduling algorithms and services and to develop new ones in order to find the best kernel mechanisms for minimizing the memory requirements without jeopardizing the timing constraints.

To reduced the RAM space, proper design techniques (like the preemption threshold techniques) have been successfully developed in the past years, allowing a significant reduction of the stack usage in the entire system.

This thesis addresses the problems of memory optimization for multiprocessors SoC, extending the existing techniques for stack minimization to multiprocessors and to dynamic scheduled systems.

1.3 Thesis contributions

The main contributions of this thesis to the state of the art in design of the firmware for mono and multiprocessor system-on-a-chip is that I developed a complete methodology for minimizing the memory utilization of real-time task sets, communicating through shared

memory, in uniprocessor and multiprocessor systems. Moreover, I developed new techniques for scheduling in heterogeneous multiprocessors.

The contributions are detailed in the following points (points 1 to 4 are related to mono processor architectures, the rest for multiprocessor architectures):

1. I designed a novel scheduling algorithm, called **SRPT**, that allows the use of one single stack for all the real-time tasks under dynamic priority scheduling (Earliest Deadline) schemes.

2. I designed an optimization procedure for assigning the scheduling parameters (preemption thresholds and grouping of tasks in non-preemptive sets) so that the maximum stack size is minimizedwithout jeopardizing the schedulability of the task set.

3. I designed a novel scheduling algorithm called **MSRP**, that allows real-time tasks, allocated on different processor, to communicate/interact through shared memory; each task is statically allocated to one processor, and all tasks on one processor share the same stack.

4. I designed and implemented an optimization procedure for assigning tasks to processors and for assigning the scheduling parameters, so to minimize the overall stack size.

5. I designed a new scheduling algorithm for scheduling heterogeneous multiprocessor systems using fixed priorities.

6. I designed a new scheduling algorithm for scheduling heterogeneous multiprocessor systems using EDF scheduling.

1.4 Thesis outline

The remainder of this thesis is structured as follows.

Chapter 2 presents some of the techniques used as a basis for the results described in this thesis. Chapter 3 details the results found in single processor scheduling, such as the SRPT algorithm, and the stack minimization procedures. Chapter 4 describes the results obtained for homogeneous multiprocessors, must notably the MSRP scheduling algorithm, its comparison with the MPCP algorithm, and the allocation procedure that assigns tasks to processors. Chapter 5 describes the results on heterogeneous multiprocessors, and finally Chapter 6 gives the implications of the results in this thesis, and some possible future works on these topics.

Chapter 2

Background

The purpose of this chapter is to give a background knowledge that will help the reader understanding the rest of the thesis.

2.1 Reference Hardware architecture

Chapter 3 to Chapter 5 propose different scheduling algorithms developed on top of various abstract hardware architectures.

In general, for the purpose of this thesis I am not interested on hardware details of a specific architecture, but only to an abstract description of the hardware in terms of the following main blocks:

CPU. The CPU is typically a microcontroller or a general purpose processing unit. In the case of multiprocessor architectures, an instance of the architecture may have one or more CPUs.

DSP. A DSP is typically a specialized hardware that can be programmed to perform some sets of computations in a very efficient way; for that reason, in Chapter 5 I will use the simplistic hypothesis that activities on a DSP can be better executed in a non preemptive fashion to avoid pipeline flushes.

Memories. Memory in general will be present on a system in two fashions: RAM and ROM. RAM will be used to store global variables that needs to be modified by the application, and to store the stacks of the tasks in execution; ROM will contain the object code of the application, and all the constant values. Please note that this thesis will not consider any kind of support for cache memories.

Bus. The bus in general will be viewed as an interconnection between the different units. The only important feature is that it connects different elements of an architecture together.

Interfaces. Interfaces are used by the CPU or the DSP to interact with the environment. In general, I suppose these interfaces will be programmed putting appropriate values into some internal registers. Then, the interfaces will interact with the rest of

the architecture raising flags on some registers, or raising interrupts to signal the completion of its job.

I will not go in further detail about hardware architectures. It is sufficient to say that the architectural blocks just introduced are general enough to describe the architecture of many multiprocessor SoC available now and in development in the near future.

2.2 Reference software architecture

A multiprogrammed system is basically an abstraction of different computational flows each one executing concurrently. In reality, there a mechanism which transparently gives the impression to the applications to execute simultaneously, even if the computational flows are more than the physical entities available on the HW architecture.

To better understand these concepts, I introduce the following definitions:

Algorithm. An algorithm is a sequence of computational steps which has to be executed to solve a particular problem.

Program. A program is an algorithm coded using a particular programming language.

Thread. A thread is a single flow of execution, characterized by a program that specifies its behavior, and a set of resources (like some memory for data and stacks, a copy of the registers of the CPU, and so on).

Process. A process is the aggregation of many threads. Each thread inside a process shares the same address space with the other threads. Each process also has some private informations, such as the address space, descriptor tables, and so on.

Task. In this thesis, the difference between threads and processes is not relevant, and for that reason the word *task* will be used to identify an executable entity, that can be a thread or a process.

When developing a concurrent application for an embedded system, designers face a set of objectives which imposes several constraints. For this reason, a typical methodology for the development of an embedded system is a V-shaped methodology similar to that described in Figure 2.1.

In particular after a first specification phase, the system is decomposed in smaller subsystems designed, implemented and tested in isolation. At the end, all the subsystems are composed together to test the system as a whole, to verify that the initial global requirements are met.

In embedded system design it often happens that the hardware architecture is specified first (for cost and other constraints). Then, the designers have to fit the software into some predefined requirements (typically CPU power and footprint). Typical constraints could be of different types, raising from power management issues to software memory footprint and response times, and so on.

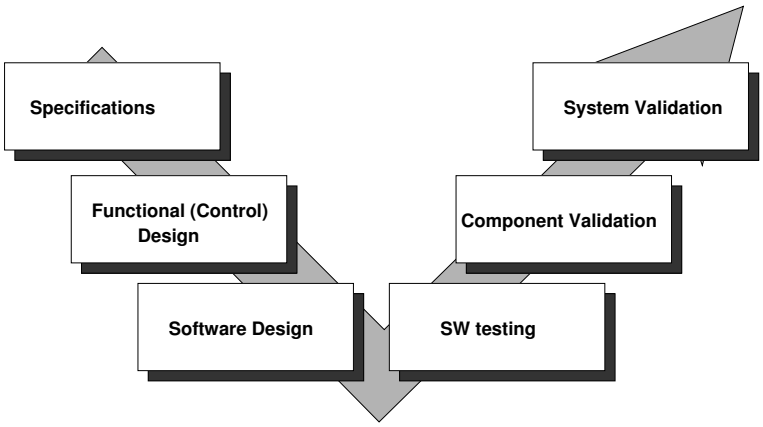

Figure 2.1: V-shaped methodology.

In particular, the focus of this thesis is on the design of software embedded systems that have to reduce the memory footprint without jeopardizing the real time guarantees imposed at design time.

The reference software architecture will be composed by two main layers:

The application layer, which basically contains the application software.

The firmware layer, which hides the details of the underlying hardware to the application layer exposing the features of a concurrent machine.

The firmware layer typically exports a common Application Program Interface (API), similar for both mono and multiprocessors, which is mainly composed by a set of components such as:

Boot code. The boot code is the set of routines that executes at power-on or when a SW reset or an exception is raised.

RTOS. The RTOS is responsible of exporting the abstraction of concurrent machine to the application layer. Typical abstractions exported by the RTOS are the concept of thread/process, the context change infrastructure, and atomic instructions for implementing mutual exclusion and synchronization.

Device Drivers. The device drivers are probably one of the most complex part of the firmware layer, because each embedded system-on-a-chip in practice implements its own set of peripherals and interfaces, and for that reason is the part that is most difficult to maintain when porting a firmware layer from an architecture to another.

Communication infrastructure. The communication infrastructure is responsible of exchanging messages between tasks located on the same or on remote CPU. Different industrial standard exists for the communication infrastructure , e.g. [34].

The purpose of this thesis is to focus on the design principles that controls the behavior of an RTOS, in particular for the part of the scheduling and synchronization algorithms between threads.

If we look in detail at the structure of a minimal RTOS for an embedded system, we can divide a generic RTOS in the following sub-components:

- interrupt handling;
- scheduling and thread/process handling;
- management of synchronization and shared resources;
- periodic reactivation handling;
- exception and error handling.

This methodology of designing the embedded system firmware has been used in several industrial standards. In particular, in this context it is worth citing the following standards:

OSEK/VDX: This is a standard for the operating system in the automotive fields, divided in different parts:

OS [33]: specifies the RTOS interface for threads, mutual exclusion, synchronization, error handling, and interrupt management.

COM: specifies a communication layer for messages inter and intra CPU.

HIS [5]: This is a working group composed by automotive car makers that aims to the production of a set of joint standards. In particular, they proposed a set of API to standardize the most common peripherals and interfaces.

2.3 Basic assumptions and terminology

The system consists of a set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of real time tasks to be executed on a single processor. A real time task τ_i is a infinite sequence of jobs (or instances) $J_{i,j}$. Every job is characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$ and a deadline $d_{i,j}$.

A task can be periodic or sporadic. A task is *periodic* if the release times of two consecutive jobs are separated by a constant *period*; a task is *sporadic* when the release times of two consecutive job are separated by a variable time interval, with a lower bound, also called minimum interarrival time.

Without loss of generality, I use the same symbol θ_i to indicate the period of a periodic task and the minimum interarrival time of a sporadic task τ_i. In the following a task will be characterized by a worst case execution time $C_i = \max\{c_{i,j}\}$ and a period θ_i. I assume that the relative deadline of a task is equal to θ_i: thus, $d_{i,j} = r_{i,j} + \theta_i$.

Tasks can access mutually exclusive resources through critical sections. Let $\mathcal{R} = \{\rho^1, \ldots, \rho^p\}$ be the set of shared resources. The k–th critical section of task τ_i on resource ρ^j is denoted by ξ_{ik}^j and its maximum duration is denoted by ω_{ik}^j.

2.4 RT Scheduling on single processors

This section recalls the main results on real-time scheduling for single processors embedded systems that will be used in the next chapters.

2.4.1 RM and EDF

The Rate Monotonic (RM) Scheduling algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates will have higher priorities. Since periods are constant, RM is a fixed priority assignment. Moreover, RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with shorter period. In 1973, Liu and Layland [44] showed that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM.

Given a task set, we can also define:

- the *utilization* of a single task, $U_i = \frac{C_i}{\theta_i}$;
- the *processor utilization* (that is a misure of the processor load), $U = \sum_i U_i$;

Although the processor utilization can be increased increasing the computation times C_i, or reducing the periods T_i, it exists an upper bound U_b, over which the schedule is no more feasible. That upper bound depends in general on the task set and on the scheduling algorithm used. Given a task set, the least upper bound U_{lub} is the minimum upper bound of all the task sets.

Liu and Layland also derived the following least upper bound of the processor utilization factor for a generic set of *n* periodic tasks using the Rate Monotonic algorithm: $U_{lub} = n(2^{\frac{1}{n}} - 1)$

This bound decreases with *n*, and for high values of *n*, the least upper bound converges to $U_{lub} = \ln 2 \cong 0.69$.

The Earliest Deadline First (EDF) algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. Specifically, tasks with earlier deadlines will be executed at higher priorities. EDF is a dynamic priority assignment. As RM it is intrinsically preemptive, and is also optimal in the sense stated by the following theorem (its proof can be found in [17]).

Theorem: A set of periodic tasks is schedulable with EDF if and only if

$$\sum_{i=1}^{n} \frac{C_i}{\theta_i} \leq 1$$

2.4.2 Priority Ceiling

The Priority Ceiling protocol (PCP) has been developed by Sha, Rajkumar and Lehoczky [59] to bound the priority inversion phenomenon[1] and prevent the formation of deadlocks and chained blocking.

The Priority Ceiling protocol can be defined as follows:

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the priority of the highest-priority job that can lock it.

[1] The Priority inversion phenomenon appears every time preemption is allowed and two tasks share an exclusive resource. In this case, the lower priority task access a shared resource and then is preempted; the higher priority task that shares the resource with the preempted task may be blocked also by a medium priority task that preempt the task locking the resource, blocking in this way also the higher priority task..

- Let J_i be the job with the highest priority among all jobs ready to run; thus, J_i is assigned the processor.
- Let S^* be the semaphore with the highest priority ceiling among all the semaphores currently locked by jobs other than J_i and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k, J_i must have a priority higher than $C(S^*)$. If $P_i \leq C(S^*)$ the lock on S_k is denied and J_i is said to be blocked on semaphore S^* by the job that holds the lock on S^*.
- When a job J_i is blocked on a semaphore, it transmits its priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with the priority of J_i. J_k is said to inherit the priority of J_i. In general, a task inherits the highest priority of the jobs blocked by it.
- When J_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k, p_k is set to the nominal priority P_k; otherwise, it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2, and J_2 blocks a job J_1, then J_3 inherits the priority of J_1 via J_2.

It can be proved that the Priority Ceiling Protocol prevents deadlocks and that a job can be blocked for at most the duration of one critical section.

2.4.3 Stack Resource Policy (SRP)

The Stack Resource Policy was proposed by Baker in [8] for scheduling a set of real-time tasks on a uniprocessor system. It can be used together with the Rate Monotonic (RM) scheduler or with the Earliest Deadline First (EDF) scheduler. According to the SRP, every real-time (periodic and sporadic) task τ_i must be assigned a priority p_i and a static preemption level λ_i, such that the following essential property holds:

task τ_i is not allowed to preempt task τ_j, unless $\lambda_i > \lambda_j$.

Under EDF and RM, the previous property is verified if preemption levels are inversely proportional to the periods of tasks:

$$\forall \tau_i \quad \lambda_i \propto \frac{1}{\theta_i}.$$

When the SRP is used together with the RM scheduler, each task is assigned a static priority that is inversely proportional to its period. Hence, under RM, the priority equals the preemption level. Instead, when the SRP is used with the EDF scheduler, in addition to the static preemption level, each job has a priority that is inversely proportional to its absolute deadline.

Every resource ρ^k is assigned a static[2] *ceiling* defined as:

$$\text{ceil}(\rho^k) = \max_i \{\lambda_i \mid \tau_i \text{ uses } \rho^k\}. \tag{2.1}$$

Finally, a dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max[\{ceil(\rho^k) \mid \rho^k \; is \; currently \; locked\} \cup \{0\}]. \tag{2.2}$$

Then, the SRP scheduling rule states that:

> *"a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling".*

The SRP ensures that once a job is started, it cannot be blocked until completion; it can only be preempted by higher priority jobs. However, the execution of a job $J_{i,k}$ with the highest priority in the system could be delayed by a lower priority job, which is locking some resource, and has raised the system ceiling to a value greater than or equal to the preemption level λ_i. This delay is called *blocking time* and denoted by B_i. Given the maximum blocking time for each task, it is possible to perform a schedulability test, depending on the scheduling algorithm.

In [8] Baker proposed the following schedulability condition for the EDF scheduler:

$$\forall i, \;\; 1 \leq i \leq n \quad \sum_{k=1}^{n} \frac{C_k}{\theta_k} + \frac{B_i}{\theta_i} \leq 1 \tag{2.3}$$

A tighter condition, proposed in [43], is the following:

$$\forall i, \;\; 1 \leq i \leq n \;\; \forall L, \;\; T_i \leq L \leq T_n \quad L \geq \sum_{k=1}^{i} \left\lfloor \frac{L}{\theta_k} \right\rfloor C_k + B_i. \tag{2.4}$$

In all cases, the maximum local blocking time for each task τ_i can be calculated as the longest critical section ξ_{jh}^k accessed by tasks with longer periods and with a ceiling greater than or equal to the preemption level of τ_i.

$$B_i = \max_{\tau_j \in \mathcal{T}, \forall h} \{\omega_{jh}^k \mid \lambda_i > \lambda_j \wedge \lambda_i \leq \text{ceil}(\rho^k)\}. \tag{2.5}$$

The Stack Resource Policy has several interesting properties. It prevents deadlock, bounds the maximum blocking times of tasks, reduces the number of context switches and can be easily extended to multi-unit resources. From an implementation viewpoint, it allows tasks to share a unique stack. In fact, a task never blocks its execution: it simply cannot start executing if its preemption level is not high enough. Moreover, the implementation of the SRP is straightforward as there is no need to implement waiting queues.

However, one problem with the SRP is the fact that it does not scale to multiprocessor systems. In Chapter 4 I propose an extension of the SRP to be used in multiprocessor

[2] In the case of multi-units resources, the ceiling of each resource is dynamic as it depends on the current number of free units.

systems.

2.4.4 Preemption Thresholds

Given a non-interleaved execution of the application tasks (obtained, for example, by using the SRP), the use of a preemptive scheduling algorithm makes the maximum number of task frames on the stack equal to the number of priority levels, whereas using a non-preemptive algorithm there can be only one frame on the stack. However, a non-preemptive algorithm in general is less responsive and could produce an infeasible schedule. Hence, the goal is to find an algorithm that selectively disables preemption in order to minimize the maximum stack size requirement while respecting the schedulability of the task set.

Based on this idea, Wang and Saksena, [58, 63] developed the concept of *Preemption Threshold*: each task τ_i is assigned *a nominal priority* π_i and a preemption threshold γ_i with $\pi_i \leq \gamma_i$. When the task is activated, it is inserted in the ready queue using the nominal priority; when the task begins execution, its priority is raised to its preemption threshold; in this way, all the tasks with priority less than or equal to the preemption threshold of the executing task cannot make preemption. According to [58], I introduce the following definitions:

Definition 1 *Two tasks τ_i and τ_j are **mutually non-preemptive** if $(\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)$.*

Definition 2 *A set of tasks $G = \{\tau_1, \tau_2, \ldots, \tau_m\}$ is a **non-preemptive group** if, for every pair of tasks $\tau_j \in G$ and $\tau_k \in G$, τ_j and τ_k are mutually non–preemptive.*

By assigning each task the appropriate preemption threshold, the number of preemptions in the system can be reduced without jeopardizing the schedulability of the tasks set. Given an assignment of preemption thresholds, the task set can be partitioned into *non–preemptive groups*. Obviously, a small number of groups results in a lower requirement for the stack size.

In Chapter 3, I will show how it is possible to efficiently implement the Preemption Threshold mechanism using the SRP, and extend it to be used under EDF.

2.5 RT scheduling on Multiprocessors

This section gives an survey of the main results on multiprocessor real-time scheduling. These results are cited in this thesis to let the reader to understand the complexity of the problem, and to position the work presented in this thesis in a broader set of research issues on multiprocessor scheduling. Although availability of inexpensive microprocessors has made practical to employ large number of processors in real-time applications, the programming of multiprocessor systems presents a rather formidable problem. In fact, Michael Dertouzos and Aloysius Ka-Lau Mok in [25] analyzed the problem of hard-real-time task scheduling in a multiprocessor environment. Representing the problem of scheduling on a multiprocessor as a game, they showed that optimal scheduling without a priori knowledge is impossible in the multiprocessor case even if there is no restriction on preemption owing to precedence or mutual exclusion constraints.

	Communication pattern	
Thread Creation	Static	Dynamic
Static	DAG	Single Program, Multiple Data (SPMD)
Dynamic	Dataflow	Unix fork/join

Table 2.1: A Simple classification of parallel programming styles.

Moreover, there is not a clear agreement on *which*, *what* and *how* a multiprocessor scheduler should work. In fact, the biggest problem in multiprocessor scheduling is the complexity of the solutions that have to be adopted to have good performances, since often the problem of scheduling a multiprocessor system is known to be NP-hard.

2.5.1 Classifications

To better understand the chosen solution for multiprocessor scheduling in SoC, I briefly introduce some notations and classifications of the various multiprocessor scheduling approaches proposed during the years.

The programming styles used to program threads and processes may in general be outlined in Table 2.1.

Moreover, the paradigm of coding and the implementation of the scheduler depends heavily on the communication architecture (*Shared memory* or *Message passing*) and on the visibility to the programmer of the interconnection network.

In [27], the authors analyze the characteristics of the scheduling algorithms for parallel computers identifying the four most commonly used or advocated techniques for programming, that are global queue, variable partitioning, dynamic partitioning and gang scheduling. Then, a system usually schedules its tasks using *time slicing* techniques (jobs share the use of the same processors) and *space slicing* techniques (each processor is allocated to a specific job until completion), or a combination of both. Usually, preemptive real-time systems uses time slicing techniques, without space slicing. Systems using time slicing techniques can also be classified on their implementation approach: in fact, there exist systems where all the CPUs are considered independent (approaches such as global or local queues are used), and systems where the CPUs are not considered independent (gang scheduling is used).

The schedulers can also be divided depending on the fact they allow or not the *migration* of the tasks. The term *migration* is related to the ability to resume a job on another processor after preemption. Usually this capability depends heavily on the performance and on the architecture of the system: allowing migration means also that the contents of the CPU cache have to be thrown away every time a job change processor. For this reason, the literature has developed a class of schedulers, called *cache-affinity* schedulers, that tries to optimize the cache performance scheduling a task on the same processor it executed when it was preempted. Tasks are partitioned on the various CPUs using bin-packing like schedulers when migration is not allowed.

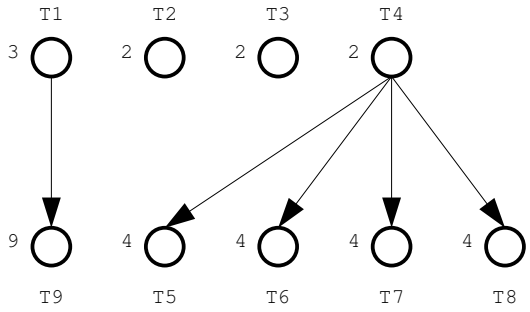

Figure 2.2: The task set is composed by nine tasks with precedence constraints and a priority proportional to their sequence number. The execution time of each task is shown near the balls.

2.5.2 Why Real-Time Multiprocessor scheduling is difficult?

When the problem of Real-Time scheduling is applied to multiprocessors, a number of difficulties arises. While Rate-Monotonic (RM) and Earliest deadline First (EDF) scheduling [44] are optimal for uniprocessor systems with fixed-priority and dynamic-priority assignment, it is, unfortunately, not so for multiprocessor systems. In fact, the problem of optimally scheduling a set of periodic tasks on a multiprocessor system using either fixed-priority or dynamic priority assignment is known to be NP-complete [42]. Hence, any practical solution to the problem of scheduling real-time tasks on multiprocessor systems presents a trade-off between computational complexity and performance.

Moreover, a negative result comes also from [25]. In particular, if we do not have a priori knowledge of any one of the following parameters:

- deadlines
- computation time
- start-times

then for any algorithm one might propose, one can always find a set of tasks which cannot be scheduled by another algorithm. The authors proved this assertion by a set of adversary arguments using then a nice scheduling game. Moreover, they proved that EDF is optimal if all the tasks have unit computation time.

The complexity of the multiprocessor scheduling is also showed by some counterexamples and some nice situations, like that reported by [32, 17], that we report here in Figures 2.2, 2.3, 2.4, 2.5, and 2.6.

Moreover, the scheduling problem may also be complicated if precedence relations, resource sharing and distributed systems are taken into account; to solve these problems some type of heuristics are used. For example in [21], heuristics putting communication overhead and other items into the objective function can result in a good trade-off.

Recently, the periodic scheduling problem has been solved using the notion of proportionate progress, or PFairness, and also interesting bounds have been found in the case of uniform multiprocessors (e.g.,[11, 9], and other marvelous works by the same authors).

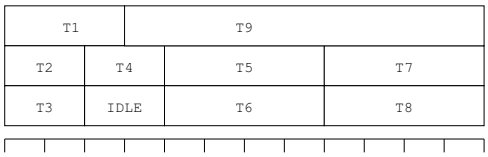

Figure 2.3: The optimal schedule (finishing time = 12).

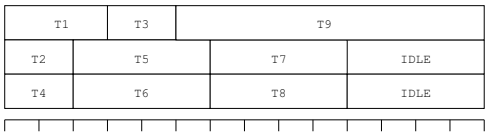

Figure 2.4: The schedule obtained changing task priorities (the order now is T1, T2, T4, T5, T6, T3, T9, T7, T8) (finishing time = 14).

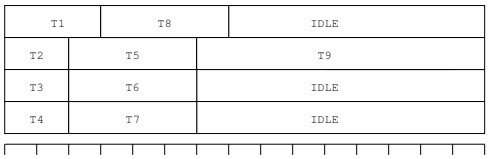

Figure 2.5: The schedule obtained adding a new processor (finishing time = 15).

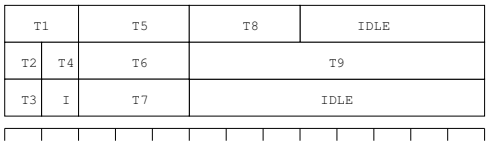

Figure 2.6: The schedule obtained reducing the execution times by 1 (finishing time = 13).

Another good work have been done by Bjorn Andersson his PhD Thesis [4] on the theoretical limits for the utilization in task sets for non-partitioned and partitioned scheduling in multiprocessors.

2.5.3 Bin-Packing Algorithms

This class of Real-Time scheduling algorithms is characterized by the absence of migration (because the overhead of a migrative solution may impose), and usually they consider the multiprocessor system as a set of CPU that have to be "filled" with independent tasks.

The various scheduling algorithms tries to schedule each processor with the classical uniprocessor scheduling schemes, like RM and EDF, partitioning the task set using heuristics to reduce the computation time (that remain NP-hard as the bin-packing problem).

Usually the goodness of the heuristics is measured using the ratio between the number of processors used by the proposed heuristic and the optimal (clairvoyant) schedule (see [16]).

In the next paragraphs, some results about bin-packing schedulers are summarized. In [32] a set of classical bin-packing heuristics are presented. The authors also analyze the cases of precedence relationships and resource constraints. In [49] the authors first did an extension of the guarantee test proposed by Liu and Layland (the same formula was recently found also by Bini et al. in [15]) and they use the result as the guarantee test for the classical First Fit and Best Fit bin packing heuristics. They propose another variant in which the tasks are divided in two subsets depending on their utilization factor $U_i = \frac{C_i}{T_i}$, where C_i is the worst case execution time of the task τ_i and T_i is its period. In [38] the authors proposed a technique in which a bin-packing technique is computed off-line and then applied every GCD of the task's periods. As PFair schedulers this scheme allow to obtain a full utilization of all the processors, but retain the problems of the use of a big number of preemptions and the heavy use of migration.

2.5.4 Resource sharing protocols

The bin-packing heuristics introduced in the previous Section usually consider the tasks to schedule as independent. While this assumption helps bounding the complexity of the problem to well-known techniques, on the other hand the scenario is not as real as it should to be used in *real* systems. In particular, these works do not consider the need of cooperation and synchronization between tasks.

The Real-Time literature proposed a number of solutions which solve the problem of sharing resources between tasks on multiprocessor systems. The solutions can be grouped fundamentally in three categories, briefly described in the following subsections:

- Classical blocking approaches.
- Wait-free approaches.
- Spin-lock and mixed approaches.

2.5.4.1 Classical blocking approaches

These approaches are based on extensions of Priority ceiling and Dynamic Priority Ceiling for uniprocessors.

Usually a new protocol is derived, the blocking factors are quantified, giving then a guarantee test similar to uniprocessors. These protocols subsumes either a shared memory multiprocessor or a distributed multiprocessor. Tasks are statically assigned to the processors and scheduled with RM or EDF. The resources are also divided in local resources or global resources depending on the fact that the tasks that use the resource are assigned to one or more than one processor.

The main algorithms that falls in this class are the MPCP and the DPCP resource sharing protocols [51]. I will describe these two protocols in chapters 4 and 5, comparing their performance against my scheduling algorithms for multiprocessor scheduling. Other interesting works in this field are [19, 46].

2.5.4.2 Wait Free approaches

The Wait-free approaches offers an attractive alternative to lock-based schemes because they eliminate priority inversion and its associated problems. The problem with such approaches is that access to such objects are not guaranteed to complete in a bounded time. Nonetheless, schedulability conditions are developed in [54] to demonstrate the applicability of lock-free objects in hard real-time systems.

Another way to implement a lock-free resource sharing protocol is to apply the concept of processor consensus. Basically the writer and the reader come to an agreement on accessing the shared data before proceeding to carry out their respective operations. For example, in [20] the authors propose a lock-free protocol based on processor consensus very similar to the uniprocessor CAB protocol used by some uniprocessor Real Time Kernels (see [18, 40, 30]).

2.5.4.3 Spin-lock and mixed approaches.

The orthogonal approach to the lock-free protocols and to the lock protocols for resource sharing is the spin-lock approach. When a process want to lock a resource and the resource is currently used, the blocked process is not preempted but it still remain into execution busy waiting the release of the lock. This protocol is very useful if the critical sections are very short, whereas it is not usable when they are long, since the task simply waste time that can be used for other processes. There exist a lot of protocols that are based to the basic spin lock idea. For example, in [47] the authors proposes a configurable lock protocol that may adapt the locking behavior from spin lock to blocking to fulfill the application needs. Another interesting result is obtained in [60], in which the authors propose a spin lock protocol whose performances are independent from the preemption overhead that the blocked tasks have to experience due to the preemption of a task that, while busing waiting for the lock, is preempted by another high priority task. The proposed solution is called SPEPP and it is based on the idea that a spinning processor may execute the critical section of another task that is currently preempted on another processor.

