Supporting Time-Sensitive Activities in a Desktop Environment

Luca Abeni

December 16, 2002

Contents

Preface

This PhD thesis represents the end of an important part of my life, in which I have been involved in real-time research. In this dissertation, you can find a summary of what I did during these 4 years, and I hope that you will find it interesting. I release my PhD thesis under the FDL (see Appendix A) as an acknowledgement to the Open Source community, and to encourage the diffusion of its contents. After all, research is sharing results.

Before going on, I ask you all, my readers, to forgive me for my terrible english and for all the linguistic and grammar errors that you will find in this manuscript. During my PhD, I learned many things and my english improved a lot, but it is still not good enough for writing a whole dissertation.

I had a very good time as a PhD student, I had the occasion to travel around the world, meeting new people and knowing new cultures. In some moments, I worked hard (and I hope I obtained some decent results), but most of the time doing real-time research has been a pleasure. Hence, I'd like to thank everyone which helped me and worked with me during this period, starting from the good old ReTiS guys, **Gerardo** and **Peppe**, who showed me how real-time research can be interesting, and can be fun.

During my PhD, I have been a visiting student in the US for two times: first in CMU, and then in OGI. I thank **professor Raj Rajkumar**, who permitted me to have a great experience in CMU and taught me many things about real-time systems and about research in general, all the members of the Real-Time Multimedia Lab (**Sourav** Ghosh, Saowanee - **Apple** - Saewong, **Dionisio** De Niz, and **Akihiko** Miyoshi) for their friendship and their help. Thanks to **professor Jonathan Walpole**, who hosted me in OGI and gave me the possibility to have a great time in Portland and to learn a lot of things about Operating Systems, **Ashvin** Goel, for his friendship and help (hey Ash, thanks for working on it!), and all the other members of the SysL in OGI.

Also, thanks to my advisor **Giorgio Buttazzo**, and to all the people who believed in me. A final special “thank you” to the brave guys who helped me in making this dissertation less unreadable, and in particular to **Tonino** and **Luigi**.

Most important, I want to say **thanks to** the One to which everyone should say thanks for everything: **the Lord our God**, who created us all, and saved us dying for us. Too many times I kept my faith out of my "real life", but now I want to recognise that nothing of what I did and obtained in the past was possible without God's help, and He deserve the biggest acknowledgement for everything He did for me, He is doing for me every day, and He will do in the future!!!

Finally, I want to say something that is completely unrelated with this thesis, but is tragically important in these days: let's say **No to the War**!!! (Hey... Do you remember my RTSS and RTLWS presentations? ;) Remember: **violence** (and hence war) **_IS NEVER A SOLUTION_!!!**

And now, after this (hopefully not too boring) preface, we are ready to talk about more technical stuff. So,

Welcome in the dark kingdom of real-time systems, where fearless knights fight against evil unpredictabilities to defend the QoS and guarantee system schedulability...

Chapter 1

Introduction

Don't believe in manuals!!!
Herman Haertig

Corollary: *Don't believe in PhD dissertations...*
Luca

The recent evolution of computer technology made personal computers powerful enough to perform new typologies of activities, like managing multimedia streams in real-time. As a result, a modern workstation can be used to run new kinds of applications, such as MultiMedia ones, as well as mixes of heterogeneous applications, each of them with different requirements. For example, it may be needed to concurrently run a word processor (requiring a large amount of memory) together with an audio/video streaming application, while a software mixer is mixing different audio sources in real time and the Operating System (OS) kernel is receiving a stream of packets from the network.

1.1 Motivation

The need to run etherogenous mixes of different classes of applications introduces new problems and requirements in handling hardware and software resources. For example, consider the most common abstraction provided in traditional servers and workstations, that is *multiprogramming*: the execution of multiple application tasks is interleaved in such a way as to create the illusion of running simultaneously. The traditional requirement is that all the applications proceed fairly (eventually according to some user specified weights) to avoid starvation, that interactive applications respond to user input in a small time (but there is no clear definition of "small time"), and that the average response time of all applications is minimised. When dealing with (for example) multimedia applications, these requirements have to be revisited.

1.1.1 Time Sensitive Applications

As said, some new applications may be characterised by additional timing requirements: for example, an audio MPEG player (such as the famous WinAMP, or XAMP) should fill the sound card buffer *before* the sound card needs the data. When this constraint is not respected, the result can be incorrect, even if AMP decodes the mpeg audio correctly, because the decoded data are generated too late. As a result, the user will hear some unpleasant noise instead of its favourite song. Hence, in these cases *the correctness of a result does not only depend on the output values, but also on the time when the result is generated* (sometime, an approximate value computed on time can be better than an exact value computed late). In this dissertation, such applications are referred as *time sensitive applications.*

As will be shown in Chapter 2, classical real-time theory (pioneered by Liu & Layland [LL73]) provides techniques for dealing with temporal constraints, but its application to generic time sensitive applications running on a workstation OS can be difficult. For example, traditional real-time theory mainly focuses on embedded control applications, which is only a subset of all possible time sensitive applications considered in this work. In fact, double thinking about it, it is possible to see that many applications are time sensitive, even if they are not traditionally considered real-time applications. An interesting example (that is also an "hot topic" in current OS research) is represented by web servers: enabling an http server to respond to a request in a specified time is fundamental to ensure that the web server will provide the required Quality of Service (QoS).

Other notable examples of time sensitive applications are multimedia applications in general (streaming, video conference, audio or video players, and so on), digital signal processing applications (such as software mixers, software modems, or audio synthesiser), virtual reality applications, and many others. Taking the above argument to the limit, we can say that all the applications are time sensitive: even a word processor (an example of "traditional computer application") is quite useless if it takes too much time to start or to print a document.

1.1.2 Current OS Support

From the previous discussion it is easy to understand that time sensitive applications are becoming more and more important, and supporting them will become a fundamental issues in future OSs. However, the most common OS kernels and applications are not designed to support time sensitive activities nor to run heterogeneous mixes of applications having contrasting requirements. As a result, resources are allocated (and tasks are scheduled) according to "general purpose" goals, such as reducing the average latency, and it is difficult for the applications to provide a controllable QoS.

The most commonly proposed solution is to increase the hardware power (and the amount of available resources), overengineering the system so that it will result to be underloaded and all the applications will be served in a reasonable way. As the power of the hardware is continuously increasing, this solution is becoming cheaper and cheaper, but it results in an inefficient exploitation of the available resources and in underutilisation of the system, encouraging a bad programming practice. Moreover, every application will likely run with the correct timing in the average case, but it cannot be guaranteed that this will always happen. For example, if a resource greedy application, such as Microsoft Word, or SUN StarWord, is launched when a streaming application is running, it is almost sure that the streaming application will experience a, hopefully transient, failure.

Hence, the inappropriateness of a traditional OS kernel for supporting time sensitive applications is generally due to design goals that did not consider support for timed activities. In particular, there are both theoretical and practical issues, such as:

- **task scheduling**. The general purpose schedulers provided by the most common OS are not designed to properly serve time sensitive activities. Note that all the system resources (not only the CPU) must be properly scheduled;

- **resource allocation policy**. As a result of the inappropriate scheduling algorithms, system resources cannot be correctly allocated to the various tasks in order to respect temporal constraints;

- **kernel structure**. Most of the current OS kernels are based on a **monolithic structure** derived from BSD. This results in a series of problems in accounting resource usage to the correct tasks, and in assigning resources to tasks in a proper way;

- **temporal resolution** of the system. General purpose OS kernels are generally based on a periodic interrupt that triggers accounting and scheduling activities, generally at a rate of 100 times per second. This solution often results in a poor scheduling and accounting.

1.2 Contribution of this Dissertation

The thesis supported in this dissertation is that the appropriate scheduling of system resources and the use of proper resource allocations policies in the OS kernel [1] permit to correctly support time sensitive applications without over-engineering the system. The use of an appropriate kernel structure (or a a proper modification of the traditional monolithic structure) is necessary to correctly implement the scheduling algorithm, to implement an accurate resource allocation policy, and to effectively schedule all the system resources. This enable to perform QoS guarantees in a workstation OS, enabling less powerful computers to support time sensitive applications in a more predictable way.

1.2.1 Scheduling and Resource Allocation

Generic resource scheduling techniques are often inadequate for respecting time constraints, hence the first element to support time sensitive applications is an appropriate scheduling algorithm. The algorithm of choice must provide a theoretical foundation that permits to provide some kind of QoS guarantee.

Using real-time theory it is possible to provide time guarantees under some (very strict) assumptions, such as the complete a-priori knowledge of the system. This includes a-priori knowledge of the task execution times (or of their upper bounds), arrival times, and so on. While this assumption is reasonable in an embedded system, where all the tasks are known in advance and can be adequately analysed, any assumption regarding a-priori information is not reasonable in a desktop operating system. In fact, in such a system the number of active tasks can vary, and cannot be predicted; moreover, the same application may need to run on a big number of different machines, making impossible to know the execution times in advance. These systems are often referred as *Open Systems* [DLS97, DL97], to distinguish them from *Closed Systems*, in which all the tasks that will run in the system are known in advance. An Open System is a general purpose computer system in which it is not possible to know a-priori neither the number nor the characteristics

[1] eventually associated with user level QoS management and application-level adaptation

of the applications that will be run. Typically, in an Open System applications with different levels of Quality of Service may coexists: hard real-time, multimedia and interactive non-real-time applications.

Hence, in an Open System it is necessary to protect applications from the misbehaviours of other applications. This property is called *Temporal Isolation*: the net effect is that each application executes as it were on a slower dedicated processor. The Resource Reservation approach [MRT93] is a good way to implement temporal isolation using real-time techniques, and has been proven to be very effective in the joint scheduling of Hard Real Time (HRT) and Soft Real Time (SRT) applications in Open Systems. Another possible way for implementing temporal isolation is *Proportional Share* (PS) scheduling [PG93, PG94].

All those scheduling algorithms are characterised by low-level scheduling parameters, that can be difficult to tune in the proper way. In particular, the scheduling algorithm constitutes a *mechanism* provided by the kernel to allocate resources to applications in a specified way, and a *policy* for allocating resources must be specified at a higher level. For this reason, a QoS manager that exports some high-level task model is needed to implement the resource allocation policy by controlling the low-level parameters of the scheduler.

1.2.2 Kernel Structure

Most of the scheduling algorithms presented in the literature assume that the scheduler has the total control of the system, and can decide when to preempt the currently running task and to schedule a new task. Moreover, scheduling decisions are assumed to be immediate, and no interference from external factors is considered.

A real OS is more complex: to preserve the integrity of some data structures and the atomicity of some operations, tasks cannot be arbitrarily preempted, the scheduler cannot be invoked at arbitrary time instants, but only when specific events (such as a timer interrupt) occur, and exactly measuring the execution time used by a task is not easy. Moreover, external events such as hardware interrupts add another level of complexity, consuming execution time and decreasing the predictability of the system.

The influence of these factors on task scheduling depend on how the kernel is internally organised. In particular, traditional OS kernels (based on the monolithic kernel structure) are not able to preempt a task when it is executing a system call, and interrupts have the precedence over all the user applications. This creates a discrepancy between the theoretical schedule and the actual one produced by the system. This discrepancy can be reduced by using an alternative kernel design, that reduces the the system calls size and permits to serve interrupts and external events in tasks scheduled by the kernel. Some real-time systems such as Real-Time Mach [TNR90], DROPS [HBB$^+$98], and similar are based on a so called μkernel architecture, that achieve those goals, but at the cost of a decreased efficiency. Other alternative kernel architectures that can be used to improve real-time performance are represented by the so called *vertically structured kernels* (such as Nemesis [Re97]), multithreaded kernels, or real-time executives.

Alternatively, the monolithic structure can be modified introducing *kernel preemptability* to reduce the system calls' size, and to introduce a more precise accounting mechanism, high-resolution timers, and other mechanisms that permit to increase the scheduler's accuracy.

1.3 Organisation of the Dissertation

This dissertation introduces some contributions both in the field of scheduling theory and resource allocation (more related to real-time research) and in the OS field (implementation, kernel structure, and so on).

The first is concerned with the use of a proper scheduling algorithm and the implementation of a QoS aware resource allocation policy. With the latter implemented either at system level or at user level.

In Chapter 2 it will be shown that in order to enable QoS aware scheduling and resource allocation a precise description of the tasks' characteristics and requirements is needed, and the concepts of *task model* and *guarantee* will be introduced to solve this problem.

Chapter 3 will review some scheduling algorithms that can be used to properly serve a time-sensitive application (namely, real-time and proportional share schedulers). Moreover, it will be shown that reservation techniques are the correct choice for a workstation OS, and the scheduling algorithm used in this dissertation (namely the CBS) will be introduced as well as some extensions used to synchronise time sensitive applications. The proposed scheduling techniques will be then analysed providing a formal model of the CBS, and the concept of *QoS guarantee* will be introduced.

Chapter 4 will address the problem of managing the system resources in a proper way, based on the task models and guarantees presented in Chapter 2 and on the scheduling algorithms presented in Chapter 3. Some adaptive techniques will be introduced.

The issues related to the implementation will be addressed in Chapter 5, where the most important kernel structures will be reviewed, and their appropriateness to serve time sensitive applications will be evaluated. Some of the most important problems and solutions will be presented, and some implementations of the techniques introduced in this dissertation will be described, showing how a general-purpose kernels (such as Linux) can be modified to support time-sensitive applications.

Finally, Chapter 6 will conclude the dissertation.

Chapter 2

Fundamental Concepts

Any sufficiently advanced technology is indistinguishable from magic
Clarke's law

Any sufficiently advanced magic is indistinguishable from technology
Murphy's reformulation of Clarke's law

Any sufficiently advanced magic is indistinguishable from a rigged demonstration
Programmer's restatement of **Murphy's reformulation of** Clarke's law

The OS kernel is the manager of all the hardware and software resources that are available in the system, and its rule is to assign resources to applications in order to properly execute them. The activities composing a time sensitive applications are characterised by some temporal constraints, and the desired QoS can be achieved only if the kernel kernel allocates the resources so that those constraints are respected. Hence, the kernel should be aware of the tasks' characteristics and requirements.

In this chapter, some basic definitions and two abstractions used to describe applications' characteristics and requirements (task models and guarantees) will be introduced.

2.1 Definitions

In a multiprogrammed system, the kernel assigns resources to the applications so that different applications give the impression to execute simultaneously. In other words, system resources have to be multiplexed between all the applications that are running in the system; depending on the kind of resources, the OS kernel can perform *spatial multiplexing* or *time multiplexing.*

Spatial multiplexing is used when the same resource can be divided in different parts, each of which can be assigned to a different application. A typical example is the system memory: when several applications are running in the system, the memory can be divided in regions, and each application is assigned a different region. When the resource cannot be split in several parts, time multiplexing must be used, and the resource is allocated to each application at different times, using a *time sharing* technique.

To better understand these concepts, some definitions are needed:

Definition 1 *An* **algorithm** *is the logical procedure that is used to solve a problem, and it can be expressed using a special formalism called* **programming language**.

Definition 2 *A* **program** *is a particular coding (implementation) of an algorithm in a well defined programming language.*

A program can execute as a sequential flow of operations, or can be composed by more than one concurrent activities, that are called *threads* or *processes*. Informally speaking, we can define threads and processes as follows:

Definition 3 *A* **thread** *is a single flow of execution, characterised by a small set of private resources, such as the CPU context, a stack and few other variables. Hence, a thread has not a large set of private resources, but generally works on public resources that it can share with other threads. In order to execute, a thread must be associated to a set of resources such as for example a memory space.*

Definition 4 *A* **process** *is composed by one or more threads, plus all the resources that they need to execute (memory space, some descriptor tables, and so on). These resources are private to the process, and cannot be accessed by other processes (unless they are explicitly shared).*

In this dissertation, the difference between threads and processes is not particularly important, and the word **task** will be used to identify an executable entity, that can be a thread or a process.

2.2 Task Models

To understand what a task model is, let's consider, for example, a task reproducing a so called *Continuous Media* (CM) stream[1]: the player task

[1]A Continuous Media is a stream of frames that should be played in a timely fashion, such as a video or an audio stream.

should decode and reproduce the stream frames periodically at a stable rate. If this rate is not maintained, the experienced QoS decreases. As we will see in Chapter 3, the use of an appropriate scheduling algorithm can help to respect temporal constraints. However, the scheduling algorithm alone is not enough since, in order to properly serve the application, the scheduling parameters should be assigned adequately. Hence, the OS should provide some way to specify the tasks' requirements and parameters: this is the role of the *task model*. A task model is an abstraction that can be used to communicate to the OS kernel the tasks' requirements and parameters, and is necessary to decouple the scheduling algorithm from the application.

Unfortunately, the traditional task model (used by general purpose OSs such as Windows or all the unix flavours) is not very useful for time sensitive applications, since it characterise a task as a continuous stream of instructions, optionally assigning an additional parameter (such as a fixed priority or a "nice" value) to the task for describing the its importance. As a result, real-time tasks can be scheduled using fixed priorities (Rate or Deadline Monotonic), dynamic priorities (Earliest Deadline First), or using some form of Proportional Share, and the scheduling parameters are assigned giving the programmer a direct visibility of those low-level parameters such as priorities, WCETs, deadlines, weights, and so on. As can be easily seen, this approach tends to mix the task model and the scheduling parameters, exposing a direct visibility of the algorithm to the user.

2.2.1 The Real-Time Task Model

Returning to the previous example of a task reproducing a CM stream: to provide a controlled QoS, frames have to be decoded periodically. This can be done by splitting the player task into instances (using each instance to process a single frame), and by executing task instances at a constant rate that is compatible with the CM requirements. This result can be obtained using the *real-time task model*, and using temporal constraints called deadlines to do performance monitoring.

A real-time task τ_i is a stream of instances, or jobs, each of them performing an independent activity, such as decoding a frame, receiving a packet from the network, serving an interrupt, and so on.

Each job $J_{i,j}$ is characterised by an arrival time (or release time) $r_{i,j}$, an execution time $c_{i,j}$, and a deadline $d_{i,j}$; in general $d_{i,j} = r_{i,j} + D_i$, where D_i is the tasks's relative deadline. When a new job $J_{i,j}$ arrives (at time $r_{i,j}$) task τ_i is inserted into the scheduler ready queue and is ready to execute when the scheduler selects it. After executing for a time $c_{i,j}$, the job finishes at time $f_{i,j}$; in order the temporal constraints to be respected, each job $J_{i,j}$ should finish before its deadline $d_{i,j}$.

In general, to perform some kind of guarantee about the respect of job's deadline it is necessary to have some information about the execution and interarrival times. The simplest way to provide those information is to specify a Worst Case Execution time (WCET) $C_i = \max_j\{c_{i,j}\}$ and a minimum interarrival time $T_i = \min_j\{r_{i,j+1} - r_{i,j}\}$ for the task. In this case, a task τ_i can be characterised by the parameters (C_i, T_i, D_i). For example, if the arrivals are periodic and the relative deadline is equal to the period (that is to say, if $r_{i,j+1} = r_{i,j} + T_i$ and $D_i = T_i$), the task is said to be *periodic*, and is described by the tuple (C_i, T_i) (the periodic task model was introduced by Liu & Layland [LL73]).

A task characterised by periodic arrivals (fixed interarrival times) but unknown execution times is referred to as a *semiperiodic task* in this dissertation. The distinction between periodic tasks and semiperiodic tasks has been introduced to distinguish the case in which a WCET C_i is known (the Liu & Layland periodic model) from a more realistic case in which no assumption on the execution times can be done.

2.2.2 The GPS Model

Returning to the CM player example, it is worth noting that to be properly served, i.e. to respect the CM temporal constraints, the CM decoding task must be assigned a proper amount of the CPU and of the other needed resources. Hence, as an alternative to the real-time task model it is possible to allow time sensitive tasks to execute at a constant rate, which permits to respect their time constraints.

Executing each task τ_i at a constant rate is the essence of the Generalised Processor Sharing (GPS) [PG93, PG94] approach: in this model, each shared resource needed by tasks (such as the CPU) is considered as a fluid that can be partitioned among the applications. Each task will instantaneously receive a fraction $f_i(t)$ of the resource at time t, where $f_i(t)$ is defined as the *share*. Note that the GPS model can be seen as the limiting form of a Weighted Round Robin policy.

To compute the share of the resource that each task τ_i will receive, in the GPS model τ_i is assigned a weight w_i, and its share is computed as

$$f_i(t) = \frac{w_i}{\sum_{\tau_j \in \Gamma(t)} w_j}$$

where $\Gamma(t)$ is the set of tasks active at time t.

Since each task consists of one or more requests for shared resources such as the CPU, tasks can block and unblock, and the $\Gamma(t)$ set can vary with time.

Hence, the share $f_i(t)$ is a time varying quantity. The minimum guaranteed share is defined as the *rate*

$$F_i = \frac{w_i}{\sum_{\tau_j \in \Gamma} w_j}.$$

Note that a correct assignment of the tasks weights permits to guarantee real-time performance to all the time sensitive tasks in the system. In fact, based on the task share, it is possible to compute a response time for each task request. The problem with this task model is that the task response time and the task throughput are not independent.

2.3 High Level Task Models

The RT and PS task model presented in the previous section can be useful to model the tasks' requirements and characteristics, but in some cases they exports some too low-level parameters. Since a user is not generally interested in the scheduling algorithm and its details, and does not often knows all the tasks parameters, in many cases the RT or PS models are very different from what the users really needs and using such models forces the programmer to assign low-level parameters according to complex mapping functions. Moreover, a similar approach presents the following disadvantages:

- the system schedulability strongly relies on the exact knowledge of WCETs, which cannot always be easily estimated;
- in some cases tasks' parameters (e.g., the PS weights) have not an easy interpretation, so the user can only assign them using heuristic rules;
- tasks' parameters are too low-level to support complex features, such as bandwidth adaptation or advanced synchronisation.

The problems mentioned above can be addressed by introducing high-level task models which provide an interface closer to the real needs. For example, in a multimedia environment the following features can be identified for the application tasks:

- each task is characterised by an importance value with respect to all the other tasks: when the system resources are not enough to fulfil each task request, the resources will be shared according to tasks' importance;
- some tasks need to execute with a constant rate, without respecting any explicit time constraint;

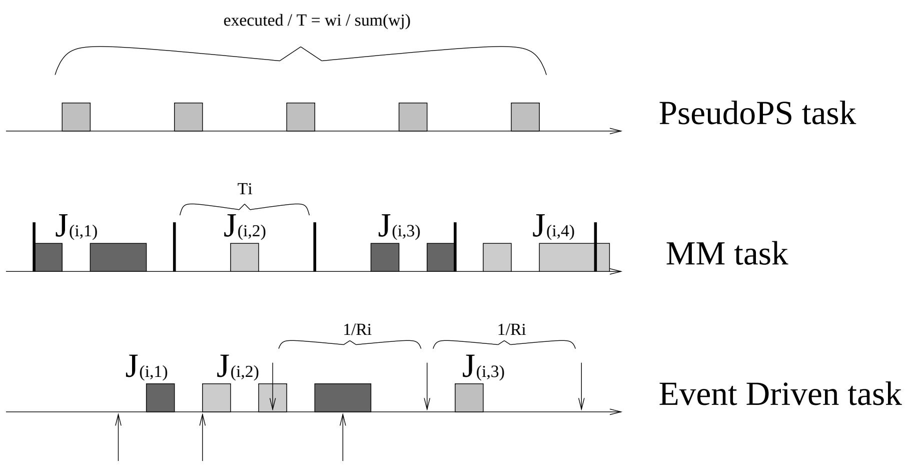

Figure 2.1: Example of the three classes of tasks.

- some tasks need to execute periodically: the task is composed of *jobs*, each of them have to be activated at a period boundary and must finish within the period end. This is a time constraint that can be expressed in terms of deadlines;
- some tasks need to respond to internal or external events, serving a minimum number of events per time unit.

To fulfil these requirements, a task τ_i can be characterised by a weight w_i, representing the task's *importance* with respect to the others. Moreover, tasks can be characterised by some *temporal constraints* (such as a period T_i). Based on these characteristics, three classes of tasks can be defined (see Figure 2.1):

- **PseudoPS (PPS) Tasks** $\tau_i = (w_i)$ are flows of instructions that execute uniformly, receiving a processor share proportional to the task weight w_i;
- **MultiMedia (MM) Tasks** $\tau_i = (w_i, T_i)$ are streams of jobs $J_{i,j}$ periodically activated with a period T_i, so that job $J_{i,j}$ arrives in the system at time $r_{i,j} = r_{i,j-1} + T_i$, and should finish before the next job starts (see the semiperiodic task model in the previous section). Using the real-time terminology, we say that $J_{i,j}$ has a soft deadline $d_{i,j} = r_{i,j+1} = r_{i,j} + T_i$. The goal of the system is to assign each task a fraction of the processor bandwidth sufficient to meet this requirement;
- **Event Driven (ED) Tasks** $\tau_i = (w_i, R_i)$ are streams of aperiodic jobs $J_{i,j}$ activated by external or internal events. The user specifies the av-

erage number R_i of jobs that should be executed per time unit, and the goal of the system is to automatically adjust the fraction of processor bandwidth assigned to each task in order to meet this requirement.

PseudoPS tasks are equivalent to GPS tasks: they execute at a uniform rate, but, since their execution time is not known, no temporal constraint can be guaranteed, although a suitable (system dependent) tuning of the tasks' weights may allow to serve conventional applications in a timely fashion, without modifying them.

MM tasks are designed to manage CM streams. Since they are composed of distinct jobs, the system can monitor each job's execution time to arrange the CPU bandwidth reserved to the task. Using this task model, the programmer has to specify the task period, but the task execution time does not need to be estimated.

ED tasks are similar to MM Tasks, in the sense that the programmer is not bound to specify the task execution time: the only mandatory task parameter is the number R_i of jobs that must execute in a time unit. The difference with MM Tasks is that Event Driven Tasks are not periodically activated by the system, but are activated by external events.

If the system is overloaded, and the CPU bandwidth is not sufficient to fulfil each task's requirement, an explicit bandwidth compression algorithm corrects the fraction of CPU bandwidth assigned to each task using the task weight w_i (the tasks with the higher weights will receive a bandwidth nearest to the requested one). This model permits to distinguish the task temporal constraint (the period T_i or the rate $R_i = 1/T_i$) from the task importance, expressed by the weight w_i. In fact, one of the biggest problems of classical real-time scheduling algorithms (such as Rate Monotonic or Earliest Deadline First) was that the task importance resulted to be proportional to the inverse of the task period.

2.4 Guarantees

As shown in Section 2.2, a time sensitive application should be served so that some temporal constraints are respected. Those constraints are expressed by the task model: for example, in the real-time task model each job $J_{i,j}$ is associated a deadline $d_{i,j}$. Hence, the goal of the OS kernel is to allocate resources in order to provide some *guarantees* about the temporal constraints: in the previous example, a simple guarantee can be that each job $J_{i,j}$ terminates before its deadline ($\forall i, j, f_{i,j} \leq d_{i,j}$).

Definition 5 *A* **guarantee** *is a contract between the system and a client* *(generally a task), regarding the amount of resources that the client will receive from the system, and the timing of this resource allocation.*

In other words, the guarantee abstraction concerns the task performance, and is used to decouple it from the scheduling algorithm an the implementation details (that is to say, a guarantee abstracts the behaviour provided by a scheduler from the scheduling algorithm itself). In Chapter 3 it will be shown that the most important issue in scheduling analysis is to prove that a scheduling algorithm provides a particular kind of guarantee. In this way, a programmer is allowed to reason in terms of *model of resource allocation*, instead of coping with the resource allocation algorithm itself.

The guarantee abstraction is particularly important in real-time systems, because it permits specify the QoS that a task will receive from the system. In this context, it is important to know if the system will be able to provide a specified guarantee, to determine if a task can be accepted in the system (without compromising the guarantee of the other tasks). This is done using an *admission test.*

Definition 6 *The* **admission test**, *or* **schedulability test** *is a condition that must be verified to provide a specified guarantee.*

Informally speaking, the admission test states that the amount of resources needed to respect a specified guarantee is less or equal than the amount of available resources. The admission test depends on the scheduling algorithm, and is used to pass from a task set and a scheduling algorithm to a guarantee that will be provided by the system. In Chapter 3 some examples of admission tests will be presented together with some real-time scheduling algorithms.

2.4.1 Hard Real-Time Guarantee

Real-time techniques were originally developed for implementing embedded control system for which the consequence of a deadline miss was considered to be catastrophic. For this reason, the first kind of guarantee that has been presented in literature is the *Hard Real-Time Guarantee*, requiring that all the deadlines in the system are respected.

More formally, a hard guarantee ensures that

$$\forall(i,j), \qquad f_{i,j} \leq d_{i,j}. \tag{2.1}$$

In order to analyse the feasibility of a hard guarantee, some additional definitions are needed:

Definition 7 *Given a real-time task τ_i, its* **demanded time** $D_i(t_1, t_2)$ *is defined as*

$$D_i(t_1, t_2) = \sum_{j: r_{i,j} \geq t_1 \wedge d_{i,j} \leq t_2} c_{i,j}$$

Definition 8 *In a similar way, the time demanded by a task set $\Gamma = \{\tau_1, \dots \tau_n\}$ can be defined as*

$$D(t_1, t_2) = \sum_i D_i$$

The concept of demanded time is fundamental to test if a task set Γ is schedulable or not, as stated by the following theorem:

Theorem 1 *A necessary condition for the task set $\Gamma = \{\tau_1, \dots \tau_n\}$ to be schedulable is that*

$$\forall t_1, t_2 : t_2 > t_1, D(t_1, t_2) \leq (t_2 - t_1)$$

2.4.2 QoS Guarantees

In recent years, it has been shown that a more relaxed guarantee can be useful too. In fact, respecting all the applications' deadline can often be overkilling, and aiming to that goal can lead to system underutilisation.

For this reason, the concept of *soft real-time tasks* has been proposed: a soft real-time task is a task that *should* respect its deadlines, but that can tolerate a "reasonable amount" of missed deadlines. It is easy to see that this definition is too vague, and the "reasonable amount" should be quantified in order to use this concept in a systematic way. In facts, the problem with soft real-time is that it is often difficult to give a formal definition of a QoS guarantee (soft guarantee). For this reason, the terms "QoS" and "soft real-time" or "soft guarantee" are often used informally, and their meaning is not well understood (for example, in all the real-time theory there is a big confusion between soft real-time tasks and aperiodic tasks).

A possible way to define the concept of QoS guarantee in a more formal way is to use probabilistic deadlines. Using this model,

$$\tau_i = (U_i(c), V_i(t))$$

where $U_i(c)$ is the probability that job $J_{i,j}$ has execution time c, and $V_i(t)$ is the probability that jobs' interarrival time is t. Hence,

$$\begin{aligned} U_i(c) &= P\{c_{i,j} = c\} \\ V_i(t) &= P\{r_{i,j+1} - r_{i,j} = t\}. \end{aligned}$$

In Chapter 3 it will be shown that a proper assignment of the scheduling parameters permits to respect all the task's deadlines. This corresponds to the Liu and Layland priority assignment and to the hard real-time guarantee.

On the contrary, a probabilistic guarantee permits to assign the scheduling parameters (Q_i^s, T_i^s) to τ_i in a less conservative way, still maintaining some control on the QoS experienced by τ_i. In this case, the concept of *probabilistic deadline* can be used to quantify the QoS experienced by each task. A probabilistic deadline δ is not required to be always respected, but can be respected by task τ_i with a probability

$$X_i(\delta) = P\{f_{i,j} \leq r_{i,j} + \delta\} < 1.$$

Performing a QoS guarantee with a probabilistic deadline δ means to guarantee that:

- if task τ_i is described by the PDFs $(U_i(c), V_i(t))$
- if the assigned scheduling parameters are (Q_i^s, T_i^s)
- then, each job $J_{i,j}$ of task τ_i has probability $X_i(\delta)$ of finishing within a relative deadline δ.

2.4.3 GPS Guarantee

As explained in Section 2.2.2, the GPS model describes a task system as a fluid flow system, in which each task τ_i is modelled as an infinitely divisible fluid, and executes at a minimum rate F_i that is proportional to a user specified weight w_i.

Hence, task τ_i is guaranteed to execute for a time $s_i(t_1, t_2) > (t_2 - t_1)F_i$ in each backlogged interval $[t_1, t_2]$. The exact definition of the GPS executed time s_i is $s_i = \int_{t_1}^{t_2} f_i(t)dt$. Hence, in the ideal fluid flow case, the tasks' execution can be described through the **GPS guarantee**:

$$\forall \tau_i \text{ active in } [t_1, t_2], \frac{exec_i(t_1, t_2)}{exec_j(t_1, t_2)} \geq \frac{w_i}{w_j} \; j = 1, 2, ..., n \tag{2.2}$$

where $exec_i(t_1, t_2)$ is the execution time of τ_i in the interval $[t_1, t_2]$.

It can be easily seen that Equation 2.2 is equivalent to $exec_i(t_1, t_2) = s_i(t_1, t_2)$.

In a real system, resources are allocated in discrete time quanta of size Q. This quantum based allocation causes an allocation error: given two active tasks τ_1 and τ_2, the allocation error in the time interval $[t_1, t_2]$ can be expressed as

$$\frac{exec_i(t_1, t_2)}{w_i} - \frac{exec_j(t_1, t_2)}{w_j}.$$

An alternative way to express this allocation error is the *maximum lag* $Lag_i = \max_{t_1,t_2}\{|exec_i(t_1, t_2) - s_i(t_1, t_2)|\}$. Hence, a more realistic version of the GPS guarantee is the following:

$$exec_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(t)dt + -Lag_i$$

2.4.4 Reservation Guarantees

An important concept that emerged in the last years is the *temporal isolation*, ensuring that the temporal behaviour of a task is not influenced by the temporal behaviour of other tasks in the system.

In other words, if a task requires "too much" resources, it must be slowed down in order not to jeopardize the other tasks' guarantee. A similar property is very important, since it permits to provide different guarantees to different tasks: for example, it is possible to perform an hard guarantee on a task, while other tasks are provided a probabilistic guarantee, or no guarantee at all.

Looking at the previous section, it is possible to see that a PS guarantee provides some form of temporal protection: if task τ_i is guaranteed to receive $f_i(t_2 - t_1)$ time units in the time interval (t_1, t_2), it means that it is possible to guarantee τ_i's performance *independently from all the other tasks.* Of course, tasks' weights w_i need to be properly arranged, and an admission test is needed, as shown in [SAWJ97]. However, imposing $\frac{exec(t_1,t_2)}{t_2-t_1}$ to be constant for all the (t_1, t_2) intervals can be a too stringent requirement (in facts, Section 2.4.3 shows that a real scheduling algorithm can only approximate a PS guarantee).

A better solution would be to guarantee that the ratio $\frac{exec(t_1,t_2)}{t_2-t_1}$ is constant *over well specified intervals*, for example between deadlines in a real-time task. This is the essence of the reservation guarantee. More formally, a reservation (Q, T, D) guarantees that an amount Q of a resource will be available to the reserved task every period T, within a deadline D from the beginning of the period. Hence,

$$\forall j_2 \geq j_1 exec(j_1 T, j_2 T + D_i) \geq (j_2 - j_1 + 1)Q$$

If $T = D$, the reservation simplifies to a (Q, T) model, and the guarantee becomes

$$\forall j_2 > j_1 \frac{exec(j_1 T, j_2 T + D_i)}{(j_2 - j_1)T} \geq \frac{Q}{T} \tag{2.3}$$

Some authors tend to distinguish *hard reservation guarantees* from *soft real-time guarantees*: following this definition, a soft reservation guarantee is based on the previous formula, whereas a hard reservation guarantees that

$$\forall j_2 > j_1 \frac{exec(j_1 T, j_2 T + D_i)}{(j_2 - j_1)T} = \frac{Q}{T} \tag{2.4}$$

Since a reservation guarantee ensures that Q time units will be served within a relative deadline D at each period T, it is possible to restate its requirements as a hard real-time guarantee, requiring that a periodic task τ with $c_{i,j} \leq Q$, period T and relative deadline D respects all its deadlines. Hence, an admission test similar to the one of Theorem 1 is required.

Chapter 3

Scheduling

My VCR is a real-time system;
it fails all the time.
I'm still alive

Rich Gerber

As said in Chapter 2, in a multiprogrammed system the kernel is responsible for multiplexing the system resources between concurrent applications. More formally, the kernel has to *schedule* resources, deciding which resource is assigned to which application. In this chapter, it will be shown that in order to properly serve time sensitive applications (that is to say, in order to respect temporal constraints of a given task model and fulfil a specified guarantee) the scheduling algorithm must be carefully chosen, and some of the most important scheduling algorithms will be presented.

3.1 Task Scheduling

To execute, each task τ needs some resources to be assigned to it (in general, it will need at least the CPU and some amount of memory); when time multiplexing is used, **a resource R is assigned to a single task τ_i at time** t, hence it is possible to describe the resource allocation using a function $\sigma_R : \mathcal{R}^+ \to \Gamma$, where $\Gamma = \{\tau_1, \ldots \tau_n\}$ is the set of all the tasks in the system. More formally,

Definition 9 *A* **schedule** $\sigma_R(t)$ *is an assignment of a resource R to a set of tasks $\Gamma = \{\tau_1, \ldots \tau_n\}$. Hence, $\sigma_R(t)$ is a function from the time domain $\mathcal{R}^+$ to the* **task set** Γ. *Note that it is possible that at time t resource R is not assigned to any task; in this case the resource is said to be* **idle**. *To cope with this situation, the schedule can be defined as $\sigma_R : \mathcal{R}^+ \to \Gamma \cup \{\phi\}$, where $\sigma_R(t) = \phi$ means that R is idle at time t.*

Definition 10 *A* **scheduling algorithm** *is an algorithm that is used to decide to which task τ_i resource R will be assigned at time t.*

Most of the scheduling algorithms are priority based: all the active tasks (that is to say, all the tasks that are competing for a resource) are listed in a *ready task queue* Γ_{ready}, and a *scheduling priority* $P(\tau_i)$ is assigned to each task τ_i. At each time, the task having the highest priority is selected (is scheduled), and the resource is allocated to it:

$$\sigma_R(t) = \tau_i : P(\tau_i) = \max_{\tau_j \in \Gamma_{ready}} \{P(\tau_j)\}$$

If the scheduler does not change the scheduling priorities (but they are assigned at task creation and can only be changed by using an explicit system call), the scheduler is said to be based on *static priorities.* Otherwise (if the scheduling priorities can be changed by the scheduler during the task execution), the scheduler is referred as a dynamic priority based one. For example, the classical Unix scheduler is based on dynamic priorities, since a task's priority decrease during task execution to avoid starvation.

3.2 Classical Real-Time Scheduling

As previously said, the scheduling algorithm is used for deciding to which task to allocate a system resource. When dealing with time sensitive applications, the goal of a scheduling algorithm is to allocate resources to a task set Γ so that some kind of guarantee is respected. Of course, the scheduler can provide a guarantee (for example, the hard real-time guarantee - all the deadlines are respected) only if the task models describing Γ are known. From this point of view, a scheduling algorithm transforms a task model (or a set of task models) into a guarantee. This can be done by using a *schedulability test* to check if given set of tasks is compatible with a specified guarantee.

To simplify the discussion, let's assume that each task in the system only needs the CPU to execute. Hence, the only scheduler present in the OS kernel is the CPU scheduler, and it is responsible to schedule tasks so that their time constraints are respected. The simplest way to do this is to consider the periodic real-time task model, and the hard real-time guarantee: in this case, each task τ_i is described by two parameters (C_i, T_i), and the goal of the scheduler is to meet all the deadlines $d_{i,j} = jT_i$. Moreover, since all tasks are periodic, Theorem 1 can be simplified in the following lemma:

Lemma 1 *If $\Gamma = \{\tau_1, \dots \tau_n\}$ is a set of periodic tasks $\tau_i = (C_i, T_i)$, then a* *necessary condition for its hard schedulability is that*

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

That is to say, if the *system utilization* $\sum_{i=1}^{n} \frac{C_i}{T_i}$ is greater than 1, then it is impossible to respect all the deadlines.

The first attempt to schedule such a task system can be to use a priority based scheduler: for example, an intuitive choice can be to use fixed priorities and to assign higher priorities to tasks characterised by shortest deadlines (smallest periods):

$$P(\tau_i) = \frac{1}{T_i}$$

this is the Rate Monotonic (RM) priority assignment, that has been analysed by Liu & Layland in [LL73]. As a confirmation of the goodness of the previous intuition, RM turns out to be an *optimal static priorities assignment.* That is to say, **if a periodic task set is schedulable using fixed priorities, then RM will schedule it properly**.

As explained above, in order to provide a guarantee a schedulability test is needed. The simplest kind of schedulability test is the utilization based one, that is expressed by the following theorem:

Theorem 2 *If* $\Gamma = \{\tau_1, \dots \tau_n\}$ *is a set of periodic tasks* $\tau_i = (C_i, T_i)$*, then RM will schedule it respecting all the deadlines if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{lub}$$

where U_{lub} *is the utilization* **least upper bound** *and is defined as* $U_{lub} = n(2^{\frac{1}{n}} - 1)$*.*

Unfortunately, the condition expressed by Theorem 2 only is a sufficient condition, and it is not a necessary one. That is to say, if $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{lub}$ then the task set will be surely schedulable with RM, but if the system utilization is greater than U_{lub} nothing can be said. Hence, restating the previous sentence, if the RM priority assignment is used and if the admission test $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{lub}$ is passed, then each task in the system will respect all its deadlines. If the admission test fails, then some deadlines *can* be missed; since RM is based on static priorities, it is possible to forecast that the tasks missing deadlines will be the lowest priority ones.

Theorem 3 *If a scheduling algorithm based on static priorities is used to schedule the periodic task set* $\Gamma = \{\tau_1, \dots \tau_n\}$ *and task* τ_i *does not miss any deadline, then each task* $\tau_j : P(\tau_j) > P(\tau_i)$ *will not miss any deadline.*

Unfortunately, the utilization least upper bound for RM is quite low (0.69 in the worst case); this problem can be addressed by using a different guarantee test based on the tasks' finishing times, as explained in [ABRT93]. Using this exact analysis, it is possible to perform a less pessimistic admission test, but there are still some task sets that are schedulable in theory (since $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$) and are not schedulable by RM. Since RM is optimal between all the fixed priority assignments, those task sets can be scheduled only using dynamic priorities. In this case, the priority of a task does not only depend on the task but it also depends on a second parameter, that can be the time t, the job number j, or a generic index i (hence, it will be expressed as $P(\tau_i, x)$). The most intuitive dynamic priority assignment is Earliest Deadline First (EDF), based on assigning priorities to the jobs, and on assigning higher priorities to jobs with the shortest *absolute* deadline:

$$P(\tau_i, j) = P(J_{i,j}) = \frac{1}{d_{i,j}}$$

EDF is an *optimal scheduling algorithm*, meaning that if a task set Γ is schedulable (that is, if an algorithm capable of scheduling Γ in order to respect every deadline exists), then EDF can schedule it respecting all the deadlines.

This concept is expressed by the following theorem:

Theorem 4 *A task set Γ is schedulable by EDF if and only if*

$$\forall t_1, t_2 : t_2 > t_1, D(t_1, t_2) \leq (t_2 - t_1)$$

Comparing Theorem 4 with Theorem 1, it is easy to see the optimality of EDF.

3.3 Proportional Share Scheduling

A Proportional Share (PS) scheduling algorithm emulates the GPS allocation model in a real system, where multiple tasks cannot execute simultaneously on the same CPU.

Hence, the ideal fluid-flow allocation is approximated using a quantum-based allocation. That is to say, in a Proportional Share scheduler the resource is allocated in discrete time quanta having maximum size Q: a process acquires a resource at the beginning of a time quantum and releases the resource at the end of the quantum (a new request is posted) or before (the process have to be expressly re-activated); this is done dividing each task τ_i in requests q_i^k of dimension Q.

As already noted in Section 2.4.3, quantum based allocation introduces an allocation error respect to the fluid flow model. The minimum theoretical error bound is $H_{i,j} = \frac{1}{2}(\frac{Q_i}{w_i} + \frac{Q_j}{w_j})$, where Q_i is the maximum dimension for τ_i requests and Q_j is the maximum dimension for τ_j requests. This allocation error influence the performance of time sensitive tasks in a way that is described by the lag. In order to understand this, consider that in the ideal GPS system task τ_i executes for a time $\int_{t_1}^{t_2} f_i(t)dt$ in the interval $[t_1, t_2]$; in a real system this is impossible (because tasks are not fluid), so the allocation error experimented by a task can be measured by the lag[1]:

$$lag_i(t_1) = \int_{t_0}^{t_1} f_i(t)dt - exec_i(t_0, t_1),$$

where t_0 is the activation time of the task.

In the following of this section, some of the most important PS scheduling algorithm are analysed, showing how they emulates the ideal GPS allocation, and evaluating their performance in terms of allocation error and lag.

3.3.1 Weighted Fair Queuing

The first known Proportional Share scheduling algorithm is Weighted Fair Queuing (WFQ), that emulate the behaviour of a GPS system using the concept of *virtual time*. The virtual time $v(t)$ is defined by increments as follows:

$$\begin{cases} v(0) &= 0 \\ dv(t) &= \frac{1}{\sum_{\tau_i \in \Gamma(t)} w_i} dt \end{cases}.$$

Each quantum request q_i^k is assigned a virtual start time $S(q_i^k)$ and a virtual finish time $F(q_i^k)$ defined as follows:

$$\begin{aligned} S(q_i^k) &= \max\{v(r_{i,k}), F(q_i^{k-1})\} \\ F(q_i^k) &= S(q_i^k) + \frac{Q_{i,k}}{w_i} \end{aligned}$$

where $r_{i,k}$ is the time at which request q_i^k is generated and $Q_{i,k}$ is the request dimension (required execution time); since $Q_{i,k}$ is not known a priori (a task may release the CPU before the end of the time quantum), it is assumed equal to the maximum value Q_i.