In Chapter 4 I will describe the MSRP protocol, that extends the SRP protocol to multiprocessor using spin-lock techniques.

2.5.5 Task Migration

The term migration is related to the ability of an operating system to resume a job on another processor after preemption. It is useful to implement an operating system that allow task migration. Intuitively a CPU, when it finishes the tasks currently allocated on it for scheduling, can resume the execution of a preempted thread that was previously executed on another processor. In that way, the system load can be balanced, reducing the mean response time of the system.

Unfortunately, the task migration has some disadvantage due to different factors, like:

- Incremented system complexity (some architectures as, for example, the NUMA architecture, does not allow simple task migration).
- Communication overhead (if the architecture does not support directly task migration, a lot of data has to be passed through the communication network).
- Incremented number of cache miss (when a task migrates from a CPU to another, the new cache does not contain any memory reference for the new task).

In large-scale supercomputers traditionally all the two approaches (migratory and non-migratory) have been pursued (see [27]), whereas the literature of Real-Time Scheduling tried to use fundamentally a non-migratory approach.

The other algorithms usually fall in two categories: a category that includes a set of algorithms that are intrinsically preemptive with migration (such as [11] and [38]) and other algorithms such as partitioned approaches based on bin-packing that usually tries not to use migration. The latter solution allow to reduce the multiprocessor scheduling problem to a uniprocessor scheduling problem that has been studied over the years and for that exists optimal results.

Other works have also shown the influence of migration on the schedulability of real-time systems. In [39] the authors analyzes the power of the scheduling algorithms which do not allow migration, giving an upper and a lower bound based on a worst case behavior and based on an algorithm that build non migratory schedules. Then, in [37], the cited results have been improved using a conflict graph and a theorem of the extremal graph theory. In that work, the authors gives a way to build a non-migratory schedule starting from a migratory one. They also prove that a migrative scheduler on m processor can always be transformed in a non-migrative scheduler on 3m-2 processors.

Finally, recently there have been some work on systems with restricted interprocessor migrations ([10], and successive works by the same authors). These works may be promising for small multiprocessor systems (such as Janus [28] or Altera NIOS II [22]) that cannot allow efficient migration of a job during its execution, but that can migrate instances of tasks between processors if needed.

Chapter 3

Single processor architectures

This chapter describes the design of a novel scheduling algorithm, called **SRPT**, that allows the use of one single stack for all the real-time tasks under dynamic priority scheduling (Earliest Deadline) schemes, and integrates preemption threshold techniques to minimize the preemptiveness between tasks.

After that, an optimization procedure for assigning the scheduling parameters (preemption thresholds and grouping of tasks in non-preemptive sets) will be described. The technique aims to the minimization of the maximum stack size without jeopardizing the schedulability of the task set.

This chapter is composed by three sections: Section **??** introduces the terminology and the basic assumptions made in this chapter; Section 3.1 describes the integration of SRP with the preemption threshold technology, and finally Section 3.2 describes the optimization algorithm I developed to find a grouping of tasks with minimal stack utilization.

3.1 Integrating Preemption Threshold with the SRP

The idea behind this work is based on the concept of non-interleaved execution. As explained in Section 2.4.3, using a protocol called Stack Resource Policy (SRP) [8], task executions are perfectly nested: if task A preempts task B, it cannot happen that B executes again before the end of A. In this way, it is possible to use a single stack for all the execution frames of the tasks. An example of this behavior is depicted in Figure 3.1.a where three periodic tasks τ_0, τ_1 and τ_2 are scheduled by SRP. In the upper part of the figure, the ascending arrows denote the task activations, whereas the descending arrows denote the task deadlines. In the lower part, the system stack size is plotted against the time.

Next, comes the following observation: if task preemption is limited to occur only between selected task groups, it is possible to bound the maximum number of task frames concurrently active in the stack, therefore reducing the maximum requirement of RAM space for stack (which is the only way the OS can limit RAM requirements). In the example of Figure 3.1.b, preemption is disabled between τ_2 and τ_1 and, consequently, only two task frames can be active at the same time: thus I can decrease the amount of memory to be reserved for the system stack. This behavior can be obtained using the preemption threshold

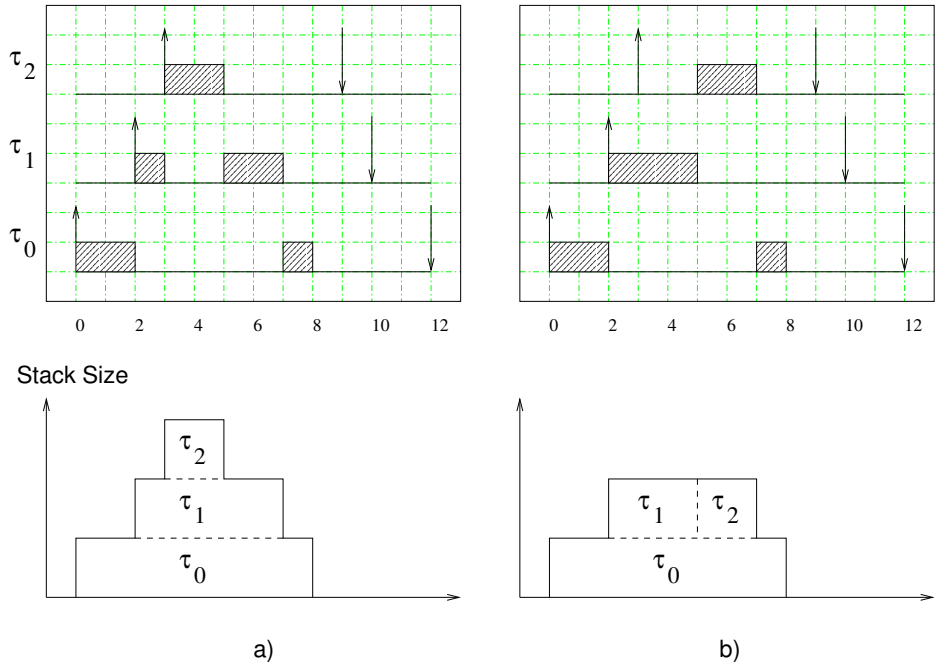

Figure 3.1: Two different schedules for the same task set: **a)** full-preemptive schedule; **b)** preemption is disabled between τ_1 and τ_2.

technique, that has been formulated by Saksena and Wang [63, 58] in the context of the fixed priority scheduling of independent tasks in uniprocessor systems. The mechanism has been implemented (in a proprietary form) in the SSX kernel from REALOGY [24] and the ThreadX kernel from Express Logic [26], both targeted to embedded system with small code requirements, and has been integrated into the OSEK/VDX standard [33].

This thesis extends these ideas along many directions. First, I consider dynamic priority scheduling instead of fixed priority. In particular, the algorithms presented in this chapter are based on the Earliest Deadline First (EDF) scheduling algorithm, which achieves better processor utilization with respect to fixed priority schedulers. Second, the objective of the algorithm presented in Section 3.2 is not to find the smallest number of non-preemptive groups that keep the set schedulable, but rather to assign tasks to non-preemptive groups in such a way that the overall requirement of stack space is minimized (which means introducing the stack requirements of each task as a factor). Third, tasks are not considered independent but are allowed to interact through mutually exclusive critical sections (i.e. shared memory).

Note that the techniques presented in this chapter will be extended in Chapter 4 to homogeneous multiprocessor systems, where tasks allocated on different processors can interact through shared memory.

My approach is based on the observation that the threshold values used in the Preemption Threshold mechanism are very similar to the resource ceilings of the SRP. In the SRP, when a task accesses a critical section, the system ceiling is raised to the maximum

between the current system ceiling and the resource ceiling. In this way, an arriving task cannot preempt the executing task unless its preemption level is greater than the current system ceiling. This mechanism can be thought as another way of limiting preemptability.

Thus, if I want to make task τ_i and task τ_j mutually non-preemptive, I can let them share a *pseudo-resource* ρ^k: the ceiling of resource ρ^k is the maximum between the preemption levels of τ_i and τ_j. At run time, instances of τ_i or τ_j will lock ρ^k when they start executing and hold the lock until they finish.

Suppose task τ_i needs a set of pseudo-resources $\rho^1, \ldots, \rho^h$. When τ_i starts execution, it locks all of them: in the SRP, this corresponds to raising the system ceiling to $\max_k \text{ceil}(\rho^k)$. I define this value as the *preemption threshold* γ_i of task τ_i. Now, the problem of finding an optimal assignment of thresholds to tasks is equivalent to finding the set of pseudo-resources for each task. In the remaining of this thesis, I will indicate this modification of the SRP as *SRPT* (SRP with Thresholds).

Since SRPT can be thought as an extension of the SRP that add pseudo-resources compatibles with the traditional SRP resources, it can be easily shown that SRPT retains all the properties of SRP.

The feasibility test for SRPT is given by one of Equations (2.3) and (2.4) in Section 2.4.3, except for the computation of the blocking time, that is:

$$B_i = max(B_i^{local}, B_i^{pseudo})$$

where B_i^{local} and B_i^{pseudo} are respectively the blocking time due to local resources and the blocking time due to pseudo-resources.

Blocking due to local resources. Assuming relative deadlines equal to periods, the maximum local blocking time for each task τ_i can be calculated using Equation (2.5) in Section 2.4.3. This can be easily proved: supposing the absence of pseudo-resources, the SRPT reduces to the SRP, and the blocking times can be calculated using equation 2.4 in Section 2.4.3.

Blocking due to pseudo-resources. A task τ_i may experience some additional blocking time due to the non-preemptability of lower priority tasks. This blocking time can be computed as follows:

$$B_i^{pseudo} = \max_{\tau_j \in \mathcal{T}} \{C_j \mid \lambda_i > \lambda_j \ \wedge \lambda_i \leq \gamma_j\}$$

The non-preemptability of lower task is due to the use of pseudo-resources. The formula of B_i^{pseudo} is another way of writing formula 2.5 in Section 2.4.3, because:

- γ_i is $\max_k \ ceil(\rho^k) = ceil(\rho^{k'})$ where $k' \in \{k : \gamma_i = ceil(\rho^k)\}$
  - C_i is the critical section duration for resource k' (remember that pseudo-resources are locked when an instance starts and is unlocked when an instance finish); moreover, I can consider only the k' critical section for each task since they all have length equal to C_i and $\forall\, k, ceil(\rho^k) \leq ceil(\rho^{k'}) = \gamma_i$.

Example. Consider the example of Figure 3.1.b. The three tasks τ_0, τ_1 and τ_2 are sporadic, and their computation times and minimum interarrival times are respectively $C_0 = 3, T_0 = 12, C_1 = 3, T_1 = 8, C_2 = 2, T_2 = 6$. By definition of preemption level, we have $\lambda_0 < \lambda_1 < \lambda_2$. I want to make τ_1 and τ_2 mutually non preemptive, so I introduce a pseudo-resource ρ^0. Every time τ_1 (or τ_2) starts executing, it locks ρ^0, and holds the lock until it finishes. The ceiling of ρ^0 is $\max(\lambda_1, \lambda_2) = \lambda_2$. By definition of preemption threshold, $\gamma_1 = \gamma_2 = ceil(\rho_0) = \lambda_2$, whereas $\gamma_0 = \lambda_0$.

In this way, we have two preemption groups, the first consists of tasks τ_1 and τ_2, the second contains only τ_0. Hence, the blocking time of τ_2 is:

$$B_2 = B_2^{pseudo} = C_1 = 3$$

and, substituting in Equation(2.3), the system results schedulable.

The algorithm works as follows (see Figure 3.1.b):

- At time $t = 0$, task τ_0 is activated and starts executing. The system ceiling Π_s is equal to γ_0.
- At time $t = 2$, task τ_1 arrives, and since its priority is the highest and $\Pi_s = \gamma_0 < \lambda_1$, it preempts task τ_0. Now the system ceiling is equal to $\Pi_s = \gamma_1$.
- At time $t = 3$, task τ_2 arrives, and even though it has the highest priority, its preemption level λ_2 is not higher than the current system ceiling. Hence, according to SRP it is blocked, and τ_1 continues to execute.
- At time $t = 5$, task τ_1 finishes, and the system ceiling returns to the previous value (γ_0). At this point, task τ_2 can start executing. The system ceiling is raised to γ_2.

Notice that, if τ_0 is also included in the same preemption group as τ_1 and τ_2, the system remains schedulable and the stack size can be reduced further. □

The SRPT algorithm presents two main advantages:

- it seamlessly integrates access to mutually exclusive resources and preemption threshold with a very little implementation effort and with no additional overhead;
- it permits to implement the preemption threshold mechanism on top of EDF.

The last issue can lead to further optimizations: the EDF scheduling algorithm has been proven optimal both in the preemptive [44, 12, 13] and in the non-preemptive[1] version [36]; furthermore, in [43] the authors claim that EDF+SRP is an optimal algorithm for scheduling sporadic task sets with shared resources. Since EDF is optimal, it is more likely that a given assignment of preemption thresholds produces a feasible schedule. Therefore,

[1] The non-preemptive version of the EDF algorithm is optimal for sporadic task sets among all the non-idle (work conserving) non-preemptive scheduling algorithms.

I expect a better chance to trade processor utilization with a reduction in the maximum stack space requirement by reducing preemption.

It is clear that in this methodology I am sacrificing task response time versus memory size. However, the response time of some task could be critical and should be maintained as low as possible. In this case, it is possible to reduce the relative deadline of that task to increase its responsiveness. For simplifying the presentation, I do not consider here the case of tasks with deadline smaller than the period.

3.2 Optimizing stack usage in Uniprocessors

In this section I present an algorithm that allows the optimization of the total stack space requirement of a set of tasks using the SRPT protocol on uniprocessor systems. The algorithm presented in this section implicitly uses pseudo resources to raise the threshold of a task. To simplify the presentation, I do not consider here the use of shared resources. Shared resources can be taken into account using the techniques presented in Section 2.4.4.

An extension to this algorithm that includes an allocation algorithm for multiprocessors will be presented in Section 4.3.

The algorithm requires each task to be characterized by its worst case execution time C_i, its period θ_i, its maximum stack requirement (in bytes) s_i and its preemption level λ_i. At the end of the optimization algorithm, each task τ_i will be assigned a preemption threshold γ_i and will be inserted in a non-preemptive group G_k. The goal of the optimization algorithm is:

step 1 to find an assignment of preemption thresholds to tasks, maintaining the feasibility of the schedule and

step 2 to find an optimal set of non-preemptive groups that minimizes the total stack size, maintaining the feasibility of the schedule.

Notice that unfortunately a preemption threshold assignment does not determine univocally a set of non-preemptive groups. Hence, after assigning the preemption threshold I still do not know the maximum number or tasks that can be present on the stack at the same time and how much memory must be allocated for the stack. For this reason, I need to perform step 2.

The optimization algorithm works as follows:

- Tasks are ordered by decreasing preemption level;
- *Step 1*: I use the algorithm described in [58] to explore the space of possible threshold assignments[2]: starting with the task having the highest preemption level, I try to raise the preemption threshold γ_i of each task τ_i, to the maximum level that allows to preserve the schedulability of all tasks. (i.e. incrementing the preemption threshold is allowed only if the task set remains schedulable.) The algorithm stops when a further increment on any task makes the system not schedulable.

[2]Since EDF is optimal, there is no need to find an initial priority assignment for the task set.

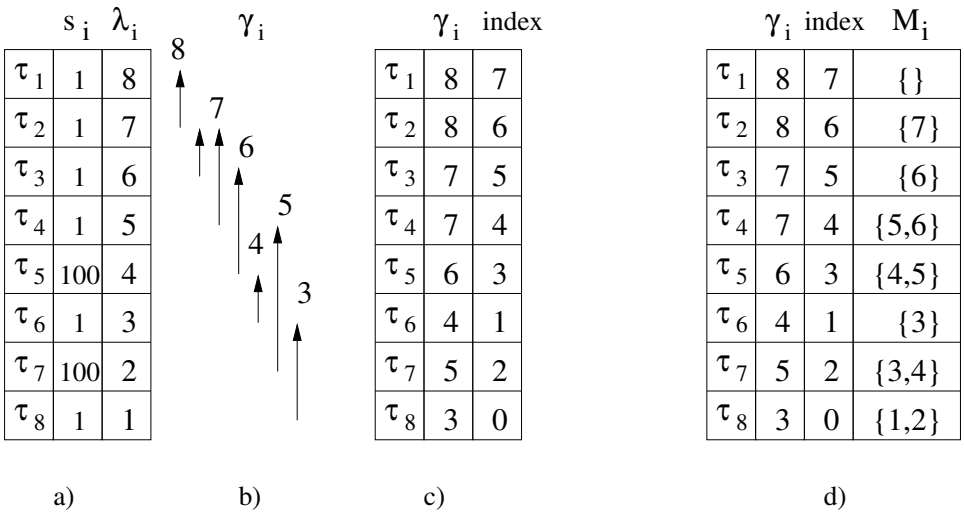

Figure 3.2: An example: The minimum total stack size does not corresponds to the minimum number of non-preemptive groups: a) Initial task set b) computation of the preemption thresholds c) reordering d) computation of the maximal groups)

- *Step 2*: Given a feasible assignment of preemption thresholds, I partition the task set into non-preemptive groups and compute the maximum required stack size.

My algorithm differs from the one in [58] in the final optimization objective: while the algorithm in [58] tries to minimize the number of non-preemptive groups, my algorithm accounts for the stack usage of each task and tries to minimize the total amount of required stack. In fact, there are cases in which the maximum overall stack requirement does not correspond to the minimum number of groups, as shown in the example of Figure 3.2. In this example there are 8 tasks, all having a stack frame size of 1 except τ_5 and τ_7 which have a stack frame size of 100 (Figure 3.2.a). The partition in non-preemptive groups provided by algorithm OPT-PARTITION in [58] is $G_1 = \{\tau_1, \tau_2\}$, $G_2 = \{\tau_3, \tau_4, \tau_5\}$, $G_3 = \{\tau_6, \tau_7, \tau_8\}$, which leads to a total stack size of 201. However, note that task τ_7 and task τ_5 are mutually non-preemptive (in fact, $\lambda_5 < \gamma_7$ and $\lambda_7 < \gamma_5$). If we consider groups $G_1 = \{\tau_1\}$, $G_2 = \{\tau_2, \tau_3\}$, $G_3 = \{\tau_4, \tau_5, \tau_7\}$, $G_4 = \{\tau_6, \tau_8\}$ the total stack requirement is 103. Note that, in this case, by using the solution with the minimum number of groups, we would have overestimated the maximum stack requirement by a factor of 2.

Figure 3.2.b shows the result of the *preemption threshold assignment phase* for the example. The preemption thresholds of all tasks (except τ_1) are raised to the values in the column marked as γ_i. The algorithm used to partition the task set into preemption groups (step 2) is described in the remaining of this section. First some definition:

Definition 3 *A **representative task** for a non-preemptive group is the task having the the smallest threshold among all the tasks in the group.*

In the following, G_i will denote a non-preemptive group with representative task τ_i.

Definition 4 *A **maximal group** for task τ_i is the biggest non-preemptive group that can be created having τ_i as a representative task.*

```
Algorithm: FindMaximalGroups
foreach τ_i in T {
   M_i = emptylist;
   foreach τ_j in {τ_k : τ_k ∈ T and k > i}
      if (λ_j ≤ γ_i) insert(M_i, j);
}
```

Figure 3.3: Algorithm for finding the maximal groups.

In the following, I denote with M_i the maximal group for task τ_i minus τ_i. For example, if the maximal group for task τ_1 is $\{\tau_1, \tau_2, \tau_3\}$, I denote $M_1 = \{\tau_2, \tau_3\}$.

With this definitions, the algorithms in *step 2* is the following:

- Tasks are ordered by increasing preemption thresholds, ties are broken in order of decreasing stack requirements. For clarifying the algorithm, after ordering I rename each task using an increasing *index* (see the example in Figure 3.2.c); this index will be used from now on for identifying the tasks. Thus, is τ_i is assigned index j, in the following it will be referred to as τ_j. Hence, according to the new ordering,

$$i < j \Rightarrow \gamma_i < \gamma_j \vee (\gamma_i = \gamma_j \wedge s_i \geq s_j)$$

  where s_i is the stack requirement for task τ_i.

- The algorithm starts by finding the set M_i for each task τ_i. Maximal groups are computed with the algorithm shown in Figure 3.3. In Figure 3.2.d, the M_i are computed for each task and are shown in the last column.

- Then, the algorithm calls a recursive function that allocates all tasks to non–preemptive groups. The function, called **create_group()**, recursively computes all possible partitions of the tasks into non-preemptive groups, and computes the maximum stack requirement for each partition. The minimum among these requirements will be the maximum memory for the stack that we need to allocate in the system.

Enumerating all possible partitions in non-preemptive groups clearly takes exponential time. I claim that the problem of finding the required stack memory size, given a preemption threshold assignment, is NP-hard. This claim is supported by the observation that the problem is somewhat similar to a bin-packing problem, which is known to be NP-hard.

Hence, a great effort has been devoted in trying to reduce the mean complexity of Algorithm `creategroup` by pruning as soon as it is possible all the solutions that are clearly non-optimal. In the following, I give a description of Algorithm **creategroup**, and proof sketches of its correctness.

The pseudo-code for **creategroup** is shown in Figure 3.4. The algorithm is recursive: at each level of recursion a new non-preemptive group is created. The following global variables are used: `minstack` contains the value of the candidate optimal solution and is initialized to the sum of the stack requirements of all tasks; F is the set of tasks that have not yet been allocated; $G_1, \ldots, G_n$ are the non-preemptive groups that will be created by

the function; $M_1, \ldots, M_n$ are the maximal groups. The parameters of `creategroup` are τ_g, which is the representative task based on which a new group G_g will to be created and `sum` that is the sum of the stack requirement of the already created groups.

The key point in understanding how **creategroup** works is that the space of candidate solutions is explored in a well defined order: in particular, all the task sets mentioned so far are ordered by increasing preemption threshold (ties are broken in order of increasing stack requirements).

`creategroup` is first invoked with τ_1 (that is the task with the lowest preemption threshold) and $sum = 0$.

When invoked (at the k-th level of recursion), function `creategroup` builds a non-preemptive group for task τ_g by inserting all tasks from M_g that are not allocated yet (lines 3 - 10).

Now, if there are still tasks to be allocated (line 12), `creategroup` tries to recursively call itself in order to compute the next group. However, this recursion is not performed if I am sure that no better solution can be found in this branch (Condition 1 at line 13 and Condition 2 at line 15).

Then, the algorithm does backtracking, by extracting tasks from G_g and inserting them back in F, in order to explore all possible configurations for this branch (lines 21 - 26). Condition 3 at line 22 further reduces the number of solution to be checked by pruning some configuration of G_g that cannot improve the optimal solution.

If this is the last level of recursion for this branch (it happens when F is empty), I check whether the current candidate solution is optimal, and if it is the case, I save the value of this solution in `minstack` and the current group configuration by calling function `NewCandidate()`. Notice that, before returning to the previous recursion level, all tasks are removed from G_g (lines 36 - 37).

Now I describe the conditions that allow `creategroup` to prune non-optimal branches. Let me define the *required tasks* as the representative tasks of the non-preemptive groups found by Algorithm OPT-PARTITION (described in [58]). These tasks are important because in every possible partition, they will always be in different non-preemptive groups. Hence, the solution is bounded from below by the sum of the stack sizes of the *required tasks*.

Condition 1 is false if the sum of `newsum` and the size of the stack of the required tasks that have not been allocated yet is greater than or equal to `minstack`.

The correctness of this condition is trivially proven.

Condition 2 is false when $\gamma_g = \gamma_f$.

Theorem 1 *If Condition 2 does not hold, (i.e. $\gamma_g = \gamma_f$), then any solution with τ_f as representative task of a new non-preemptive group cannot achieve a solution with a lower stack requirement than the already explored solutions.*

Proof.

First I will prove that $G_f \subset M_g$. Consider $\tau_i \in G_f$, it follows that $\lambda_i \leq \gamma_f$, $\lambda_f \leq \gamma_i$ and $\gamma_f \leq \gamma_i$.1 Since $\gamma_f = \gamma_g$, it follows that $\lambda_i \leq \gamma_f \leq \gamma_g$, and $\lambda_g \leq \gamma_g = \gamma_f \leq \gamma_i$.

```
int minstack = ∑ s_i;
F = T;
∀i  G_i = {};
 1:     creategroup(τ_g, sum) {
 2:        int newsum;
 3:        G_g.insert(τ_g);
 4:        τ_i = M_g.queryFirst();
 5:        end = false;
 6:        do {
 7:              ∀τ_j ∈ M_g if (τ_j ∈ F and j ≥ i) {
 8:                    G_g.insert(τ_j);
 9:                    F.remove(τ_j);
10:              }
11:              newsum = sum + stackUsage(G_g);
12:              if (!empty(F)) {
13:                    if (condition1) {
14:                          τ_f = F.removeFirst();
15:                          if (condition2)
creategroup(τ_f,newsum);
16:                          else {
17:                                F.insert(τ_f);
18:                                end = true;
19:                          }
20:                    }
21:                    if (G_g != {τ_g} and !end) {
22:                          while (G_g ≠ τ_g and condition3) {
23:                                τ_h = G_g.removeLast();
24:                                F.insert(τ_h);
25:                          }
26:                          τ_i = M_g.queryNext(τ_h);
26:                    } else end = true;
28:              } else {
29:                    if (newsum < minstack) {
30:                          minstack = newsum;
31:                          NewCandidate();
32:                    }
33:                    end = true;
34:              }
35:        } while (!end);
36:        F.insertAll(G_g);
37:        G_g.removeAll();
38:     }
```

Figure 3.4: The create_group() recursive function.

As a consequence, τ_i and τ_g are mutually non preemptive, and that τ_i is in M_g. Hence, $G_f \subset M_g$.

Since the current iteration that have τ_g as a representative tasks visits all the subsets of M_g, and $G_f \subset M_g$ it follows that any configuration produced by calling recursively the `creategroup` algorithm with representative task τ_f leads to an overall stack usage that is bounded from below by the following expression:

$$\begin{gathered}\texttt{stackUsage}(G_g) + \texttt{stackUsage}(G_f) \geq \\ \geq \texttt{stackUsage}(G_g \cup G_f) \geq \texttt{stackUsage}(M_g).\end{gathered}$$

However, M_g is a branch that has already been explored (it is the first branch that algorithm `creategroup` explores). Hence, the theorem follows. □

Condition 3 is true when the task with the maximum stack requirement in G_g has been removed, that is when the stack usage of G_g has been reduced.

Theorem 2 *All the branches in which the task with the maximum stack requirement has not been removed from G_g cannot lead to a better solution than the already explored ones.*

Proof Sketch.
The basic idea is based on the fact that solutions are explored in a certain order. In particular, the first solution is the *greedy* solution, where G_g is the biggest non-preemptive group that can be built.

When considering the next configuration for G_g, it must have a lower stack requirement than the previous one. In fact, all solutions in which G_g has the same stack requirement are bounded from below by the first solution explored. Hence, Condition 3 forces the removal of tasks from G_g until its stack requirement is reduced. □

As an example, a typical run of the algorithm on the task set of Table 3.2 will work as follows:

- To find the first solution, three recursive calls are needed, creating groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_4, \tau_5\}$, and $G_6 = \{\tau_6, \tau_7\}$, with a total stack of 201. This first solution is equal to that found by the algorithm OPT-PARTITION proposed in [58].

- Then, group G_6 is rolled back. Task 5 is removed from group G_3, and τ_i is set to the next task (τ_5 will not be reconsidered for inclusion in the next group configuration). The recursive calls produce groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_4\}$, $G_5 = \{\tau_5, \tau_6\}$, $G_7 = \{\tau_7\}$.

- Group G_7 is rolled back and τ_6 is removed from group G_5. The recursive calls produce groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_4\}$, $G_5 = \{\tau_5\}$, $G_6 = \{\tau_6, \tau_7\}$.

- Then, groups G_6 and G_5 are rolled back, and τ_4 is removed from G_3. Now τ_i is moved past τ_4, and τ_5 is re-inserted into G_3. Next, τ_4 and τ_7 are chosen as rep-

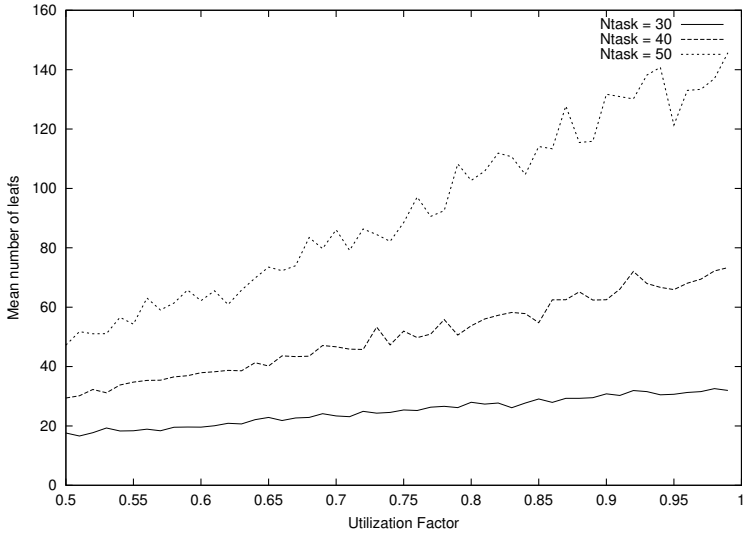

Figure 3.5: Mean number of explored solutions for different task set sizes.

resentative tasks giving groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_5\}$, $G_4 = \{\tau_4, \tau_6\}$, $G_7 = \{\tau_7\}$.