[1]remember that the maximum lag has already be cited a measure of the allocation error

Tasks' requests are scheduled in order of increasing virtual finish time: in the virtual time domain, each request will finish before the virtual finish time.

WFQ provide fairness (bounding the allocation error) in static systems, where all the tasks are always active, but presents some problems:

- it needs the frequent recalculation of $v(t)$;
- it does not perform well in dynamic systems (when a task activates or blocks, the fairness of the schedule is compromised);
- it assumes each requests size equal the maximum value (scheduling quantum): in a real situation this assumption is not correct;

3.3.2 Start Fair Queuing

In [GGV96], a proportional share scheduler is used to subdivide the CPU bandwidth between various application classes: the proposed algorithm, Start Fair Queuing (SFQ), is similar to WFQ but defines the virtual time in a different manner and schedules the requests in order of increasing virtual start time. The virtual time $v(t)$ is defined as follows:

$$v(t) = \begin{cases} 0 & \text{if } t = 0 \\ 0 \text{ or any value} & \text{if the CPU is idle} \\ S(q_i^k) & \text{if request } q_i^k \text{ is executing} \end{cases}$$

SFQ guarantees an allocation error bound of $2H_{i,j}$, so it is near-optimal. Moreover, SFQ calculates $v(t)$ in a simpler way (introducing less overhead) and does not need the virtual finish time of a request to schedule it, so it does not require any a priori knowledge of the request execution time ($F(q_i^k)$ can be calculated at the end of q_i^k execution).

A Proportional Share algorithm schedules the tasks in order to reduce the allocation error experimented by each of them; to provide some form of real-time execution it is important to guarantee that $lag_i(t)$ is bounded.

SFQ and WFQ provides an optimal upper bound for the lag: $\max_t\{lag_i(t)\} = Q_i$, but do not provide an optimal bound for the absolute value of the lag: for example, for SFQ this bound $\max_t\{|lag_i(t)|\} = Q_i + f_i \sum Q_j$ that depends on the number of active tasks.

3.3.3 Earliest Eligible Virtual Deadline First

In [SAWJ+96] the authors propose a scheduling algorithm, called Earliest Eligible Deadline First (EEVDF), that provide a bound on the lag experimented by each task.

EEVDF defines the virtual time as WFQ and schedules the requests by virtual finish times (in this case called virtual deadlines), but use the virtual start time (called virtual eligible time) to decide if a task can be scheduled (is eligible): if the virtual eligible time is grater than the actual virtual time, the request is not eligible. Virtual eligible and finish time are defined as follows:

$$\begin{aligned} S(q_i^k) &= \max\{v(r_{i,k}), E(q_i^{k-1} + \frac{Q_{i,k-1}}{w_i}\} \\ F(q_i^k) &= S(q_i^k) + \frac{Q_{i,k}}{w_i}. \end{aligned}$$

When a task joins or leaves the competition (activates or blocks), $v(t)$ is adjusted in order to maintain the fairness in dynamic system.

The minimum theoretical bound for the lag absolute value is Q, that is guaranteed by the EEVDF algorithm; for this reason, EEVDF is said to be optimal. EEVDF can also schedule dynamic task sets and can use fractional and non uniform quantum size, so it can be used in a real operating system. To the best knowledge of the authors, EEVDF is the only algorithm that provides a fixed lag bound.

If the lag is bounded, real-time execution can be obtained maintaining constant the share of each real-time task:

$$f_i(t) = \frac{C_i + \max_t\{lag_i(t)\}}{D_i}.$$

3.4 Reservation Based Scheduling

Based on classical real-time scheduling (EDF or RM priority assignment), it is possible to implement a reservation guarantee by simply enabling a task τ to execute as a real-time task (scheduled, for example, by EDF or RM) for a time Q, and then blocking it (or scheduling it in background as a non real-time task) until the next period. In this way, a task is *reshaped* so that it behaves like a periodic real-time task with parameters (Q, T) and can be properly scheduled by a classical real-time scheduler. A similar technique is used in computer networks by the traffic shapers, such as the leaky bucket or the token bucket. More formally,

- a reservation scheduler is characterised by two parameters (Q, T)
- a *budget*, or *capacity* is associated to each reservation
- at the beginning of each reservation period, the budget is recharged to Q

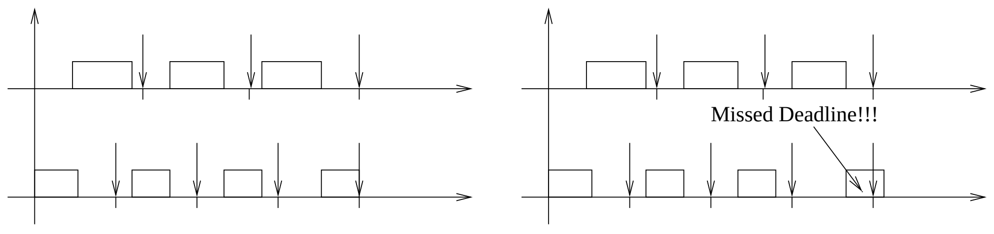

Figure 3.1: Resource Reservations with aperiodic arrivals.

- when the reserved task executes, the budget is decreased accordingly
- when the budget arrives to 0, the reservation is said to be *depleted*, and an appropriate action should be taken.

As previously said, when a reservation is depleted the reserved task can be blocked, or it can be "downgraded" to be a non real-time task. By blocking the task, it is possible to implement a hard reservation, whereas if the task is downgraded to non real-time a soft reservation behaviour can be implemented.

Note that the reservation parameters (Q, T) are different from the task parameters (C, T), and this separation can be useful to control the tasks' QoS (as it will be shown in the next sections). In order to avoid confusion, the reservation's parameters will be indicated with (Q^s, T^s), whereas the task parameters will be indicated with (C, T) as usual.

3.4.1 Reservation Systems on Dynamic Priority Systems

A generic reservation based scheduling algorithm can in general have some problems in handling aperiodic task's arrivals. For example, let's consider two tasks $\tau_1 = (2, 4)$ and $\tau_2 = (1.5, 3)$ served by two reservations $RSV_1 = (2, 4)$, and $RSV_2 = (1.5, 3)$. As shown in Figure 3.1, if the EDF priority assignment is used to implement the reservation scheme, then the task set is schedulable (and each task will respect all its deadlines). However, if an instance of one of the two tasks is activated later (the third instance of τ_1 in the example), a task may miss a deadline. Other similar problems can be highlighted when a lot of reservation are created and immediately destroyed consecutively in a short time.

When implementing reservations over a dynamic priority scheme (such as the EDF priority assignment), it is possible to take advantage of dynamic priorities in order to fix all these problems, and to better exploit the CPU time. This can be done by properly assigning a dynamic *scheduling deadline* to each task and by scheduling tasks by EDF based on their scheduling deadlines.

Definition 11 *A* **scheduling deadline** $d^s_{i,j}$ *is a dynamic deadline assigned to a job* $J_{i,j}$ *in order to schedule it by EDF.*

Note that a scheduling deadline is something completely different from the job deadline $d_{i,j}$, that in this case is only used for performance monitoring.

The abstract entity that is responsible for assigning a correct scheduling deadline to each job is called **aperiodic server**.

Definition 12 *A* **server** *is a mechanism used to assign scheduling deadlines to jobs in order to schedule them so that some properties (such as the reservation guarantee) are respected.*

Hence, the server assigns to each job $J_{i,j}$ an absolute time-varying deadline $d^s_{i,j}$ which can be dynamically changed. This fact can be modelled by splitting each job $J_{i,j}$ in *chunks* $H_{i,j,k}$, each of whose is assigned a fixed scheduling deadline $d^s_{i,j,k}$.

Definition 13 *A* **chunk** $H_{i,j,k}$ *is a part of the job* $J_{i,j}$ *characterised by a fixed scheduling deadline* $d^s_{i,j,k}$. *Each chunk* $H_{i,j,k}$ *is characterised by an arrival time* $a_{i,j,k}$, *an execution time* $e_{i,j,k}$ *and by its scheduling deadline. Note that the arrival time* $a_{i,j,0}$ *of the first chunk of a job* $J_{i,j}$ *is equal to the job release time:* $a_{i,j,0} = r_{i,j}$.

In order to be useful to implement a resource reservation strategy, an aperiodic server must assign scheduling deadlines to tasks so that the utilization of the served task is less than a *server utilization* U^s. This concept can be better understood by extending the demanded time definition given in Section 2.4.1 [2].

Definition 14 *Given a server* S_i, *its* **demanded time** $D^s_i(t_1, t_2)$ *is defined as*

$$D^s_i(t_1, t_2) = \sum_{j : r_{i,j} \geq t_1 \wedge d^s_{i,j} \leq t_2} e_{i,j}$$

Where $d^s_{i,j}$ *is the* j^th *deadline generated by server* S_i, *and* $e_{i,j}$ *is the amount of time that the served task will execute with deadline* $d^s_{i,j}$.

[2] note that the demanded time was defined in the context of the real-time guarantee, and we are extending the definition to the reservation guarantee

Based on these definitions, a server must generate scheduling deadlines so that

$$D_i^s(t_1, t_2) \leq (t_2 - t_1)B_i^s$$

in this way, a set of servers is schedulable (that is to say, each scheduling deadline is respected) if $\sum_{i=1}^{n} B_i^s \leq 1$.

3.4.2 The Constant Bandwidth Server

The service mechanism proposed in this dissertation is the Constant Bandwidth Server (CBS), a work conserving server (implementing soft reservations) that has been inspired by the Total Bandwidth Server and by the Dynamic Sporadic Server (for a better comparison between these service mechanisms, see [Abe98, AB98].

The CBS algorithm is formally defined as follows:

- A CBS S is characterised by a budget c^s and by a ordered pair (Q^s, T^s), where Q^s is the *server maximum budget* and T^s is the *server period.* The ratio $B^s = Q^s/T^s$ is denoted as the *server bandwidth.* At each instant, a fixed deadline d_k^s is associated with the server. At the beginning $d_0^s = 0$.

- Each served job $J_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline d_k^s.

- Whenever a served job $J_{i,j}$ executes, the budget c^s of the server S serving τ_i is decreased by the same amount.

- When $c^s = 0$, the server budget is recharged to the maximum value Q^s and a new server deadline is generated as $d_{k+1}^s = d_k^s + T^s$. Notice that there are no finite intervals of time in which the budget is equal to zero.

- A CBS is said to be active at time t if there are pending jobs (remember the budget c^s is always greater than 0); that is, if there exists a served job $J_{i,j}$ such that $r_{i,j} \leq t < f_{i,j}$. A CBS is said to be idle at time t if it is not active.

- When a job $J_{i,j}$ arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) non-preemptive discipline (e.g., FIFO).

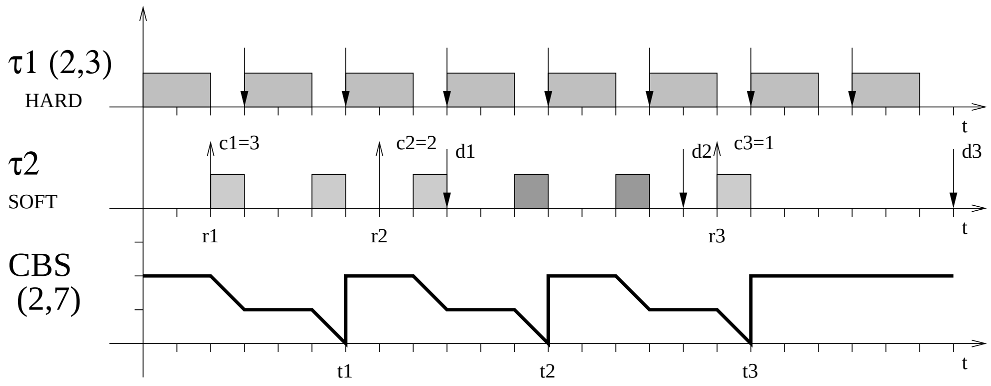

Figure 3.2: Simple example of CBS scheduling.

- When a job $J_{i,j}$ arrives and the server is idle, if $c^s \geq (d^s_k - r_{i,j})B^s$ the server generates a new deadline $d^s_{k+1} = r_{i,j} + T^s$ and c^s is recharged to the maximum value Q^s, otherwise the job is served with the last server deadline d^s_k using the current budget.

- When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.

- At any instant, a job is assigned the last deadline generated by the server.

Figure 3.2 illustrates an example in which a hard periodic task (directly scheduled by EDF) τ_1 is scheduled together with a soft task τ_2, served by a CBS having a budget $Q^s = 2$ and a period $T^s = 7$. The first job of τ_2 arrives at time $r_1 = 2$, when the server is idle. Being $c^s \geq (d^s_0 - r_1)B^s$, the deadline assigned to the job is $d^s_1 = r_1 + T^s = 9$ and c^s is recharged at $Q^s = 2$. At time $t_1 = 6$ the budget is exhausted, so a new deadline $d^s_2 = d^s_1 + T^s = 16$ is generated and c^s is replenished. At time r_2 the second job arrives when the server is active, so the request is enqueued. When the first job finishes the second job is served with the actual server deadline ($d^s_2 = 16$). At time $t_2 = 16$ the server budget is exhausted so a new server deadline $d^s_3 = d^s_2 + T^s = 23$ is generated and c^s is replenished to Q^s. The third job arrives at time 17, when the server is idle and $c^s = 1 < (d^s_3 - r_3)B^s = (23 - 17)\frac{2}{7} = 1.71$, so it is scheduled with the actual server deadline d^s_3 without changing the budget.

In Figure 3.3, a hard periodic task τ_1 is scheduled together with a soft task τ_2, having fixed inter-arrival time ($T_2 = 7$) and variable computation time, with a mean value equal to $C_2 = 2$. This situation is typical in applications that manage continuous media: for example, a video stream requires

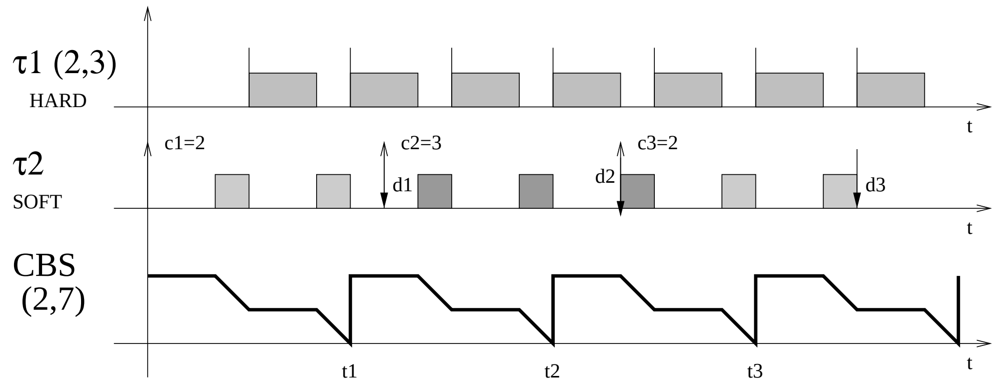

Figure 3.3: Example of CBS serving a task with variable execution time and constant inter-arrival time.

to be played periodically, but the decoding/playing time of each frame is not constant. In order to optimise the processor utilization, τ_2 is served by a CBS with a maximum budget equal to the mean computation time of the task ($Q^s = 2$) and a period equal to the task period ($T^s = T_2 = 7$).

As we can see from Figure 3.3, the second job of task τ_2 is first assigned a deadline $d_2^s = r_2 + T^s$. At time t_2, however, since c^s is exhausted and the job is not finished, the job is scheduled with a new deadline $d_3^s = d_2^s + T^s$. As a result of a longer execution, only the soft task is delayed, while the hard task meets all its deadlines. Moreover, the exceeding portion of the late job is not executed in background, but is scheduled with a suitable dynamic priority.

In other situations, frequently encountered in CM applications, tasks have fixed computation times but variable inter-arrival times. For example, this is the case of a task activated by external events, such a driver process activated by interrupts coming from a communication network. In this case, the CBS behaves exactly like a TBS with a bandwidth $B^s = Q^s/T^s$. In fact, if $C_i = Q^s$ each job finishes exactly when the budget arrives to 0, so the server deadline is increased of T^s. It is also interesting to observe that, in this situation, the CBS is also equivalent to a Rate-Based Execution (RBE) model [JB95] with parameters $x = 1, y = T_i, D = T_i$. An example of such a scenario is depicted in Figure 3.4.

Finally, Figure 3.5 shows how the tasks presented in Figure 3.1 are scheduled by a CBS when an instance arrives late. Since the CBS assigns a correct deadline to the instance arriving late (the third instance of τ_1), τ_2 does not miss any deadline, and temporal protection is preserved.

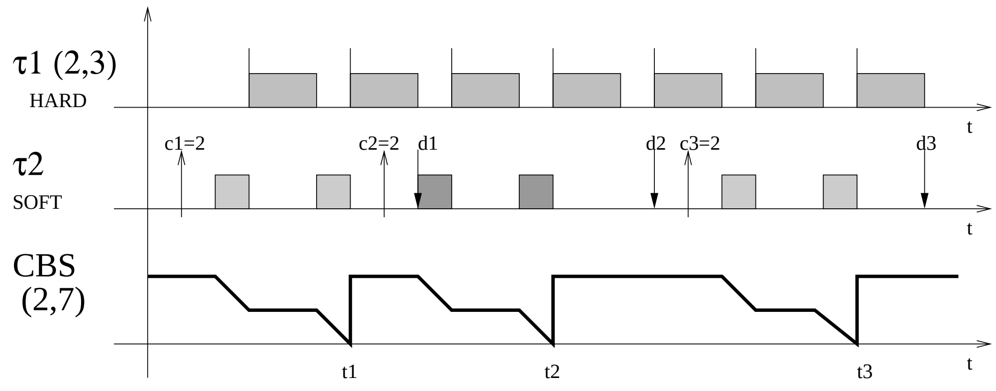

Figure 3.4: Example of CBS serving a task with constant execution time and variable inter-arrival time.

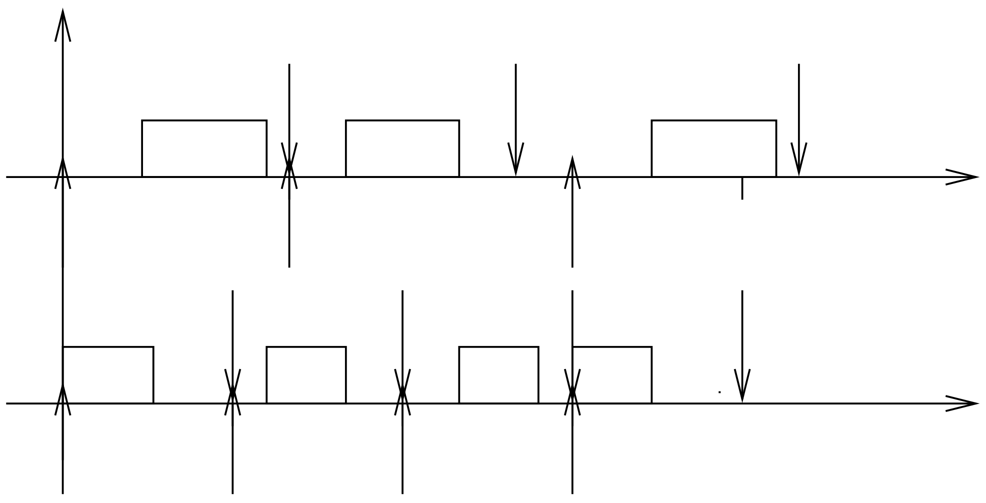

Figure 3.5: CBS with aperiodic arrivals.

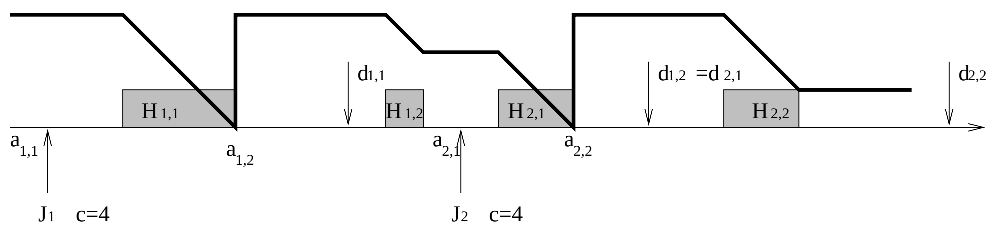

Figure 3.6: Serving some jobs divided in chunks.

3.4.3 CBS Properties

The proposed CBS service mechanism presents some interesting properties that make it suitable for supporting CM applications. The most important one, the *the isolation property* is formally expressed by the following theorem.

Theorem 5 *A CBS with parameters* (Q^s, T^s) *demands a bandwidth* $U^s = \frac{Q^s}{T^s}$

Proof.
In order to prove that a CBS with parameters (Q^s, T^s) cannot demand a bandwidth greater than $B^s = Q^s/T^s$, it is sufficient to prove that

$$\forall t_1, t_2 \in N : t_2 > t_1, \quad D^s(t_1, t_2) \leq B^s(t_2 - t_1).$$

Remember that each job J_j can be thought as consisting of a number of chunks $H_{j,k}$, each characterised by a release time $a_{j,k}$ and a fixed deadline $d^s_{j,k}$. An example of chunks produced by a CBS is shown in Figure 3.6. To simplify the notation, all the chunks generated by a server will be referred with an increasing index k (in the example of Figure 3.6, $H_{1,1} = H_1$, $H_{1,2} = H_2$, $H_{2,1} = H_3$, and so on).

The release time and the deadline of the k^{th} chunk generated by the server will be denoted by a_k and d_k, c will indicate the actual budget and n the number of requests in server queue. These variables are initialised in the following manner:

$$\begin{aligned} d^s_0 &= 0 \\ c^s &= 0 \\ n &= 0 \\ k &= 0 \end{aligned}$$

Using these notations, the server behaviour can be expressed as in Figure 3.7.

```
When job $J_j$ arrives at time $r_j$
      enqueue the request in the server pending request queue;
      n = n + 1;
      if (n == 1) /* (the server is idle) */
        if ($r_j$ + ($c^s$ / $Q^s$) * $T^s$ >= $d^s_k$)
          /*---------------Rule 1---------------*/
          k = k + 1;
          $a_k$ = $r_j$;
          $d^s_k$ = $a_k$ + $T^s$;
          $c^s$ = $Q^s$;
        else
          /*---------------Rule 2---------------*/
          k = k + 1;
          $a_k$ = $r_j$;
          $d^s_k$ = $d^s_{k-1}$;
          /* $c^s$ remains unchanged */
When job $J_j$ terminates
      dequeue $J_j$ from the server queues;
      n = n - 1;
      if (n != 0) begin to serve the next job in queue with deadline $d^s_k$;
When job $J_j$ served by $S_s$ executes for a time unit
      $c^s$ = $c^s$ - 1;
When ($c^s$ == 0)
      /*---------------Rule 3---------------*/
      k = k + 1;
      $a_k$ = actual_time();
      $d^s_k$ = $d^s_{k-1}$ + $T^s$;
      $c^s$ = $Q^s$;
```

Figure 3.7: The CB algorithm.

Note that the execution time of chunk H_k, e_k, is the server time demanded in the interval $[a_k, d^s_k]$: $e_k = D^s(a_k, d^s_k)$. Hence,

$$\forall t_1, t_2, \quad \exists k_1, k_2 : \quad D^s(t_1, t_2) \ = \sum_{k: a_k \geq t_1 \wedge d^s_k \leq t_2} e_k \ = \sum_{k=k_1}^{k2} e_k.$$

If $c^s(t)$ is the server budget at time t and f_k is the time at which chunk H_k ends to execute, it is possible to see that $c^s(f_k) = c^s(a_k) - e_k$, while $c^s(a_{k+1})$ is calculated from $c^s(f_k)$ in the following manner:

$$c^s(a_{k+1}) = \begin{cases} c^s(f_k) & \text{if } d^s_{k+1} \text{ was generated by Rule 2} \\ Q^s & \text{if } d^s_{k+1} \text{ was generated by Rule 1 or 3.} \end{cases}$$

Based on these observations, the theorem can be proved by showing that:

$$D^s(a_{k_1}, d^s_{k_2}) + c^s(f_{k_2}) \leq (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s},$$

and this property can be proved by proceeding by induction on $k_2 - k_1$, using the algorithmic definition of CBS shown in Figure 3.7.

Inductive base. If in $[t_1, t_2]$ there is only one active chunk ($k_1 = k_2 = k$), then two cases have to be considered.

Case a: $d^s_k < a_k + T^s$.

If $d^s_k < a_k + T^s$, then d^s_k is generated by Rule 2, so $a_k + \frac{c^s(f_{k-1})}{Q^s}T^s < d^s_k$ and $a_k = f_{k-1}$, that is

$$a_k + \frac{c^s(a_k)}{Q^s}T^s < d^s_k.$$

Being $c^s(f_k) = c^s(a_k) - e_k = c^s(a_k) - D^s(a_k, d^s_k)$, it is possible to see that

$$a_k + \frac{D^s(a_k, d^s_k) + c^s(f_k)}{Q^s}T^s < d^s_k$$

hence

$$D^s(a_k, d^s k) + c^s(f_k) < (d^s_k - a_k)\frac{Q^s}{T^s}.$$

Case b: $d^s_k = a_k + T^s$.

If $d^s_k = a_k + T^s$, then $D^s(a_k, d^s_k) + c^s(f_k) = e_k + c^s(f_k) = Q^s$.

Hence, in both cases

$$D^s(a_{k_1}, d^s_{k_2}) + c^s(f_{k_2}) = D^s(a_k, d^s_k) + c^s(f_k) \leq (d^s_k - a_k)\frac{Q^s}{T^s} = (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s}.$$

Inductive step. The inductive hypothesis

$$D^s(a_{k_1}, d^s_{k_2-1}) + c^s(f_{k_2-1}) \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s}$$

is used to prove that

$$D^s(a_{k_1}, d^s_{k_2}) + c^s(f_{k_2}) \leq (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s}.$$

Given the possible relations between d^s_k and d^s_{k-1}, three cases have to be considered:

- $d^s_k \geq d^s_{k-1} + T^s$. That is, d^s_k is generated by Rule 3 or Rule 1 when $r_j \geq d^s_{j-1}$.
- $d^s_k = d^s_{k-1}$. That is, d^s_k is generated by Rule 2.
- $d^s_{k-1} < d^s_k < d^s_{k-1} + T^s$. That is, d^s_k is generated by Rule 1 when $r_j < d^s_{j-1}$.

<u>*Case a*</u>: $d_{k_2} = d_{k_2-1} + T_s$.

In this case $d^s_{k_2}$ can be generated only by Rule 1 or 3. Adding e_{k_2} to both sides of the inductive hypothesis, the following disequation can be obtained:

$$\sum_{k=k_1}^{k_2-1} e_k + e_{k_2} \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2-1}) + e_{k_2}$$

and from $c^s(f_k) = c^s(a_k) - e_k$ it follows that

$$\sum_{k=k_1}^{k_2} e_k \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2-1}) + c^s(a_{k_2}) - c^s(f_{k_2}).$$

Since $d^s_{k_2}$ is generated by Rule 1 or 3, it must be $c^s(a_{k_2}) = Q^s$, therefore:

$$\sum_{k=k_1}^{k_2} e_k \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2-1}) + Q^s - c^s(f_{k_2})$$

$$\sum_{k=k_1}^{k_2} e_k + c^s(f_{k_2}) \leq (d^s_{k2-1} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2-1}) + Q^s \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} + Q^s$$

and finally

$$D^s(a_{k_1}, d^s_{k_2}) + c^s(f_{k_2}) \leq (d^s_{k2-1} - a_{k_1})\frac{Q^s}{T^s} + Q^s = (d^s_{k2-1} + T^s - a_{k_1})\frac{Q^s}{T^s}$$

$$D^s(a_{k_1}, d^s_{k_2}) + c^s(f_{k_2}) \leq (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s}.$$

<u>*Case b*</u>: $d^s_{k_2} = d^s_{k_2-1}$.

If $d^s_{k_2} = d^s_{k_2-1}$, then $d^s_{k_2}$ is generated by Rule 2. In this case,

$$\sum_{k=k_1}^{k_2-1} e_k + e_{k_2} \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2-1}) + e_{k_2}$$

but, being $d^s_{k_2} = d^s_{k_2-1}$, $c^s(f_{k_2}) + e_k = c^s(a_{k_2})$ and $c^s(a_{k_2}) = c^s(f_{k_2-1})$ (by Rule 2), it results:

$$\sum_{k=k_1}^{k_2} e_k \leq (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s} - c^s(a_{k_2}) + e_{k_2} = (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2})$$

hence

$$D^s(k_1, k_2) + c^s(f_{k_2}) = \sum_{k=k_1}^{k_2} e_k \leq (d^s_{k_2} - a_{k_1})\frac{Q^s}{T^s}.$$

<u>*Case c*</u>: $d^s_{k_2-1} < d^s_{k_2} < d^s_{k_2-1} + T^s$.

If $d^s_{k_2} < d^s_{k_2-1} + T^s$, then $d^s_{k_2}$ is generated by Rule 1, hence $a_{k_2} + \frac{c^s(f_{k_2-1})}{Q^s}T^s \geq d^s_{k_2-1}$, and $c(f_{k_2-1}) \geq (d^s_{k_2-1} - a_{k_2})\frac{Q^s}{T^s}$. Applying the inductive hypothesis, the following disequation can be obtained:

$$\sum_{k=k_1}^{k_2-1} e_k + e_{k_2} \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2-1}) + e_{k_2}$$

from which it follows that

$$\sum_{k=k_1}^{k_2} e_k \leq (d^s_{k_2-1} - a_{k_1})\frac{Q^s}{T^s} - (d^s_{k_2-1} - a_{k_2})\frac{Q^s}{T^s} + e_{k_2}$$

$$\sum_{k=k_1}^{k_2} e_k \leq (d^s_{k_2-1} - d^s_{k_2-1} - a_{k_1} + a_{k_2})\frac{Q^s}{T^s} + e_{k_2}.$$

Now, being $e_{k_2} = Q^s - c^s(f_{k_2})$, we have:

$$\sum_{k=k_1}^{k_2} e_k \leq (-a_{k_1} + a_{k_2})\frac{Q^s}{T^s} + Q^s - c^s(f_{k_2}) = (a_{k_2} + T^s - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2})$$

but, from Rule 1 and 3, it results that $d_k^s = a_k + T^s$, hence

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k_2}^s - a_{k_1})\frac{Q^s}{T^s} - c^s(f_{k_2})$$

hence

$$D^s(k_1, k_2) + c^s(f_{k_2}) = \sum_{k=k_1}^{k_2} e_k \leq (d_{k_2}^s - a_{k_1})\frac{Q^s}{T^s}.$$

□

The isolation property allows to use a bandwidth reservation strategy to allocate a fraction of the CPU time to each task that cannot be guaranteed a priori. The most important consequence of this result is that soft tasks can be scheduled together with hard tasks without affecting the a priori guarantee even in the case in which soft requests exceed the expected load.

In addition to the isolation property, the CBS has the following characteristics:

- No hypothesis are required on the WCET and the minimum inter-arrival time of the served tasks: this allows the same program to be used on different systems without recalculating the computation times. In other words, this property is the one that permits to decouple the task model from the scheduling parameters.

- If the task's parameters are known in advance, a hard real-time guarantee can be performed. This is the *hard schedulability* property, expressed by the following lemma:

  Lemma 2 *A hard task τ_i with parameters (C_i, T_i) is schedulable by a CBS with parameters $Q_i^s = C_i$ and $T_i^s = T_i$ if and only if τ_i is schedulable without the CBS.*

  Proof.
  For any job of τ_i, $r_{i,j+1} - r_{i,j} \geq T_i$ and $c_{i,j} \leq Q_i$. Hence, by definition of the CBS, each job $J_{i,j}$ is assigned a scheduling deadline $d_{i,j}^s = r_{i,j} + T_i^s$ (since $r_{i,j}$ is always greater than $d_{i,j-1}^s$) and it is scheduled with a budget $Q_i^s = C_i$. Moreover, since $c_{i,j} \leq Q_i^s$, each job finishes no later than the budget is exhausted, hence the deadline assigned to a job does not change and is exactly the same as the one used by EDF. □

- The CBS automatically reclaims any spare time caused by early completions. This is due to the fact that whenever the budget is exhausted, it is always immediately replenished at its full value and the server deadline is postponed. In this way, the server remains eligible and the budget can be exploited by the pending requests with the current deadline.

- Knowing the statistical distribution of the computation time of a task served by a CBS, it is possible to perform a statistical guarantee, expressed in terms of probability for each served job to meet its deadline.

3.4.4 A Model of the CBS

In order to perform a formal analysis of a reservation based scheduler (and of the CBS in particular), a mathematical model of the system is needed. If a task τ_i is scheduled by a reservation based scheduler (for example, if it is served by a CBS) with parameters (Q_i^s, T_i^s), then τ_i can be modelled with a queue. Moreover, if the task's interarrival times are multiple of T_i^s, they can be expressed as $r_{i,j+1} - r_{i,j} = z_{i,j}T_i^s$, hence:

1. each T_i^s units of time, $Q_i^s = B_i T^s$ units of time of task τ_i can be served;
2. the arrival of job $J_{i,j}$ corresponds to a request of $c_{i,j}$ units of time entering the queue;
3. when a job arrives, the next request of c_{j+1} units will arrive after $r_{j+1} - r_j = z_{i,j}T_i^s$ units of time.

Using this model, the evolution of task τ_i can be described by a state variable $x_{i,j}$ defined as follows:

$$\begin{cases} x_{i,1} & = & c_{i,1} \\ x_{i,j} & = & max\{0, x_{i,j-1} - z_{i,j}Q_i^s\} + c_{i,j} \end{cases} \tag{3.1}$$

where $x_{i,j}$ indicates the length of the queue (in time units) immediately after job $J_{i,j}$ arrival.

When job $J_{i,j}$ arrives, it will be served at a rate of Q_i^s units of time each T_i^s, hence, if there are $x_{i,j}$ units of time to serve immediately after $J_{i,j}$ arrival (at time $r_{i,j}$), $J_{i,j}$ is guaranteed to be served in $\left\lceil \frac{x_{i,j}}{Q_i^s} \right\rceil T_i^s$ time units. As a result, job $J_{i,j}$ will finish before time

$$r_{i,j} + \left\lceil \frac{x_{i,j}}{Q_i^s} \right\rceil T_i^s = = r_{i,j} + \left\lceil \frac{x_{i,j}}{T_i^s B_i} \right\rceil T_i^s. \tag{3.2}$$

3.5 Stochastic Analysis of a Reservation Based System

One of the advantages of using a reservation based scheduling approach such as the CBS, is that the scheduling parameters (Q_i^s, T_i^s) can be separated from the task characteristics (such as execution and interarrival times). In this way, if task τ_i is described by a pair of Probability Distribution Functions (PDFs) of the execution and interarrival times, then it is possible to perform a *probabilistic guarantee*, as defined in Section 2.4.2. A simplified stochastic analisys of the CBS (only considering *semiperiodic tasks* and *generalized sporadic tasks*) is presented in [AB99]; this section extends the previous results by generalizing the analisys to tasks characterized a stochastic behaviour in both execution and interarrival times.

In order to perform a stochastic analysis of a generic reservation based scheduling algorithm, the simplified case in which $r_{i,j+1} - r_{i,j}$ is a multiple of T_i^s is considered first, and the model presented in Equation 3.1 is used. Since the execution and interarrival times are random variables described by the PDFs $U_i(c)$ and $V_i(t)$, the state variable x is a random variable too, and is described by a PDF $\pi_k^{(i,j)} = P\{x_{i,j} = k\}$.

According to Equation 3.2, job $J_{i,j}$ will finish before time

$$r_{i,j} + \left\lceil \frac{v_{i,j}}{Q_i^s} \right\rceil T^s$$

hence the probability $\pi_k^{(i,j)}$ that the queue length $x_{i,j}$ is k immediately after a job arrival is a lower bound to the probability that the job finishes before the probabilistic deadline

$$\delta_i = \left\lceil \frac{k}{Q_i^s} \right\rceil T_i^s.$$

Being the interarrival times multiple of T_i^s, it is possible to define $V_i'(z) = P\{r_{i,j} - r_{i,j-1} = zT^S\}$ as probability that the interarrival time between two consecutive jobs is zT_i^s. Hence,

$$V(t) = \begin{cases} 0 & if \quad t \bmod T^S \neq 0 \\ V'(\frac{t}{T^S}) & \text{otherwise.} \end{cases} \tag{3.3}$$

Note that since $c_{i,j}$ and $r_{i,j+1} - r_{i,j}$ are time invariant, $U_i(c)$ and $V_i'(z)$ do not depend on j. Under these assumptions, it is possible to compute $\pi_k^{(i,j)}$ as follows:

$$\pi_k^{(i,j)} = P\{x_{i,j} = k\} P\{\max\{0, x_{i,j-1} - z_{i,j}Q^s\} + c_{i,j} = k\} =$$

$$= \sum_{h=-\infty}^{\infty} P\{\max\{0, x_{i,j-1} - z_{i,j}Q^s\} + c_{i,j} = k \wedge x_{i,j-1} = h\} =$$

$$= \sum_{z=-\infty}^{\infty} \sum_{h=-\infty}^{\infty} P\{max\{0, x_{i,j-1} - z_{i,j}Q^s\} + c_{i,j} = k \wedge x_{i,j-1} = h \wedge z_{i,j} = z\}.$$

Being $x_{i,j}$ and $z_{i,j}$ greater than 0 by definition, the sums can be computed for h and z going from 0 to infinity:

$$\pi_k^{(i,j)} = \sum_{z=0}^{\infty} \sum_{h=0}^{\infty} P\{max\{0, h - zQ^s\} + c_{i,j} = k\} P\{x_{i,j-1} = h\} P\{z_{i,j} = z\}$$

$$= \sum_{h=0}^{\infty} \sum_{z=0}^{\infty} P\{max\{0, h - zQ^s\} + c_{i,j} = k\} V_i'(z) \pi_h^{(i,j-1)} =$$

$$= \sum_{h=0}^{\infty} \sum_{z=0}^{\infty} P\{c_{i,j} = k - max\{0, h - zQ^s\}\} V_i'(z) \pi_h^{(i,j-1)} =$$

$$= \sum_{h=0}^{\infty} \sum_{z=0}^{\infty} U(k - max\{0, h - zQ^s\}) V_i'(z) \pi_h^{(i,j-1)}$$

Hence,

$$\pi_h^{(i,j)} = \sum_{h=0}^{\infty} m_{h,k} \pi_h^{(i,j-1)}$$

with

$$m_{h,k}^i = \sum_{z=0}^{\infty} U_i(k - max\{0, h - zQ_i^s\}) V_i'(z). \tag{3.4}$$

Considering $m_{h,k}^i$ as an element of a matrix M^i, $\pi_k^{(i,i)}$ can be computed by solving the equation

$$\Pi^{(i,j)} = M^i \Pi^{(i,j-1)} \tag{3.5}$$

where

$$\Pi^{(i,j)} = \begin{pmatrix} \pi_0^{(i,j)} \\ \pi_1^{(i,j)} \\ \pi_2^{(i,j)} \\ \pi_3^{(i,j)} \\ . \\ . \\ . \end{pmatrix}.$$

3.5.1 Stability Considerations

For a generic queue, it is known that the queue is stable (i.e., the number of elements in the queue do not diverge to infinity) if

$$\rho = \frac{mean\ interarrival\ time}{mean\ service\ time} < 1.$$

Hence, the stability can be achieved under the condition

$$\frac{E[C_i]}{E[T_i]} < \frac{Q_i^s}{T_i^s}$$

where $E[C_i]$ is the execution time expectation and $E[T_i]$ is the interarrival time expectation.

If this condition is not satisfied the difference $f_{i,j} - r_{i,j}$ between the finishing time $f_{i,j}$ and arrival the time $r_{i,j}$ of each job $J_{i,j}$ of task τ_i will increase indefinitely diverging to infinity as j increases:

$$\lim_{j \to \infty} f_{i,j} - r_{i,j} = \infty.$$

This means that, for preserving the schedulability of the other tasks, τ_i will slow down in an unpredictable manner.

If a queue is stable, a stationary solution of the Markov chain describing the queue can be found; that is, there exists a finite solution Π^i such that $\Pi^i = \lim_{j\to\infty} \Pi^{(i,j)}$. Since $\Pi^{(i,j)} = M^i \Pi^{(i,j-1)}$, we can compute Π as follows:

$$\Pi^i = \lim_{j\to\infty} \Pi^{(i,j)} =$$

$$= \lim_{j\to\infty} M^i \Pi^{(i,j-1)} =$$

$$= M^i \lim_{j\to\infty} \Pi^{(i,j-1)} = M^i \Pi^i.$$

Hence, Π^i can be computed by solving the eigenvector problem

$$\Pi^i = M^i \Pi^i.$$

This solution can be approximated by truncating the infinite dimension matrix M^i to an nxn matrix $\tilde{M^i}$ and solving the eigenvector problem $\tilde{\Pi^i} = \tilde{M^i}\tilde{\Pi^i}$ with some numerical calculus technique.

3.5.2 Relaxing the hypothesis on interarrival times

In the previous analysis, task interarrival times are assumed to be multiple of an integer value T_i^s so that Equation 3.3 is verified. This assumption results to be very useful in order to simplify the queue analysis, but can be unrealistic in some practical situations.

Using some appropriate approximation, it is possible to relax the assumption on the interarrival times without compromising the analysis based on it. When Equation 3.3 is not respected, it is convenient to introduce a new distribution $\tilde{V}_i(t)$ which approximates $V_i(t)$ for enabling the previously developed analysis. In this way, it is possible to analyse the task behaviour based on the approximate PDF $\tilde{V}_i(t)$ instead of the actual PDF $V_i(t)$. In order this approximation to be correct, $\tilde{V}_i(t)$ must:

- be conservative (pessimistic);
- verify Equation (3.3).

"Being conservative" means that if a probabilistic deadline can be guaranteed using $\tilde{V}_i(t)$, it is guaranteed also according to the real distribution $V_i(t)$. Since the opposite is not true, this approach is pessimistic.

The new PDF $\tilde{V}_i(t)$ is conservative if

$$\forall k, \sum_{n=0}^{k} \tilde{V}_i(n) \geq \sum_{n=0}^{k} V_i(n), \tag{3.6}$$

while the second requirement states that

$$\tilde{V}_i(t) = \begin{cases} 0 & if \quad t \bmod T_i^s \neq 0 \\ V_i'(\frac{t}{T_i^s}) & \quad\quad \text{otherwise.} \end{cases}$$

Equation (3.6) states that the approximated interarrival times Cumulative Distribution Function (CDF) $\tilde{W}_i(t)$ computed from $\tilde{V}_i(t)$ must be greater than or equal to the interarrival times CDF $W_i(t)$ computed from $V_i(t)$. Recall that the CDF of a stochastic variable expresses the probability that the variable is less than or equal to a given value. The CDF $W(t)$ of a stochastic variable t can be computed as $W(t) = \sum_{n=0}^{t} V(n)$, where $V(t)$ is the PDF of t, as shown in Figure 3.8.

In practice, the intuitive interpretation of Equation 3.6 is that a $\tilde{V}_i(t)$ is conservative if the probability that the interarrival time is smaller than t according to $\tilde{V}_i(t)$ is bigger than according to $V_i(t)$. Figure 3.9 explains this concept.

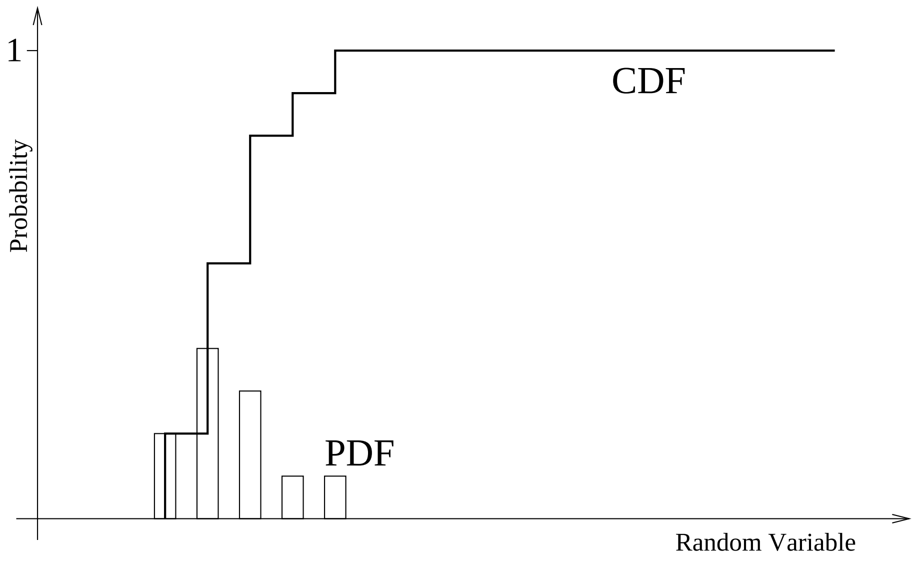

Figure 3.8: CDF vs PDF.