- Again, G_7 is emptied and τ_6 is removed from G_4. Group G_6 is created, giving groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_5\}$, $G_4 = \{\tau_4\}$, $G_6 = \{\tau_6, \tau_7\}$.
- At this point, groups G_6 and G_4 are removed; τ_5 is also removed from G_3. Then, groups G_4 and G_7 will be created, giving groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3\}$, $G_4 = \{\tau_4, \tau_5, \tau_6\}$, $G_7 = \{\tau_7\}$.
- After some other non optimal solutions, the first recursive call of **creategroup()** will remove τ_2 from G_0, letting the creation of group $G_2 = \{\tau_2, \tau_3, \tau_4\}$ that will bring the algorithm to the optimal solution.

As already mentioned, the complexity of the algorithm is exponential in the number of tasks. However, since the number of groups in the optimal solution is often small, the number of combinations to evaluate is limited. Thanks to the efficiency of the pruning, the number of solutions is further reduced. In Figure 3.5 and 3.6 the average number of explored solutions (leafs) is plotted against the load of the system and for different number of tasks: the resulting average number is quite low even for large task sets. I conclude that, for typical embedded systems in the domain of automotive applications where the number of tasks is relatively small, the problem is tractable with modern computers.

3.3 Experimental evaluation

I extensively evaluated the performance of the optimization algorithms on a wide range of task set configurations. In every experiment, tasks' periods are randomly chosen between 2 and 100. The total system load U ranges from 0.5 to 0.99, with a step of 0.01: the worst

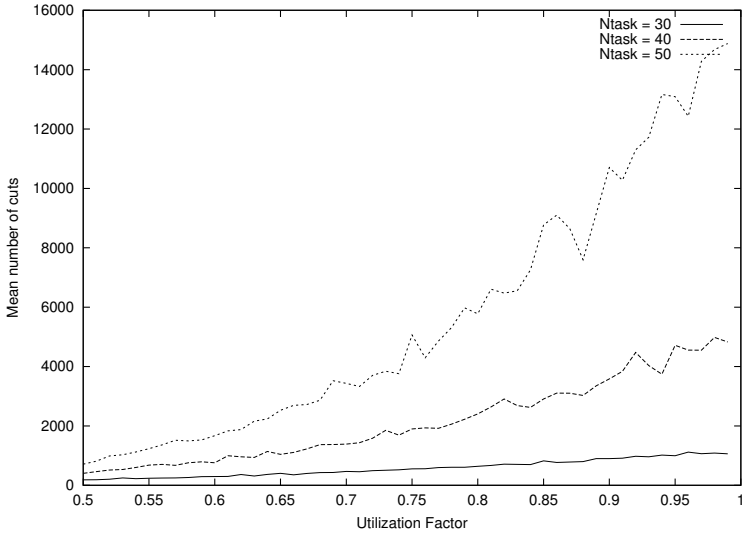

Figure 3.6: Mean number of *cuts* for different task set sizes.

case execution time of every task is randomly chosen such that the utilization factors sums up to U. The number of tasks in the task set ranges from 1 to 100, and the stack frame size is a random variable chosen between 10 and 100 bytes excepts for the experiments of Figure 3.9 in which the stack size ranges between 10 and 400 bytes.

In Figure 3.7 the average number of preemption groups is shown. Figure 3.8 is a cross-cut section of Figure 3.7.

Note that:

- The figure has a maximum for NTASK = 4 and U=0.99. As the number of tasks increases, the number of preemption groups tends to 2; this can be explained with the fact that, when the number of tasks grows, each task has a smaller worst case execution time; hence, the schedule produced by a non-preemptive scheduler does not differ significantly from the schedule produced by a preemptive scheduler. On the contrary, with a small number of tasks, the worst case execution time of each task is comparable with the period; hence it is more difficult to find a feasible non-preemptive schedule.

- Figure 3.7 shows how the average number of preemption groups is almost independent of the utilization factor and of the number of tasks, except for a very limited number of tasks (< 10) and a high utilization factor (> 0.8).

- The average number of groups is not only constant but also very close to 2. This means that the application of Preemption Threshold techniques, together with EDF, allows a great reduction in the number of preemption levels and great savings in the amount of RAM needed for saving the task stack frames. RAM reduction in the order of 3 to 16 times less the original requirements can easily be obtained.

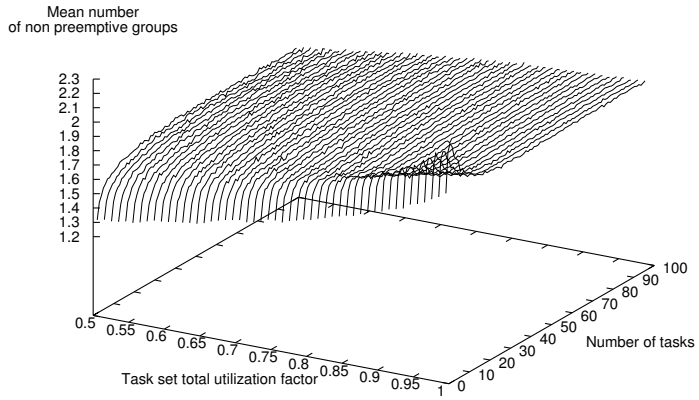

Figure 3.7: Average number of preemption groups.

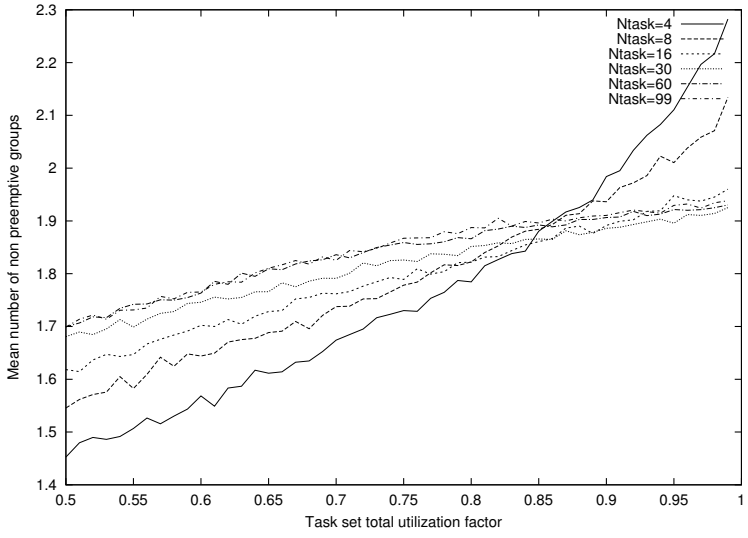

Figure 3.8: Average number of preemption groups for different task set sizes.

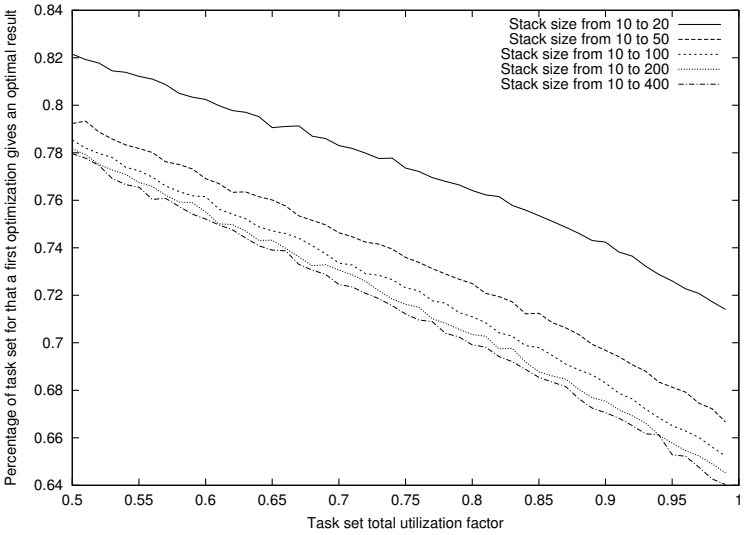

Figure 3.9: Ratio of improvement given by my optimization algorithm.

In Figure 3.9, I compare the optimization algorithm presented in [58] (which does not take into account the stack frame size of the tasks) and the algorithm, to show the improvement in the optimization results. The figure shows the fraction of experiments where the optimal solution has been found by the original algorithm. The ratio appears as a function of the system load and for different stack sizes. In most cases (from 60% to 80%), the algorithm proposed in [58] finds the optimal partition of the task set in preemption groups. This ratio decreases as the load increases and as the range of the stack size requirements is widened.

Chapter 4

Homogeneous multiprocessors architectures

This chapter describes the design of the MSRP algorithm, a scheduling algorithm for homogeneous multiprocessors. MSRP allows tasks to share the stack on the same CPU reducing the memory footprint needed on the global system, and allows the use of local and global resources without jeopardizing the real-time scheduling guarantees imposed in the design of the real-time application. The algorithm can be easily extended to use preemption thresholds to further reduce stack usage. An allocation algorithm is also proposed to allow tasks to be statically mapped on the various processors with the final goal of reducing the overall memory required by the system.

This chapter also contains an experimental evaluation that compares the performance of my algorithm with a solution based on Rate Monotonic and MPCP in the context of the Janus multiple processor architecture. I report on two sets of experiments: the first addresses a range of generic task configurations to see if one of the algorithms can clearly outperform the other. The results show MSRP to be better for random task periods but are probably not conclusive. Later, I focus on a more application-specific (also more restrictive) architecture design representing a typical automotive application: a power-train controller. In this case, MSRP clearly performs better. The performance gap between the two policies can be further increased when considering that MSRP is much simpler to implement, it has a lower overhead, and it allows RAM memory optimization.

This chapter is composed by three sections: Section 4.1 introduces the terminology and the basic assumptions made in this chapter, and recalls the basics of MPCP scheduling; Section 4.2 describes the MSRP algorithm and the related schedulability analysis; Section 4.3 describes the allocation algorithm; Section 4.4 presents a comparison between MPCP and MSRP, and finally Section 4.5 describes an evaluation of MSRP, and its comparison with MPCP on synthetic loads and on an automotive application scenario.

4.1 Background

This section is divided in two parts: the first part of this section introduces the terminology used through this chapter, and it is an extension of the terminology described in Section **??**; the second part recalls the MPCP scheduling algorithm that will be later used for a comparison with MSRP.

4.1.1 Basic assumption and terminology

The system consists of a set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of real time tasks to be executed on a set $\mathcal{P} = \{P_1, \ldots, P_m\}$ of processors. The subset of tasks assigned to processor P_k will be denoted by $T_{P_k} \subset \mathcal{T}$.

A real time task τ_i is a infinite sequence of jobs (or instances) $J_{i,j}$. Every job is characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$ and a deadline $d_{i,j}$ and a priority p_i.

A task can be periodic or sporadic. A task is *periodic* if the release times of two consecutive jobs are separated by a constant *period*; a task is *sporadic* when the release times of two consecutive job are separated by a variable time interval, with a lower bound, also called minimum interarrival time.

Without loss of generality, I use the same symbol θ_i to indicate the period of a periodic task and the minimum interarrival time of a sporadic task τ_i. In the following a task will be characterized by a worst case execution time $C_i = \max\{c_{i,j}\}$ and a period θ_i. I assume that the relative deadline of a task is equal to θ_i: thus, $d_{i,j} = r_{i,j} + \theta_i$.

Tasks can access mutually exclusive resources through critical sections. Let $\mathcal{R} = \{\rho^1, \ldots, \rho^p\}$ be the set of shared resources. The k–th critical section of task τ_i on resource ρ^j is denoted by ξ_{ik}^j and its maximum duration is denoted by ω_{ik}^j.

Finally, I suppose that tasks have already been allocated to processors. Depending on this allocation, resources can be divided in *local* and *global* resources. A local resource is used only by tasks belonging to the same processor, whereas a global resource is used by task belonging to different processors. A critical section protecting a global resource is called *global critical section* or *gcs*.

4.1.2 The MPCP Multiprocessor Priority Ceiling Protocol

The *Multiprocessor Priority Ceiling Protocol* (MPCP) has been proposed by Rajkumar in [52] for scheduling a set of real-time tasks with shared resource on a multi-processor. It extends the Priority Ceiling Protocol [59] for global resources. Since this policy is the term of comparison for the MSRP policy I will spend some extra time discussing its features.

If tasks block on semaphores protecting global resources, the concept of blocking needs to include also *remote blocking* (when a job has to wait for the execution of a task of any priority assigned to another processor.) MPCP extends the priority ceiling protocol to multiprocessor systems with the assumption that tasks are statically bound to processors and scheduled according to the rate monotonic policy.

The goal of MPCP is to bound the remote blocking duration of a job as a function of the duration of critical sections of other jobs and not as a function of the duration of non-critical code. As a direct consequence, it is necessary that global critical sections are assigned a ceiling that is higher than the priority of any other task in the system. If p_H is the highest priority among all tasks, a priority of $p_H + 1 + \max_i\{p_i \mid \tau_i \quad \text{uses} \quad \rho^k\}$ is the priority ceiling for the semaphore protecting the global resource ρ_k. Other important design choices of MPCP are the following:

- jobs are suspended when they try to access a locked gcs;
- when a higher priority task is blocked on a global critical section local tasks can be executed and may even try a lock on local or global critical sections;
- when a global resource is released the task waiting on top of the semaphore list is awakened and inherits the priority of the global critical section.

One very important consequence of letting lower priority local tasks execute and possibly inherit the priority of global critical sections is the possibility of priority inversion occurring while a high priority task is blocked on a gcs. Other assumptions are the following: local critical sections do not make nested access to global resources and vice versa, furthermore, nested accesses to global critical sections are prohibited.

MPCP allows for a bounded blocking time and a formula exists for checking the schedulability of real-time tasks. The formula is an adaptation of Formula 2.3 (to be evaluated for each processor) with the only difference that the blocking factor B_i must account for local and global priority inversions. In order to simplify the formulation of the five factors that add up to form the factor B_i the following additional definitions are introduced.

Task τ_i can access local (i.e. allocated on the same processor) or global resources. The number of global critical sections executed by τ_i is n_i^G. $NL_{i,j}$ is the number of jobs with a lower priority than J_i on its processor. $\{J'_{(i)_r}\}$ is the set of jobs on processor P_r with gcs having priority higher than global critical sections that can directly block J_i. $NH_{i,r,j}$ is the number of global critical sections of job $J_j \in \{J'_{(i)_r}\}$ with higher priority than a global critical section on processor P_r, which can directly block J_i. $\{nGS_{i,j}\}$ is the set of global semaphores locked by both J_i and J_j. Finally, $NC_{i,j}$ is the number of global critical sections entered by J_j and guarded by elements of $\{nGS_{i,j}\}$.

The blocking time for a job J_i on processor P_j consists of up to five different factors:

$$B_i = B_{i_1} + B_{i_2} + B_{i_3} + B_{i_4} + B_{i_5}$$

where

- $B_{i_1} = n_i^G \omega_i^{local}$ (where ω_i^{local} is the longest critical section accessed by jobs with a priority lower than J_i executing on the same processor), since each time J_i needs a global semaphore may suspend, letting lower priority jobs execute on its processor. These low priority jobs can lock local semaphores and block J_i when it resumes its execution.

- $B_{i_2} = n_i^G \omega_j^{global}$ (where ω_j^{global} is the longest global critical section accessed by jobs with a priority lower than J_i executing on other processor) when job J_i tries to access a global critical section and finds it is accessed by a lower priority job on another processor.

- $B_{i_3} = NC_{i,j}\lceil T_i/T_j \rceil \omega_j^{global}$ for each higher priority job executing on a processor different from P_i and requesting the same global semaphore as J_i.

- $B_{i_4} = NH_{i,r,j}\lceil T_i/T_j \rceil \omega_j^{global}$ for higher priority global critical sections, which can preempt the global critical sections of lower priority jobs directly blocking J_i.

- $B_{i_5} = min(n_i^G + 1, n_k^G)\omega_j^{global}$ each time J_i tries to access a global critical section it can suspend letting lower priority jobs execute on its processor. These jobs can lock or queue on global semaphores and eventually execute at a priority higher than P_i and preempt it when it executes non global code.

4.2 Sharing Resources in Multiprocessors

When considering **multiprocessor** symmetric architectures, I wish to keep the nice properties of EDF and SRP, that is high processor utilization, predictability and perfectly nested task executions on local processors. Unfortunately, the SRP cannot be directly applied to multiprocessor systems.

In this section, I first propose an extension of the SRP protocol to multi-processor systems and a schedulability analysis for the new policy. In the next section, I propose a simulated annealing based algorithm for allocating tasks to processors that minimizes the overall memory requirements.

4.2.1 Multiprocessor Stack Resource Policy (MSRP)

I concentrate my efforts on the policy for accessing global resources. If a task tries to access a global resource and the resource is already locked by some other task on another processor, there are two possibilities:

- the task is suspended (as in the MPCP algorithm);

- the task performs a busy wait (also called *spin lock*).

I want to maintain the properties of the SRP: in particular, I want to let all tasks belonging to a processor to share the same stack. Hence, I choose the second solution. However, the *spin lock time* is wasted time and should be reduced as much as possible (the resource should be freed as soon as possible). For this reason, when a task executes a critical section on a global resource, its priority is raised to the maximum priority on that processor and the critical section becomes non-preemptable.

In order to simplify the implementation of the algorithm, the amount of information shared between processors is minimal. For this reason, the priority assigned to a task when

accessing resources does not depend on the status of the tasks on other processors or on their priority. The only global information is the status of the global resources.

The MSRP algorithm works as follows:

- For local resources, the algorithm is the same as the SRP algorithm. In particular, I define a preemption level for every task, a ceiling for every local resource, and a system ceiling Π_k for every processor P_k.
- Tasks are allowed to access local resource through nested critical sections. It is possible to nest local and global resources. However, it is not possible to nest global critical sections, otherwise a deadlock can occur.
- For each global resource, every processor P_k defines a ceiling greater than or equal to the maximum preemption level of the tasks on P_k.
- When a task τ_i, allocated to processor P_k accesses a global resource ρ^j, the system ceiling Π_k is raised to ceil(ρ^j) making the task non–preemptable. Then, the task checks if the resource is free: in this case, it locks the resource and executes the critical section. Otherwise, the task is inserted in a FCFS queue on the global resource, and then performs a busy wait.
- When a task τ_i, allocated to processor P_k, releases a global resource ρ^j, the algorithm checks the corresponding FCFS queue, and, in case some other task τ_j is waiting, it grants access to the resource, otherwise the resource is unlocked. Then, the system ceiling Π_k is restored to the previous value.

Example. Consider a system consisting of two processors and five tasks as shown in Figure 4.1. Tasks τ_1, τ_2 and τ_3 are allocated to processor P_1: task τ_3 uses local resource ρ^1, task τ_2 uses resources ρ^1 and ρ^2 through nested critical sections, and τ_1 does not use any resource. Tasks τ_4 and τ_5 are allocated to processor P_2: task τ_4 uses the global resource ρ^1 and τ_5 does not uses resources. The parameters of the tasks are reported in Table 4.1. The ceiling for resource ρ^1 is 2. The ceiling for resource ρ^2 on processor P_1 is 3, and on processor P_2 is 2. A possible schedule is shown in Figure 4.2. Notice that:

- At time $t = 3$, task τ_2 is blocked because its preemption level $\lambda_2 = 2$ is equal to the current system ceiling $\Pi_1 = 2$ on processor P_1.
- At time $t = 5$, task τ_3 locks resource ρ^2 and raises the system ceiling Π_1 to 3.
- At time $t = 6$, task τ_4 tries to access the global resource ρ^2 which is currently locked by τ_2. Thus, it raises the system ceiling of processor P_2 to 2 and performs a busy wait.
- At time $t = 7$, both τ_1 and τ_5 are blocked, because the system ceilings of the two processors are set to the maximum.

	C_i	λ_i	ω_{ij}^1	ω_{ij}^2	ts_i	C_i'	B_i^{local}	B_i^{global}
τ_1	2	3	0	0	0	2	0	7
τ_2	6	2	2	0	0	6	9	7
τ_3	11	1	9	4	3	14	0	0
τ_4	7	1	0	3	4	11	0	0
τ_5	2	2	0	0	0	2	0	7

Table 4.1: The example task set.

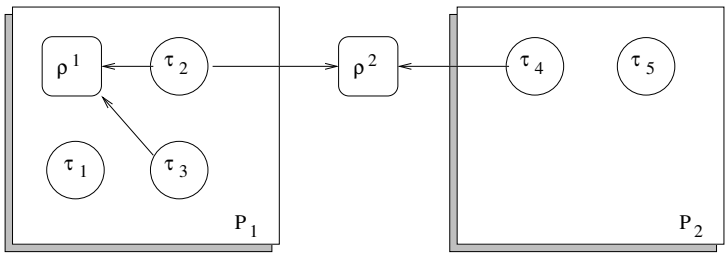

Figure 4.1: Structure of the example.

- At time $t = 8$, task τ_3 releases the global resource ρ^2 and task τ_4 can enter the critical section on ρ^2. At the same time, the system ceiling of processor P_1 is set back to 2, and task τ_1 can make preemption.

4.2.2 Schedulability analysis of the MSRP

First, I give an upper bound on the time that task τ_i, allocated to processor P_k, can spend waiting for a global resource ρ^j. In the following, I refer to this time as *spin lock time* and denote it as $spin(\rho^j, P_k)$.

Lemma 1 *The spin lock time that every task allocated to processor P_k needs to spend for accessing a global resource $\rho^j \in \mathcal{R}$ is bounded from above by:*

$$spin(\rho^j,\, P_k) = \sum_{p \in \{\mathcal{P} - P_k\}} \max_{\tau_i \in T_p,\, \forall\, h} \omega_{ih}^j.$$

Proof.
On each processor, only one task can be inside a global critical section or waiting for a global resource. In fact, when a task tries to access a critical section on a global resource, it first raises the system ceiling to the maximum possible value, becoming non–preemptable. Tasks that are waiting on a global critical section are served in a FCFS order: hence, a task allocated to P_k that tries to access ρ^j, has to wait for at most the duration of the longest global critical section on ρ^j for each processor $p \neq P_k$. This condition does not depend on the particular task on processor P_k. Hence, the lemma follows. □

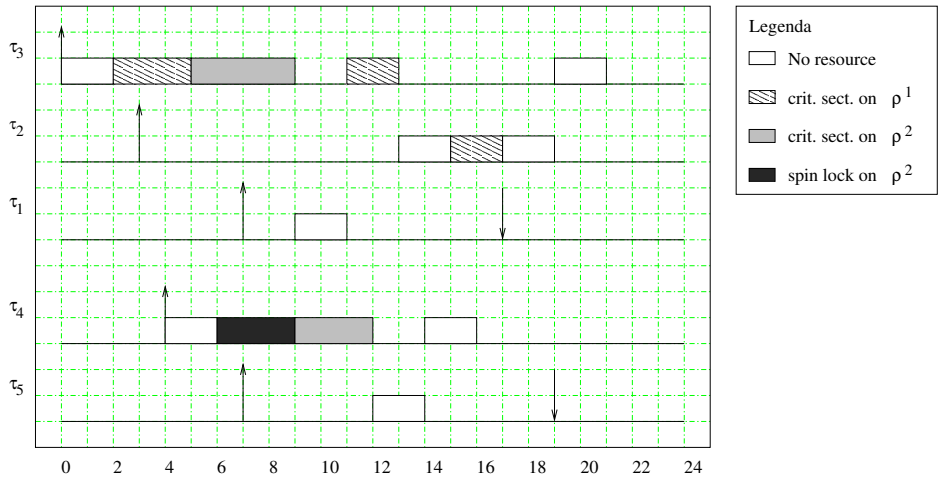

Figure 4.2: An example of schedule produce by the MSRP on two processors.

Basically, the spin lock time increments the duration ω_{ih}^{j} of every global critical section ξ_{ih}^{j}, and, consequently, the worst case execution time C_i of τ_i. Moreover, it also increments the blocking time of the tasks allocated to the same processor with a preemption level greater than λ_i.

I define $totalspin_i$ as the maximum total spin lock time experienced by task τ_i. From the previous lemma,

$$totalspin_i = \sum_{\xi_{ih}^{j}} spin(\rho^j,\, P_i)$$

I also define the *actual worst case computation time* C_i' for task τ_i as the worst case computation time plus the total spin lock time:

$$C_i' = C_i + totalspin_i$$

Now, I demonstrate that the MSRP maintains the same basic properties of the SRP, as shown by the following theorems.

Lemma 2 *When task τ_j starts executing:*

1. *all the local resources required by the τ_j are unlocked;*

2. *all the local resources required by every task that can preempt τ_j are unlocked.*

Proof.
By contradiction.

1. Suppose that, after τ_j starts executing, it needs resource ρ^k which is locked by some other task. Since τ_j is executing, $\lambda_j > \Pi_s$. Moreover, since τ_j needs ρ^k, $ceil(\rho^k) \geq \lambda_j$. But ρ^k is locked, so $\Pi_s \geq ceil(\rho^k) \geq \lambda_j$, and this is a contradiction.

2. Suppose that at time t a task τ_H preempts τ_j, and that it needs a local resource ρ^k which is locked. By hypothesis, $\lambda_H > \lambda_j > \Pi_s$ (because τ_H can preempt τ_j) and $\Pi_s \geq ceil(\rho^k)$ (because ρ^k is locked). The lemma follows because τ_H uses ρ^k, that implies $ceil(\rho^k) \geq \lambda_H$.

□

Theorem 3 *Once a job starts executing it cannot be blocked, but only preempted by higher priority jobs.*

Proof.

I prove the theorem by induction. Suppose there are n tasks that can preempt τ_j.

If $n = 0$, no task can preempt τ_j. Since when τ_j started executing $\lambda_j > \Pi_s(t)$, Lemma 2 guarantees that all the resources required by task τ_j are free, so it can run to completion without blocking.

If $n > 0$, suppose that a task τ_H preempt τ_j. By induction hypothesis τ_H cannot be blocked, so when it finishes it will release all the resources that it locked, and the task τ_j will continue until the end since it has all the resources free. □

Note that a job can be delayed before starting execution by the fact that the system ceiling is greater than or equal to its preemption level. This delay is called *blocking time*. The following theorem gives an upper bound to the blocking time of a task.

Theorem 4 *A job can experience a blocking time at most equal to the duration of one critical section (plus the spin lock time, if the resource is global) of a task with lower preemption level.*

Proof.

By contradiction. Suppose that a task τ_j is blocked for the duration of at least 2 critical sections corresponding to resources ρ^1 and ρ^2. First, suppose that ρ^1 is a local resource (ρ^2 can be either local or global). It must be the case that there are two lower priority tasks (τ_{L1} and τ_{L2}). The first task locked a resource ρ^1 and, while in critical section, it is preempted by τ_{L2} that locked another resource ρ^2. While τ_{L2} is still in critical section, τ_j arrives, and it has to wait for ρ^1 and ρ^2 to be free. This scenario cannot happen, because I have that $ceil(\rho^1) \geq \lambda_j > \lambda_{L2}$ (since τ_j uses ρ^1 and preempted τ_{L2}), and $\lambda_{L2} > ceil(\rho^1)$ (since τ_{L2} preempted τ_{L1}, when τ_{L1} locked ρ^1).

Now suppose ρ^1 is a global resource and consider the previous scenario. When τ_{L1} locked ρ^1, τ_{L1} become non-preemptable, so τ_{L2} cannot preempt τ_{L1}, and any scenario like that cannot apply.

Finally note that every global critical section has a length that is composed by the critical section itself, plus the spin-lock time due to the access to global resources. □

It is noteworthy that the execution of all the tasks allocated on a processor is perfectly nested (because once a task starts executing it cannot be blocked), therefore all tasks can share the same stack.

For simplicity, the blocking time for a task can be divided into blocking time due to local and global resources. In addition, if I consider also the preemption threshold mechanism, I have to take into account also the blocking time due to the pseudo-resources:

$$B_i = max(B_i^{local}, B_i^{global}, B_i^{pseudo})$$

where B_i^{local}, B_i^{global} and B_i^{pseudo} are:

Blocking time due to local resources: This blocking time is equal to the longest critical section ξ_{jh}^k among those (of a task τ_j) with a ceiling greater than or equal to the preemption level of τ_i:

$$B_i^{local} = \max_{j,h,k}\{\omega_{jh}^k \mid (\tau_j \in T_{P_i}) \wedge (\rho^k \text{ is local to } P_i) \wedge (\lambda_i > \lambda_j) \wedge (\lambda_i \leq ceil(\rho^k))\}$$

Blocking time due to global resources: Assume the task τ_i, assigned to processor P_i, is blocked by a task τ_j ($\lambda_j < \lambda_i$) which is assigned to the same processor P_i, and which is waiting for, or it is inside to, a global critical section ξ_{jh}^k. In this case, the blocking time for task τ_i is,

$$B_i^{global} = \max_{j,h,k}\{\omega_{jh}^k + spin(\rho^k, P_i) \mid (\tau_j \in T_{P_i}) \wedge (\rho^k \text{ is global}) \wedge (\lambda_i > \lambda_j)\}$$

Blocking time due to pseudo resources: As explained in the previous sections, this blocking time is due to the fact that a task τ_i can be mutually non–preemptive with other tasks on the same processor: here, the only difference with the SRPT is that I have to consider the *actual worst case execution time* instead of the worst case execution time.

$$B_i^{pseudo} = \max_{\tau_j \in T_{P_i}} \{C'_j \mid \lambda_i > \lambda_j \wedge \lambda_i \leq \gamma_j\}$$

Theorem 5 *Suppose that tasks on processor P_k are ordered by decreasing preemption level. The schedulability test is as follows:*

$$\forall\, P_k \in \mathcal{P} \quad T_{P_k} = \{\tau_1, \tau_2, \ldots, \tau_{n_k}\} \quad \forall i = 1, \ldots, n_k \quad \sum_{l=1}^{i} \frac{C'_l}{\theta_l} + \frac{B_i}{\theta_i} \leq 1 \qquad (4.1)$$

Proof.