Given a generic interarrival times PDF $V_i(t)$, it is possible to generate a conservative approximation $\tilde{V}_i(t)$ if $\exists k : t < k \Rightarrow V_i(t) = 0$. In this case, it is possible to set $T_i^s < k$ and to compute

$$\tilde{V}_i(t) = \begin{cases} 0 & if \quad t \bmod T_i^s \neq 0 \\ \sum_{i=t-T_i^s+1}^{t} V_i(t) & \text{otherwise.} \end{cases} \tag{3.7}$$

It can be easily verified that if $\tilde{V}_i(t)$ is computed according to Equation 3.7, then it will have both the required properties. However, every conservative approximation $\tilde{V}_i(t)$ respecting Equation (3.6) can be used: an extreme case is using

$$\tilde{V}_i(t) = \begin{cases} 1 & if \quad t = T^S \\ 0 & \text{otherwise.} \end{cases}$$

This is a very pessimistic approximation and corresponds to the worst case sporadic task analysis, based on considering task τ_i as a periodic task with period equal to the MIT. In this case,

$$V_i'(z) = \begin{cases} 1 & if \quad z = 1 \\ 0 & \text{otherwise} \end{cases}$$

and Equation (3.4) becomes

$$m_{h,k}^i = U_i(k - \max\{0, h - Q_i^s\})$$

that is coherent with the results in [AB99].

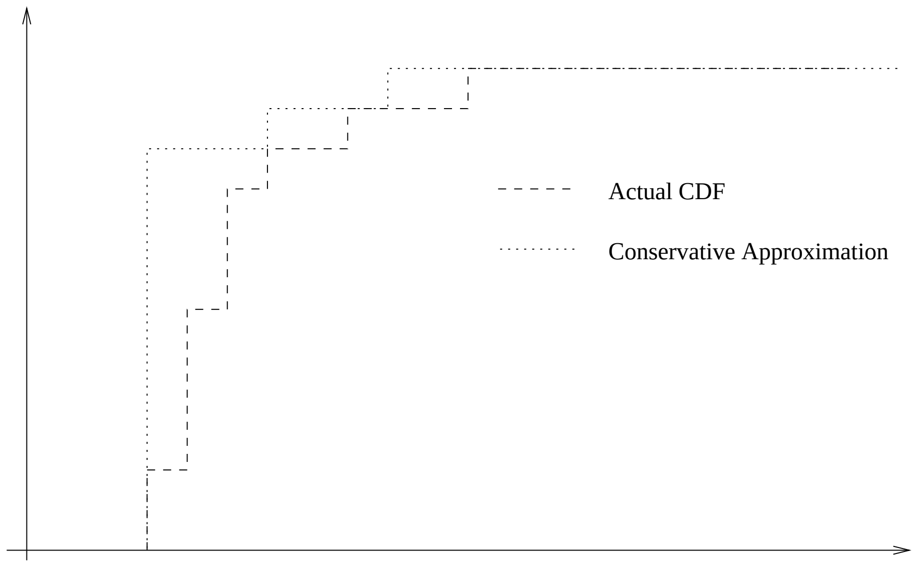

Figure 3.9: Conservative approximation of a CDF.

The $\exists k : t < k \Rightarrow V_i(t) = 0$ assumption is realistic (an interarrival time must have a lower bound) and does not impose serious limits to the applicability of the analysis. However, in some occasions the lower bound can be too small, resulting in a small T_i^s value that unnecessarily increases the number of context switches; in some other cases, a continuous distribution can be used to approximate $V_i(t)$, making difficult to compute the lower bound.

In these cases, some approximation can be introduced by truncating the interarrival times PDF. In practice, this can be done by considering $V_i(t) = 0$ if $t < t_0$, with $V_i(t_0) << 1$; in this way, it is possible to assign $T_i^s \leq t_0$.

Chapter 4

Adaptive Scheduling

When something does not work, reboot the system and restart from beginning.
Computer Scientist approach

When something does not work, try to decompose the problem in smaller problems, so that you will have many problems instead of one.
Computer Engineer approach

When something does not work, keep randomly changing things until it works...
Luca's approach

In Section 2.3, three high-level task models have been introduced (PPS tasks, MM tasks, and ED tasks). Those high level models associate a weight w_i to each task τ_i, and characterise time-sensitive tasks (MM and ED Tasks) with proper temporal constraints. In Chapter 3 some scheduling algorithm that are suitable for managing time sensitive applications have been introduced; however, it is not clear how PPS, MM, and ED tasks can be implemented in terms of those scheduling algorithms.

For example, to guarantee the respect of tasks deadlines (when using priority based scheduling), or to reserve the correct amount of resources to each task (when using a reservation strategy or a PS scheduler), some knowledge about the tasks' execution times is required. But since the tasks' WCETs are not specified in the PPS, MM or ED model, some form of adaptation is needed for performing a correct tuning of the scheduling parameters.

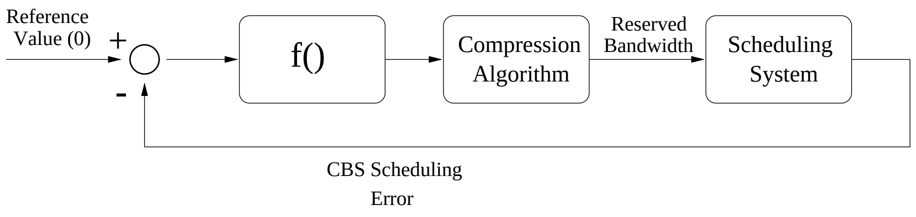

Figure 4.1: The control scheme.

4.1 The Feedback Mechanism: Adaptive Bandwidth Reservations

When the task execution or interarrival times are unknown, some form of adaptation is needed to estimate them. Such an adaptation mechanism can be designed following these steps:

1. choose a suitable low-level **scheduling algorithm**, on top of which the adaptive mechanism will be implemented;
2. **map** the task model's parameters to the scheduling algorithm parameters;
3. design a **feedback mechanism** to adjust the tasks' reserved bandwidth on line.

Since the basic idea is to control each task independently from the others, the proposed feedback scheme is based on a scheduler providing temporal protection between tasks. Every algorithm that provides temporal protection (such as a reservation scheme or a PS scheduler) can be used to implement the low-level scheduler, whose scheduling parameters are adapted by the feedback mechanism so that the user does not have to cope with them. The scheduling algorithm used in this work is based on the Earliest Deadline First (EDF) [LL73], since it achieves full CPU utilization. Based on EDF, temporal protection is provided using a bandwidth reservation mechanism, hence serving each task with a dedicated CBS is the natural choice. Since a CBS S_i is characterised by the pair (Q_i^s, T_i^s) (or (B_i^s, T_i^s)), the PPS, MM, and ED task models must be mapped to such server parameters.

For what concerns PPS tasks, Q_i^s can be considered as the *scheduling quantum*, and it is assigned an initial default value of $\tilde{Q}^s$, which can be changed using a specific system call. Then, each PPS task τ_i is assigned a bandwidth

$$B_i = \frac{w_i}{\sum_{j:\tau_j \in PPS} w_j} B_{PPS} \tag{4.1}$$

where B_{PPS} is the total bandwidth assigned to PPS tasks. Thus, T_i^s is computed as $T_i^s = \frac{Q_i^s}{B_i}$.

Equation 4.1 shows how the total PPS bandwidth is shared among the PPS tasks according to the tasks weights (hence, it results to be identical to the share f_i in a PS system). In this way, the fraction of CPU bandwidth B_{PPS} assigned to PPS tasks will be shared among them proportionally to the weights, like in a PS system [1].

For what concerns MM and ED tasks, the T_i^s parameter is fixed and equal to $T_i = 1/R_i$, where R_i is the *task rate* specified by the user. The Q_i^s parameter is adjusted by the system in order to meet the tasks' temporal requirements. As said, this can be done using a feedback mechanism that controls the scheduling parameters based on an **observed value**. When a CBS is used to serve a time sensitive task τ_i, the natural choice for the observed value is the the *CBS scheduling error*

$$\epsilon_{i,j}^s = d_{i,j}^s - (r_{i,j} + T_i)$$

defined as the difference between the last CBS scheduling deadline $d_{i,j}^s$ assigned to job $J_{i,j}$ and the task's soft deadline $d_{i,j} = r_{i,j} + T_i$.

Since the underlying priority assignment is based on EDF, if the server is schedulable each instance $J_{i,j}$ is guaranteed to finish within the last assigned server deadline $d_{i,j}^s$. Hence, the CBS scheduling error $\epsilon_{i,j}^s$ represents the difference between the deadline $d_{i,j}^s$ that $J_{i,j}$ *is guaranteed* to respect and the deadline $d_{i,j} = r_{i,j} + T_i$ that it *should* respect. When this value is 0, the task is guaranteed to respect its soft deadline, whereas when the scheduling error is different from 0, some task instance *can* terminate after its (soft) deadline because the reserved bandwidth $\frac{Q_i^s}{T_i^s}$ is not enough to properly serve it.

Hence, the objective of the system is to control the scheduling error ϵ^s to 0: if this value increases, Q_i^s has to be increased accordingly, otherwise it can be left unchanged. Based on this idea, a feedback control mechanism can be used to adapt the amount of resources reserved to each task. In practice, the amount of CPU bandwidth $B_{i,j}$ reserved to job $J_{i,j}$ can be different from the amount of CPU bandwidth reserved to other jobs of the same task, and is adjusted according to a *feedback function* $B_{i,j} = f(B_{i,j-1}, \epsilon_{i,j-1}^s)$. Different kinds of feedback functions can be used, but the intuitive requirement is that $\epsilon_{i,j}^s > 0 \Rightarrow f(B_{i,j}, \epsilon_{i,j}^s) > B_{i,j}$.

Note that all this mechanism works correctly under the assumption that all the scheduling deadlines $d_{i,j}^s$ are respected, and this is true if and only

[1]Each PPS task will receive an amount Q_i^s of CPU time every T_i^s, so Q_i^s can be considered as the scheduling quantum in a conventional time sharing system, as said above.

if a schedulability condition $\sum B_i^s \leq 1$ is verified. To better express this requirement, some additional definitions are needed:

Definition 15 *Given a task set* $\Gamma = \{\tau_1, \dots \tau_n\}$ *composed of* n *tasks, a* **bandwidth assignment** $\hat{B}$ *is a vector* $\hat{B} = (B_1^s, \dots B_n^s) \in R^n$ *such that* $\forall i \leq n, 0 \leq B_i^s \leq 1$*, and at every time* $B_i^s = B_{i,j}$*.*

Definition 16 *A bandwidth assignment* $\hat{B}$ *is said to be* **feasible** *if* $\sum_i B_i^s \leq 1$*.*

The feasibility of a bandwidth assignment is a *global* condition, because it depends on all the servers S_i in the system. However, the feedback function $f()$ only performs a *local* adaptation, since it operates only on a single task and does not consider any schedulability (or feasibility) condition[2]. As a result, it is possible that the reserved bandwidths are increased "too much" and the bandwidth assignment is not feasible (that is to say, $\sum_{j:\tau_j \in MM \cup ED} B_j > B^{max}$). In this case, some form of global mechanism is necessary to decrease the tasks' reserved bandwidths so that the assignment is feasible. This compression of the reserved bandwidths is performed by the *compression function* $\hat{B}' = g(\hat{B})$.

The compression function is a function $g : R^n \to R^n$ that transforms an infeasible bandwidth assignment into a feasible one; in practice, if $\hat{B}' = g(\hat{B})$, then $\sum_i B_i^{s\prime} \leq 1$. In particular,

$$B_i^{s\prime} = \begin{cases} B_i^s & \text{if } \sum_i B_i^s \leq 1 \\ g_i(\hat{B}) & \text{otherwise} \end{cases}$$

It is worth noting that according to these considerations it is possible to define a feedback mechanism in which the reserved CPU bandwidth B^s is decreased only in overload conditions (when $\sum_i B_i^s > 1$). The specified task weights w_i can be used to decrease the tasks' reserved bandwidth in overload conditions. This solution has the advantage of avoiding unnecessary bandwidth adaptations, but could be more difficult to analyse.

A simple solution to perform such a bandwidth compression is to scale the tasks' utilizations in a proportional way:

$$B_i^{s\prime} = B_i^s s_i$$

being s_i the scaling factor. Since the compression must be done according to the tasks' weights, s_i must be proportional to w_i: $s_i = w_i M$. Imposing

[2]because it is not aware of all the other reserved tasks in the system.

$\sum_{j:\tau_j \in MM \cup ED} B_j^{s\prime} = B^{max}$, it results:

$$\begin{aligned}
\sum_{j:\tau_j \in MM \cup ED} B_j^{s\prime} &= B^{max} \Rightarrow \\
\sum_{j:\tau_j \in MM \cup ED} B_j^s s_j &= B^{max} \Rightarrow \\
\sum_{j:\tau_j \in MM \cup ED} B_j^s w_j M &= B^{max} \Rightarrow \\
M \sum_{j:\tau_j \in MM \cup ED} B_j^s w_j &= B^{max} \Rightarrow \\
M &= \frac{B^{max}}{\sum_{j:\tau_j \in MM \cup ED} B_j^s w_j}
\end{aligned}$$

Hence,

$$B_i^{s\prime} = B_i^s w_i \frac{B^{max}}{\sum_{j:\tau_j \in MM \cup ED} B_j^s w_j} \tag{4.2}$$

This simple rule can be slightly modified to guarantee a minimum bandwidth B_{min} to each task.

The described closed loop control used to adjust the reserved bandwidth is shown in Figure 4.1.

4.2 Performance of Adaptive Reservations

When implementing an Adaptive Reservation abstraction, it is important to design the feedback function so that the resulting adaptive scheduler is able to assign the correct amount of resource to each task (when possible) in a short time and with an acceptable accuracy. Using the control theory terminology, the closed loop system must be stable, and the response time, overshoot, and steady-state error must be compliant with some specifications.

Although designing a proper feedback function $f()$ might seem to be simple, things are more complicated than what appears at a first glance [Goe02]. Hence, a careful analysis of the closed loop scheduler is needed; in this section, after a simple analysis based on the CBS model developed in Section 3.4.4, a control theoretical approach will be proposed.

4.2.1 Analysis of a Simple Feedback Scheme

Using Adaptive Reservations, the bandwidth reserved to an adaptive task τ_i varies from instance to instance, hence it will be indicated as $B_{i,j}$. If the bandwidth assignment is feasible, $B_{i,j+1} = f(B_{i,j}, \epsilon_{i,j})$; hence, according to Equations 3.1 and 3.2, each task dynamics is described as follows:

$$\begin{cases} x_{i,1} &= c_{i,1} \\ x_{i,j} &= max\{0, x_{i,j-1} - z_{i,j} B_{i,j} T_i^s\} + c_{i,j} \\ \epsilon_{i,j} &= \left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil T_i^s - T_i \\ B_{i,j+1} &= f(B_{i,j}, \epsilon_{i,j}) \end{cases}$$

If the feedback function $f(B, \epsilon)$ is properly designed, it is possible to prove that $B_{i,j}$ will converge to a correct value $\tilde{B}_i > \frac{\overline{c_i}}{T_i}$. For example, if the feedback equation is $f(B, \epsilon) = B + \alpha\frac{\epsilon}{T}B$, then the new bandwidth assigned to job $J_{i,j+1}$ results to be

$$B_{i,j+1} = B_{i,j} + \alpha\frac{\epsilon_{i,j}}{T_i}B_{i,j} = B_{i,j} + \alpha\frac{\left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil T_i^s - T_i}{T_i}B_{i,j} =$$

$$= B_{i,j} + \alpha\left(\left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil \frac{T_i^s}{T_i} - 1\right) B_{i,j}$$

But if the task is semiperiodic, then $T_i = zT_i^s$ (remember that T_i is a multiple of T_i^s), hence

$$B_{i,j+1} = B_{i,j} + \alpha\left(\left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil \frac{1}{z} - 1\right) B_{i,j}$$

Now, defining $S_{i,j} = \left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil B_{i,j}$, it is possible to obtain

$$B_{i,j+1} = B_{i,j} + \alpha(\frac{S_{i,j}}{z} - B_{i,j}) = (1-\alpha)B_{i,j} + \alpha\frac{S_{i,j}}{z} \qquad (4.3)$$

Note that $S_{i,j}$ is an estimation of the CPU bandwidth required to serve $J_{i,j}$ in a server period, hence $\frac{S_{i,j}}{z}$ is an estimation of the CPU bandwidth needed to schedule $J_{i,j}$ in $zT_i^s = T_i$ time units (that is, $S_{i,j}$ should converge to the amount $\tilde{S}_i$ of CPU bandwidth needed by the task to control the scheduling error to 0). Since a succession $a(n+1) = (1-\alpha)a(n) + \alpha S$ converges to S for $n \to \infty$, $\lim_{j\to\infty} B_{i,j} = \tilde{B}_i$ will be equal to the estimated bandwidth requirement $\frac{\tilde{S}_i}{z}$ if the compression equation is not used. Hence, if a feasible bandwidth assignment that controls all the scheduling errors to 0 exists, the reserved bandwidths will converge to it.

More formally, given $\eta > 0$ it exists j_0 such that $\forall j \geq j_0, |B_{i,j} - S_{i,j}| \leq \eta$. Hence,

$$\frac{S_{i,j}}{z} - \eta \leq B_{i,j} \leq \frac{S_{i,j}}{z} + \eta \Rightarrow \frac{\left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil B_{i,j}}{z} - \eta \leq B_{i,j} \leq \frac{\left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil B_{i,j}}{z} + \eta \Rightarrow$$

$$\left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil - \eta \leq z \leq \left\lceil \frac{x_{i,j}}{T_i^s B_{i,j}} \right\rceil + \eta \Rightarrow$$

$$\begin{cases} \frac{x_{i,j}}{T_i^s B_{i,j}} - \eta & \leq & z \\ \frac{x_{i,j}}{T_i^s B_{i,j}} + \eta & \geq & z-1 \end{cases} \Rightarrow \begin{cases} B_{i,j} & \geq & \frac{x_{i,j}}{zT_i^s} - \eta \\ B_{i,j} & \leq & \frac{x_{i,j}}{(z-1)T_i^s} + \eta \end{cases}$$

If $j \geq j_0$, then $B_{i,j}$ will be constrained into the interval $\frac{x_{i,j}}{zT_i^s} \leq B_{i,j} \leq \frac{x_{i,j}}{(z-1)T_i^s}$ of size $\frac{x_{i,j}}{(z-1)T_i^s} - \frac{x_{i,j}}{zT_i^s} = \frac{x}{z(z-1)T_i^s}$. It is easy to see that increasing z the reserved bandwidth will converge to a better estimation of the requested bandwidth. On the other hand, increasing z will decrease the server period, increasing the number of context switches and the kernel overhead.

From this argument, it is possible to understand that the server period has to be carefully chosen in order to find a good trade-off between a more precise resource allocation and a low kernel overhead.

4.2.2 A Control Theoretical Approach

The feedback function can be designed using different approaches, either theoretically founded or empirically proven to be effective. Since closed-loop systems similar to an adaptive reservation have been studied at long in control theory, an idea for properly designing the feedback function is to apply some results form control theory. In fact, control theory provides powerful tools that are very useful in analysing closed-loop systems, proving their stability, and evaluating their dynamic properties.

Additional Definitions

To extend the scheduling error concept t a generic reservation based system, it is useful to define the *latest possible finishing time* for a job. The latest possible finishing time $\mathrm{LFT}_{i,j}$ for job $J_{i,j-1}$ is the end of the latest reservation period used by the job, minus the job arrival time: for example, if $J_{i,j-1}$ has execution time $c_{i,j-1} = 5$, it has been reserved a bandwidth $B_i = 0.5$, and the reservation period is $T_i^s = 4$, then it uses $\left\lceil \frac{5}{0.5 \cdot 4} \right\rceil = 3$ reservation periods, and its latest possible finishing time is 15.

Note that, for CBS, when a job finishes the deadline of the server minus the job arrival time is equal to the latest possible finishing time: $\mathrm{LFT}_{i,j} = d_i^s - r_{i,j-1}$.

Mathematical Model of a Reservation

A proper feedback scheme providing the required characteristics can be designed only based on an accurate model of the system. In this section, a model of a reservation system alternative to the one presented in Section 3.4.4 will be developed.

The notation will be simplified by removing the task index from all the quantities: hence, Q will be used instead of Q_i, T^s will be used instead of T_i^s, J_j will be used instead of $J_{i,j}$, and so on.

The goal of the feedback scheduler is to control LFT to T; thus, the *scheduling error* ϵ_k is defined as the difference between the latest possible finishing time LFT_k and the job relative deadline T. Note that, if $\mathrm{LFT}_k > T$, then job J_{k-1} consumes some of the time that should be used by the next job, which will have less time to execute. In this case, jobs J_{j-1} and J_j share a reservation period, and LFT_{j+1} depends on LFT_j. To express this dependency, and write the dynamic equations of our system, it is useful to introduce another state variable that represents the amount of time used by J_{j-1} in its last reservation that it shares with J_j.

As said, the scheduling error is defined as the difference between the *latest possible finishing time* and the task period:

$$\epsilon_k = \mathrm{LFT}_k - T.$$

Notice that the scheduling error is a discrete variable and it is multiple of T^s.

It is also useful to define a state variable x_k that represents the amount time consumed by job J_{k-1} on the latest reservation period, if shared with job J_k. To help clarify the meaning of x_k, an example is shown in Figure 4.2. In Figure 4.2.a, J_1 uses only 2 reservation periods and finishes before the end of its period: J_1 and J_2 do not share any reservation, and $x_2 = 0$. In Figure 4.2.b, J_1 uses 3 reservation periods: therefore, $x_2 = 1$. In the following, x_k will be assumed to be not measurable.