Consider a generic task τ_i on processor P_i. To be schedulable under MSRP with preemption thresholds, it must be schedulable under EDF considering all blocking times and spin locks. Hence, a guarantee formula for task τ_i can be written as

$$\sum_{l=1}^{i-1} \frac{C'_l}{\theta_l} + \frac{C'_i}{\theta_i} + \frac{B_i}{\theta_i} \leq 1$$

where the first part is the bandwidth stolen by tasks that preempted τ_i, and C'_l consider also the effect of the wasted bandwidth of the spin-lock time for each preempter task. The second part accounts for the execution time and the spin-lock time of the task to be guaranteed. The third part accounts for the largest blocking time experienced by τ_i due to the use of resources by lower priority tasks. □

In the same way, I can rewrite Equation (2.4) as follows:

$$\begin{array}{c} \forall\, P_k \in \mathcal{P} \quad T_{P_k} = \{\tau_1, \tau_2, \ldots, \tau_{n_k}\} \quad \forall i, \quad 1 \leq i \leq n_k \\ \forall L, \quad \theta_i \leq L \leq \theta_{n_k} \quad L \geq \sum_{l=1}^{i} \left\lfloor \frac{L}{\theta_l} \right\rfloor C'_l + B_i \end{array} \tag{4.2}$$

Please note that the blocking factor influences only one element of the guarantee formula, whereas the spin lock time influences both the blocking time and the worst case execution time. This implies that, when designing an allocation algorithm, one of the goals is to reduce the spin lock time as much as possible. Another noteworthy observation is that, using the MSRP, each processor works almost independently from the others. In particular, it is possible to easily apply this algorithm to non–homogeneous multiprocessor systems.

Example. For the task set of the previous example, the total spin lock time ts_i, the actual worst case execution time C'_i, the local and global blocking times are reported in Table 4.1.□

The main differences between the MSRP and the MPCP are the following:

- Unlike MPCP, with the MSRP it is possible to use one single stack for all the tasks allocated to the same processor.
- The MPCP is more complex and difficult to implement than the MSRP. In fact, the MSRP does not need semaphores or blocking queues for local resources, whereas global resources need only a FIFO queue (an efficient implementation can be found in [23]).
- The MSRP, like the SRP, tends to reduce the number of preemptions in the systems, hence there is less overhead. However, this comes at the cost of a potentially dangerous spin lock time.

I'll describe with more details the differences between the two algorithms in Section 4.4.

4.3 Optimizing stack usage in Multiprocessors

Sections 3.1 and 4.2 provide the basis for the implementation of run-time mechanisms for global and local resource sharing in multiprocessor systems. Given a task allocation, the

policies and algorithms presented in Section 3.2 allow to search for the optimal assignment of preemption thresholds to tasks and to selectively group tasks in order to reduce RAM consumption. However, the final outcome depends on the quality of the decisions taken in the task allocation phase. Moving one task from one processor to another can change the placement of (some of) the shared resources accessed by it (some global resources become local and vice versa) and the final composition of the non–preemptive groups on each processor. Unfortunately, the task allocation problem has exponential complexity even if I limit ourselves to the simple case of deadline-constrained scheduling.

A simulated annealing algorithm is a well-known solution approach to this class of problems. Simulated annealing techniques (SA for short) have been used in [62, 58] to find the optimal processor binding for real-time tasks to be scheduled according to fixed-priority policies, in [48] to solve the problem of scheduling with minimum jitter in complex distributed systems and in [63] to assign preemption thresholds when scheduling real-time tasks with fixed priorities on a uniprocessor. In the following I show how to transform the allocation and scheduling problem which is the subject of this chapter into a form that is amenable to the application of simulated annealing. The solution space S (all possible assignments of tasks to processors) has dimension p^n where p is the number of processors and n is the number of tasks. I am interested in those task assignments that produce a feasible schedule and, among those, I seek the assignment that has minimum RAM requirements. Therefore I need to define an objective function to be minimized and the space over which the function is defined.

The SA algorithm searches the solution space for the optimal solution as follows: a transition function TR is defined between any pair of task allocation solutions $(A_i, A_j) \in S$ and a neighborhood structure S_i is defined for each solution A_i containing all the solutions that are reachable from A_i by means of TR. A starting solution A_0 is defined and its cost (the value of the objective function) is evaluated. The algorithm randomly selects a neighbor solution and evaluates its cost. If the new solution has lower cost, then it is accepted as the current solution. If it has higher cost, then it is accepted with a probability exponentially decreasing with the cost difference and slowly lowered with time according to a parameter which is called temperature.

I will not explain in detail the SA mechanism and why it works in many combinatorial optimization problems. Please refer to [1] for more details.

The transition function consists in the random selection of a number of tasks and in changing the binding of the selected tasks to randomly selected processors. This simple function allows to generate new solutions (bindings) at each round starting from a selected solution. Some of the solutions generated in this way may be non schedulable, and therefore should be eventually rejected. Unfortunately, if non–schedulable solutions are rejected before the optimization procedure is finished, there is no guarantee that the transition function can approach a global optimum. In fact, it is possible that every possible path from the starting solution to the optimal solution requires going through intermediate non–schedulable solutions (see Figure 4.3).

If non-schedulable solutions are acceptable as intermediate steps, then they should be evaluated very poorly. Therefore, I define a cost function with the following properties:

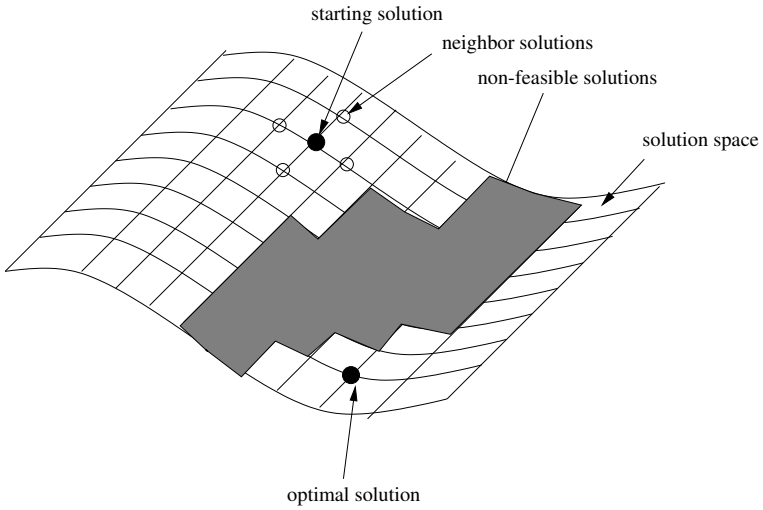

Figure 4.3: Non feasible solutions must be accepted in order to reach the optimal solution.

- schedulable solutions must always have energy lower than non–schedulable solutions;
- the energy of non schedulable solutions must be proportional to the maximum excess utilization resulting from the evaluation of formula (4.1) for non–schedulable tasks;
- the energy of schedulable solution must be proportional to the worst case overall RAM requirements for stack usage.
- If $TotalStack$ is the overall stack requirement, obtained by adding up the stack requirements of all tasks, and $OptStack$ is the overall stack requirement, evaluated for schedulable sets after the computation of optimal preemption thresholds and task groups (see Section 3.2), then the cost function is the following:

$$\begin{cases} \max_{\forall \tau_i} \left(\sum_{k=i}^{n} \frac{C'_k}{T_k} + \frac{B_i}{T_i} \right) * TotalStack & \text{non schedulable assignment} \\ TotalStack + \Delta * (OptStack - TotalStack) & \text{schedulable assignment} \end{cases}$$

  When the assignment is non schedulable, I use the result of the guarantee test (Equation 2.3) as an index of schedulability. In fact, as the system load, blocking time or spin-lock time increase, the system becomes *less* schedulable. When the assignment is schedulable, the cost function does not depend on processor load but returns a value that is proportional to the reduction of stack with respect to the total stack requirement.

The Δ factor estimates the average ratio between the stack requirements before task grouping and the stack requirements after optimization and is defined as:

$$\Delta = \frac{ncpu * meanstack * meangroups}{ntask * meanstack}$$

```
visited = 0;
// generate a first solution using only the Ui
FirstBinPackingAssignment();
Enew = Eold = Energy();
while (C > stopTemp) {
  internal = 0;
  while (internal < maxTries) {
    internal++;
    GenerateNeighbour();
    // for each processor, optimize the stack using SRPT
    if (Schedulable) OptimizeStack();
    Enew = Energy();
    if (Schedulable) {
      // I stop after a maximum number of schedulable solutions
      visited++;
      if (visited == maxVisited) return;
    }
    if (new_solution->Energy < old_solution->Energy)
      Eold = Enew
    else {
      // Upward energy jump
      if (exp((Eold-Enew)/C) >= frand(0,1)) Eold = Enew;
      else Enew = Eold;
    }
  }
  C = C * coolingStep; // Temperature Cooling
}
```

Figure 4.4: Simulated Annealing Algorithm.

where *ncpu* is the number of CPU in the system, *meanstack* is the mean stack value of all tasks, *meangroups* estimates the typical number of preemption groups on a uniprocessor. The Δ factor has been introduced to smooth the steep increase in the cost function when going from schedulable solutions to non–schedulable assignments. This improves the chances for the simulated annealing algorithm to escape from local minima (which might require accepting a non–schedulable solution).

The experimental results in Section 4.5 show the effectiveness of the SA-based binding algorithm when simulating task sets scheduled on 4-processor system-on-a-chip architectures.

4.4 Comparing MSRP and MPCP

One of the possible drawbacks of the MSRP policy is the cost of spin locking in multiprocessor real-time systems when compared to other policies. In contrast, the multiprocessor priority ceiling protocol or MPCP, probably the best known policy for bounding blocking time in a predictably way in multiprocessor systems, avoids spin-locking, but does not allow sharing the stack space of tasks. Furthermore, it requires a non trivial run-time support, which results in greater overhead when compared to the implementation of MSRP.

In order to settle this dispute, I performed experiments comparing MSRP with MPCP in

Janus (see Section 4.5). The experiments are in two stages. In the first stage, the simulator evaluates the schedulability of a number of generic task sets to see if one of the algorithms can clearly outperform the other. The results are not conclusive, except that (as expected) MSRP is better when considering few global resources and short critical sections. In the second stage I focus on a domain-specific example: a task set implementing a power-train controller, which is the representative of a typical automotive application. For this case, MSRP clearly outperforms MPCP, proving the viability of a spin-lock based approach for sharing resources on the Janus platform.

The following subsections discusses a comparison on the blocking times and on the implementation of the two algorithms.

4.4.1 Comparing the blocking factors of MSRP and MPCP

The blocking factors $B_{i_1}, \ldots B_{i_5}$ of MPCP are the result of a worst case analysis and can be reduced by carefully examining the task set at hand. Nonetheless the guarantee formula is clearly extremely complicated. Consider also that PCP requires keeping track of local and global priority ceilings and the previous formula holds if the period enforcing technique (described in [52]) is used.

If, on the other hand, MSRP is used, I can expect to waste more local processor time due to the use of spin locks when trying to lock global resources. The guarantee formula of MSRP is simpler since I do not have to account for the events that cause the blocking factors B_{i_1} and B_{i_5} which are the consequence of suspending a task when trying to access a locked critical section.

At first sight, it would appear that, whenever global critical sections are sufficiently short, the MSRP approach would perform better (besides being much simpler to implement). On the other hand, MPCP should be better when global critical sections grow larger. I performed a first set of experiments trying to determine where this boundary lies and in what conditions should designers expect MSRP to perform adequately. Following the results of these experiments, I focused the analysis on a power-train application: a typical case study from the automotive domain.

4.4.2 Comparing the implementation of MSRP and MPCP

An implementation of the MPCP protocol can be basically divided in two parts [52]: the implementation of a local priority ceiling protocol and the implementation of the global inter–processor synchronization.

The local part of the protocol implementation can easily be done using a priority ordered Task queue, where the highest priority task in the queue is the running task. Moreover, a list of locked semaphores (ordered by ceiling) has to be maintained to allow the implementation of the inheritance of the priority.

The global part of the protocol subsumes the existence of a shared data structure that records the state of a global mutex. In particular, an ordered queue of the tasks that are blocked on the global resource has to be implemented. The low-level access to that data structure has to be done in mutual exclusion, and that is usually done using a spin-lock

approach (the duration of the spin lock is not accounted into the guarantee equations, since it is bounded by the maximum time needed to handle the internal data structure) or using an inter-processor interrupt. Moreover, to guarantee a bounded blocking time the Period Enforcer technique must be implemented.

When using SRP there is no need to implement semaphores and queues for blocked tasks, and the blocking time experienced by each task can be predictably bound. Furthermore, the SRP allows multiple tasks to share a single stack. For these reasons, the SRP can be implemented with a small overhead and memory occupation. The implementation of MSRP on the Janus platform has been simplified by taking advantage of the fact that there are only two processors contending for the use of global resources. In particular, only one processor at a time can be blocked on a global resource, so FCFS queues are not needed for waiting tasks. Moreover, implementing a spin-lock mechanism on Janus only requires a negligible amount of code. Since all memory is shared between the two processors, a single memory location can be used to synchronize all tasks using the `swpb` ARM instruction.

4.5 Experimental evaluation

The MSRP experimental evaluation has been divided in three parts: first, I performed some simulative evaluation to validate the MSRP allocation algorithm described in section 4.3. Then, a sets of experiments is presented on a range of generic task configurations to see if one of the algorithms (MPCP and MSRP) can clearly outperform the other. Finally, a set of more application-specific experiments on a architecture design representing a typical automotive application are presented.

4.5.1 Multiprocessor experiments

In the first set of experiments, I consider 4 CPU, 40 resources, and 40 tasks. Tasks' periods are randomly chosen between 1 and 1000. The total system load U ranges from 2.76 to 3.96, with a step of 0.2. The stack frame size of each task is a random variable chosen between 10 and 100 bytes. Each task has 0 to 4 critical sections that lock randomly selected resources; the sum of the worst case execution times of the critical section accessed by each single task is in the range of 0-20%, 5-25%, 10-30%, 15-35%, 20-40% of the task worst case execution time (depending on the simulation, see Figure 3.9).

In Figure 4.5 I plot the stack gain ratio between the overall stack requirement before optimization and the stack memory requirement of the solution found by the SA algorithm. In all experimental runs the solution found by the SA routine saves a considerable amount of RAM even when compared to the first schedulable (and optimized for RAM consumption) solution found. The average improvement in 58 runs is 34.6% (min 18%, max 49%).

Running times can be a concern when using a simulated annealing solution. The algorithm can be run in a few hours on modern computers (Figure 4.6 shows typical computation times as a function of the task set size). The execution of the simulated annealing routine takes 6 to 30 hours on an Intel Pentium III 700Mhz to complete the cooling. For example, a typical execution (Total U = 2.76, critical section ratio 0.10 to 0.30) vis-

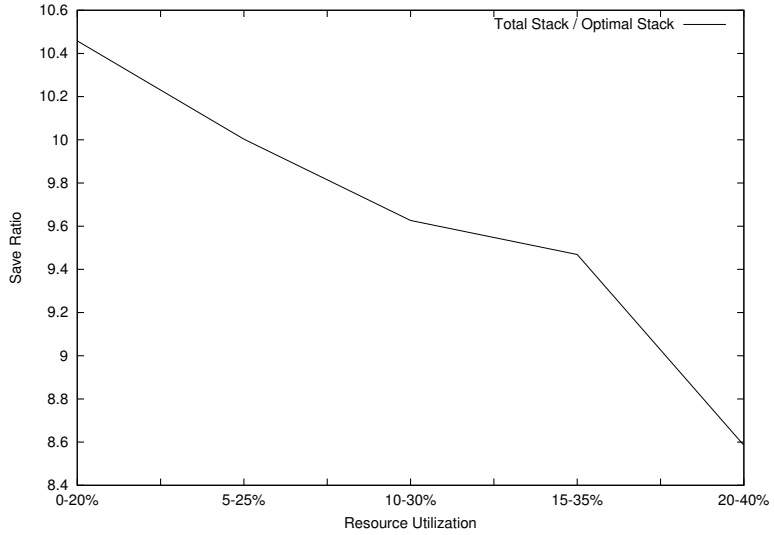

Figure 4.5: Ratio of improvement given by my multiprocessor optimization algorithm when varying the utilization of shared resources.

ited 15,900,000 assignments (one every 4 ms) and found 6,855,560 schedulable solutions. These results are quite acceptable considered that task allocation is a typical design time activity.

4.5.2 MPCP vs. MSRP comparison on generic task sets

In the first set of experiments I compare the performance of the MSRP and MPCP algorithms on a range of task configurations (random load) mapped on the 2 Janus processors.

The experiments consider a set of 6 to 10 tasks statically allocated to each CPU. Depending on the experiments, task periods are chosen randomly between 1 and 100 or by selecting appropriate harmonic values. Harmonic periods are generated in the following way: the period of the first is 1; the period of the next task is given by the period of the previous task multiplied by a random factor between 1 and 4 (ratio 1 has the 30% of probability, ratios 2,3,4 share the remaining 70%).

If U is the system utilization, defined as $U = \Sigma_i c_i/\theta_i$, the total load on each CPU ranges from U_{min} and U_{max}, where U_{min} ranges from 0.025 and 0.925 with steps of 0.025, and U_{max} ranges from Umin+0.025 to 0.95 with steps of 0.025. The number of local resources is always 6 for each processor, plus 6 global resources. The number of critical sections accessed by each task is a random value chosen in the intervals (0,2), (1,4), (2,6) depending on the experiment. Tasks spend a percentage of their computation time into critical sections. The fraction of execution time that is spent in a critical section (local or global) ranges between C_{min} and C_{max}, where C_{min} and C_{max} belong to the set {0.0, 0.5, 0.10, 0.15, 0.2}, and C_{max} is always greater than C_{min}. For each task set I consider a set of 101 possible configurations, obtained considering that the time spent in a critical section is allocated for a percentage x to local critical sections, and for a percentage (1-x) to global

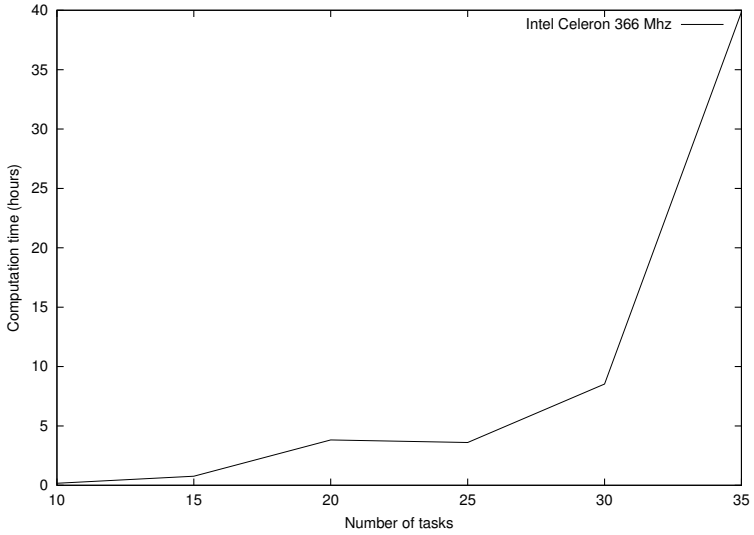

Figure 4.6: Average computation times for the simulated annealing algorithm as a function of the problem size.

critical sections, with x ranging from 0 to 1 with steps of 0.01. On each configuration I check if the MSRP and the MPCP tests can guarantee the task set as schedulable (more than 520 million configurations were tried).

The first set of experiments is performed on task sets where periods are randomly chosen. The graphs show the percentage of tasks sets that can be guaranteed to be schedulable. It is easy to see how the MSRP policy performs better than MPCP mainly because of the higher utilization bound of EDF when compared to Rate Monotonic. For higher utilizations the guarantee rate decreases, most notably for the MPCP solution, where it finally approaches a hyperbolic bound for higher values (the hyperbolic bound for rate monotonic scheduling defined in [15] is used).

In general, in all the experiments on random task periods, MSRP always performed better. Even if this is mostly due to the use of EDF as a task scheduling policy, it is my opinion that this advantage should not be easily dismissed. A comparison which does not give an a priori advantage to MSRP because of the higher schedulability bound of EDF can be obtained by selecting task sets where periods are harmonic, therefore having a utilization bound of 1 for the Rate Monotonic policy. The results for this case (Figure 4.8) show that there is no algorithm performing better on the whole scale of the spectrum (for all the possible percentages of local resources). As one should expect MPCP performs better for a higher percentage of global resources while MSRP is better if a greater percentage of local resources is simulated.

The MPCP curves are always between the minimum and the maximum curves for MSRP. This implies the existence of a crossing point which identifies the percentage of local access to resources that, for each U separates the zone where MPCP performs better from the range where MSRP guarantees an higher percentage of schedulable sets.

To better highlight these regions it is useful to plot the data with a different X axis

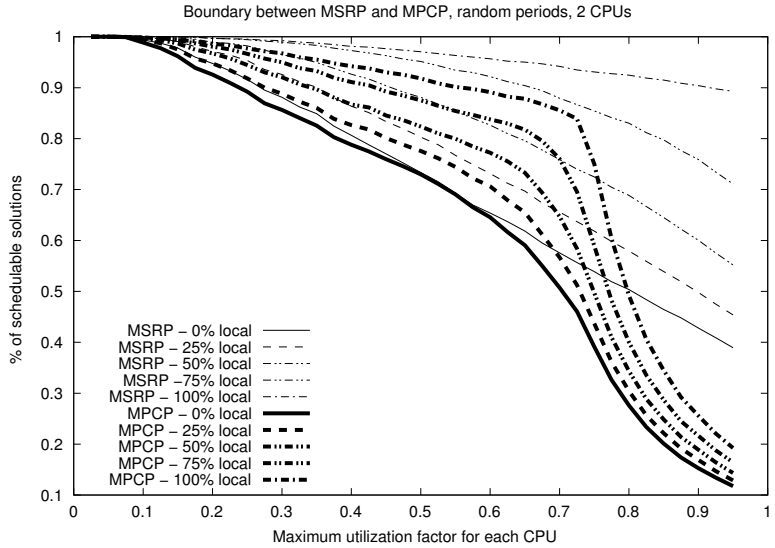

Figure 4.7: Percentage of schedulable solutions, random periods, variable percentage of local resource utilization.

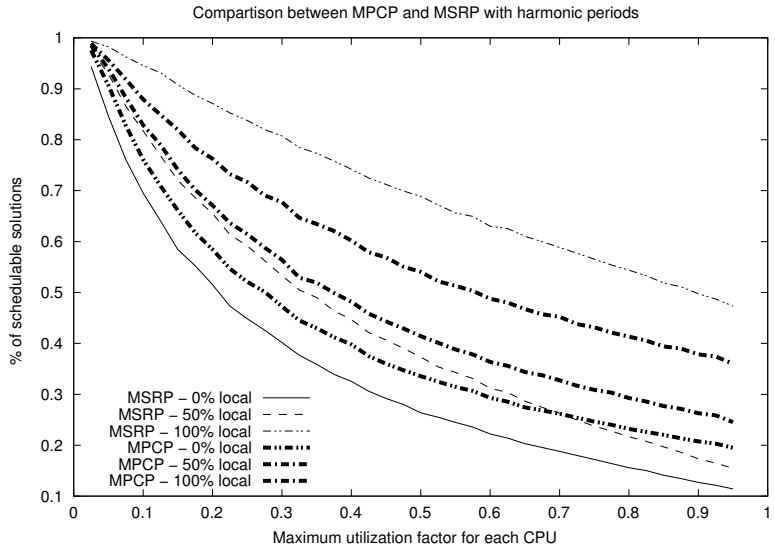

Figure 4.8: Percentage of schedulable solutions, harmonic periods, variable percentage of local resource utilization.

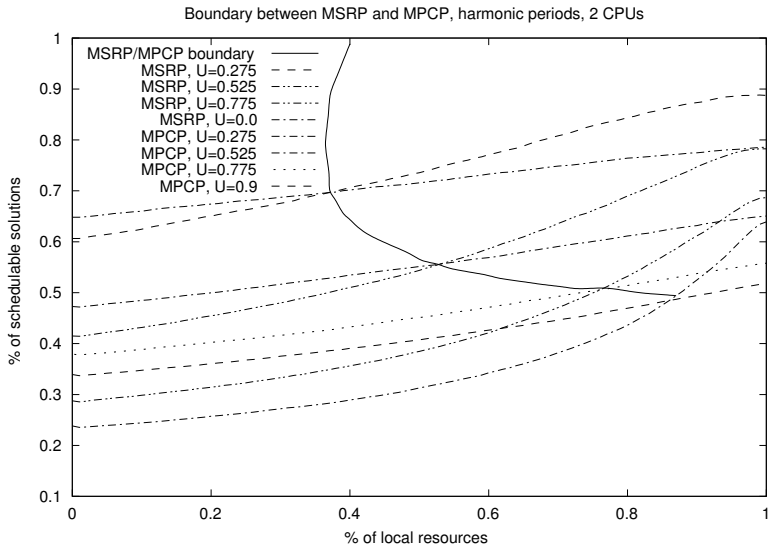

Figure 4.9: Comparison of MPCP and MSRP with the performance boundary (Y=percentage of schedulable solutions, X=percentage of local critical sections).

variable: the percentage of local resource utilization. For example, in Figure 4.9 MSRP outperforms MPCP for high local resource usage, that is when at least 40% of the resource access time is on local resources (the upper right region in Figure 4.9).

It can be noted that, as U increases, the lines decrease (since the system load is greater, fewer schedulable solutions can be found). Moreover, when the use of global resources increases (X axis going to 0) there is a point (the boundary in the figure) where MPCP starts to perform better (which can be explained because the spin locking term influences not only the blocking time, but also the task computation time). A continuous spline, interpolating the crossing points in the figure gives an idea of the boundary between the areas where the two algorithms perform better.

Experiments clearly show how the area where the MSRP protocol performs better increases for a higher use of shared resources. This is a side effect of the reduction of schedulability caused by a higher use of shared resources. In case of Figure 4.10, the lines are not simply splines, but are the result of comparative experiments for points of the plane (U, % of local resources).

4.5.3 MPCP vs. MSRP comparison on a power-train case

4.5.3.1 The Power-train Control Application

The goal of power-train control systems is to offer appropriate driving performance (e.g. driveability, comfort, safety) while minimizing fuel consumption and pollutant emissions. In an engine management system, the fuel injection and air intake are controlled to produce the desired mix to be transformed, by the combustion process, in torque and emissions.

The combustion process takes place in the cylinders and the starting time is controlled by the sparks generated from the spark plugs. The produced torque is then applied to the

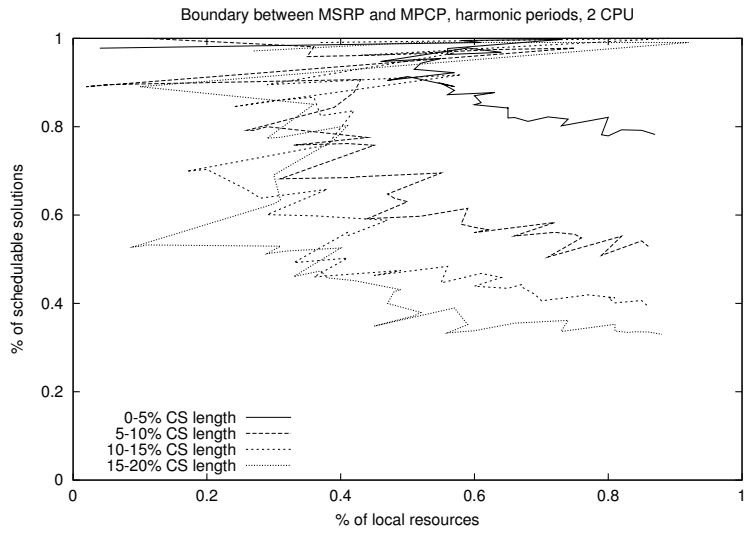

Figure 4.10: Boundary obtained considering 2 CPUs with various resource usages.

power-train, which is controlled by the gear selection and clutch position. The final result is the force applied, through the wheels, to the entire vehicle. Driveability is an informal measure of how favorably this force is perceived by the driver under his/her action.