By definition,

$$\begin{cases} x_0 & = 0 \\ \mathrm{LFT}_1 & = \left\lceil \frac{C_0}{B_0 T^s} \right\rceil T^s \end{cases}$$

The general equations for x_k and LFT_k are the following:

$$x_k = \begin{cases} c_{k-1} + x_{k-1} - (\mathrm{LFT}_k - T^s) B_{k-1} & \mathrm{LFT}_k > T \\ 0 & \mathrm{LFT}_k \le T \end{cases}$$

$$\mathrm{LFT}_k = \begin{cases} \mathrm{LFT}_{k-1} - T - T^s + \left\lceil \frac{c_{k-1} + x_{k-1}}{B_{k-1} T^s} \right\rceil T^s & \mathrm{LFT}_{k-1} > T \\ \left\lceil \frac{c_{k-1}}{B_{k-1} T^s} \right\rceil T^s & \mathrm{LFT}_{k-1} \le T \end{cases}$$

From the previous equation, it is possible to derive the scheduling error:

$$\epsilon_k = \begin{cases} \epsilon_{k-1} - T - T^s + \left\lceil \frac{c_{k-1} + x_{k-1}}{B_{k-1} T^s} \right\rceil T^s & \epsilon_{k-1} \ge T^s \\ \left\lceil \frac{c_{k-1}}{B_{k-1} T^s} \right\rceil T^s - T & \epsilon_{k-1} < T^s \end{cases} \tag{4.4}$$

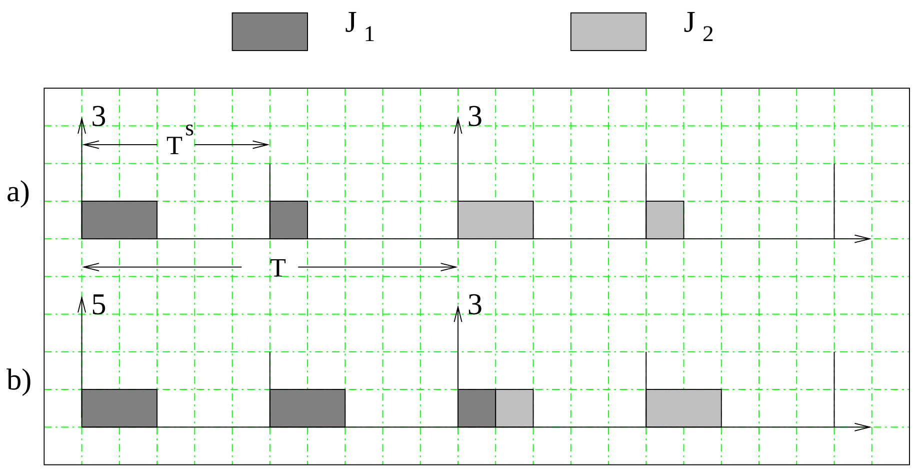

Figure 4.2: Internal state x_j. In case a), the first job finishes before the end of its period, hence $x_2 = 0$; in case b), the first job consumes 3 reservation periods, and consumes 1 capacity unit in the last reservation period, hence $x_2 = 1$.

Now, a *quantisation error* QE_k can be introduced considering two different cases: $\epsilon_{k-1} \geq T^s$ and $\epsilon_{k-1} < T^s$. In the first case, ϵ_k depends on x_{k-1} that is not measurable. However, x_{k-1} is always in the range $[0, B_{k-1}T^s]$. Hence, the following upper bound for the scheduling error holds:

$$\epsilon_k = \epsilon_{k-1} - T - T^s + \left\lceil \frac{c_{k-1} + B_{k-1}T^s}{B_{k-1}T^s} \right\rceil T^s$$

The quantisation error can be defined as:

$$\mathrm{QE}_k = \left\lceil \frac{c_{k-1} + B_{k-1}T^s}{B_{k-1}T^s} \right\rceil - \frac{c_{k-1} + B_{k-1}T^s}{B_{k-1}T^s}$$

In the second case, the quantisation error is defined as follows:

$$\mathrm{QE}_k = \left\lceil \frac{c_{k-1}}{B_{k-1}T^s} \right\rceil - \frac{c_{k-1}}{B_{k-1}T^s}$$

Finally, the scheduling error is redefined as follows:

$$\tilde{\epsilon}_k = \epsilon_k - \mathrm{QE}_k T^s.$$

By substituting,

$$\tilde{\epsilon}_{k+1} = \begin{cases} \tilde{\epsilon}_k + \frac{c_k}{B_k} - T & \tilde{\epsilon}_k \geq T^s \\ \frac{c_k}{B_k} - T & \tilde{\epsilon}_k < T^s \end{cases} \tag{4.5}$$

Controller Design

As shown, a reservation-based scheduler with period T^s can be dealt with as a dynamical system described by the following equations:

$$\epsilon_{k+1} = \begin{cases} \epsilon_k + \frac{c_k}{B_k} - T & \text{if } \epsilon_k \geq T^s \\ \frac{c_k}{B_k} - T & \text{if } \epsilon_k < T^s \end{cases} \tag{4.6}$$

where ϵ_k represents the scheduling error. Equation 4.6 describes an approximation of the scheduling error where the quantisation error QE_j is neglected (in the this issue will be addressed in the sequel). The goal of this section is to propose techniques for effectively designing feedback controllers for this system. This task is hindered by the possibility for the system dynamics of switching between two different modes corresponding to $\epsilon_k \geq T^s$ and $\epsilon_k < T^s$.

The classical "pole-placement" technique can be used to synthesise a controller in each mode; in this way it is possible both to comply with requirements on the closed loop dynamics (i.e. the evolution of the scheduler under the action of a feedback controller).

Let's start to design the controller for the first operating mode (the same consideration apply to the second one): if $\epsilon_k \geq T^s$, then

$$\epsilon_{k+1} = \epsilon_k + c_k u_k - T$$

where u_k is defined as $\frac{1}{B_k}$.

Quantities ϵ_k, c_k, and B_k can be expressed as a constant value plus a variation: $\epsilon_k = \Delta\epsilon_k + \overline{\epsilon}$, $c_k = \overline{c} + \Delta c_k$ and $u_k = \overline{u} + \Delta u_k$. At the steady state it must hold $\overline{c} = \frac{T}{\overline{u}}$.

Assuming small variations around the linearization point, the relation between the variations can be found *via* differentiation:

$$\Delta\epsilon_{k+1} = \Delta\epsilon_k + \overline{c}\ \Delta u_k + \overline{u}\ \Delta c_k = \Delta\epsilon_k + \frac{T}{\overline{u}}\ \Delta u_k + \overline{u}\ \Delta c_k. \tag{4.7}$$

For notational simplicity and with a slight abuse of notation, in the rest of the section the symbol Δ will be dropped. Hence, unless otherwise stated, ϵ_k, u_k and c_k represent variations of the original quantities around the $\overline{\epsilon}, \overline{u}, \overline{c}$ respectively.

As the difference Equation 4.7 is linear, it is possible to compute the Z transform:

$$\epsilon(Z) = H_c(Z)c(Z) + H_u(Z)u(Z),$$

where $H_c(Z) = \frac{\overline{u}}{Z-1}$ and $H_u(Z) = \frac{T}{\overline{u}(Z-1)}$.

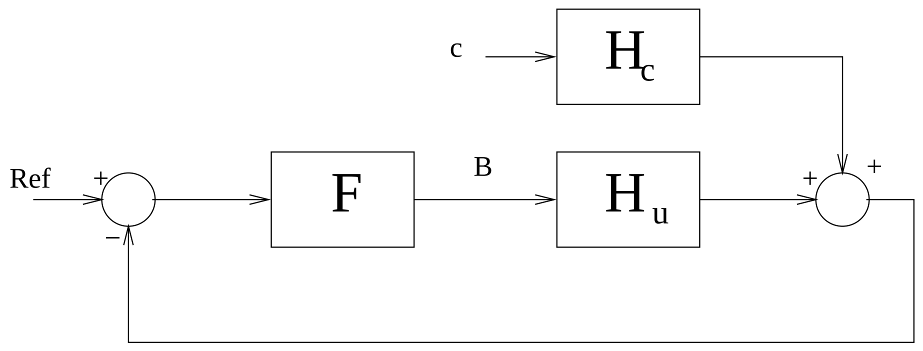

Figure 4.3: Dynamic system representing a linearised reservation with a feedback mechanism.

To achieve the control goals, a feedback controller $F(Z)$ is used as in Figure 4.3: $u(Z) = -F(Z)\epsilon(Z)$. The closed loop dynamics is described by the transfer function $H(Z)$ between $c(Z)$ and $\epsilon(Z)$:

$$\epsilon(Z) = H(Z)c(Z) = \frac{H_c(Z)}{1 + F(Z)H_u(Z)}c(Z) \tag{4.8}$$

The simplicity of the system (whose dynamic equations are similar to those of a tank) suggested the use of a PI controller. A PI controller is described by:

$$u_k = c_1(-\epsilon_k) + c_2\sum_{j=0}^{k-1}(-\epsilon_{k-1})$$

where c_1 and c_2 are the coefficients of the proportional and integral actions respectively. By subtracting the expression for u_{k-1} from the expression for u_k the equation can be written as:

$$u_k = u_{k-1} + \alpha(-\epsilon_k) + \beta(-\epsilon_{k-1}).$$

where $\alpha = c_1$ and $\beta = c_2 - c_1$. The transfer function $F(Z)$ is given by:

$$F(Z) = \frac{\alpha Z + \beta}{Z - 1}.$$

Plugging $F(Z)$ into Equation 4.8,

$$\epsilon(Z) = H(Z)c(Z) = \frac{\overline{u}(Z-1)}{Z^2 + (\frac{T}{\overline{u}}\alpha - 2)Z + \beta\frac{T}{\overline{u}} + 1}c(Z). \tag{4.9}$$

The closed loop system is stable if the zeros Z_i of the denominator in Equation 4.9 (i.e. the poles of the closed loop system), have norm strictly lower than 1: $||Z_i|| < 1$. Observe that the use of the PI controller enables the choice of the two closed loop poles poles. As a matter of fact, to place the closed loop poles in Z_1 and Z_2 it is sufficient to impose:

$$Z^2 + (\frac{T}{\overline{u}}\alpha - 2)Z + \beta\frac{T}{\overline{u}} + 1 = Z^2 - (Z_1 + Z_2)Z + Z_1 Z_2.$$

Solving for α, β yields:

$$\begin{aligned}\alpha &= \frac{\overline{u}(2 - (Z_1 + Z_2))}{T} \\ \beta &= \frac{\overline{u}(Z_1 Z_2 - 1)}{T}.\end{aligned}$$

Moreover, the decay rate ρ is given by the maximum norm of the poles.

Repeating the computations for $\epsilon_{k-1} < T^s$, it is possible to obtain:

$$\begin{aligned}\alpha &= \frac{\overline{u}(1 - (Z_1 + Z_2))}{T} \\ \beta &= \frac{\overline{u}(Z_1 Z_2)}{T}.\end{aligned}$$

All subsequent results can similarly be rephrased.

Accounting for the Quantisation Error

According to Equation 4.5

$$\epsilon_k = \tilde{\epsilon}_k + \mathrm{QE}_k T^s.$$

Let's consider an equilibrium point where the quantisation error has a value $\tilde{\mathrm{QE}}_k$ and repeat the analysis considering the variation around the equilibrium $\mathrm{QE}_k = \tilde{\mathrm{QE}}_k T^s + \Delta\mathrm{QE}_k$, where T^s has been absorbed into $\Delta\mathrm{QE}_k$. Hence, $0 \le \Delta\mathrm{QE}_k \le T^s$. Considering now the linearised system, it is possible to treat $\Delta\mathrm{QE}_k$ as an additional norm-bounded disturbance (see Figure 4.4). The transfer function from such a disturbance to ϵ_k is given by $\frac{1}{1+F_u(Z)\ G(Z)}$.

Thereby, it is possible to use standard results from control theory to compute a bound for the effect of quantisation. Considering for simplicity the case of distinct and real poles, such a bound is provided by $\frac{2\ T^s\overline{u}}{||Z_2 - Z_1||}$. This bound has to be added to the one computed for the uncertainties of the computation time c_k. As one would expect, diminishing T^s (and hence the quantisation grain) results into higher and higher precision for the control. Again, observe that a less conservative bound can be obtained by numerically computing $E = \sum ||f_k||$.

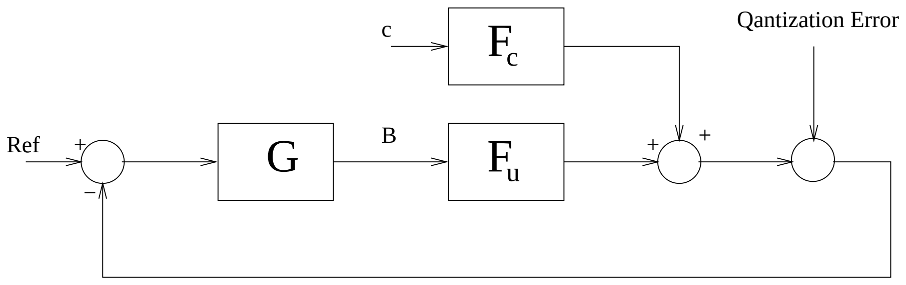

Figure 4.4: Dynamic system representing a linearised CBS with a feedback mechanism.

Moreover, if the controller is able to stabilise the system into a point rather than into a set, it is possible to apply the *Steady State Worst Case Analysis* developed by Slaughter [Sla64]: the worst case steady state quantisation error on ϵ is lower than or equal to $|\frac{1}{1+H_u(Z)F(Z)}|_{Z=1}T^s$. Replacing H_u and F with the expressions provided above, it is possible to conclude that $|\frac{1}{1+H_u(Z)F(Z)}|_{Z=1}T^s = 0$. Therefore, if it is possible to stabilise the system into a point then the steady state value for the effect of the quantisation error is 0. The effect of quantisation is, in this case, an overestimation of the bandwidth $\tilde{B}$ assigned to the task. In fact, imposing the equilibrium condition $\epsilon_{k+1} = \epsilon_k$ in equation 4.4 it is possible to obtain:

$$\left\lceil \frac{\overline{c}}{\tilde{B}T^s} \right\rceil T^s - T = 0.$$

Observing that $x \leq \lceil x \rceil < x + 1$, this results in

$$\frac{\overline{c}}{T} \leq \tilde{B} \leq \frac{\overline{c}}{T - T^s}.$$

Experimental Results

To test the effectiveness of the proposed adaptive scheme, an adaptive reservation controlled by the PI designed in Section 4.2.2 was simulated, comparing the results obtained with different poles assignments and different server periods. These first experiments were performed using synthetic workloads (as proposed in [LSA$^+$00]) to estimate the performance of a feedback scheduler. Then, a more realistic workload (such as the execution times of an MPEG decoder) was simulated to show how a controller that gives good responses to a step can have problems with real workloads. In this case, a proper assignment of the poles is a critical task that needs further investigations.

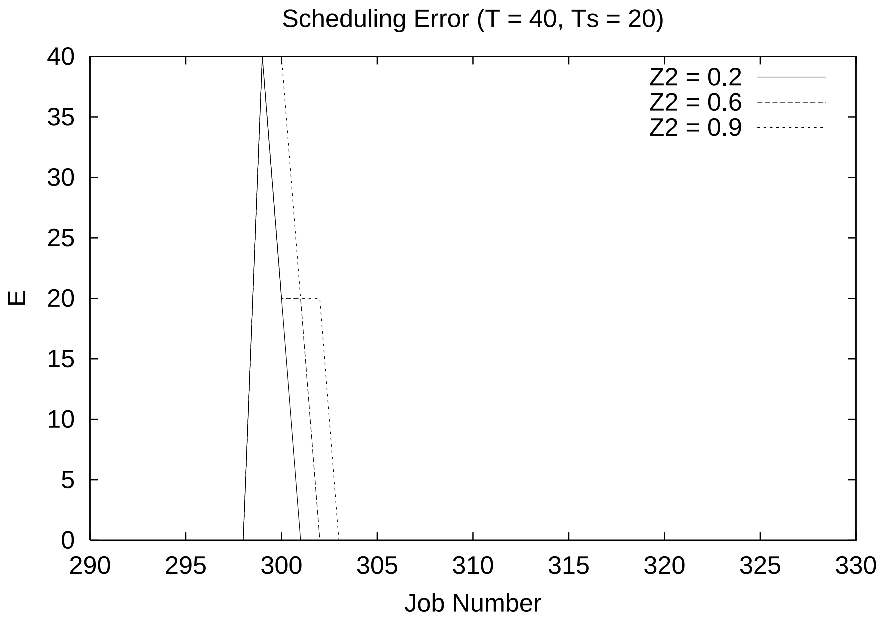

Figure 4.5: Scheduling Error obtained using an adaptive reservation with $T^s = 20$ in response to a step in the load.

Evaluating the performance of a feedback scheduler is not trivial: schedulers that seem to work properly at a first glance [LSTS99] may result to be unstable when evaluated more systematically [LSA+00]. To properly evaluate our adaptive reservation mechanism, the system response to a step and a ramp in the system load were used, since they have been proved to be a good test case [LSA+00].

Although a wide set of experiments was performed, for the sake of brevity only some meaningful experiments are reported here. In particular, in the following results consider a task τ with period $T = 40$ and execution time $c_j = 5$ if $j < 300$, $c_j = 15$ otherwise.

Figure 4.5 shows ϵ_j when $T^s = 20$ (the closed loop poles are assigned as in the previous simulation). When, at job J_{299}, the execution time increases from 5 to 15, the scheduling error raises to 40 (two times the reservation period), and it is controlled to 0 in a short time. Note that in this case the overshot is smaller than in the previous experiment: this is due to the quantisation error caused by the non-accessible internal state. However, when the system reaches the steady state, the quantisation error is 0, as expected. As in the previous case, moving Z_2 from 0.2 to 0.9 the decay rate is higher.

Figure 4.6 shows the evolution of the reserved time, and is probably more interesting. In this case, the impact of the quantisation error is an overestima-

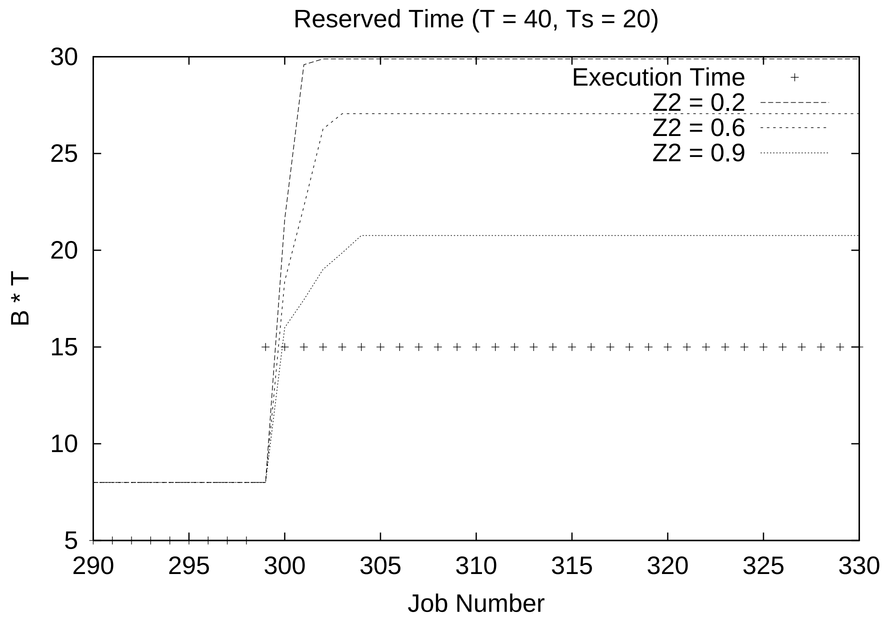

Figure 4.6: Bandwidth reserved by an adaptive reservation with $T^s = 20$ in response to a step in the load.

tion of the reserved bandwidth, which in the worst case results to be 0.747198 instead of $0.375 = 15/40$. Hence, the overestimation is $0.747198 - 0.375 = 0.37220$; this value is compatible with the worst case estimation developed in Section 4.2.2, which is $B^0(T^s/(T - T^s)) = 0.375(20/(40-20)) = 0.375$. Note that, in this case, the quantisation error tends to increase when Z_2 moves to 0.2: in fact, a faster controller tends to "overreact" to the execution times variation, and the quantisation error prevents B_j from decreasing after the first adaptation.

Figures 4.7 and 4.8 plot the evolution of the scheduling error and of B_jT^s when $T^s = 10$, respectively. In this case, the quantisation error is lower and the response becomes closer to the one of model without quantisation. In this case, faster controllers ($Z_2 = 0.2$ and $Z_2 = 0.5$) have an underrun in the scheduling error, that was previously masked by the quantisation error.

The same experiments were repeated using a ramp on the input, and gave similar results.

As previously stated, the first set of experiments was performed based on a synthetic workload that has been recognised as particularly significant for evaluating system performance [LSA$^+$00]. However, some experiments performed using a more realistic workload highlighted new problems.

To generate a realistic workload, an MPEG player running on Linux has

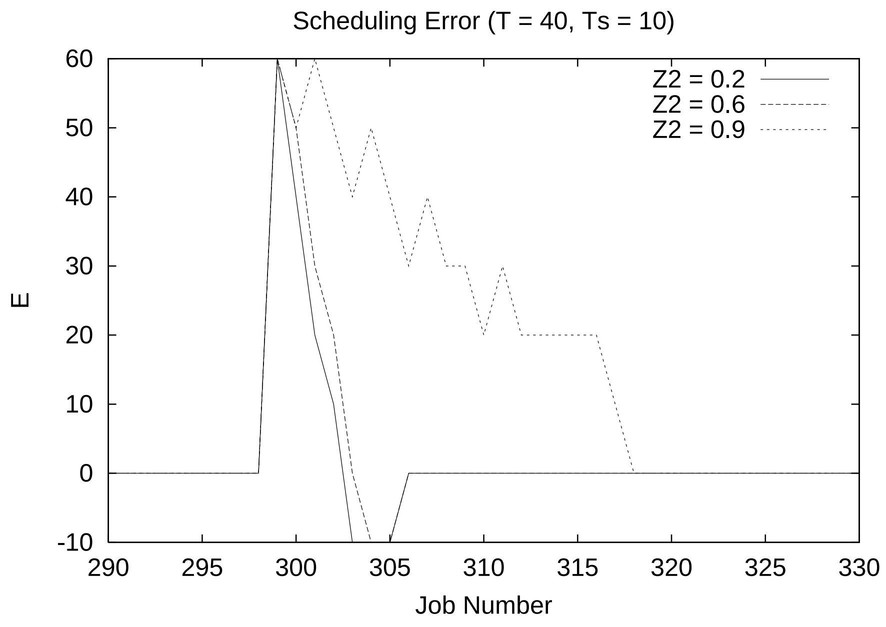

Figure 4.7: Scheduling Error obtained using an adaptive reservation with $T^s = 10$ in response to a step in the load.

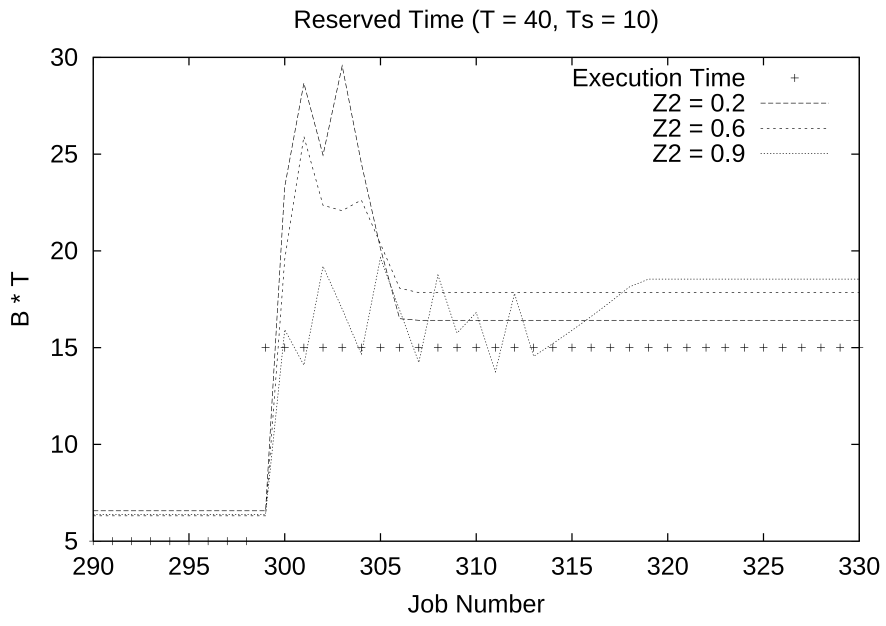

Figure 4.8: Bandwidth reserved by an adaptive reservation with $T^s = 10$ in response to a step in the load.