The control strategy goal is achieved by means of several control inputs such as throttle position, fuel injection, spark ignition, gear selection and clutch position. Fuel injection, spark ignition and part of the gear-box control are angle-based, i.e. they must be synchronized with the *engine position*[1] *or drive-line angle*. The other control variables do not have these synchronization constraints and are called time based. To compute the engine position, the engine has two sensors (the crankshaft and cam-shaft toothed wheel sensors) providing two angular references used for injection and ignition synchronization. Synchronization is essential for timing the opening of fuel injectors and the ignition of the spark plugs. The supplied torque and the emitted pollutants depend crucially on the accuracy of these operations.

In order to evaluate the performance of the resource sharing algorithms for the target application, I used a model view representing the thread architecture. The view must define the typical abstractions used in schedulability analysis, such as the real-time threads, each characterized by its activation mode (periodic or sporadic), and its timing characteristics (such as the WCET) and the shared resources used by the threads, with the execution times of the methods called upon them.

An extremely short introduction to a typical automotive software development process can help understanding the nature of the application threads and the relationships between them and the set of shared resources. The threads running under the control of the RTOS are the result of a software development process, which starts from the definition of a high level model (usually a functional model obtained from a tool like Simulink) and continues

[1] For engine position I mean both the angular position of the crankshaft and the working phase (i.e. intake, compression, expansion or exhaust) of each cylinder.

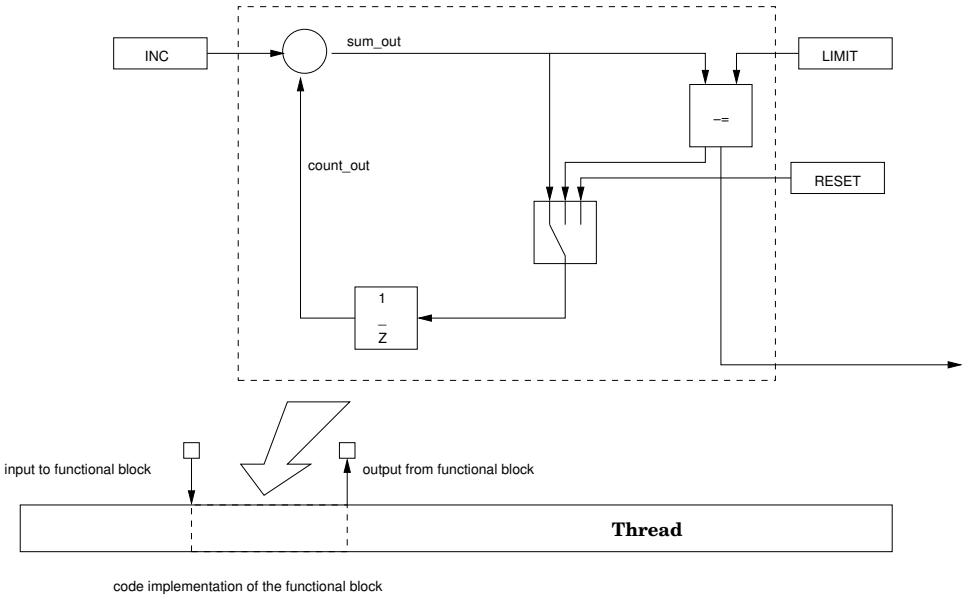

Figure 4.11: A thread contains the implementation of several functional blocks

with the automatic production of software code implementing the functional blocks defined in the model. The code implementing the functional blocks is statically scheduled in the context of a thread (see figure 4.11). As a result, each thread performs many read and write accesses to the input and output variables or devices defined by the function blocks implemented in it. These sets of input and output variables/devices are possibly implemented as shared resources and the resulting graph of use relationships among tasks and resources is quite densely connected, with each task accessing many resources.

Unfortunately, the exact specification of the application architecture and its performance/timing data (as implemented in the current version of the controller) are considered extremely sensitive industrial property. Furthermore, the current implementation is on a single-cpu controller and it is expected that it will change when ported to the new Janus architecture. Given this restriction, the analysis had to settle for realistic data on the application threads and resources, which could be used for measuring the quality of the algorithms and comparing their performance. The model view I analyzed can be considered a good abstraction of the current implementation and the starting point for evaluating algorithms and solutions (on the worst-case side) for the upcoming Janus implementation.

Based on the analysis of the current implementation and based on the number and complexity of the function points in the new implementation, I considered from 10 to 20 periodic tasks and from 2 to 6 aperiodic tasks with periods ranging from 1 ms to about 100 ms (given the dependency from specification requirements, such as the maximum rpm of the engine, the rate requirements should be considered quite reliable data). Tasks are divided into 3 classes:

high rate: from 1 ms to 5 ms.

medium rate: from 5 to 20 ms.

low rate: from 50 to 100 ms.

Tasks are distributed among the three classes in this way: 50% of the tasks belong to the medium rate type, the other types account for 25% of the tasks each. The processors are quite heavily utilized, utilization ratios above 70% should be expected for each processor. The fraction of the processor utilization required by each class is the following: 50% of the processor time is used to serve high rate tasks, 30% is allocated to the medium rate class, and the last 20% will be used to execute the low rate class. As for resources, tasks share both physical and logical resources. The Janus physical resources shared by tasks are the I/O channels for Analog to Digital (A/D) conversion and the serial ports that are used for communication with the outside systems. The logical resources consist of memory buffers for communication. Both kind of resources are protected by priority ceiling (MSRP or MPCP) semaphores.

Access to the shared I/O channels can be characterized as follows: the serial bus is expected to work at high speeds (the target rate is 500 kb/s) transmitting one byte at a time, corresponding to about 50 μs of required access time. The serial communication will be used only once for each task. Two serial ports (UARTs) are implemented in Janus. The A/D conversion device can be used multiple times, from 5 up to a maximum of 10 times for each task. The A/D access time is dominated by the setup time, resulting in critical sections of about 5 μs.

Tasks are expected to communicate through shared memory resources, which are of two types: switched (no-wait) buffers and one-position mailboxes. Resources of the first type do not actually need semaphores, since the pointer swapping instruction is provided as an atomic instruction by the ARM processors and only one writer task is expected for these resources. As for the second type of resources, tasks are expected to cooperate by exchanging information on their internal state as a set of shared variables. These sets consist of 20 to 50 sets, each one containing between 10 and 300 variables, which must be written and read atomically, in order to keep the state consistent. Each variable is implemented using a 16 or 32 bit data type.

These shared memory requirements actually represent a worst case approximation and in no case will the overall memory allocated to shared variables exceed an architecture-specific bound of 16 KBytes. Write operations are expected to affect all the variables in the data set, and read operations only address a subset (uniformly distributed between 10% and 100%) of the variables in the set. Each task is expected to perform from 3 up to 20 read accesses and from 2 to 8 write accesses to the sets of variables. Finally, in order to ease the schedulability of the task set, a large percentage of the resources accessed by high rate tasks is implemented by using switched buffers (when allowed by the communication semantics).

4.5.3.2 Experimental setup

The results of the experiments on generic task sets can hardly be considered conclusive for the harmonic periods case. In the second set of experiments I focused on the power-train specifications, to see if more knowledge could be gained when restricting the application

domain.

The task sets used for the evaluation of the power-train case study were created using the abstract architecture specification defined in Section 4.5.3.1.

Utilization A set of experiments was performed for different values of the system (2 CPUs) utilization. I considered utilization values from 1.4 to 1.96 with steps of 0.04. In the graphs, the utilization value is the variable on the X axis.

Tasks The total number of tasks in each experiment is a random variable with integer values uniformly distributed in the interval $[12, 26]$. Tasks are divided in three subclasses according to their rate of execution. I generate tasks with random periods and with harmonic periods. Periods have integer values (in msecs). Worst case execution times are chosen in a way that the utilization of each class of tasks sums to the desired value for the class. Task allocation is performed by a simulated annealing algorithm (described in [31]).

Resources and Critical Sections Physical resources are modeled as follows. The Janus chip has two serial ports (UARTs). I assume each serial port is allocated to the tasks running on one of the CPUs. In this way the serial port is a resource shared only among local tasks. Resource index 0 is reserved to the "Serial I/O" channel. The critical sections that use the UART are assumed as 50 μs long. Resource 1 is the "A/D converter". The critical sections that use this resource are 5 μs long. The remaining resources are shared "memory resources". Given the requirements of Section 4.5.3.1, each memory resource uses from 20 to 1200 bytes of memory. For the target (ARM-based) Janus platform, the maximum length of a critical section is estimated as $4 + \frac{83-4}{1200-20} * memorysize$ μs[2]. The simulator loop that generates the resource sets stops at the 16 Kbytes limit and the critical sections computed in step 4 are accepted until the total critical section time is lower than the task WCET. Finally, every critical section accessed by a high rate task has a 40% probability to be deleted, to account for the fact that high priority tasks will use switched buffers when possible in order to reduce their blocking time.

4.5.3.3 Results

I ran experiments for increasing processor utilization factors from 1.4 (approximately 0.7 for each CPU) to 2.0. First, sets of tasks with random integer periods were tried. After processing about 6000 task sets generated according to the specifications, I obtained the results shown in Figure 4.12. This time the performance difference between the two algorithms is striking: not a single task set is found schedulable with MPCP and the schedulability ratio provided by MSRP goes down (almost linearly) from about 50% at 1.4 utilization (0.7 for each CPU) to about 0 at 1.98 utilization. In the graph of Figure 4.12, the MPCP curve is not visible since it is completely hidden by the X axis. The situation does not improve significantly for MPCP when the task periods are forced to be harmonic. The MPCP guarantee ratio goes barely up for 1.4 utilization but it is always below 10%. No schedulable set is found under MPCP for utilization values higher than 1.7. In contrast, MSRP continues to deliver an acceptable performance going from more than 40% of schedulable sets at 1.4 utilization, to virtually no schedulable solution at 1.8/1.9 utilization.

[2] 4 to 83 μs is the expected time to write the data on a 40Mhz Janus platform.

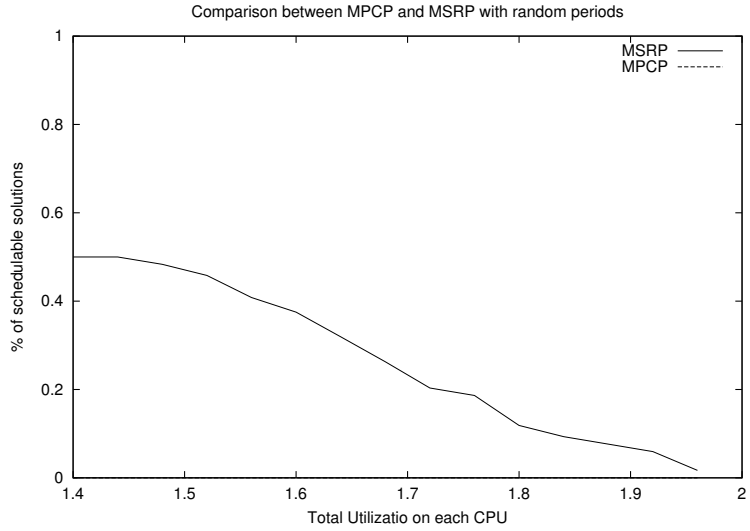

Figure 4.12: Percentage of schedulable task sets with randomly selected periods on Janus by MPCP/MSRP.

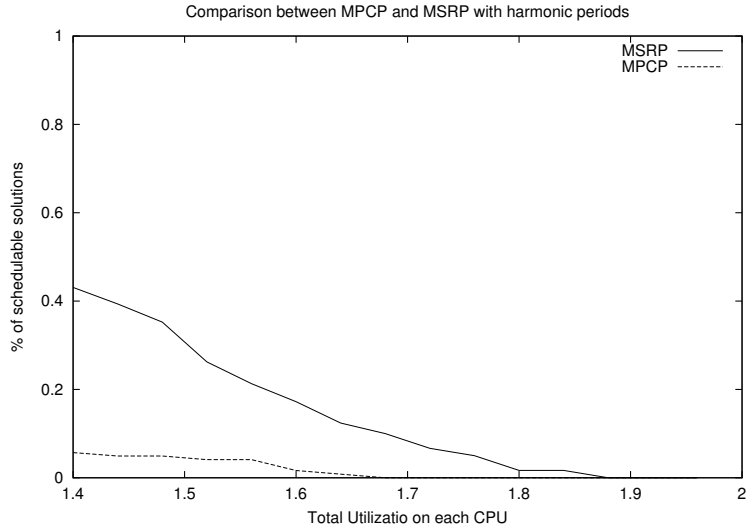

Figure 4.13: Percentage of schedulable task sets with harmonic periods on Janus by MPCP/MSRP.

When compared with the generic task graphs tried in the previous set of experiments, the power-train case study has at least two striking differences:

- Each critical section is quite short when compared to the execution time of the tasks. In the power-train case high rate tasks spend up to 20% of their time while accessing critical resources, but the time spent by medium and low rate tasks is significantly lower. Furthermore, in the test case, the time spent in each critical section is quite small when compared to the previous experiments, since tasks perform more accesses but with shorter execution times. In the context of the results on the generic task sets this means I expect the power-train application to be in the range of quite low resource usage.

- Each task uses many resources and each resource is accessed by many tasks. In the previous case, tasks used from 0 up to a maximum of 6 critical sections each. In the power-train case there is a much more connected graph of task-resource use relationships. In turn, this means more pessimism in the evaluation of the worst case assumptions of MPCP, since the factors n_i^G, $NC_{i,j}$ and $NH_{i,r,j}$ from which the blocking factors of MPCP depend linearly are now significantly higher. On the other side, the blocking factors of MSRP depend only on the worst case length of individual critical sections.

Since I expect both characteristics to be quite common for automotive applications developed according to the guidelines described in Section 4.5.3.1 I expect MSRP to retain a significant advantage over MPCP even under significant changes in the number of task and/or resources in the final implementation.

4.5.4 Final comments

Using spin-lock for accessing mutual exclusive resources in real-time multi-processor systems can possibly lead to a non-schedulable system, because the worst case execution time of a task is increased while keeping the processor idle. When I were faced with the problem of designing a concurrency control protocol for the multiprocessor Janus platform, the goal was to obtain a simple and effective algorithm that allows extending the SRP protocol and therefore, permits to share the stack among all the tasks that are allocated to one processor. As a solution, I proposed a spin-lock mechanism for accessing global resources. I expected an advantage in terms of implementation complexity and a disadvantage in terms of schedulability.

After performing an extensive set of simulations, I discovered that the spin-lock mechanism is not necessarily a disadvantage, but performs even better (in terms of schedulability guarantees) for given application contexts. The simulations show that no algorithm outperforms the other on the whole spectrum of the possible task sets. Which one is the best in terms of the schedulability bound depends on the characteristics of the task set: when access to local resources is clearly dominating with respect to the use of global critical sections, and when the critical sections are short, MSRP presents a better schedulability bound than MPCP.

A second set of experiments, performed on a power-train case study, clearly showed how MSRP can guarantee a higher percentage of task sets when compared to MPCP.

Finally, even in the cases in which MPCP is better than MSRP, it should be considered that MSRP is very simple to implement and has a lower overhead than MPCP. In the Janus case, simplicity and memory optimization were the primary goals.

Regarding other possible approaches to resource sharing, an interesting possibility is to use lock-free algorithms. Lock-free approaches to real-time scheduling were proposed by Anderson, Ramamurthy and Jeffay in [2]. In this approach, a task can execute a critical section more than once, because of the possible conflicts during access. However, when considering periodic real-time tasks, the number of retries is bounded. Intuitively, these approaches can be used especially for short critical sections. However, a deeper study (not covered by this thesis) is needed.

Chapter 5

Heterogeneous multiprocessors architectures

This chapter presents my results on heterogeneous multiprocessor scheduling. In particular, I developed two techniques to schedule heterogeneous bi-processors composed by a CPU and a DSP.

The main idea in this chapter is that a generic hard real-time system can consider a DSP as an accelerator that can speedup critical computations for a given application.

This chapter is divided in the following sections: Section 5.1 and 5.2 present the reference architecture, highlighting the main design problems. Section 5.3 presents the problems that I want to solve in this chapter. Section 5.4 describes some results for systems scheduled using fixed priority, and Section 5.5 presents some results for systems scheduled using EDF.

5.1 System Model

Many architectures available today on the market are heterogeneous multiprocessor architectures, that promise the availability of asymmetric multiprocessors composed by a RISC processor (or a microcontroller) and one or more DSPs [55, 35]. For example, the Texas Instruments SMJ320C80 is a single-chip MIMD parallel processor that consists of a 32-bit RISC master processor, four 32-bit parallel DSPs, a transfer controller, and a video controller. All the processors are tightly coupled through an on-chip crossbar switch that provides access to a shared on-chip RAM. These architectures are expressly designed to handle multimedia streams using dedicated units for signal processing, reducing the computational power needed on the main CPU. Moreover, DSP architectures are designed to have predictable execution times, so they offer a viable alternative to the implementation of fast general purpose multiprocessors.

Moreover, new technologies bring the promise to have really good performance by using accelerators technologies. For example, the Altera FPGA [22] with the NIOS embedded processor promise the availability of cheap reconfigurable hardware that opens big

possibilities for implementing hardware accelerators on the same chip where the main CPU resides.

The challenging issue addressed in this chapter is to verify whether the use of a dedicated processor can effectively enhance the performance of an embedded system still maintaining some kind of real-time guarantee.

The strong hypothesis made in this chapter is that a DSP is usually designed to execute algorithms on a set of data without interruption. Hence, the *natural* way of scheduling a DSP is typically non-preemptive, whereas the CPU schedules both the application tasks and the non-preemptive tasks running on the DSP. In practice, the DSP can be used as a DSP accelerator responding to requests of the master (or main) CPU [14].

I admit that this point of view has some limitations because it is difficult to exploit the full power of a DSP if its schedule depends on the schedule of the main CPU. In fact, if the request of hard real-time guarantee is removed, I can make better use of the DSP computational power by queuing pending requests for the various DSPs in the system, as done in [56].

The problem of DSP scheduling in asymmetric multiprocessor architectures can be viewed as a special case of scheduling with shared resources in multiprocessor distributed systems. In [53, 50], Rajkumar addressed this problem for fixed priority scheduling, providing two algorithms called MPCP (see Section 4.1.2) and DPCP, that allow resource sharing in generic multiprocessors. Although DPCP can be applied to this particular case, the guarantee test provided in [53, 50] is too pessimistic, since it relies on a very generic scenario that does not take in account the characteristics of the considered architecture.

An interesting approach for coping with tasks requiring multiple processing resources is proposed in [57], where the *Co-Scheduling* approach is presented. Co-Scheduling can be used when a task is composed by multiple phases and requires a different resource in each phase. The basic idea of co-scheduling is to divide each job into chunks, associating suitable deadlines to each chunk in order to meet the job deadline (that is, the deadline of the last chunk). In the original paper, the CPU and the disk were considered as the co-scheduled resources, but I believe that this approach could be also applied to the DSP.

I consider an abstract architecture composed by a general purpose CPU and a specialized CPU. The two computational units share a common bus and can freely use some RAM memory and some ROM (Flash) memory, that I suppose is built in the same chip. Other peripherals can be directly controlled by the general purpose CPU. These assumptions are not far from reality since a system on a chip architecture, like the Altera NIOS, can be modeled in a similar way.

At the present stage of the analysis, I assume that the general purpose CPU and the specialized CPU communicate with negligible overhead. This assumption is justified by the fact that the two processors are supposed to be built on the same chip, and both processors can share the full (or part of the) memory address space of the architecture. However, possible communication overheads can be accounted to the task that invokes the service on the specialized CPU. In the following, the general purpose CPU will be identified as the *master processor*, whereas the specialized CPU as the *DSP*. A block diagram of the considered architecture is depicted in Figure 5.1.

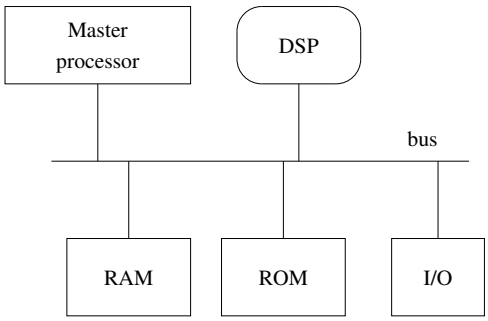

Figure 5.1: Block diagram of the system architecture.

I assume that all the jobs executing on a DSP are invoked using a remote procedure call (RPC) paradigm and are executed in a non-preemptive fashion. Such RPCs will be called *DSP activities*. To simplify the analysis I also assume that on the master processor the DSP activities are scheduled one at a time. Hence, it is a responsibility of the real-time kernel on the master processor to avoid that a task issues a DSP request while the DSP is active.

5.2 Task Model

The real-time task model considered in this chapter is illustrated in Figure 5.2: each task τ_i is a stream of instances, or jobs; each job $J_{i,j}$ arrives at time $r_{i,j}$, executes for C_i units of time on the master CPU, and may require a DSP activity for C_i^{DSP} units of time. I assume that each job performs at most one DSP request, after C_i^{pre} units of time, and then executes for other C_i^{post} units, such that $C_i^{pre}+C_i^{post} = C_i$. Job arrivals can be periodic (if $r_{i,j} - r_{i,j-1} = T_i$, being T_i the task's period) or sporadic (if $r_{i,j} - r_{i,j-1} \geq MIT_i$, MIT_i being the minimum interarrival time). If a task requires a DSP activity it is denoted as a *DSP task*, otherwise it is denoted as a *regular task*. Note that a regular task is equivalent to a DSP task with $C_i^{DSP} = 0$. Whenever a fixed priority scheduling algorithm is used, P_i denotes the priority of task τ_i.

5.3 Problem definition

To present the problem that may occur when dealing with the task model introduced in Figure 5.2, let me make some considerations about the structure of a DSP task.

When executing a DSP task, a *hole* within each job is generated in the schedule of the master processor. Such holes are created because the DSP task executes a DSP activity on another processor. The main idea is to exploit these holes to schedule some other tasks on the master processor in order to improve the schedulability bound of the system.

Suppose, for example, to have a task τ_1 with a period $T_1 = 4$ units of time, $C_1^{pre} = C_1^{post} = 1$, and $C_1^{DSP} = 2$. Note that, although τ_1 uses the master processor only for fifty percent of the time, it must start *exactly* when it arrives, otherwise it will miss its deadline. This constraint is independent from the scheduling algorithm used in the system. Hence, if

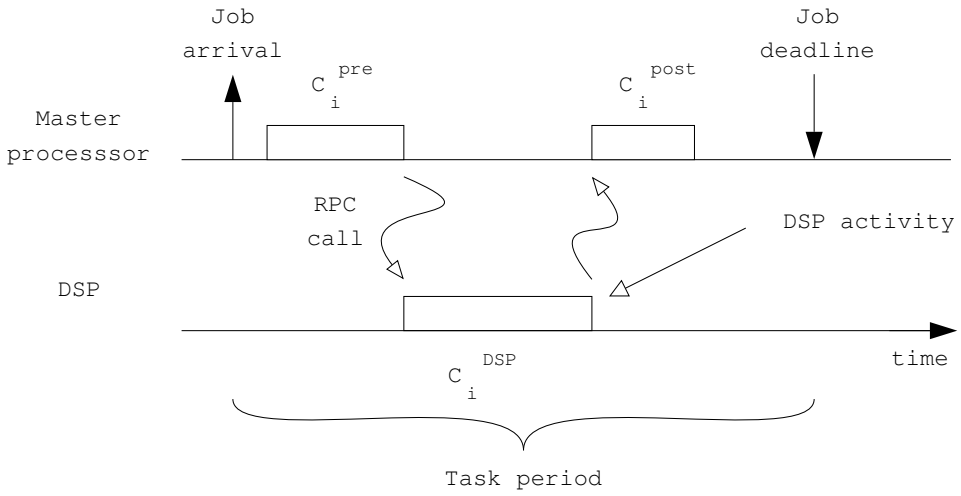

Figure 5.2: Structure of a DSP task.

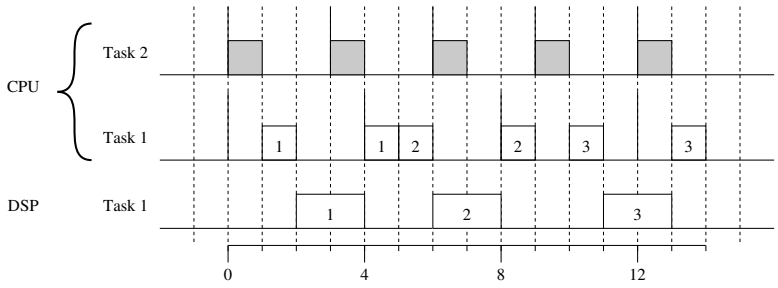

Figure 5.3: A task set that cannot be feasibly scheduled by RM and EDF (jobs of task τ_1 are numbered to facilitate interpretation): task τ_1 misses all its deadlines.

a regular task τ_2 (which does not use the DSP) with period $T_2 = 3$ and computation time $C_2 = 1$ is added to the system, both the Rate Monotonic (RM) algorithm and the Earliest Deadline First (EDF) algorithm fail to generate a feasible schedule, because if tasks start at the same time, τ_2 will always have precedence to τ_1. This situation is shown in Figure 5.3.

However, the task set can be schedulable using a fixed priority assignment by simply assigning $P_1 > P_2$. Figure 5.4 shows the feasible schedule generated using such a priority assignment.

Note that the system model I introduced can be viewed as a particular case of a more general model proposed by Rajkumar in [53, 50], where the Distributed Priority Ceiling Protocol (DPCP) is used to access shared resources on a distributed system. In particular, the DPCP protocol can be used for all the tasks allocated on the main CPU and the DSP activities can be considered as global critical sections executed on the DSP (which acts as a synchronization processor). According to the DPCP approach, the schedulability of the task set is guaranteed by the following test:

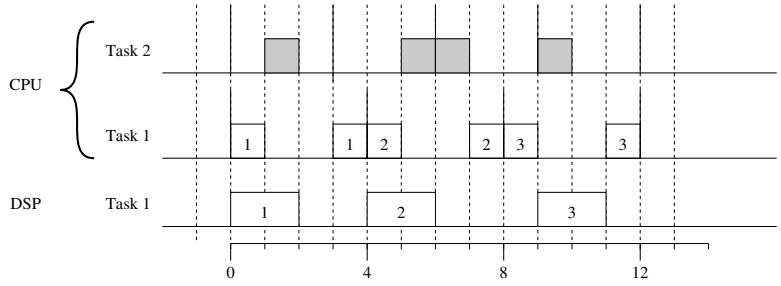

Figure 5.4: A feasible schedule achieved by a different priority assignment ($P_1 > P_2$).

$$\forall i = 1, \ldots, n \quad \sum_{P_j > P_i} \frac{C_j + C_j^{DSP}}{T_j} + \frac{C_i + C_i^{DSP} + B_i}{T_i} \leq U_{lub}(i),$$

where $U_{lub}(i) = i(2^{1/i} - 1)$, and B_i is a blocking factor computed as follows:

$$B_i = \begin{cases} \max_{P_j < P_i}\{C_j^{DSP}\} + \sum_{P_j > P_i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j^{DSP} & for\, a\, DSP\, task \\ 0 & for\, a\, regular\, task \end{cases} \quad (5.1)$$

The major problem of this approach is that the execution time of the DSP activities (C_i^{DSP}) is considered as part of the computation time of every task, including regular tasks. Such a pessimistic computation, although correct, drastically reduces the schedulability of the system. Indeed, in the papers [53, 50], *the authors claim that the C_i^{DSP} factor can also be removed from the computation times, but no proof is provided for that claim.*

Extending the previous analysis to dynamic priorities unfortunately is not trivial. In fact, the admission test presented in Equation (5.3) can be applied to dynamic priorities only if the blocking terms are due to resource sharing [7, 8]. In the case the blocking times are also caused by the synchronization between the master processor and the DSP.

To better clarify the problem that can arise in this case, let me consider a task set $\Gamma = \{\tau_1, \tau_2\}$, with $C_1^{pre} = C_1^{post} = 1$, $C_1^{DSP} = 2$, $T_1 = 4$, $C_2 = \epsilon$, $C_2^{DSP} = 0$ (τ_2 is not accessing the DSP), and $T_2 = 4 + \epsilon$. According to equation (5.3), the task set should be schedulable: in fact $B_1 = 2$ and $B_2 = 0$, hence for task τ_1 I have $\frac{1}{2} + \frac{1}{2} \leq 1$, and for task τ_2 I have $\frac{1}{2} + \frac{\epsilon}{4+\epsilon} \leq 1$, so the admission test is passed. Unfortunately, as can be seen in Figure 5.5, if τ_2 arrives at time $t = 3$, it executes before the second instance of τ_1. As a result, $J_{1,2}$ is not scheduled as soon as it is released and it misses its deadline.

To use dynamic priorities in this context, one could think to split each job in more chunks and assign each chunk a different relative deadline (for example, a relative deadline of $T_i - C_i^{DSP} - C_i^{post}$ could be given to the chunk executed before the DSP activity).

However, I note that, although this assignment works properly in the scenario of Figure 5.4, it does not work in general. For example consider two tasks, τ_1 and τ_2. τ_1 does not use the DSP and has $T_1 = 2$ and $C_1 = 1$. τ_2 has $T_2 = 7$, $C_2 = 5$, $C_2^{DSP} = 2$ and $C_2^{post} = 2$. Figure 5.6 shows that this task set is not schedulable with the method suggested above. I

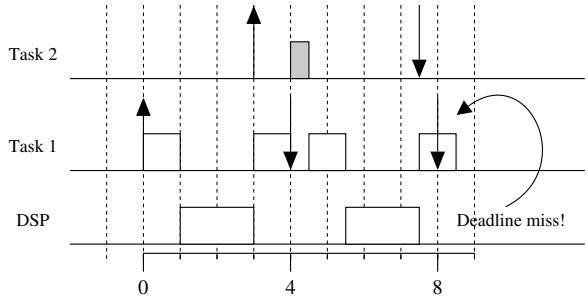

Figure 5.5: EDF does not work always.

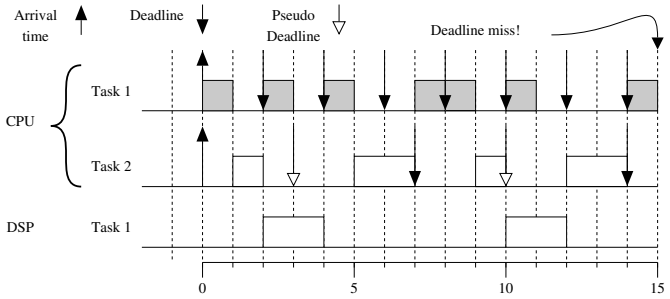

Figure 5.6: Also EDF with modified deadlines does not work always.

will propose a schedulability test for these systems in Section 5.5.

The problem of exploiting the holes to improve the schedulability of the system can be posed in terms of fixed or dynamic priorities. This chapter provide a scheduling algorithm and an analysis for both cases:

- In Section 5.4, I will provide a formal analysis of the DSP tasks for fixed priorities, and I will propose a more efficient method for scheduling DSP tasks and DSP activities. The method is then compared with the DPCP in Section 5.4.3.
- In Section 5.5 I propose a solution for the case of dynamic priorities (using a task model that extends the task model presented in Section 5.2).

5.4 DSP scheduling under fixed priorities

As observed in the previous section, the idle time created in the master processor by executing some activities on the DSP can be used for increasing the system schedulability to guarantee more real-time tasks.

In this section, I show how to schedule tasks on the master processor and how to perform an admission test which exploits the time left by the activities executing on the DSP. The basic idea is to re-arrange the scheduling and guarantee algorithms in order to account the DSP time C_i^{DSP} only to tasks that use the DSP (without influencing the schedule and the guarantee of the regular tasks). This can be achieved by modelling the DSP activity as a

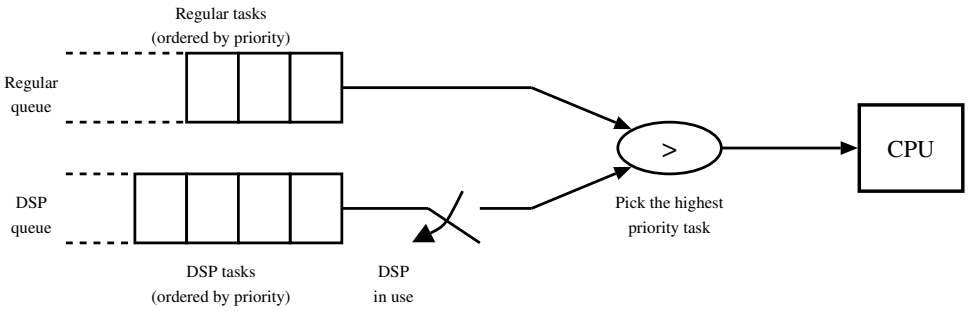

Figure 5.7: My scheduling approach. When the DSP is active, the scheduler selects tasks from the regular queue only.

blocking time: when a DSP task τ_i requests a DSP activity, it blocks for a time B_i waiting for its completion (since I are using an RPC protocol). Moreover, the scheduler has to be modified in such a way that the B_i factor affects only τ_i in the schedulability analysis.

In the next subsection, I show how to modify the scheduler and how to compute the blocking factors when tasks are scheduled using a fixed priority assignment. The case of dynamic priorities is more difficult to analyze and, at present, I just discuss some issues that may help to address the problem.

5.4.1 Enhancing schedulability under fixed priorities

As already said in Section 5.2, to simplify the analysis I assume that DSP requests are scheduled by the master processor one at a time, so that no DSP activity is issued while the DSP is active. This can be achieved by enqueuing regular tasks and DSP tasks in two separate queues that are ordered by priority as shown in Figure 5.7. When the DSP is idle, the scheduler always selects the task with the highest priority between those at the head of the two queues. When the DSP is active, the scheduler selects the task with the highest priority from the head of the regular queue only.

In this way, a task using the DSP blocks all the other tasks requiring the DSP, but not the regular tasks, which can freely execute on the master processor in the holes created by DSP activities.

A similar result can be achieved by implementing each DSP request through a blocking primitive which suspends the calling task in a queue. A waiting task is then awakened by the DSP as the activity has been completed. This solution has also the advantage of allowing other DSP tasks to execute on the master processor while a DSP request is being served.

Note that this approach is different from using an inheritance-based protocol [59], which would prevent a regular task with medium priority to execute while a high priority DSP task is blocked on the DSP resource. A regular and a DSP tasks can only be delayed by other DSP tasks that where delayed.

That is, the blocking factor B_i of a regular task can be computed as:

$$B_i^{reg} = B_i^{def}$$

where

$$B_i^{def} = \sum_{P_j > P_i} min(C_j^{post}, C_j^{DSP})$$

is a duration that accounts for the deferred execution requests of higher priority DSP tasks.

A DSP task, however, can also be delayed by other tasks which may hold the DSP. In particular, a DSP task τ_i can be delayed by a single lower priority job which is already using the DSP, and by those higher priority jobs that that may interfere with τ_i before it is scheduled.

Hence, the blocking factor B_i of a DSP task can be computed as the sum of three terms:

$$B_i^{DSP} = C_i^{DSP} + B_i^{lp} + B_i^{hp} + B_i^{def}, \tag{5.2}$$

where B_i^{lp} denotes the blocking caused by the (single) lower priority task and B_i^{hp} denotes the blocking due to the interference of higher priority tasks. Figure 5.8 illustrates an example showing how a task τ_3 can be blocked by a task τ_4, with lower priority, and by two tasks, τ_1 and τ_2, having higher priority. As also done in [53, 50], the two blocking terms can be computed as follows:

$$B_i^{lp} = \max_{P_j < P_i} \{C_j^{DSP}\}$$

$$B_i^{hp} = \sum_{P_j > P_i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j^{DSP}.$$

Therefore, an upper bound for the blocking time B_i experienced by task τ_i is given by

$$B_i = \begin{cases} B_i^{DSP} & for\, a\, DSP\, task \\ B_i^{reg} & for\, a\, regular\, task \end{cases}$$

This value can be used in the classical admission test for fixed priority systems [59], that is:

$$\forall i = 1, \ldots, n \quad \sum_{P_j \geq P_i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq U_{lub}(i), \tag{5.3}$$

where $U_{lub}(i) = i(2^{1/i} - 1)$.

It is worth observing that to enhance schedulability and accept more tasks in the system, the admission test can be performed by using the hyperbolic bound [15] or the response time analysis [41, 6]. For large task sets where the admission test has to be performed on line, then the hyperbolic bound is more suited, having an $O(n)$ complexity, whereas the response time analysis has a pseudo-polynomial complexity.

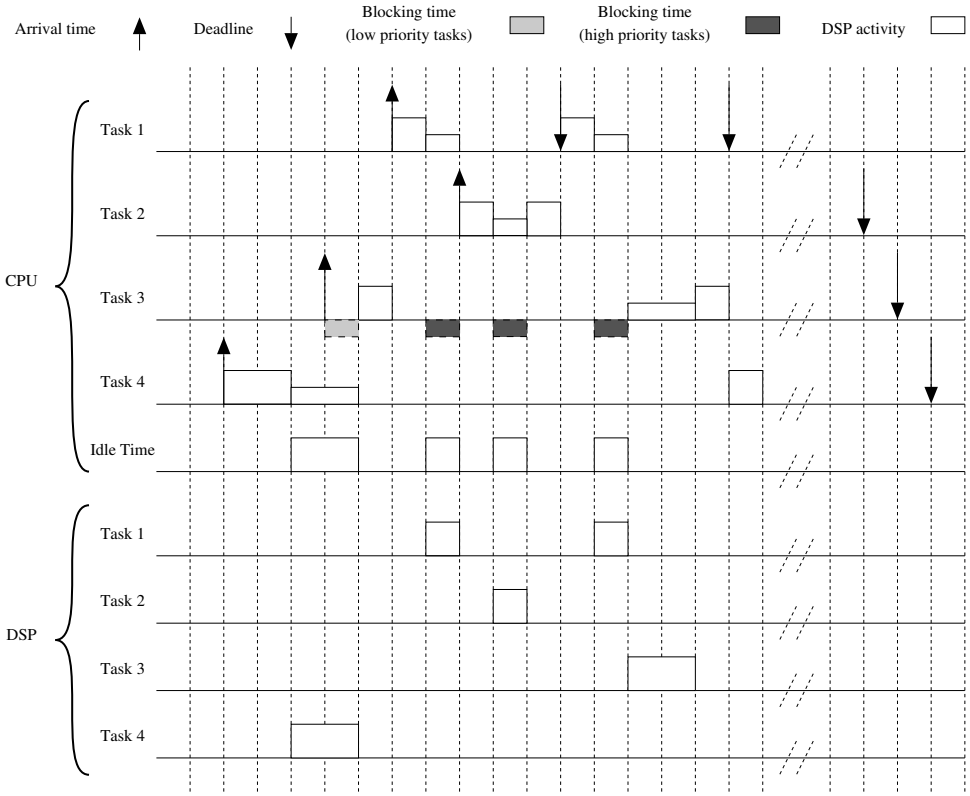

Figure 5.8: Example of scenario where task τ_3 is blocked by some high priority (τ_1 and τ_2) and low priority (τ_4) tasks.

Using the hyperbolic bound [15] the admission test becomes:

$$\forall i = 1, \ldots, n \quad \prod_{P_j > P_i} \left(\frac{C_j}{T_j} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2.$$

In Section 5.4.3, I also present some simulation experiments aimed at estimating the advantage of using different admission tests.

5.4.2 Allowing interleaving DSP requests

In this section I consider the possibility of extending the analysis of the blocking times by removing the assumption of having one DSP request at a time.

If the constraint of having only one DSP activity at a time is removed and no special protocol is used for accessing the DSP, the blocking time B_i for task τ_i is equal to the response time of the DSP request. In this case, such a response time must be explicitly computed by performing a finishing time analysis on the DSP schedule. Since I am using an RPC protocol, the DSP scheduling is clearly non-preemptive; hence, *if the requests for DSP activities are ordered according to the RM priority assignment, then the problem of computing the blocking time B_i is equivalent to the problem of finding the response time of a task τ_i with computation time C_i^{DSP} and period T_i when a nonpreeemptive RM scheduler is used and some release jitter is present*.

The exact finishing time of a task under a fixed priority preemptable scheduler can be computed as shown in [6], the nonpreemptability of the scheduler can be accounted by adding a blocking term equal to $\max_{P_j < P_i}\{C_j^{DSP}\}$ to each blocking time B_i, whereas the release jitter can be accounted as shown in [61].

5.4.3 Simulation results

The performance of the scheduling algorithm presented in Section 5.4 has been evaluated by simulation on a large number of task sets (more than 64 million experiments). For each task set I computed the blocking time of each task and I checked the schedulability using both my approach and the DPCP protocol.

Task sets were generated using random parameters with uniform distribution with the following characteristics:

- The number of tasks was chosen as a random variable from 2 to 50.
- Task periods were generated from 10 to 1000.
- Task worst-case execution times ($C_i' = C_i + C_i^{DSP}$) were chosen in such a way that $\sum_i \frac{C_i'}{T_i}$ varied from 0.01 to 0.99.
- DSP tasks were generated to be 80% (in the average) of the total number of tasks.
- C_i^{DSP} was generated to be a random variable with uniform distribution in the range of 10% to 80% of the C_i'.

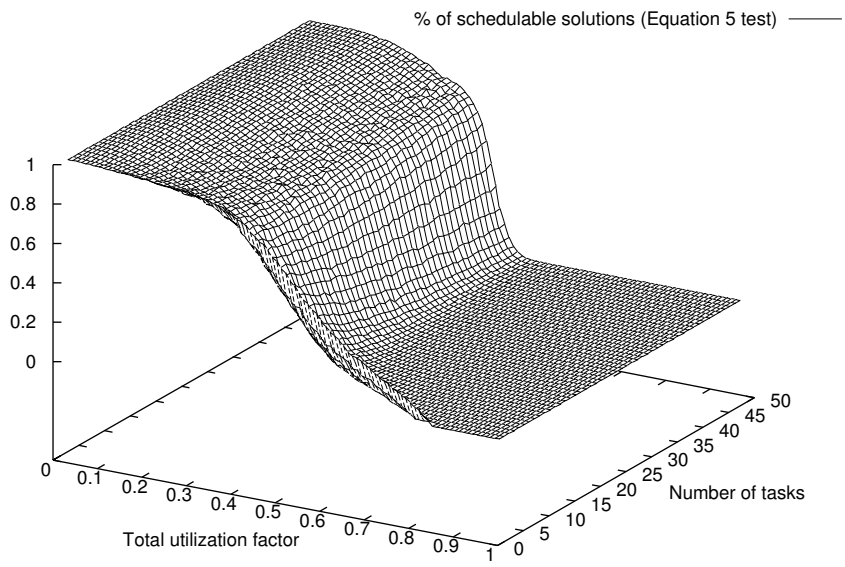

Figure 5.9: Schedulability results of my approach when varying the total utilization factor and the number of tasks in the task set (using Equation (5.3)).

In a first experiment I compared the performance of my method against the DPCP approach in terms of average schedulability, using the admission test given by Equation (5.3). Figures 5.9 and 5.10 show the percentage of schedulable task sets, for my approach and for the DPCP method, as a function of the number of tasks and the total utilization factor ($\sum_i \frac{C'_i}{T_i}$). As clear from the graphs, both methods have a performance degradation as the total utilization factor increases, whereas they are quite insensitive to the number of tasks in the set.

To better evaluate the enhancement achieved with my approach, Figure 5.11 reports the difference between the two previous graphs. I note that the advantage of my approach with respect to DPCP is more sensitive for task sets with total utilization in the range from 0.3 to 0.6.

A second experiment has been carried out to evaluate the improvement that can be achieved using my method with the hyperbolic bound in place of Equation (5.3). Figure 5.12 shows that for large task sets with utilization around 50% the hyperbolic bound improves the acceptance rate up to 30%.

A third experiment has been performed to compare the two approaches using the response time analysis. Figure 5.13 shows the performance differences between the two methods. I note that the surface has the same shape as the one in Figure 5.11, presenting a peak translated around 0.6 in the utilization axis. Hence, my approach basically outperforms DPCP when the utilization factor is near to the RM schedulability bound. Figure 5.14 reports the performance of the two approaches and their difference as a function of the utilization factor for task sets composed by 30 tasks.

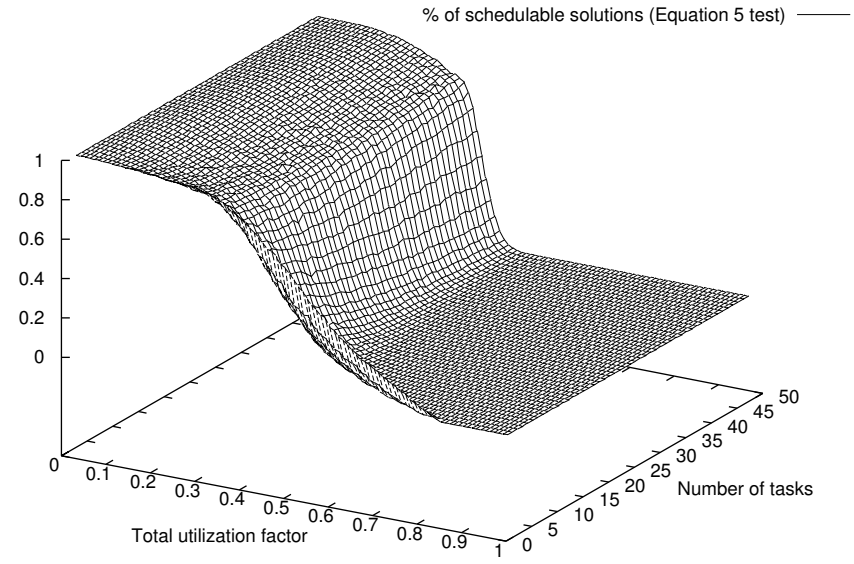

Figure 5.10: Schedulability results of DPCP when varying the total utilization factor and the number of tasks in the task set (using Equation (5.3)).

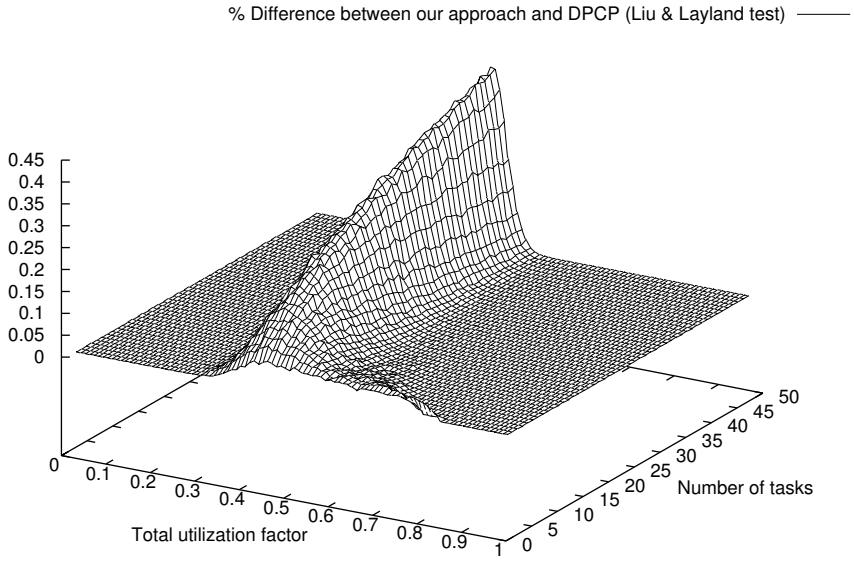

Figure 5.11: Difference between the two approaches (using Equation (5.3)).

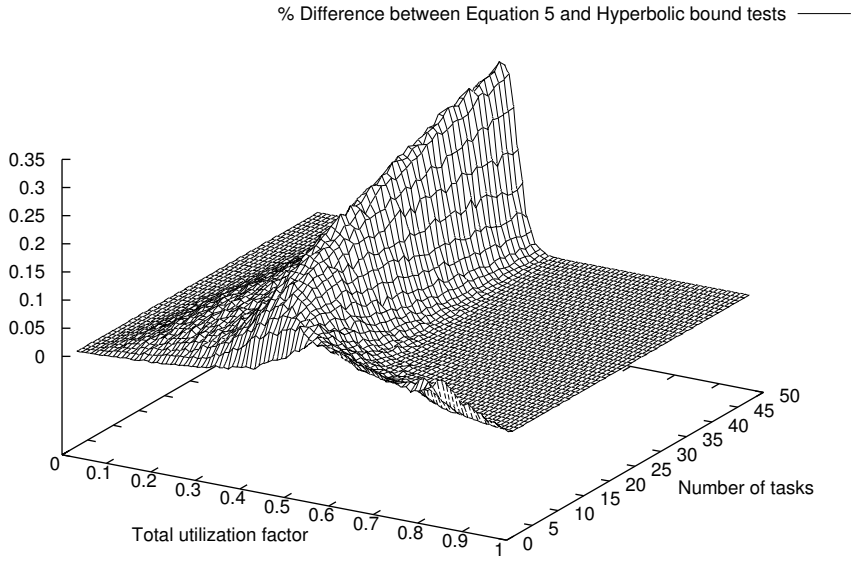

Figure 5.12: Improvement achieved using the Hyperbolic Bound.

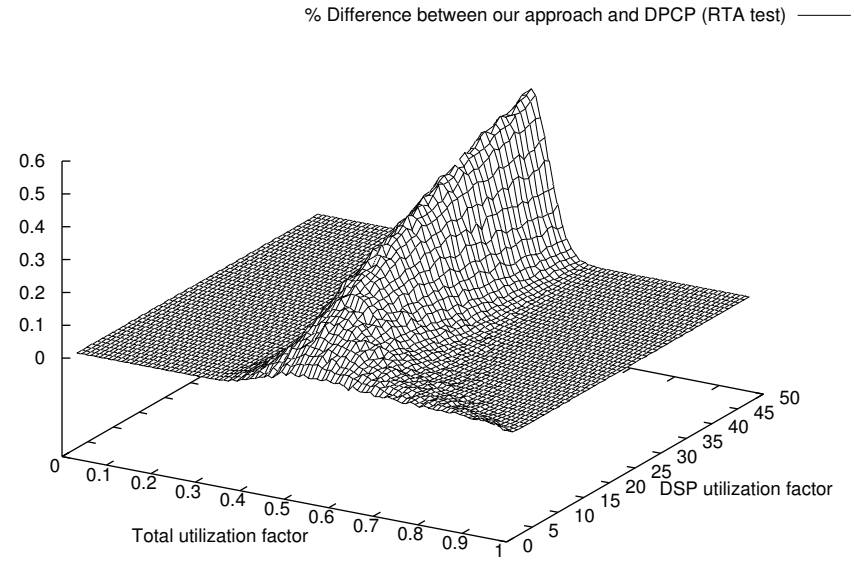

Figure 5.13: Difference between the two approaches (using response time analysis).

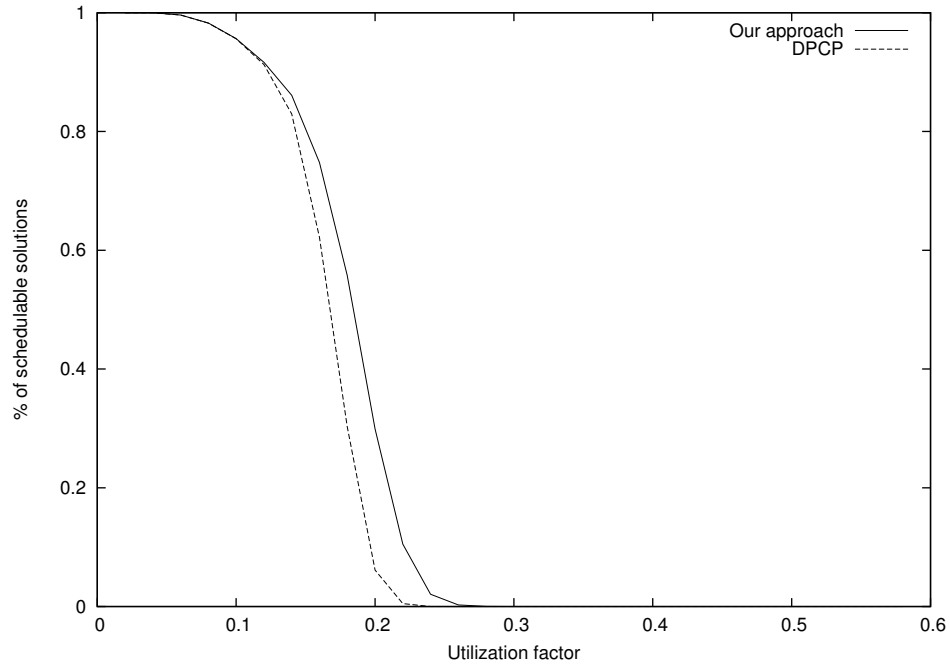

Figure 5.14: Performance of the two approaches and their difference as a function of the utilization factor for task sets composed by 30 tasks.

Within the same experiment performed with the response time test, I evaluated the influence of the DSP usage ($\sum_i C_i^{DSP}$) on the schedulability results. Figure 5.16 shows that the higher the DSP usage, the lower the average acceptance rate.

Finally, the difference of schedulability percentage between my approach and the DPCP is reported in Figure 5.16, which shows that my algorithm outperforms DPCP for task sets with high utilization and high DSP usage.

5.5 DSP scheduling under dynamic priorities

This Section propose an analysis based on the problem definition in Section 5.3. In particular, I propose two new uniprocessor scheduling algorithms, called CEDF and CEDF+SRP, which are derived from the well-known EDF and SRP algorithms. In order to efficiently exploit the time left on the general processor CPU from DSP execution, the algorithms divide each task instance into chunks, which are scheduled under EDF with a suitable deadline (less than the task period).

The main contributions of this section are:

- It presents two new uniprocessor scheduling algorithms, called CEDF and CEDF+SRP, that are derived from the well-known Earliest Deadline First (EDF) [44] and Stack Resource Policy (SRP) [7] algorithms using a technique based on checkpoints. The two algorithms are then formally presented.

- Then, the CEDF+SRP algorithm is used to extend the results obtained in [29] to

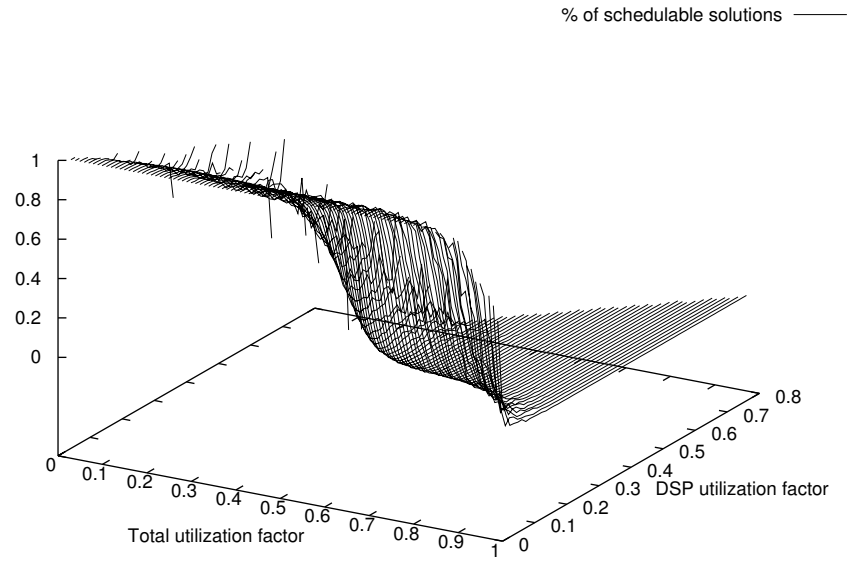

Figure 5.15: Difference in the percentage of scheduled tasks set between my approach and DPCP when considering the influence of DSP utilization.

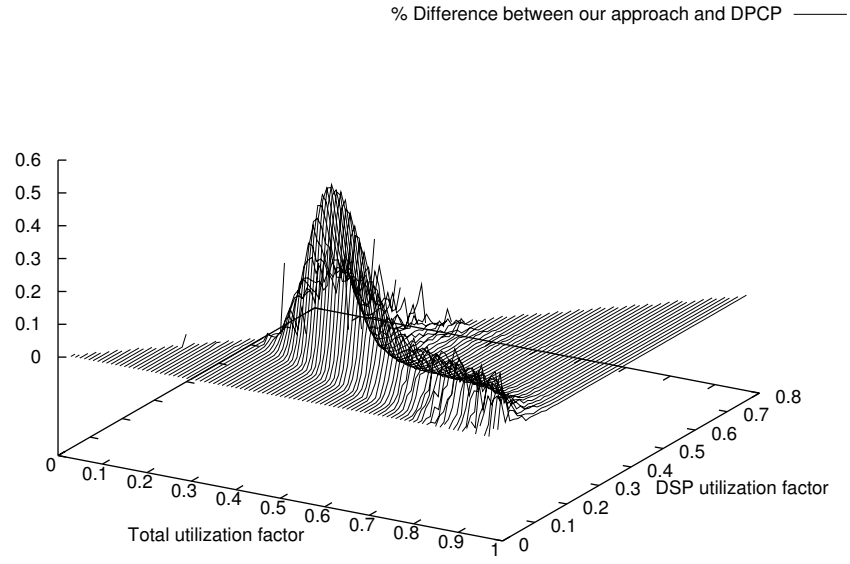

Figure 5.16: Influence of DSP utilization on the schedulability.

dynamic priority systems, giving a way to efficiently reuse on the main CPU the time spent on the DSP.

As it will be described later, the checkpoint technique divides each task instance into chunks, where every chunk has its own deadline. This approach has some similarities with some other work in the literature (see also Chapter 2). In particular, the class of PFair algorithms [11, 3] and its derivates divides every task instance into unit-size chunks, every one with a window and a pseudo deadline. The technique proposed in this article is slightly different in the way deadlines are chosen and in the dimension of the chunks. Moreover, PFair algorithms are multiprocessor scheduling techniques, whereas the CEDF+SRP algorithm is basically a uniprocessor scheduling algorithm. Another work in multiprocessor scheduling that has some similarities with the proposed approach is [38], where tasks instances are split into different parts; in that work, however, the scheduling is made off-line with a bin-packing approach, and then statically repeated.