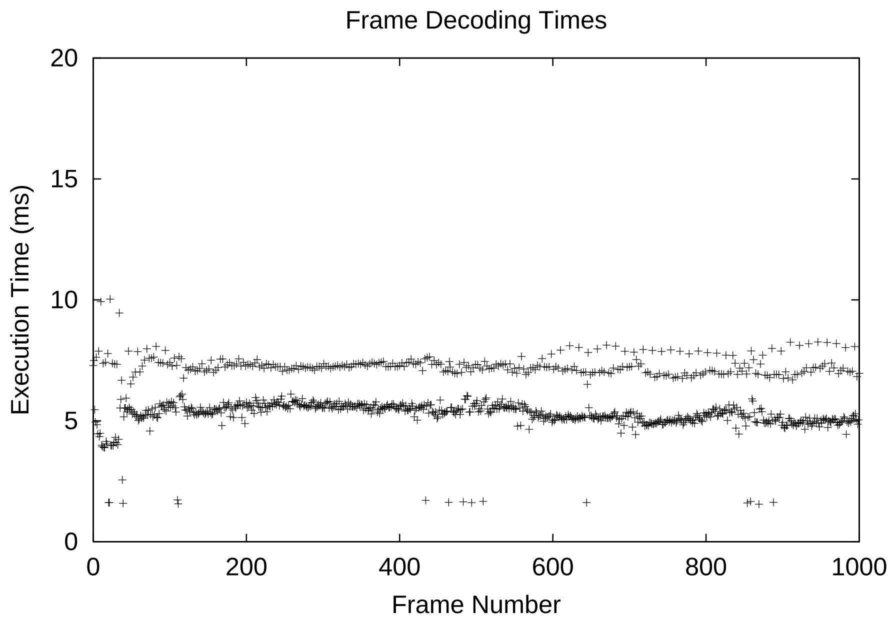

Figure 4.9: Frame decoding times for the Star Wars Episode 1 trailer, measured on a P4 1.80GHz CPU running Linux and XFree86.

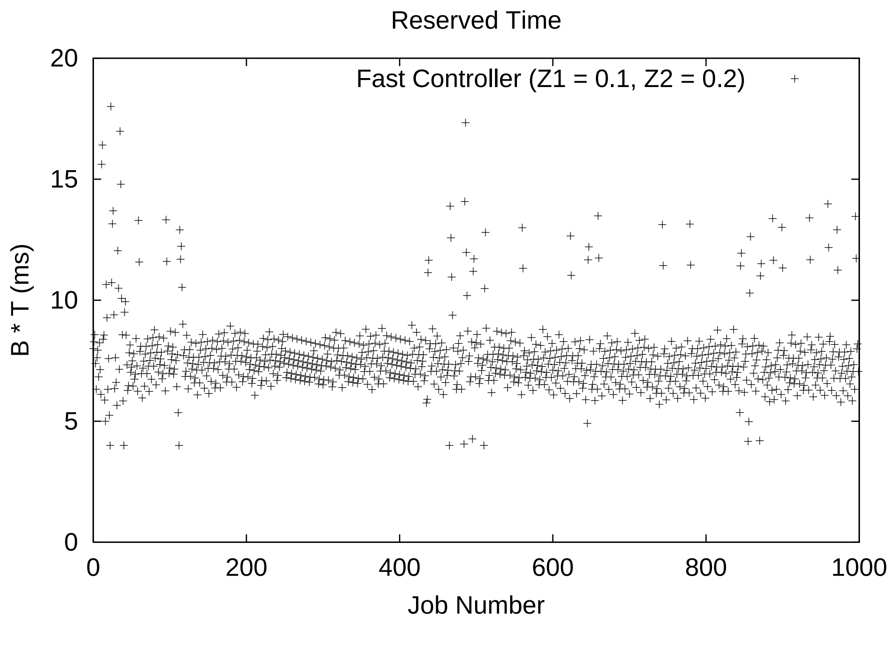

Figure 4.10: Reserved amount of time under a realistic workload (fast PI controller).

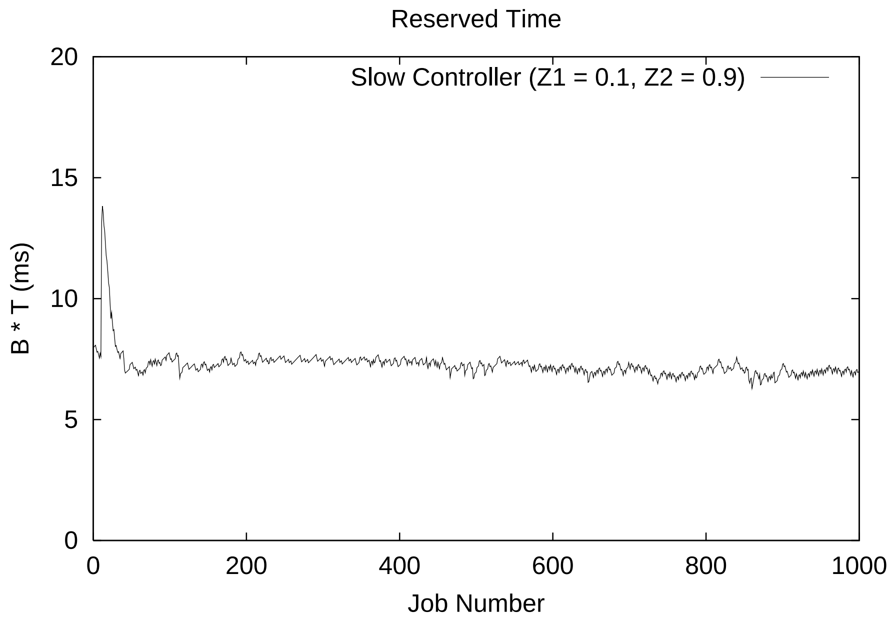

Figure 4.11: Reserved amount of time under a realistic workload (slow PI controller).

been instrumented to measure the frame decoding times for the trailer of Star Wars Episode 1 [Luc99], shown in Figure 4.9. As it is possible to see, the execution times are highly variable. Since the goal of the PI controller is to control the scheduling error to 0, it can be expected that this variability in the execution times will be reflected in a high variability in the reserved time. Figure 4.10 shows the evolution of the reserved time for a PI controller (the simulation was performed setting $T = 33ms$ - 33.3 frames per second, $T^s = T/4 = 8.25ms$, $Z_1 = 0.1$ and $Z_2 = 0.2$. By comparing the two figures, it is clear that the reserved bandwidth does not stabilise properly; as a result, the scheduling error does not stabilise to 0, but continues to oscillate. This kind of problem can be expected from the theory of control, because the system's input is highly variable. Since the system is practically stable and the variations in the input are bounded, the variations on the scheduling error are also bounded (and the average of the scheduling error is 0).

This problem can be addressed by filtering out the higher frequencies (this can be done by moving one of the two poles near to 1). The results are shown in Figure 4.11. By comparing Figures 4.11 and 4.9, it is clear that the reserved bandwidth results to be more stable, and this permits to better control the scheduling error. The first controller (with $Z_2 = 0.2$) tends to "over-react" to execution time variations, presenting a bigger overshot: even after the initial transient, the scheduling error raises to more than $33ms$. On the contrary, moving the second pole to $Z_2 = 0.9$, the maximum scheduling error registered after the initial transient is $8.75ms$.

Summing up, while considering the response to a step or to a ramp the position of the poles Z_1 and Z_2 only influences the overshoot and the response time, when a more realistic workload is applied as input to the system, the position of the poles becomes critical for the system performance.

4.3 The QoS Manager

To test the effectiveness of the proposed approach, the adaptive reservation abstraction described in the previous sections has been implemented through a QoS Manager that realizes the control loop used to adjust the scheduling parameters.

Most of the functionalities of the control loop are coded in the user-level QoS manager; in this way, the kernel is only required to:

- provide temporal protection (hence, the kernel scheduler has to use an appropriate scheduling algorithm;
- give the possibility of changing the scheduling parameters of each task;
- export some kind of performance parameter that can be used as an observed value for the control loop. As said, if the kernel implements the CBS algorithm, it can export the CBS scheduling error.

In this vision, the kernel provides a *mechanism*, the scheduling algorithm, that is used by the QoS manager to implement a resource management *policy*. The tasks whose scheduling parameters are controlled by the QoS Manager are referred as adaptive tasks, whereas the other tasks (characterised by fixed scheduling parameters) will be referred as regular tasks.

Since the QoS Manager needs to have a global system visibility to implement the compression equation (and to eliminate the problems described in [CT94]), it is a regular task (indicated as `qosman` in Figure 4.12). The QoS Manager task is used to create adaptive tasks and to manage their bandwidths according to some user defined policy.

All the adaptive applications have to be linked against the QoS library, that interfaces them with the QoS manager, providing some library calls to communicate with it.

When the `qosman` task is created, it asks the system for all the available CPU bandwidth in order to distribute it among adaptive tasks. When an application needs to create a new adaptive task, it must issue a request to the

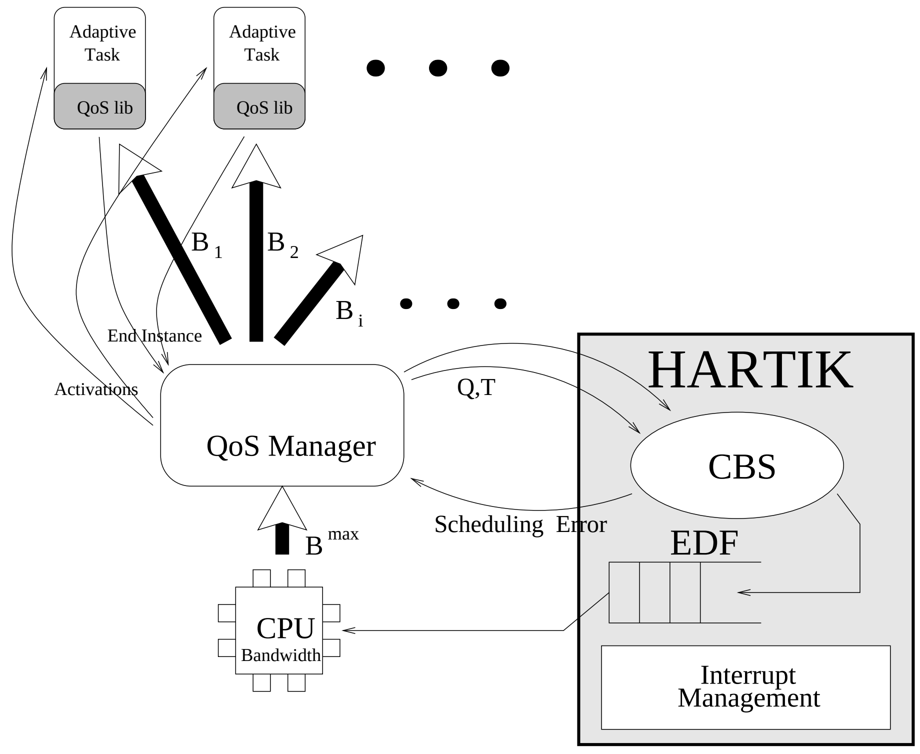

Figure 4.12: The Adaptive QoS Manager.

`qosman` (through the `qset_addtask()` library call). After this call the task is created and added to the set of the tasks handled by the QoS Manager.

At the beginning of each period, the adaptive task is activated (i.e., a new job is created for that task). When the job finishes, the task has to notify this event to the `qosman` task (through the `qtask_endinstance()` library call); in this way, `qosman` has the possibility to monitor the performance of the adaptive task. In this case, performance monitoring is done by measuring the scheduling error, that will be used to compute the new requested bandwidth by applying the feedback function $f()$. After that, the QoS manager will apply the compression function $g()$, and will adjust the parameters of all the adaptive tasks in the system (note that if the requested bandwidth assignment is feasible, the scheduling parameters of only one task need to be changed).

4.3.1 Implementation on the HARTIK kernel

A first version of the QoS manager has been coded on HARTIK [AB00] (that directly provides the CBS scheduling inside the kernel), taking advantage of the particular HARTIK structure to simplify and making more efficient the implementation.

HARTIK is in fact a real-time executive that must be directly linked to the application program, sharing code and data with it. In particular, the kernel structures are not protected from the application, and all the application threads share the same address space. The first prototypal implementation of the QoS Manager, based on HARTIK, took advantage of these peculiarities of the HARTIK kernel to improve the efficiency and reduce the overhead. On the other hand, this implementation is not portable.

After that the first prototype showed the effectiveness of adaptive reservations, the QoS manager has been reimplemented in a portable way, to provide support for different kinds of kernels.

4.3.2 Portable Reimplementation

To make the QoS manager independent from the OS kernel, its functionalities have been split between user tasks (inside the QoS Library) and the `qosman` task. In this way, the adaptation mechanism is distributed between the application address space and the QoS manager address space, and some IPC mechanism is used to allow communication between the QoS library (in the application space) and the QoS manager. In a unix-like system, such as Linux, this mechanism can be provided through some form of IPC (for example, a FIFO, or named pipe); in non-protected systems, such as

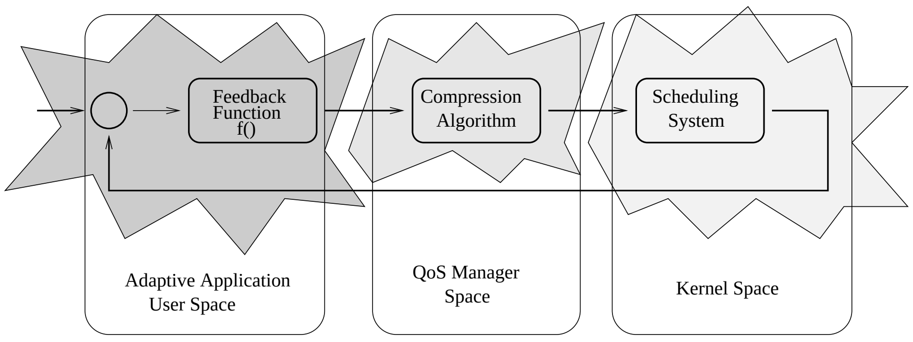

Figure 4.13: The closed-loop controller: Client/Server architecture.

HARTIK, this communication can be more efficiently performed using shared memory.

Hence, the portable implementation of the QoS manager and library is based on a two-layer approach, in which the upper layer is system independent, whereas the lower layer is system defendant and is responsible for:

- providing a simple and efficient communication mechanism between the QoS library and the `qosman` task;
- implementing the interactions between the QoS manager and the kernel (that is to say, reading the scheduling error and modifying the scheduling parameters).

This new implementation results in a client/server structure, and this approach also helps to better isolate the various functionalities into specific modules, as shown in Figure 4.13:

1. the QoS library code, running in the adaptive application address space, is responsible for reading the observed value and computing the feedback function,
2. the QoS manager receives requests from the adaptive applications, and performs the resource assignment applying the compression function,
3. the kernel schedules the tasks according to the parameters set by the QoS manager, and produces a new scheduling error.

All the interface calls are implemented by QoS library functions that send the appropriate requests to `qosman`. In particular, the `qman_endinstance()` call reads the scheduling error (using an appropriate call to the OS kernel, or passing through the QoS manager), computes the feedback function (only

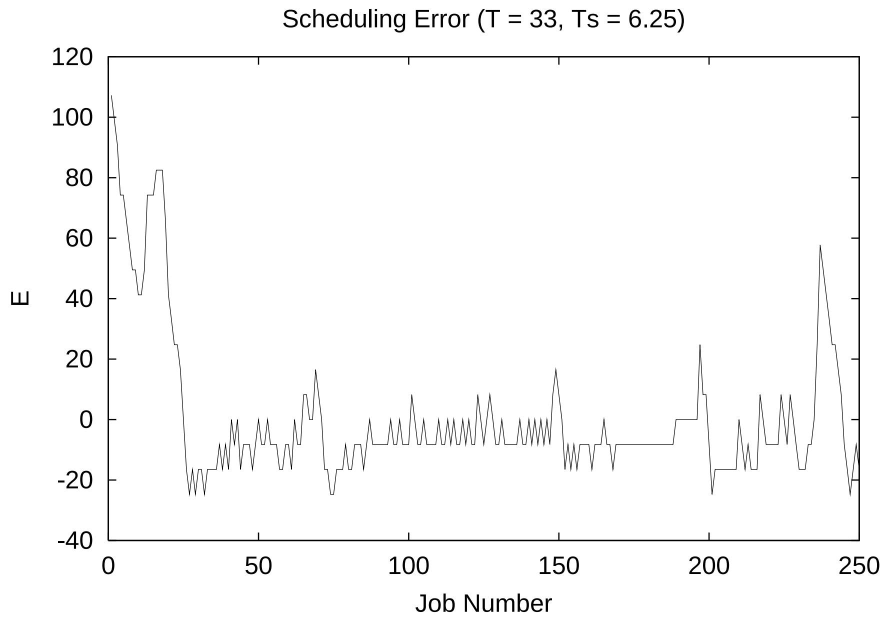

Figure 4.14: Scheduling Error for an MPEG player with $T = 33ms$ and $T^s = 6.75ms$.

using the read value and the status information contained in the application's address space), and sends the new bandwidth requirement to `qosman`.

The QoS Manager task receives bandwidth requests from the application tasks, and serves them by adapting the tasks' scheduling parameters. If the sum of the CPU bandwidths requested by all the clients (the adaptive tasks) through the QoS library is greater than U_{lub}, then the compression function is applied, and the CPU bandwidth assigned to all the adaptive tasks is updated. Otherwise, only the CPU bandwidth reserved to the task performing the request is changed. Note that `qosman` is the only task that can modify the scheduling parameters in the kernel.

The portable reimplementation of the QoS manager currently runs on HARTIK and on the Linux kernel (Linux/RK [RAdN$^+$00] in particular).

4.3.3 Experimental Evaluation

The effectiveness of the QoS Manager was tested implementing the controller described in Section 4.2.2; the implementation of the PI controller was a simple task and required less than half an hour.

Using this implementation, the feedback scheduler was tested by running two simultaneous MPEG players (at $33.3Fps$ and $20Fps$) attached to two

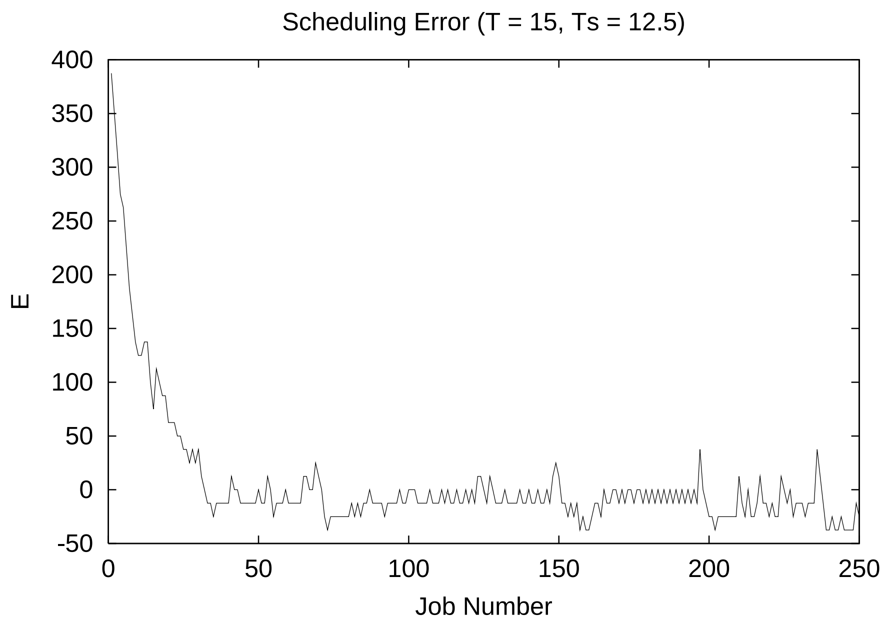

Figure 4.15: Scheduling Error for an MPEG player with $T = 50ms$ and $T^s = 12.5ms$.

adaptive reservations, with periods $33/4 = 8.25ms$ and $50/4 = 12.5ms$. The scheduling errors for the two players are shown in Figures 4.14 and 4.15. These experiments were performed setting $Z_1 = 0.1$ and $Z_2 = 0.8$.

After an initial transient, the feedback controller is able to adapt the reserved bandwidths so that the scheduling error is controlled to about 0. Since the execution times are highly variable, the scheduling error cannot be constant, but it is important to note that $\epsilon \leq 0$ most of the time (remember that a negative scheduling error is not bad for the perceived QoS). In coincidence with big variations in the execution times, the scheduling error increases, but it is immediately controlled to 0 again. It is important to note that these plots refer to *real experiments* performed on a *real Linux system*, and that the two players run simultaneously and share some important resource, such as the X server.

4.4 User Level Adaptation

In this section, the previously introduced adaptation mechanism will be described from a different point of view, analysing it in terms of *demanded bandwidth* and *requested bandwidth*. The concepts of demanded bandwidth and reserved bandwidth have been informally used in the previous sections, and will be more formally defined here.

The demanded bandwidth can be defined based on the time $D_i^s(t_1, t_2)$ demanded by server S_i. In fact, it has been proved that $D_i^s(t_1, t_2) \leq (t_2 - t_1)B_i^s$, hence

Definition 17 *The* **demanded** *bandwidth is be defined as*

$$B_i^{demanded} = \max_{(t_1,t_2)} \frac{D_i^s(t_1, t_2)}{t_2 - t_1}.$$

First of all, note that

$$B_i^{demanded} = \max_{(t_1,t_2)} \frac{D_i^s(t_1, t_2)}{t_2 - t_1} \Rightarrow B_i^{demanded} \leq \frac{(t_2 - t_1)B_i^s}{t_2 - t_1} = B_i^s.$$

Moreover, it is easy to find a case (a continuously backlogged task) in which the demanded bandwidth is equal to the reserved bandwidth, hence $B_i^{demanded} \geq B_i^s$.

As a result, we obtain

$$\left\{ \begin{array}{lcl} B_i^{demanded} & \leq & B_i^s \\ B_i^{demanded} & \geq & B_i^s \end{array} \right. \Rightarrow B_i^{demanded} = B_i^s$$

Since the demanded bandwidth results to be equal to the reserved bandwidth, they will be both referenced as B_i^s in the future.