5.5.1 EDF with Checkpoints

Consider a set $\mathcal{T}$ of periodic tasks with start time s_i, period $T_i\,(i = 1 \ldots n)$, a Worst Case Execution Time (WCET) C_i, and a relative deadline D_i equal to the period. In this section I will modify the EDF policy (see Chapter 2) defining a new scheduling algorithm called Checkpointed EDF (CEDF), that retains the same property of optimality shown by EDF; the CEDF algorithm is useful because it is the base I used to derive the CEDF+SRP algorithm.

Considering a generic task τ_i, the CEDF algorithm divides each job τ_{ij} (with start time s_{ij}) into m_i chunks τ_{ijk}[1] with computational time C_{ik}, such that

$$\sum_{k=1}^{m_i} C_{ik} = C_i.$$

The number m_i and the size C_{ik} of each chunk can be arbitrarily chosen. Every chunk τ_{ijk} is scheduled following the EDF rules using a relative deadline defined as

$$D_{ik} = D_i - \sum_{h=k+1}^{m_i} C_{ih}.$$

In practice, when a task finishes a chunk, its deadline is set to the deadline of the next chunk, that is, the deadline is postponed during the execution of the task; Figure 5.17 shows the structure of a task scheduled by EDF, and the structure of the same task as scheduled by CEDF supposing it is divided into three chunks.

The CEDF algorithm has the following property:

Theorem 6 ***(Optimality of CEDF):***

Algorithm CEDF is optimal, in the sense that a task set is schedulable by CEDF if and only if it is schedulable by EDF.

[1] That is the k^{th} chunk related to the instance j of τ_i.

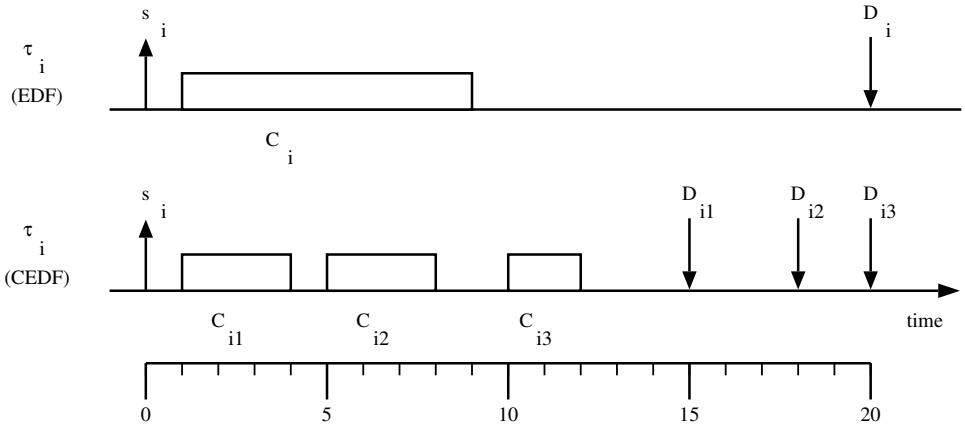

Figure 5.17: A task scheduled by EDF and CEDF. The task has the following structure: $D_i = 20$, $C_i = 8$, $m_i = 3$, $C_{i1} = 3$, $C_{i2} = 3$, $C_{i3} = 2$.

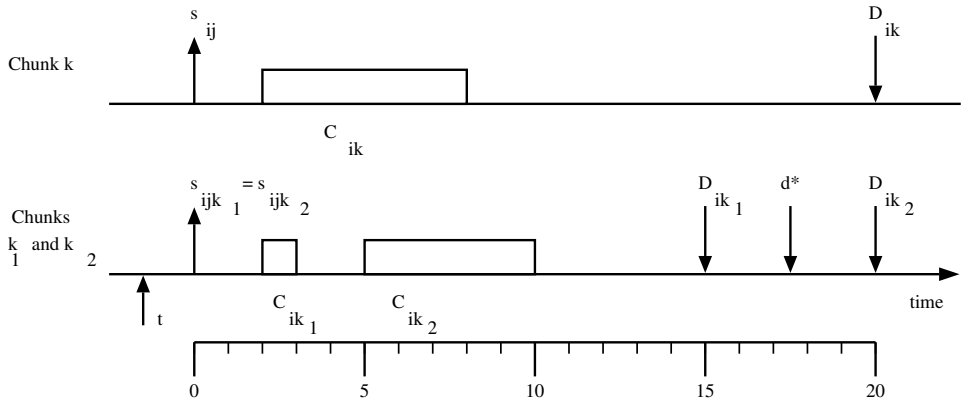

Figure 5.18: A single transformation step.

Proof.

If part. Consider a feasible finite EDF schedule. The initial EDF schedule can be viewed as a set of chunks, where each chunk corresponds to a whole instance of the task. Each chunk has the same start time, deadline and computation time as the corresponding EDF task instance.

The proof is done defining a transformation step that takes as input a feasible schedule composed by a set of n chunks τ_{ijk} and gives as output another feasible schedule with $n + 1$ chunks. Each step divides, for an instance j, a chunk τ_{ijk} in two chunks τ_{ijk_1} and τ_{ijk_2} with start times $s_{ijk_1} = s_{ijk_2} = s_{ij}$, WCET C_{ik_1} and C_{ik_2} ($C_{ik_1} + C_{ik_2} = C_{ik}$), deadlines $D_{ik_1} = D_{ik} - C_{k_2}$ and $D_{ik_2} = D_{ik}$, as shown in Figure 5.18.

C_{ik_2} is chosen in a way that each chunk τ_{ijk_2} produced by the transformation step corresponds to a chunk in the CEDF schedule. The first time the transformation step is applied, it produces the chunk τ_{ijm_i} in the CEDF schedule, and so on.

I now prove that the proposed transformation step maintains the feasibility of the sched-

ule. That is, if I consider a feasible schedule and I apply the proposed transformation, the resulting schedule is still feasible.

By contradiction, if the resulted schedule is not feasible there must be a chunk related to a task τ^* *different* from τ_i that misses its deadline. Let d^* be the missed deadline. Since CEDF uses the earliest deadline to schedule the task's chunks, I note that the chunks with deadline less than d_{ijk_1} or greater than d_{ijk_2} are not affected by the transformation step, so it must be that $d_{ijk_1} \leq d^* \leq d_{ijk_2}$.

If a chunk misses its deadline d^* I have that there must exists an instant t such that the computational demand in $[t,\ d^*]$ exceeds the length of the interval. That is,

$$D(t,\ d^*) > d^* - t.$$

I note that the computational demand in $[t,\ d_{ijk_2}]$ is:

$$D(t,\ d_{ijk_2}) \geq C_{ik_2} + D(t,\ d^*) > C_{ik_2} + d^* - t > C_{ik_2} + d_{ijk_1} - t = d_{ijk_2} - t$$

(where the first $\geq$ is due to the fact that, at least, $D(t,\ d_i^*)$ does not include the chunk τ_{ijk_2}).

That is, the demand in $[t,\ d_{ijk_2}]$ is greater than $d_{ijk_2} - t$, meaning that the task set was not schedulable before the transformation step, contradicting the hypothesis.

Hence, the single transformation step preserves the feasibility of the schedule. The transformation step can be recursively applied until, after a finite number of steps, a given finite EDF schedule will be transformed into a CEDF schedule, where each job T_i is splitted in m_i chunks.

Only if part: Given a CEDF schedule, I need to prove that the same set of chunks are schedulable under plain EDF. First, for each task instance I consider all the chunks related to that instance and I postpone their deadline setting it to $s_{ij} + D_i$ (note that this postponement does not reduce the schedulability of the system). Then, I obtain for each instance a set of chunks with the same start time and the same deadline of the EDF schedule I want to build. Then, the proof comes directly from the EDF optimality. □

5.5.2 Resources and checkpoints

The next step in the presentation of my approach is the extension of the CEDF algorithm to cover the case in which tasks can share resources. The results presented in this section are based on the Stack Resource Policy (SRP) scheduling algorithm (see also Chapter 2). In this section, I modify the SRP scheduling policy by introducing checkpoints. I will call this extension the CEDF+SRP algorithm.

Basically, the CEDF+SRP algorithm works as follows:

- Every real-time (periodic and sporadic) task τ_i must be assigned a static preemption level π_i inversely proportional to the task period.
- Every real-time task is divided into chunks as in the CEDF algorithm; checkpoints are set as follows:

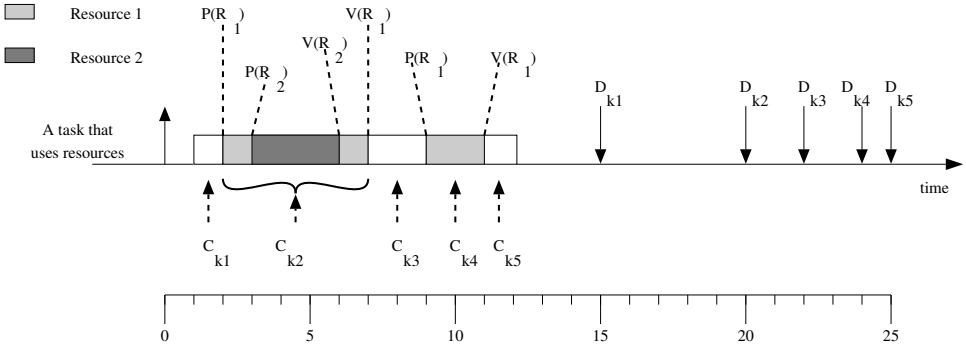

Figure 5.19: A typical checkpoint assignment used in the CEDF+SRP Algorithm.

1. just before a task enters a critical section;
2. just after a task exits a critical section (if the critical sections are nested, the checkpoints are put on the outermost critical sections);
3. at the end of the task (as in the CEDF algorithm);

Figure 5.19 shows a typical checkpoint assignment.

- The static ceiling and the system ceiling works like in SRP, Equations (2.1) and (2.2)
- Every chunk is assigned a dynamic priority using the CEDF algorithm.

Then, the CEDF+SRP scheduling rule (that is the same as the SRP scheduling rule) states that:

> *"a job is not allowed to start executing until its priority is the highest among those of the active jobs and its preemption level is greater than the system ceiling".*

The priority assignment of the CEDF+SRP algorithm ensures that a task, once started, cannot be blocked until completion (mainly because the algorithm is based on SRP). The only difference with respect to SRP is that the execution of a job τ_{ik} could be delayed by a job with a lower preemption level (not priority!), which is locking some resource, and has raised the system ceiling to a value greater than or equal to the preemption level π_i.

Note that the CEDF+SRP algorithm does not ensure that tasks' executions are perfectly nested, as it can be seen in Figure 5.20. The example consists of two tasks τ_1 and τ_2, with period $T_i = T_j = 18$, computational times $C_{i1} = C_{j2} = 2$, $C_{i2} = C_{j1} = 4$, and no resources. Due to the different deadlines of the chunks, tasks are scheduled in an interleaved way. That means the stack sharing property of SRP is no longer maintained.

However, the CEDF+SRP algorithm retains all the other properties of SRP; in particular the maximum blocking time of a task under CEDF+SRP is the same as in the "plain" SRP. Moreover, if a task set is schedulable under SRP, then it is also schedulable under CEDF+SRP. These facts are proven by the following Lemma and the following Theorem.

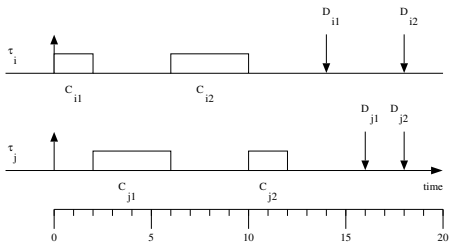

Figure 5.20: Task executions are not nested under CEDF+SRP. Note that task τ_i postpones its deadline at time 2; task τ_j postpones its deadline at time 6.

Lemma 3 *The maximum blocking time that a task can experience under the CEDF+ SRP algorithm is the same as that it can experience under the SRP algorithm.*

Proof.
Consider that the blocking time a task τ_i can experience is due to the fact that when a task arrives, some task with a lower preemption level can have locked a resource, raising the system ceiling to a value greater than π_i. Please also remember that the preemption level of a task depends only on its period.

Once a task starts and experiences its blocking time, there cannot be other blocking. In fact, a task cannot be blocked when a chunk ends and the deadline is postponed, because the checkpoints are always inserted in points where a task does not use any resource. When a new chunk is started with a postponed deadline (from D_{ik} to D_{ik+1}), it cannot cause early blocking on other tasks, because it simply does not contribute to raise the system ceiling. The only effect that a postponed deadline can have is that the task can be preempted by a task with an earlier deadline.

As a consequence blocking can only occur when a task arrives, so the blocking time experienced by the CEDF+SRP algorithm is the same as the one experienced under the plain SRP. □

It is worth noting that, although the blocking time is the same as in the SRP case, a task can be preempted by a task with a lower or equal preemption level (see Figure 4, where the two tasks have the same preemption level). This fact, however, is not important for the global scheduling properties of CEDF+SRP. In fact, the following theorem holds:

Theorem 7 *A set $\mathcal{T}$ of tasks sharing resources can be feasibly scheduled by CEDF+ SRP if it can be feasibly scheduled by SRP.*

Proof.
The scheme of the proof is the same as the one used for Theorem 6. The proof is done by defining a transformation step that takes as input a feasible schedule composed by a set of n chunks τ_{ijk} and gives as output another feasible schedule with $n + 1$ chunks. Each step divides, for an instance j, a chunk τ_{ijk} in two chunks τ_{ijk_1} and τ_{ijk_2} with start times $s_{ijk_1} =$

$s_{ijk_2} = s_{ij}$, WCET C_{ik_1} and C_{ik_2} ($C_{ik_1} + C_{ik_2} = C_{ik}$), deadlines $D_{ik_1} = D_{ik} - C_{k_2}$ and $D_{ik_2} = D_{ik}$, as shown in Figure 5.18.

The transformation step is done in a way that each chunk τ_{ijk_2} produced by the transformation step corresponds to a chunk in the CEDF+SRP schedule, that is the first time the transformation step is applied, it produces the chunk with index m_i in the CEDF+SRP schedule, and so on (see Figures 5.18 and 5.19).

Again, I need to prove that the proposed transformation step preserves the feasibility of the schedule, considering the blocking time experienced by every task.

Let $B_i(t_1, t_2)$ be the maximum blocking time experienced by the first chunk of task τ_i in a generic interval $[t_1, t_2]$ due to tasks with a lower preemption level.

If the resulting schedule is not feasible there must be a chunk belonging to a task τ_x that misses its deadline. Let d^* be such a deadline. Since CEDF+SRP uses the earliest deadline to schedule the task's chunks, I note that the chunks with deadline less than d_{ijk_1} or greater than d_{ijk_2} are not affected by the transformation step, so it must be that $d_{ijk_1} \leq d^* \leq d_{ijk_2}$.

If a chunk of τ_x misses its deadline d^* I have that it must exist an instant t where the computational demand in $[t, d^*]$ plus the blocking time (experienced by the first chunk only) exceeds the length of the interval. That is,

$$D(t, d^*) + B_x(t, d^*) > d^* - t.$$

Since the system was schedulable before splitting the chunk τ_{ijk}, the computational demand in $[t, d^*]$ *before* dividing chunk k is such that

$$D(t, d^*) + B_x(t, d^*) \leq d^* - t.$$

I need now to have a more strict version of the inequality above. For that, I define $B'_i(t_1, t_2, \tau_j)$ as the maximum blocking time experienced by the first chunk of task τ_i in a generic interval $[t_1, t_2]$ due to tasks with a lower preemption level, *except* for task τ_j.

Since I know that chunk τ_{ijk} was schedulable before splitting it, the scheduling of the chunk related to task τ_x that misses the deadline must have finished before d^* to enable chunk k to complete before its deadline. Hence, I have:

$$D(t, d^*) + B'_x(t, d^*, \tau_i) \leq d_{ijk_2} - C_{ik} - t.$$

I now use Lemma 3 to say that $B'_x(t, d^*, \tau_i)$ does not change when chunk τ_{ijk} is splitted in two. Then I sum C_{ik_1} on both sides, obtaining the computational demand *after* splitting chunk k.

$$\begin{aligned} D(t, d^*) + C_{ik_1} + B'_x(t, d^*, \tau_i) \leq d_{ijk_2} - C_{ik} - t + C_{ik_1} = \\ = d_{ijk_2} - C_{ik_2} - t = d_{ijk_1} - t \leq d^* - t. \end{aligned}$$

Since the computational demand in $[t, d^*]$ (considering the blocking time for task τ_x) is less than or equal to $d^* - t$, the final schedule is still feasible. □

Theorem 7 is important because it states a sort of equivalence between SRP and CEDF+SRP,

and this is important because SRP is widely studied in the real-time literature, and there are schedulability conditions that can be used without modifications.

5.5.3 Implementation issues

The implementation of the CEDF algorithm is straightforward. In practice, every task should start having its relative deadline set to the deadline of the first chunk. Such a primitive should be provided by the scheduler, in a way that the task can insert it in its code. The behavior of that primitive should set the current deadline of the task to the deadline of the next chunk, and then it should include the preemption check that ensures that the earliest deadline task is always the running task.

Implementing the CEDF+SRP algorithm and its application to DSP scheduling is of similar complexity. In particular, the resource lock/unlock primitives should be modified to implement the change of the deadline and the preemption check before/after their normal behavior. I can also suppose that at the end of the DSP code an internal interface is programmed to raise an interrupt on the master processor. When implementing the RPC that executes the DSP code, the deadline postponement should be put just before the RPC and just after that interrupt.

5.5.4 Using CEDF+SRP for DSP scheduling

In this section, I will reconsider the problem definition introduced in Section 5.3 using the CEDF+SRP approach, designing a way to account in the scheduling guarantee for the time left free by the DSP scheduling using the CEDF+SRP. Unfortunately, the time I can reuse is not uniformly distributed like a bandwidth, but it depends on the period of its utilization, because the position of DSP holes strictly depends on the execution of the DSP tasks.

Without loss of generality, in the rest of this section I will map a regular task to a task composed by only one chunk, and, a DSP task τ_i to a task with three chunks defined in the following way:

$$C_{i1} \equiv C_i^{pre}$$

$$C_{i2} \equiv C_i^{DSP}$$

$$C_{i3} \equiv C_i^{post}.$$

The following subsections show how much bandwidth can be collected from a set of tasks that use the DSP, and how to derive a good schedulability condition.

5.5.4.1 Collecting Bandwidth

The main problem in collecting spare time from the DSP schedule is that the DSP time is not uniformly distributed, but it depends on the scheduling of the tasks that use the DSP. For that reason the DSP time can be exploited only under particular conditions, that is

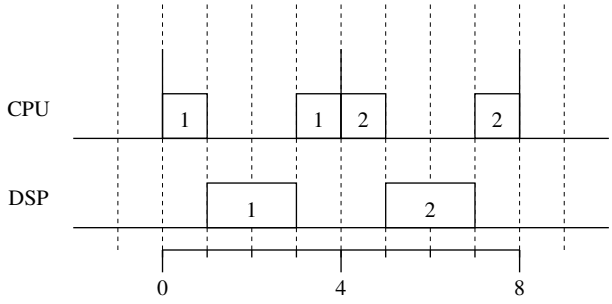

Figure 5.21: An example. The Figure shows only two instances (numbered with '1' and '2') of the periodic task.

with particular periods. For example, consider a task set composed by only one task τ_1 (see Figure 5.21), with $C_1 = 4$, divided in three chunks, $C_{11} = 1$, $C_{12} = 2$, $C_{13} = 1$; moreover, τ_1 has $D_1 = D_{13} = 4$, $D_{11} = 1$ and $D_{12} = 3$. The second chunk executes on the DSP.

Under the pure SRP scheduling algorithm (without chunks and with deadlines equal to periods), the DSP time can be exploited on the main CPU only using background scheduling. In fact, adding a task with period less than 4 always results in a deadline miss of τ_1 [29]. Using CEDF+SRP, the DSP time can be exploited also with periods less than 4. For example, a task τ_2 with $C_2 = 1$ and $T_2 = 3$ can be added. However, it is still not possible (and it will never be!) to add a task with period less than 2. Task τ_1 will always miss its deadline if such a task arrives when the third chunk is in execution (i.e., at time 3).

The approach I propose is to find a lower bound on the DSP time made available by the various tasks in the system, and use that lower bound as an estimation of the DSP time it can be collected under CEDF+SRP. In the following paragraphs, I propose a method for estimating the lower bound on a simple case (more complex case can be analyzed in a similar way). Then, in the following subsection I will use that bound to prove that the system remains schedulable under the CEDF+SRP algorithm adding a task that meets the bound.

I consider a task divided in 3 chunks, like task τ_1 in Figure 5.21. The worst case of the collected time from a single task is depicted in Figure 5.22. In practice, I consider a starting point that is equal to the end of the DSP part if the task is scheduled just after its activation, and I consider that the collected time starts just when the second instance of a task (executed as late as possible) uses the DSP. In the example, the collected time $\gamma_i(t)$ is zero if the period I want to use is less than 14. Defining $\eta(t) = \left\lfloor \frac{\max(t-T_i+C_{i1}+C_{i2},0)}{T_i} \right\rfloor$, the exact formula of $\gamma_i(t)$ is:

$$\gamma_i(t) = \eta(t)\, C_{i2} + \min\left\{C_{i2},\, \max\left[0,\, t - (2+\eta(t))T_i + C_i + C_{i2}\right]\right\}. \qquad (5.4)$$

Note that if only the DSP time is known, but not the distribution of the DSP time in the task, a conservative approach such that shown in Figure 5.23 can be applied. In the following, I suppose the use of Equation 5.4.

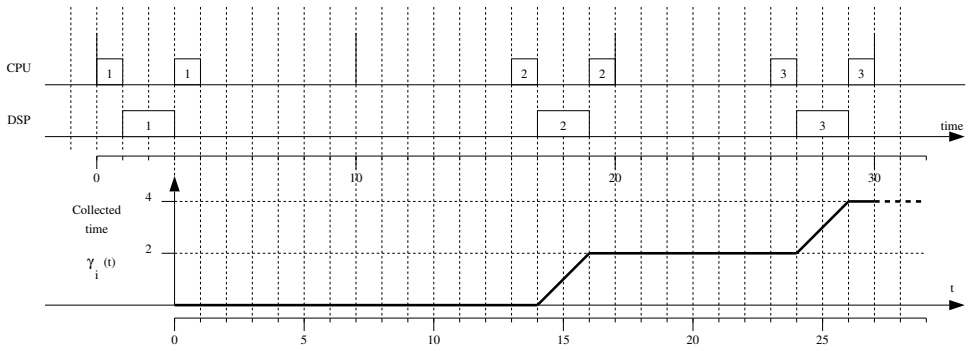

Figure 5.22: The lower bound on the collected time $\gamma_i(t)$ of a task τ_i with $T_i = 10$ and $C_i = 4$; task τ_i is divided in three chunks (the second runs on a DSP) with capacities $C_{i1} = 1$, $C_{i2} = 2$, and $C_{i3} = 1$.

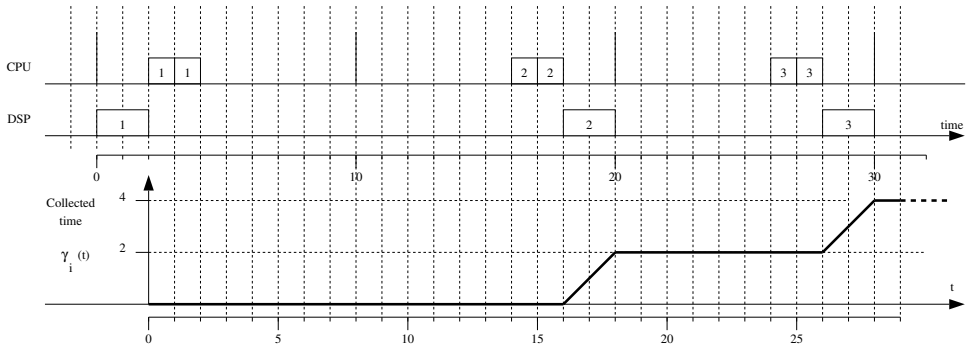

Figure 5.23: If the exact distribution of the DSP computation is not known, a conservative approach can be applied (compare this figure with Figure 5.22).

Before stating the schedulability condition I prove the following lemma:

Lemma 4 *For each t,* $n\gamma_i(t) \leq \gamma_i(nt)$.

Proof.

I first show that $n\eta(t) \leq \eta(nt)$.
$n\,\eta(t) = n\,\left\lfloor \frac{\max(t-T_i+C_{i1}+C_{i2},0)}{T_i} \right\rfloor \leq \left\lfloor n\,\frac{\max(t-T_i+C_{i1}+C_{i2},0)}{T_i} \right\rfloor =$
$= \left\lfloor \frac{\max(n(t-T_i+C_{i1}+C_{i2}),0)}{T_i} \right\rfloor = \eta(nt).$
Now, since $C_{i1} + C_{i2} - T_i \leq 0$, I have that
$\left\lfloor \frac{\max(n(t-T_i+C_{i1}+C_{i2}),0)}{T_i} \right\rfloor \leq \left\lfloor \frac{\max(nt-T_i+C_{i1}+C_{i2},0)}{T_i} \right\rfloor.$
Then,
$n \cdot \min\left(C_{i2},\, \max\left(0,\, t - (2+\eta(t))T_i + C_i + C_{i2}\right)\right)$
$\leq \min\left(C_{i2},\, \max\left(0,\, n(t - (2+\eta(t))T_i + C_i + C_{i2})\right)\right)$
then, since $C_{i1} + C_{i2} - 2T \leq 0$, I have that
$\leq \min\left(C_{i2},\, \max\left(0,\, nt - (2+n\eta(t))T_i + C_i + C_{i2}\right)\right)$
$\leq \min\left(C_{i2},\, \max\left(0,\, nt - (2+\eta(nt))T_i + C_i + C_{i2}\right)\right).$

Given the two previous results, I have that

$n\gamma_i(t) = n\eta(t)\,C_{i2} + n\min\left(C_{i2},\, \max\left(0,\, t - (2+\eta(t))T_i + C_i + C_{i2}\right)\right)$
$\leq \gamma_i(nt) = \eta(nt)\,C_{i2} + \min\left(C_{i2},\, \max\left(0,\, nt - (2+\eta(nt))T_i + C_i + C_{i2}\right)\right) =$
$= \gamma_i(nt).$ □

5.5.4.2 Using collected bandwidth for DSP scheduling

The $\gamma(t)$ function basically gives a measurement of the capacity that can be exploited given a certain period. This fact means that a given schedulable task set remains feasible if a regular task τ' with period P consumes $\gamma(P)$ units of time more than declared or, similarly, if a regular task τ' with period P and capacity $\gamma(P)$ is added to the system. This property holds if the CEDF+SRP scheduling algorithm is used to schedule the task set, as stated by the following theorem:

Theorem 8 *If a task set* $\mathcal{T}$ *can be feasibly scheduled under the CEDF+SRP algorithm, then* $\mathcal{T}' = \mathcal{T} \cup \tau'$*, where* τ' *has period* P *and execution time* $\gamma(P)$*, is also feasible under CEDF+SRP.*

Proof.

Let me consider a feasible schedule of $\mathcal{T}$, where the CPU is idle when the DSP is in execution. When τ' is added to the system, all the chunks executing on the DSP that contributes to the capacity $\gamma(P)$ have always a deadline less than or equal to the deadline of τ'.

I now prove that the insertion of τ' maintains the schedulability of the system. If the resulting schedule is not feasible there must be at least one chunk belonging to a task τ_i that miss its deadline. Let d^* be such a deadline. Since the task set was schedulable before the insertion of τ', I have

$$D(t,\, d^*) < d^* - t - \gamma(d^* - t)$$

because in the interval $[d^*, t]$ there is at least $\gamma(d^* - t)$ DSP time. Adding the computational time of τ' to each side, I have:

$$D(t,\, d^*) + \left\lfloor \frac{d^* - t}{P} \right\rfloor \gamma(P) \leq d^* - t - \gamma(d^* - t) + \left\lfloor \frac{d^* - t}{P} \right\rfloor \gamma(P)$$

From Lemma 4, I have that

$$\left\lfloor \frac{d^* - t}{P} \right\rfloor \gamma(P) \leq \gamma\left(\left\lfloor \frac{d^* - t}{P} \right\rfloor P \right).$$

Hence, since $\gamma(t)$ is a monotonic increasing function,

$$\gamma\left(\left\lfloor \frac{d^* - t}{P} \right\rfloor P \right) \leq \gamma\left(d^* - t\right),$$

and therefore

$$D(t,\, d^*) + \left\lfloor \frac{d^* - t}{P} \right\rfloor \gamma(P) \leq d^* - t$$

This proves that task τ_i cannot miss its deadline. □

Note that a priori is not known which is the task that will be scheduled into the DSP time (in general, any regular task can be scheduled there). What happens is that it is guaranteed that task τ' will be scheduled, and that the system is still feasible.

Also note that Theorem 8 does not hold if the plain SRP scheduling algorithm is used. That is because SRP does not separate the deadlines of each chunk; the additional task can be scheduled under SRP maintaining the feasibility of the system only if its deadline is always greater than those of the instances that contributes to $\gamma(t)$, that leads to a more pessimistic usage of the DSP time on the master processor.

Using the results of Theorem 8, it is possible to reclaim the execution time of a single DSP task. Figure 5.24 shows how Theorem 8 can be used to design an acceptance test that consider the contribution of the reclaiming from different DSP tasks. The basic idea of the acceptance test is to first try to guarantee each DSP task, and then use the capacity reclaimed to guarantee the other regular tasks.

To do that, I introduce a function $\mu_i(t)$, that models the worst case execution time utilization of a regular task τ_i (see Figure 5.25), and a function $\Gamma(t)$ that at each step sums together the functions $\gamma(t)$ and $\mu(t)$ of the accepted tasks. In general, $\Gamma(t)$ represents the time (depending on the period) that can be reclaimed at a given step of the guarantee algorithm.

$U_{tot} = 0$
$\Gamma(t) = 0$

```
 1:   bool guarantee(T, C1, C2, C3)
 2:   {
 3:         C = C1 + C2 + C3
 4:         if (Γ(T) > 0) {
 5:               reclaim = min(Γ(t), C)
 6:               C = C − reclaim
 7:         }
 8:         else
 9:               reclaim = 0
10:         U = C/T
11:         if (U_tot + U > 1)
12:               return false
13:         else {
14:               if (reclaim > 0) Γ(t) = Γ(t) − μ_{reclaim,T}(t)
15:               U_tot = U_tot + U
16:               if (C2 > 0) Γ(t) = Γ(t) + γ_{T,C1,C2,C3}(t)
17:               return true
18:         }
19:   }
```

Figure 5.24: An acceptance test that consider more than one DSP task.

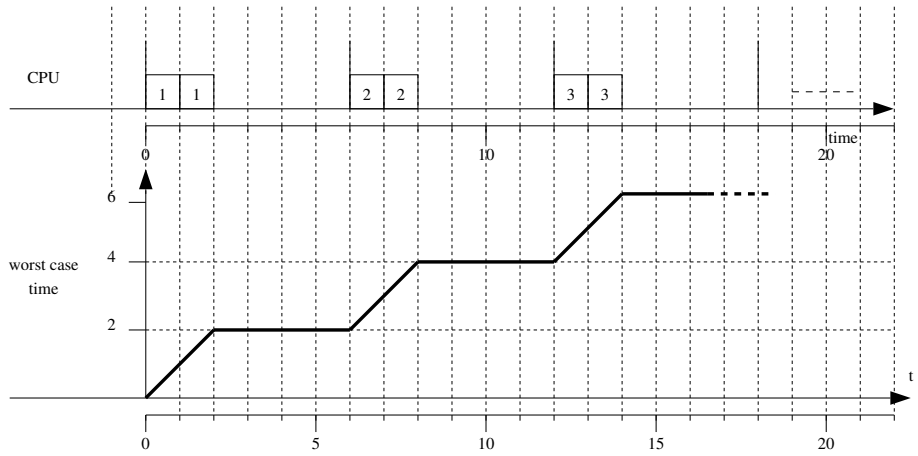

Figure 5.25: A function $\mu(t)$ for a task with execution time $C_i = 2$ and period $T_i = 5$.

The algorithm works as follows: first of all, the function $\Gamma(t)$is initialized to zero for each t, and the total utilization factor of the guaranteed tasks U_{tot} is set to 0. Then, function `guarantee()` have to be called for each task $\tau_i$of the task set, starting from DSP tasks, in order of increasing periods, until the function returns *false* or all the tasks have been accepted. Lines 4 to 9 tries to reclaim some execution time from DSP tasks already accepted. Line 11 tests if the task set already guaranteed plus the task under acceptance can be guaranteed. Lines 14 to 16 updates the global variables. The notation $\mu_{reclaim,T}(t)$ means a function $\mu_i(t)$ for a task $\tau_i$with $C_i = reclaim$, and $T_i = T$. The notation $\gamma_{T,C_1,C_2,C_3}(t)$means a function $\gamma_i(t)$ for a task τ_i with $C_{i1} = C_1$, $C_{i2} = C_2$,$C_{i3} = C_3$, and $T_i = T$.

Please note that:

- if we define the real load on the CPU as

$$U_{CPU} = \sum_{DSP\,tasks} \frac{C_{i1} + C_{i3}}{T_i} + \sum_{regular\,tasks} \frac{C_i}{T_i},$$

  then it holds that $U_{tot} > U_{CPU}$ (because U_{tot} also includes the DSP time of DSP tasks);

- the designer can make use of function $\Gamma(t)$ to know at which period a given computation time will be available, to appropriately tune the design of the system;

- the time *made available* by a function $\gamma_i(t)$ in $\Gamma(t)$ scales with a slope comparable to $\frac{C_{i2}}{T_i}$, whereas the time that can be *used* by a periodic task scales with a slope $\frac{\Gamma(t)}{T} < \frac{C_{i2}}{T_i}$, that means that *a single periodic task cannot reclaim all the available bandwidth made available by* $\Gamma(t)$. In general, only an infinite sequence of reclaimed tasks with increasing period can use all the bandwidth reclaimed using a function $\gamma(t)$.

These notes will be further discussed with an example in Section 5.5.5.

5.5.5 Simulation results

In this section I present some simulation results that show the peculiarities of the approach I propose. In particular, the simulations show the behavior of the $\gamma(t)$ function considering only a single task. This allows to better highlight the relation between $\gamma(t)$ and t, showing how much of the DSP time can be collected using my method. The results with more than one task heavily depends on the task parameters and they can be derived in a similar way summing the contributions of each task. Figures report the behavior of $\gamma(t)$ measured in multiples of the task period. The plots are influenced only by the ratios between the period T_i and the execution times C_{i1}, C_{i2}, and C_{i3}, and not by their absolute values.

In particular, Figure 5.26 plots the percentage of DSP utilization considering the task described in Figure 5.22. It can be noted that the percentage of the DSP time collected using the proposed approach increases as the period of the additional task increases. Please also note that the DSP time cannot be exploited with very small periods.

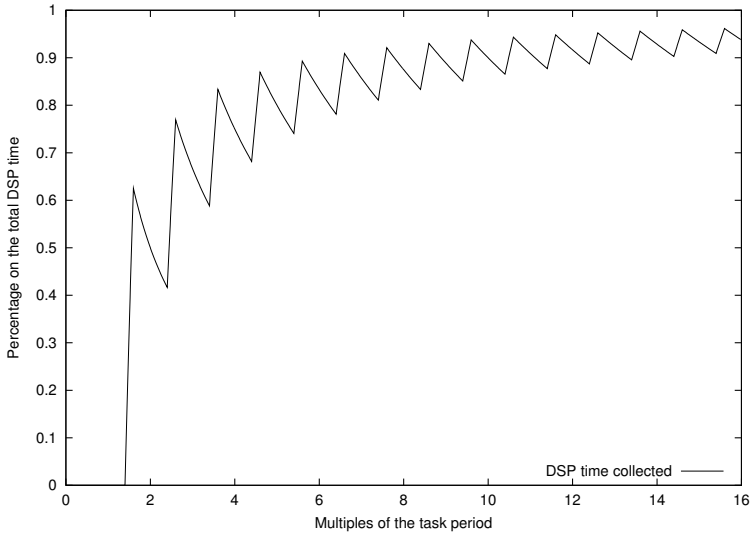

Figure 5.26: Percentage of DSP time collected by $\gamma_i(t)$ using the settings on Figure 5.22.

Figure 5.27 shows the percentage of DSP time collected in three configurations consisting of a single DSP task characterized by the following parameters: $C_{i1} = C_{i3} = \frac{P(1-d)}{2}$, $C_{i2} = Pd$, where P is the considered period for the additional task, and d is the DSP percentage with values 0.25, 0.5 and 0.75. Note that there are points (multiples of P) in which the DSP time can be collected totally.

Figure 5.28 compares the DSP time collected by my approach and the time that can be collected under the SRP protocol using a task with a *big* relative deadline (i.e., in background). As it can be noted, the approach significantly outperforms the latter in some regions, making possible a better utilization of the DSP time. In particular, Figure 5.29 shows the difference between the two plots, highlighting the regions where CEDF+SRP gives better results.

As an example of the usage of the constructive method of Section **??** consider a task set composed by a DSP task τ_1 and by a set of regular tasks τ_i, with $i > 1$. Task τ_1 has $C_{11} = 1$, $C_{12} = 2$, $C_{13} = 1$, $T_1 = 6$; regular tasks τ_i have $C_i = 2$. The purpose of the example is to assign the smallest period possible to each task τ_i.

Following the acceptance test described in Figure 5.24, task τ_1 is accepted first. After accepting τ_1, $U_{tot} = \frac{4}{6} = 0.667$, $U_{CPU} = \frac{2}{6} = 0.333$, and $\Gamma(t) = \gamma_{T_1,C_{11},C_{12},C_{13}}(t)$ ($\Gamma(t)$ is plotted in Figure 5.30.a).

Task $\tau_2$can then be accepted at period $T_2 = 6$ (using all the spare bandwidth left by τ_1), giving $U_{tot} = 1$, $U_{CPU} = \frac{4}{6} = 0.667$, and $\Gamma(t)$ unaltered.

Task τ_3 will be accepted at period $T_3 = 8$ (because $\Gamma(8) = 2$, see Figure 5.30.a). Since we are reclaiming computation time from $\Gamma(t)$, we have to compute the function $\mu_3(t)$ (see Figure 5.30.b, that plots $-\mu_3(t)$). At this step, U_{tot} is (and will be in the next steps) unaltered, $U_{CPU} = 0.916667$, and $\Gamma(t) = \gamma_1(t) - \mu_3(t)$ (Figure 5.30.c).

At that point, the only available capacity is after time 31. That capacity can be used to accept other tasks. Please note that the function $\Gamma(t)$ is not monotonic ascending, because

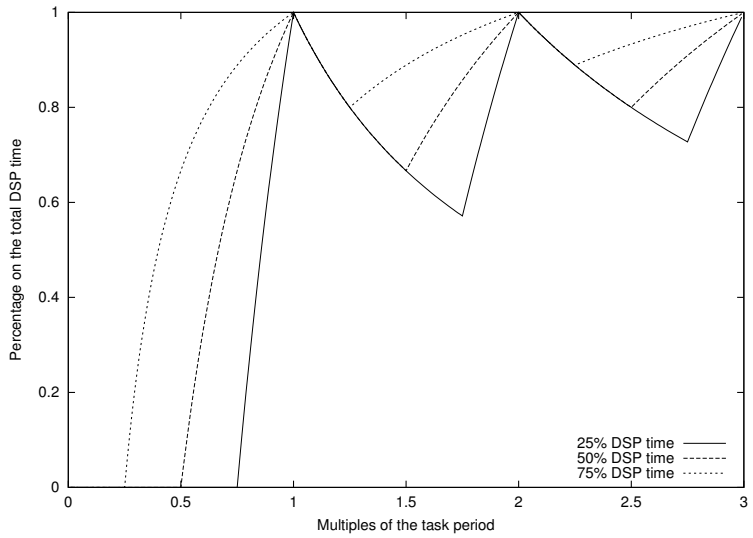

Figure 5.27: Percentage of DSP Time collected with different task settings.

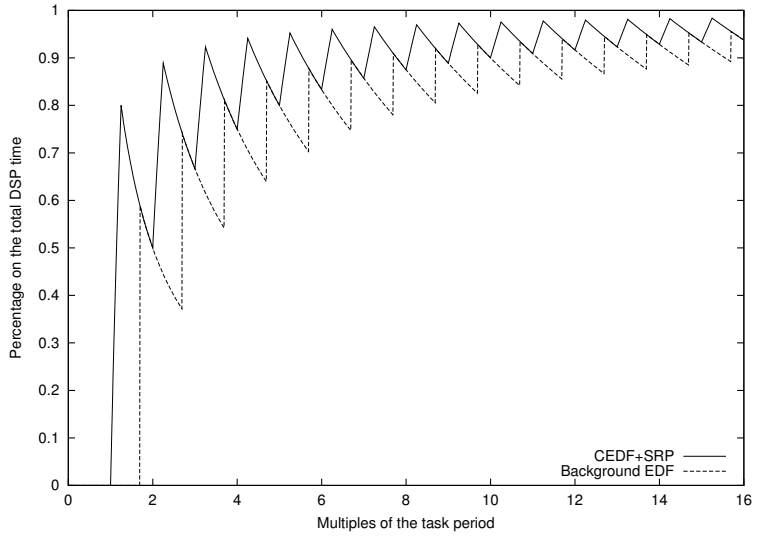

Figure 5.28: Comparison between the collected DSP time using CEDF+SRP and SRP with a big relative deadline. The parameter of the DSP task were $P = 100$, $C_{i1} = 5$, $C_{i3} = 45$, $C_{i2} = 25$.

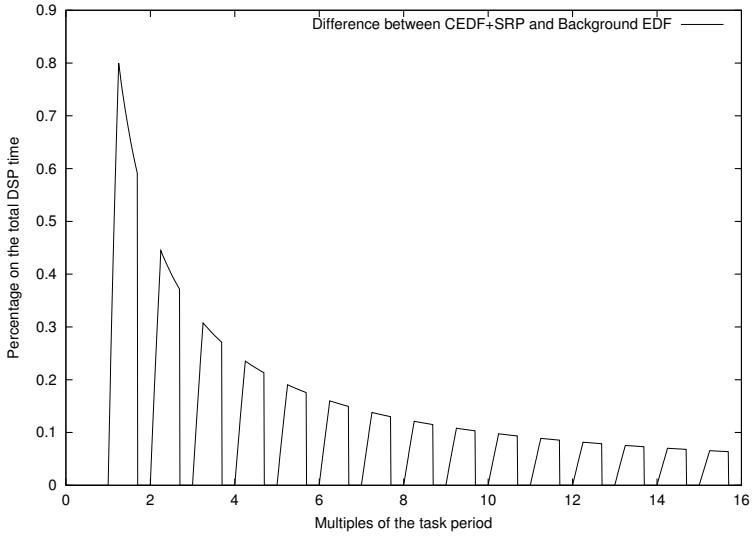

Figure 5.29: Difference between the two plots of Figure 5.28.

τ_i	T_i	U_{CPU} after accepting task τ_i
τ_4	32	0.979167
τ_5	128	0.994792
τ_6	512	0.998698
τ_7	2048	0.999674
τ_8	8192	0.999919
τ_9	32768	0.999980
τ_{10}	131072	0.999995
τ_{11}	524288	0.999999

Table 5.1: Periods and CPU utilization for tasks $\tau_4$to τ_{11}.

of the conservative assumption made with $\mu_j(t)$ starting just when arrived.

Table 5.1 shows the next periods and U_{CPU} values for task τ_4 to task τ_{11}. Please note that the CPU utilization U_{CPU} after accepting task τ_{11} is really near to 1, meaning that it is possible to design systems really near to the full utilization of the CPU computation time.

Finally note that, as explained in Section 5.5.4.2, a finite number of regular tasks cannot use all the available bandwidth left by DSP tasks.

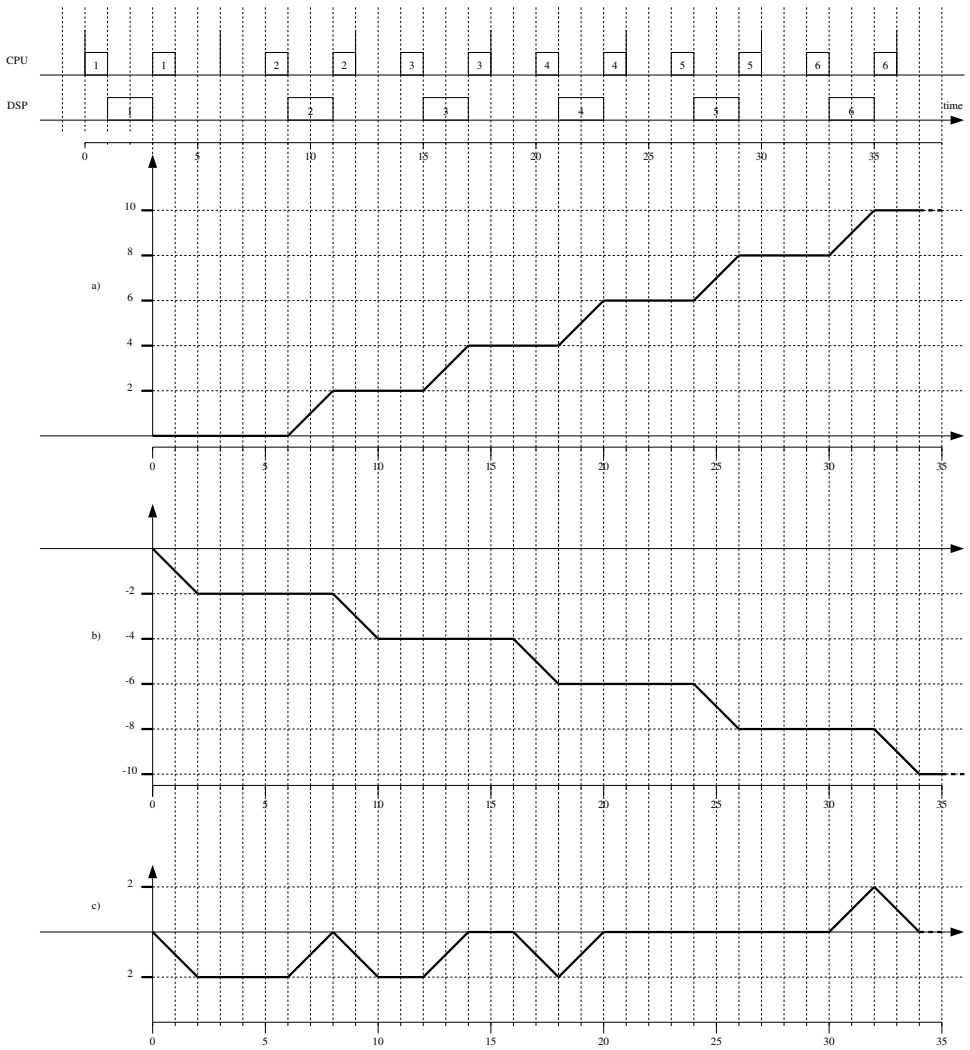

Figure 5.30: An example showing the constructive method of Section **??**: a) $\Gamma(t) = \gamma_1(t)$ after accepting τ_1; b) the function $-\mu_3(t)$; c) $\Gamma(t) = \gamma_1(1) - \mu_3(t)$ after accepting τ_3.

Chapter 6

Conclusions

This thesis has been strongly motivated by the arrival of the new generation of multiple-processor on-a-chip systems for embedded applications. These platforms not only require real-time executives, but also ask for kernel mechanism that save as much RAM space as possible, RAM memory being one of the most expensive components in those systems.

This thesis addressed the problem:

> What is the best way to design minimal real-time operating systems for embedded systems-on-a-chip?

To this end, I designed a set of scheduling algorithms and optimization techniques for mono and multiprocessors, providing evidence of their effectiveness. The thesis can be divided in two parts, the first one related to the design of single and homogeneous multiprocessors, and a second part for the design of solutions for heterogeneous multiprocessors.

In particular, for the first part

- I designed a scheduling algorithm that merges EDF techniques together with preemption threshold techniques, ideal for scheduling uniprocessor systems with small kernels for SoC that have synchronization requirements as well as low memory footprint.
- I developed an optimization algorithm for the assignment of preemption thresholds and the grouping of tasks in non-preemptive sets. The methodology allows to evaluate the schedulability of task sets and to find the schedulable solution (the task groups) that minimize the RAM requirements for stack.
- I designed an extension of the SRP policy to multiprocessor systems and global shared resources (MSRP).
- I developed task allocation algorithm based on simulated annealing.

The main contribution of this first part consists in realizing that real-time schedulability and the minimization of the required RAM space are tightly coupled problems and can be efficiently solved only by devising innovative solutions. The objective of RAM minimization

guides the selection of all scheduling parameters and is a factor in all my algorithms. The experimental runs show an extremely effective reduction in the occupation of RAM space when compared to conventional algorithms.

In the second part, I addressed the problem of scheduling a set of tasks in an asymmetric multiprocessor consisting of a general purpose CPU and a DSP. Although this kind of architecture can be considered as a special case of a multiprocessor system, its peculiarity allows to perform more specific analysis which is less pessimistic than the one typically used in distributed systems with shared resources.

In particular,

- I designed a method for computing blocking times, which has been showed to be more effective than the classical method adopted in the Distributed Priority Ceiling Protocol [53, 50].
- I designed CEDF and CEDF+SRP, two new scheduling algorithms for uniprocessors, obtained by modifying the traditional EDF and SRP scheduling algorithms using a checkpoint technique. Using these results, I addressed the problem of scheduling a set of tasks in an asymmetric multiprocessor consisting of a general purpose CPU and a DSP.

A complete analysis of the approach has been presented together with some simulations that show the performance enhancements that can be obtained using the proposed techniques.

List of Papers

This thesis is based on and extends the work and results presented in the following papers and publications:

1. Paolo Gai, Giuseppe Lipari, Marco di Natale, Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip, Proceedings of the 22th Real-Time Systems Symposium, December 2001.

2. Paolo Gai, Giuseppe Lipari, Marco Di Natale, Design Methodologies and Tools for Real-Time Embedded Systems, Special Issue of Design Automation for Embedded Systems, 2002.

3. Paolo Gai, Luca Abeni, Giorgio Buttazzo, Multiprocessor DSP Scheduling in System-on-a-chip Architectures, Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems, June 2002.

4. Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini and Paolo Marceca, A comparison of MPCP amd MSRP when Sharing Resources in the Janus Multiple Processor on a Chip Platform, Proceedings of RTAS 2003, Washington DC, May 2003.

The following papers and publications are related but not covered in this thesis:

1. Paolo Gai, Giuseppe Lipari, Luca Abeni, Marco di Natale and Enrico Bini, Architecture for a Portable Open Source Real-Time Kernel Environment, Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial, November 2000.

2. Paolo Gai, Luca Abeni, Massimiliano Giorgi and Giorgio Buttazzo, A New Kernel Approach for Modular Real-Time systems Development, Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems, June 2001.

3. Paolo Gai, Giuseppe Lipari, Marco di Natale, A Flexible and Configurable Real-Time Kernel for time predictability and Minimal RAM Requirements, Scuola Superiore S. Anna, 2001.

4. Paulo Pedreiras, Luis Almeida and Paolo Gai, The FTT-ethernet protocol: Merging flexibility, timeliness and efficiency, Proceedings of the 14th IEEE Euromicro Conference in Real-Time Systems, June 2002, Vienna.

5. Paulo Pedreiras, Luis Almeida, Paolo Gai and Giorgio Buttazzo, FTT-Ethernet: A Platform to Implement the Elastic Task Model over Message Streams, Proceedings of the WCFS '02, 2002, August, Vasteras.

6. Paolo Gai and Giorgio Buttazzo, Mutual exclusion in operating systems with application defined scheduling, Proceedings of ARTOSS Workshop, Porto, July 2003.

7. Paolo Gai and Giorgio Buttazzo, An Open Source Real-Time Kernel for Control Applications, Proceedings of the 47imo Convegno nazionale ANIPLA, Brescia, Italy, November 2003.

8. G. Lipari, P. Gai, M. Trimarchi, G. Guidi and P. Ancilotti, A Hierarchical Framework for Component-Based Real-Time Systems, International Symposium on Component-based Software Engineering (CBSE7), Edimburgh, May 2004.

9. G. Lipari, P. Gai, M. Trimarchi and G. Guidi, A Hierarchical Framework for Component-Based Real-Time Systems, Workshop on Test and Analysis of Component Based Systems (TACoS 04), April 2004.

10. Michele Cirinei, Antonio Mancina, Davide Cantini, Paolo Gai and Luigi Palopoli, An Educational Open Source Real-Time Kernel for small embedded control systems, to appear on the Proceeding of ISCIS 2004, Turkey, November 2004.

Bibliography

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons, 1989.

[2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):pp. 134–165, May 1997.

[3] James Anderson and Anand Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, December 2000.

[4] Bjorn Andersson. *Static-priority scheduling on multiprocessors*. PhD thesis, Chalmers University, Goteborg, Sweden, September 2003.

[5] Audi, BMW, DaimlerChrysler, Porsche, and Volkswagen. Herstellerinitiative software (his). http://www.his-automotive.de, April 2004.

[6] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(8):284–292, Sep 1993.

[7] T. P. Baker. A stack-based allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, december 1990.

[8] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

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

[10] Sanjoy Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. In *Proceedings of the EuroMicro Conference on Real-Time Systems, Porto, Portugal*, pages 195–202, July 2003.

[11] S.K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 6, 1996.

[12] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, December 1990.

[13] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *The Journal of Real-Time Systems*, 2, 1990.

[14] Robert Baumgartl and Hermann Hartig. Dsps as flexible multimedia accelerators. In *Second European DSP Education and Research Conference (EDRC'98)*, Paris, September 1998.

[15] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[16] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 1995.

[17] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[18] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[19] Chia-Mei Chen and Satish K. Tripathi. Multiprocessor priority ceiling based protocols. Technical report, Univ. of Maryland CS, 1994.

[20] Jing Chen and Alan Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proc.10th EuroMicro Workshop on Real-Time Systems (EuroMicro '98)*, 1998.

[21] Wesley W. Chu and Lance M-T Lan. Task allocation and precedence relations for distributed real-time systems. *IEEE Transactions on computers*, C-36(6), June 1987.

[22] Altera Corporation. The NIOS II embedded processor. http://www.altera.com/nios, May 2004.

[23] T. S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1993.

[24] Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[25] M. L. Dertouzos and Aloysius Ka-Lau Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on software engineering*, 15(12), December 1989.

[26] Inc. Express Logic. http://www.threadx.com. available on Internet.

[27] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. *Parallel Processing, Springer-Verlag*, pages 1–18, 1995.

[28] A. Ferrari, S. Garue, M Peri, S. Pezzini, L.Valsecchi, F. Andretta, and W. Nesci. The design and implementation of a dual-core platform for power-train systems. In *Convergence 2000*, Detroit (MI), USA, October 2000.

[29] Paolo Gai, Luca Abeni, and Giorgio Buttazzo. Multiprocessor dsp scheduling in system-on-a-chip architectures. In *Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems*, June 2002.

[30] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[31] 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. In *Proceedings of the 22th Real-Time Systems Symposium*, December 2001.

[32] R.L. Graham. *Bounds on the performance of scheduling algorithms*, chapter 5. Coffman Jr. E. G. (ed.) Computer and JobShop Scheduling Theory, Wiley, New Yorj, 1976.

[33] OSEK Group. *OSEK/VDX Operating System Specification 2.2*. available at http://www.osek-vdx.org, 2001.

[34] OSEK Group. *OSEK COM Operating System Specification 3.0.1*. available at http://www.osek-vdx.org, 2004.

[35] Texas Instruments. *Military Semiconductor Products Fact Sheet SM320C80 / SMJ320C80 / 5962-9679101 SGYV006C*, August 2000.

[36] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[37] Bala Kalyanasundaram and Kirk R. Pruhs. Eliminating migration in multi-processor scheduling. *To appear in a special issue of Journal of Algorithms devoted to selected papers from the ACM/SIAM Symposium on Discrete Algorithms*, 1999.

[38] A. Khemka and R. K. Shyamasunda. Multiprocessor scheduling of periodic tasks in a hard real-time environment. Technical report, Tata Institute of Fundamental Research, 1990.

[39] Gilad Koren, Amihood Amir, and Emanuel Dar. The power of migration in multiprocessor scheduling of real-time systems. In *In Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 226–235, San Francisco, California, January 1998.

[40] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Real-Time Computing Systems and Applications*, October 1997.

[41] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.

[42] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[43] Giuseppe Lipari and Giorgio Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46:327–338, 2000.

[44] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[45] The national italian project madess ii. http://www.madess.cnr.it, 2002.

[46] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec 1999.

[47] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with configurable locks for multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–205–II–208, Boca Raton, FL, 1993. CRC Press.

[48] M. Di Natale and J. Stankovic. Scheduling distributed real-time tasks with minimum jitter. *Transaction on Computer*, 49(4), 2000.

[49] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Journal on Real Time Systems*, 9, 1995.

[50] R. Rajkumar. Synchronization in multiple processor systems. In *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[51] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, chapter 3. Kluwer Academic Publishers, 1991.

[52] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[53] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *In Proceedings of the 1988 Real Time System Symposium*, 1988.

[54] Srikanth Ramamurthy. A lock-free approach to object sharing in real-time systems. Master's thesis, University of North Carolina at Chapel Jill, 1997.

[55] Ken Kim Philips Research. Increasing functionality in set-top boxes. In *Proceedings of IIC-Korea, Seoul*, 2001.

[56] M. J. Rutten, J. T. J. van Eijndhoven, and E. J. D. Pol. Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors. In *Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems*, June 2002.

[57] Saowanee Saewong and Ragunathan (Raj) Rajkumar. Cooperative scheduling of multiple resources. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1999.

[58] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the Real Time Systems Symposium*, December 2000.

[59] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[60] Hiroaki Takada and Ken Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. Technical Report 97-01, Dept. of Information Science, University of Tokyo, Jan 1997.

[61] K. Tindell. An extendible approach for analysing fixed priority hard real-time tasks. Technical Report YCS 189, Department of Computer Science, University of York, December 1992.

[62] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks (an np-hard problem made easy). *Real-Time Systems Journal*, 1992.

[63] Yun Wang and Manas Saksena. Fixed priority scheduling with preemption threshold. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.