The requested bandwidth can be defined based on the tasks' soft deadlines, in order to describe the amount of the CPU bandwidth that the task should be reserved to fulfil its time constraints.

Definition 18 *Given a task τ_i, its* **requested** *bandwidth $B_i^{\mathcal{R}}$ is defined as*

$$\max_{t_1,t_2} \frac{D_i(t_1, t_2)}{t_2 - t_1}$$

Where $D_i(t_1, t_2)$ is the time demanded by the tasks' soft deadlines, as previously defined in Chapter 2.

Now, let's remember that a task served by a CBS S_i cannot demand more than the reserved bandwidth B_i^s: if the task requested bandwidth B_i is greater than B_i^s, the task will slow down in order not to affect the others. This can be better understood in the following way:

$$B_i^{req} = \lim_{t \to \infty} \frac{D_i(0, t)}{t} = \lim_{t \to \infty} \frac{\sum_{d_{i,j} \leq t} c_{i,j}}{t} = \lim_{k \to \infty} \frac{\sum_{j=0}^{k} c_{i,j}}{t} =$$

$$= \lim_{k\to\infty} \frac{\sum_{j=0}^{k} c_{i,j}}{r_{i,k}+a} = \lim_{k\to\infty} \frac{\sum_{j=0}^{k} c_{i,j}}{k} \frac{k}{r_{i,k}+a} = \lim_{k\to\infty} \frac{\sum_{j=0}^{k} c_{i,j}}{k} \lim_{k\to\infty} \frac{k}{r_{i,k}+a} \Rightarrow$$

$$B_i^{req} = E[U(c)]\frac{1}{E[V(t)]} = \frac{E[U(t)]}{E[V(t)]}$$

As shown in Section 3.5, if $\frac{E[U(t)}{E[V(t)]} < B_i^s$, the task QoS can be controlled, otherwise the scheduling deadline will be postponed in an unpredictable way. Since $\frac{E[U(t)}{E[V(t)]} < B_i^{req}$, the previous condition can be rewritten as $B_i^{req} < B_i^s$.

Hence, if a task "requests too much bandwidth" (i.e., if the requested bandwidth is greater than the reserved bandwidth: $B_i^{req} > B_i^s$) its schedule is no more predictable, and its QoS cannot be controlled. In this dissertation, a task requiring too much bandwidth is referred as an *overloaded task*.

Definition 19 *Task τ_i is said to be* **overloaded** *if*

$$B_i^{req} \geq B_i^s. \tag{4.10}$$

The "task overload" situation can be removed in two ways (that may also be combined together):

1. By increasing the reserved bandwidth B_i^s
2. By decreasing the task requested bandwidth B_i^{req}

The first strategy is used by adaptive reservations, where the scheduler or a QoS manager adapts the reserved bandwidths in order to resolve all the task overload situations (if possible). In the second strategy, each application explicitly scales down its QoS (and consequently its resource requests), in order to make $B_i^{req} < B_i^s$, thus removing the overload condition. This is referred as *application level adaptation*, since in this case each application has the responsibility to cope with its own overloads (each application can scale down its QoS in different ways, and it is the only entity to know how to perform such a QoS adaptation). Numerous solutions for performing such an application level adaptation have been proposed in the literature and are well known in the multimedia community, ranging from enlarging the task's period to skipping some tasks' instances.

Note that if the sum of all the requested bandwidths $\sum_i B_i^{req}$ is less than the maximum available CPU bandwidth B^{max}, then the adaptive reservation mechanism will be able to use the first strategy (global adaptation) to find a feasible bandwidth assignment $\hat{B} = (B_1^s, \dots B_n^s)$ such that $\forall i, B_i^s \geq B_i^{req}$.

If, on the other hand, $\sum_i B_i^{req} \geq B^{max}$, then the less important tasks can suffer from local overloads. Indeed, the goal of the global adaptive reservation mechanism is to isolate task overloads in the less important tasks, independently from their requirements and periods. In this aspect, adaptive reservations differ from classical real-time techniques, in which task importance is inversely proportional to its period.

In this case, an overloaded task can use application level adaptation to try to scale down its requirements (by decreasing its QoS). If such an adaptation is performed, the task may exit the overload condition, *reaching a lower QoS level in a controlled fashion*, otherwise the QoS degradation can be unpredictable.

If a task τ_i does not implement the application level adaptation, the less important tasks (the tasks τ_j with $w_j \leq w_i$) will be more penalised in terms of reserved bandwidth, since the global adaptive reservation mechanism performs the compression based on task importances w_i. Hence, the bandwidth of the less important tasks will be used to satisfy the QoS requirements of the most important tasks. Such a system behaviour is consistent with the proposed QoS model (avoiding overloads in the most important tasks). A possible concern can be that a misbehaved task having a high importance can compromise the QoS experienced by all the applications in the system. However, the importance w_i is assigned by the user, and can be used as a mechanism to penalise misbehaved tasks or applications that do not adapt their QoS properly.

Since the amount of resources requested by a task to provide a specified level of QoS is not always known (and only a feedback mechanism can be used to control the QoS) the global adaptive reservation mechanism alone may not be able to guarantee a minimum QoS to each task.

If application level QoS adaptation is implemented, the task can scale down its resource requirements in order to provide a minimum QoS, if the task is guaranteed to receive a minimum amount of resources. For this reason, the original adaptive reservations scheme can be enhanced in order to guarantee a minimum fraction of the CPU bandwidth to each task. Note that this modification only affects the compression equation, and does not change anything in the original feedback scheme.

4.4.1 Hierarchical QoS feedback control

As shown, when application level adaptation is used together with the adaptive reservation approach, there are two *orthogonal* forms of adaptation:

- the reserved bandwidth adaptation realized by an active entity having a global system visibility, such as a QoS manager or the scheduler itself;
- the application level QoS adaptation, as presented in the previous section.

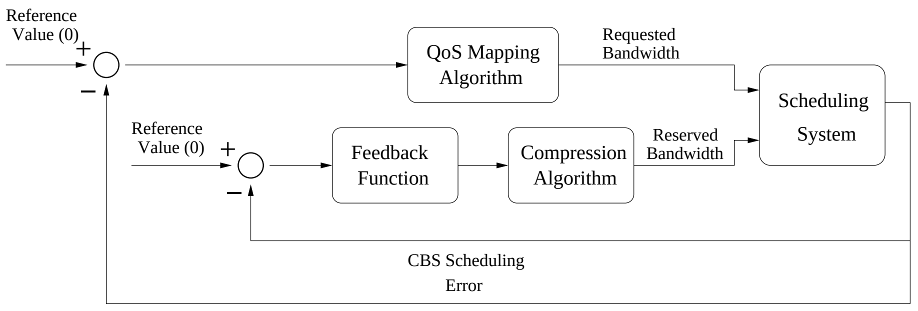

Figure 4.16: Two-Level Feedback.

This integrated approach, referred in this dissertation as *hierarchical adaptation*, presents the advantages of both methods, allowing the applications to scale their QoS when the bandwidth adaptation is not able to serve them properly. In fact, it can be shown that adaptive reservations can suffer when all the tasks require too much resources (basically, when $\sum_i B_i^{req} > B^{max}$), and the QoS adaptation mechanism can solve this problem. On the other hand, the bandwidth adaptation mechanism allows applications to obtain the desired QoS without requiring any a-priori knowledge on their resource requirements.

To use the hierarchical QoS management approach, a new level of feedback has to be added to the feedback scheme of Figure 4.1, as shown in Figure 4.16. The inner loop controls the bandwidth B_i^s reserved by the global adaptive reservation, while the outer loop controls the bandwidth B_i^{req} requested by the application, using the local method. As explained above, the goal of the control loops is to obtain $B_i^s > B_i^{req}$. One of the major problems with this kind of hierarchy is that it can easily reach unstable conditions. For example, let's consider two tasks τ_1 and τ_2: by reacting to a transient overload, the global adaptive reservation mechanism can decrease B_1^s; if τ_1 reacts immediately by decreasing its QoS, when the transient overload finishes the bandwidth adaptation mechanism can increase B_2^s. In this way, τ_2 increases its QoS level, stealing bandwidth from τ_1, preventing it to recover its initial QoS level.

To solve this problem, the application level QoS adaptation action has been made slower than the bandwidth adaptation one, so that QoS is changed only when the overload condition is long (in most cases, the QoS is not scaled in response to transient overloads).

More information about hierarchical QoS adaptation can be found in [AB01].

Chapter 5

OS

5 years from now everyone will be running free GNU on their 200 MIPS, 64M SPARCstation-5
Andrew S. Tanenbaum, **30 Jan 1992**

In the first part of this dissertation, scheduling and resource allocation techniques suitable for serving time sensitive applications have been presented. However, those issues have been addressed from a "purely mathematical" point of view, without considering real implementations. In this chapter, the implementation of the previously described techniques will be analysed, showing the most important problems and some possible solutions.

5.1 Kernel Structures

Since the structure of the kernel can heavily influence the accuracy of the scheduler, imposing or removing constraints and assumption on the resource allocation strategies, in this section the most diffuse kernel structures will be presented and evaluated from the real-time perspective.

5.1.1 Executives

The simplest way to organise system and user code is the one used by executives. An executive is a bunch of library code that can be linked to an application, providing some "kernel functionalities", such as multithreading, interrupt management, and so on. As for traditional kernels, the role of an executive is to abstract the hardware machine, implementing a higher level programming interface.

The main difference with a kernel-based OS is that executives do not create a real distinction between application code and system code, and everything is mixed together. For this reason, someone tends to see an executive as a LibOS (library Operating System).

Using an executive, the "system services" can be invoked through simple near or far calls: no interrupt, trap, or gate mechanism is needed. As a result, an executive is generally more efficient and introduces less overhead, providing good real-time performance and introducing less unpredictabilities in the scheduling. For this reason, executives are often chosen to implement simple real time systems, such as RTEM [rte], HARTIK/SHARK [AB00, But93, LLB$^+$97, GAGB01], and similar. On the other hand, the increased efficiency achieved by eliminating the barrier between system code and user code results in a decreased flexibility and in the total absence of any kind of protection.

5.1.2 DOS-like Systems

Respect to executives, in this kind of systems (sometime called "systems without kernel"), there is a better distinction between application code and system code. However, the application still has complete access to the hardware (and this fact permits to increase the efficiency and predictability of device drivers). In this way, system services are only "facilities" that applications may or may not use. Protection is not enforced, and each application is free to do everything (even crash the system): someone sees this fact as a drawback (lack of protection), someone else loves this kind of freedom (better predictability).

Since system code and application code are separated, applications can require system services through system calls, that are implemented using an INT/TRAP mechanism (as in MSDOS) or some system entry table (as in Amiga OS). However, due to the lack of some concepts like protection, and similar, it is not appropriate to talk about a "real kernel".

Examples of this kind of systems are MSDOS and its clones (such as FreeDOS [fre]), AmigaDOS, and similar. Note that, thanks to their simplicity and predictability, these OSs are often used in embedded and real-time systems.

5.1.3 Monolitihic Kernels

This is the most common OS structure: a single program, the *kernel*, running in privileged mode (system mode), abstracts the hardware providing a high level Application Binary Interface (ABI). Since protection is enforced by the kernel (using appropriate hardware facilities such as the MMU), applications cannot directly access hardware resources, but must require such an access to the kernel.

The kernel is implemented as a single-threaded program, hence only one single execution flow can run in system mode at each time. The kernel responds to two different kinds of requests, coming from up (application requests) or down (hardware requests); application requests are the system calls, conforming to the kernel exported ABI.

Application requests are often called *top halves* in Unix terminology, while bottom requests are called *bottom halves* (in Linux), *soft interrupts* (in the *BSD world), or *Deferred Procedure Calls* - DPCs - (in Windows & similia). As said above, it is avoided to execute more than one top half simultaneously; this requirement is often enforced using non-preemptable system calls.

Since a task cannot be preempted during the execution of a system call, only one top half per time is active. Moreover, top halves also need to synchronise with bottom halves: bottom halves are executed atomically, according to kernel-defined priorities, immediately before returning from system mode to user mode. When a hardware interrupt fires, the system executes an Interrupt Service Routine (ISR) that acknowledges the hardware and queues a request for a bottom half execution. Since the bottom half will be executed immediately before returning to user mode, if the interrupt interrupted a user mode program the bottom half will execute immediately before the ISR, whereas if the interrupt interrupted a top half the bottom half will execute after the top half. In this way the atomicity between top halves and bottom halves is guaranteed; to synchronise with ISRs, a top half needs to explicitly disable and reenable interrupts. For this reason, monolithic kernels are often referenced as non-preemptive and single-threaded kernels.

To correctly manage multiple processors (I.E., SMP machines), a monolithic kernel requires strong modifications. This is due to the fact that in a multiprocessor environment the simple top half/bottom half synchronisation scheme does not work (for example, nothing in the scheme presented above prevents two top halves from executing simultaneously on two different CPUs), and more complex mechanisms (such as spinlocks) must be used. Note that the modifications needed to use a monolithic kernel on an SMP machine make it more similar to a multithreaded kernel.

Finally, note that nonpreemptable system calls and bottom halves can create scheduling anomalies by removing the "full preemptability" hypothesis used in real-time theory, and by introducing priority inversions. Hence, although the monolithic structure permits to enforce protection between user applications and to achieve high throughput, it is not suitable for real-time systems.

5.1.4 Multithread Kernels

This kind of kernels remove the limitation of having one single execution flow inside the kernel. In this way, the kernel can also be preemptable.

In a multithreaded kernel, different execution flows can be used for processing interrupt requests (without the need for making them atomic like the bottom halves). Synchronisation between the various execution flows must be explicitly enforced using a combination of interrupt disabling and busy waiting named *spinlock*. For this reason, the extension to SMP machines is much simpler.

A spinlock provides two operations, *lock* and *unlock*, and acts as a mutex, ensuring the atomicity of sections contained between lock and unlock. The difference respect to mutexes is that a spinlock will not use the process block/unblock mechanism. On a single processor machine, a lock operation is equivalent to disabling interrupts (an x86 `cli` instruction), which will be reenabled by the unlock operation. On a SMP machine, a lock will disable interrupts and, if the spinlock is locked, will perform a busy wait (with a polling cycle) until the spinlock is unlocked.

Examples of multithreaded kernels are Solaris or AtheOS. Note that, due to their internal structure and to the possibility of running interrupt handlers in dedicated thread, multithreaded kernels creates less scheduling anomalies than monolithic kernels in real-time systems.

5.1.5 μkernel systems

The *mu*kernel idea is not new, being born in '70s. The basic concept is to reduce the number of abstractions exported by the kernel to a minimum, implementing in user space the higher level abstractions provided by traditional monolithic kernels. The minimum abstractions that the *u*kernel must provide are address spaces, threads, and some IPC mechanism (channels or ports). All the rest of the OS can be implemented through user level programs.

Using such a μkernel based design, an operating system kernel can be implemented in two possible ways: as a single user process (server), or as a set of cooperating servers. An example of the first approach is the Lite server [Hel94], implementing a BSD style kernel on top of Mach, or the mklinux server [dPSR96], implementing Linux on top of OFS/Mach. An example of the second approach is the GNU Hurd [TB].

In a multi server implementation, kernel functionalities are split in groups implemented by different servers (for example, a EXT2 file system server, a process server, an authentication serve, and so on). This approach can result to be more flexible, and has been recently discovered as more efficient

[GJP+00].

Since a μkernel only implements very simple functionalities, its execution paths will be very short, hence it will not create big anomalies in real-time scheduling. Moreover, device drivers can be implemented externally to the kernel (in dedicated server), so that they do not influence the real-time performance of the system. For this reason there are a lot of real-time systems implemented over μkernels [TNR90, Hil92, HBB+98, Meh99].

Fat μkernels

First generation μkernels, such as Mach and Chorus, were developed using the "traditional kernel" design, with the result of obtaining big kernels (in fact, the "μ" does not mean "small"), often incorporating device drivers (and thus also loosing some real-time properties).

These "fat" μkernels resulted in a less efficient (although more flexible) implementation of the OS functionalities due to various problems like IPC overhead and cache effects.

As a result, a single server implementation of a monolithic kernel running on Mach can incur in a 30% performance penalty. A possible solution are co-located servers that, running in the same μkernel address space, do not incur in the IPC overhead. In this way, one of the biggest advantages of μkernel systems (protection between servers) is lost. Windows NT uses a similar design (NT drivers are in fact co-located servers).

Small μkernels (Second Generation)

To solve the problems encountered in fat μkernels, a second generation of μkernels has been designed. These new μkernels, such as L4 and QNX, only implements the basic needed functionalities, that have been identified in:

1. Threads
2. Address Spaces
3. an IPC mechanism
4. an Interrupt Handling mechanism.

By exporting a minimal interface, that only provides few fundamental services, the kernel size can be minimised so that the whole kernel fits in cache. Moreover, the most critical IPC paths can be optimized by using the CPU registers to pass message data.

The performance improvement obtained by the second generation μkernels is remarkable, and these OSs result to be particularly suited for real-time and embedded systems.

5.1.6 RTLinux-like systems

As seen, the predictability requirements of a real-time system often contrasts with the throughput and flexibility requirements of a general-purpose system. Sometimes, a general-purpose system can be useful for development, and being able to run real-time programs on if can greatly speed-up the development process. However, general-purpose systems are generally based on a monolithic structure.

If real-time performance are not important for the applications running on the monolithic kernel, but are only important for tasks that do not use the general-purpose kernel features, then it could be possible to run the general-purpose kernel *over* a real-time executive that directly accesses the hardware. This requires some kind of *interrupt virtualization mechanism*: interrupt are directly managed by the real-time executive, and are forwarded to the non real-time kernel running over it when appropriate. Instead of disabling interrupts, the non real-time kernel can ask the real-time executive to stop forwarding interrupts, so that hardware interrupts are disabled/reenabled (and managed) only by the real-time executive. In this way, real-time applications get very good real-time performance, and predictable delays and latencies, as proved by RTLinux [BY96], RTAI [MBDP00], and similar systems.

Using this kind of solutions, two different subsystems (an executive or a DOS-like system used by real-time applications, and a monolithic kernel running in background over it) coexist in the same machine, trying to achieve the best of the two worlds. Of course, things can also be seen in the other way around: applications running on the monolithic kernel will get very bad real-time performance and a bad throughput (the non real-time monolithic kernel is scheduled in background), and real-time applications will not be able to access the services provided by the monolithic kernel and will be able to crash or starve the whole system (the real-time executive does not provide any kind of protection).

The second problem (lack of protection in the real-time executive) can be solved by adopting a μkernel structure, and using a high-priority real-time server instead of a real-time executive. In this way, interrupts are not virtualized, but forwarded by the μkernel, and the non real-time server is scheduled in background because of its low priority. This solution has been implemented in L4-RTL [Meh99], achieving real-time performance comparable with the one of RTLinux/RTAI while enforcing protection between real-time applications [MHSH01].

5.2 Scheduling Latency

As explained, real kernels often generate a schedule that is different from the expected one, due to the strategies used to enforce mutual exclusion or the guarantee the consistency of internal data. The difference between the actual schedule produced by the kernel and the ideal schedule can be quantified using a metric called *kernel latency.*

Definition 20 *Let τ be a task belonging to a time-sensitive application that should be ideally scheduled at time t, and let t' be the time at which τ is actually scheduled; the kernel latency experienced by τ is defined as $L = t' - t$.*

According to the previous description, several sources of kernel latency can be identified; the two most important sources being *timer resolution* and *non-preemptive sections* in the kernel. In this section, the kernel latency of a monolithic kernel will be analysed, and some techniques for reducing that latency will be described.

Timer resolution latency occurs because kernel timers are generally implemented using a periodic tick interrupt. For example, consider a periodic task τ that needs to execute every $T \mu s$. Typically, the task will be woken up by a kernel timer that is triggered by the periodic tick interrupt with say, period T^{tick}. Hence, a task that sleeps for an arbitrary amount of time T can experience some *timer resolution* latency L^{timer} if its expected activation time is not on a tick boundary.

Another source of latency, the *non-preemptable section latency* is caused by non-preemptable sections in the kernel or in the drivers. In a monolithic kernel, this component of latency includes the effects of ISRs and bottom halves. Consider an example where interrupts are disabled at time t. Task τ can only enter the ready queue later when interrupts are re-enabled. In addition, even if τ enters the ready queue at the correct time t, it may still not be scheduled if preemption is disabled for some reason. In this case, τ will be scheduled when preemption is re-enabled at time t', contributing a non-preemptable section latency $L^{np} = t' - t$.

5.2.1 Timer Resolution

As said, in a traditional kernel, timers are triggered by a periodic tick interrupt, which on x86 machines is generated by the Programmable Interval Timer (PIT) and has a period $T^{tick} = 10ms$. As a result, the maximum latency due to the timer resolution $\max\{L^{timer}\}$ is $T^{tick} = 10ms$. Thus, this value can be reduced by reducing T^{tick}. However, decreasing T^{tick} increases system overhead because more tick interrupts are generated. In addition, there is a lower bound on L^{timer} which is equal to the execution time required for servicing the tick interrupt.

The fact that a periodic timer interrupt is not an appropriate solution for a real-time kernel is well known in the literature, and thus most of the existing real-time kernels provide *high resolution timers* based on an aperiodic interrupt source[ST93]. In an x86 architecture, the PIT or the CPU APIC (Advanced Programmable Interrupt Controller present in many modern x86 CPUs) can be programmed to generate aperiodic interrupts for this purpose. Thus, high resolution timers could reduce L^{timer} to the interrupt service time without significantly increasing the kernel overhead, because these interrupts are generated only when a timer expires.

5.2.2 Non-Preemptable Sections

The second term contributing to the maximum kernel latency is the non-preemptable section latency $\max\{L^{np}\}$. According to the previous description of the various kernel structures, in a monolithic kernel $\max\{L^{np}\}$ is equal to the maximum length of a system call (which, we recall, is non-preemptable) plus the processing time of all the interrupts that fire before returning to user mode. Unfortunately, in a standard monolithic kernel such a Linux this value can be as large as $28ms$ as shown in Section 5.2.3. In a μkernel system, system calls are still non-preemptable, but they are shorter (because of the simplicity of the μkernel), and the interrupt processing time does not affect L^{np}. This is the reason why some real-time systems such as RT-Mach [TNR90], QNX [Hil92], and DROPS [HBB^{+}98] are based on a μkernel. Multithreaded kernels can also be used to reduce the effect of non-preemptable sections by removing the effect of ISR and bottom halves, but this solution also affects the throughput of the system.

An alternative solution to decrease L^{np} is to modify the monolithic approach by decreasing the size of the kernel non-preemptable sections or by introducing full kernel preemptability. Hence, three new kernel structures have to be considered:

Low-Latency kernel: This approach “corrects” the monolithic structure by inserting explicit preemption points (also called rescheduling points) inside the kernel. In this approach, when a task is executing inside the kernel it can explicitly decide to yield the CPU to some other task. In this way, the size of non-preemptable sections is reduced, thus decreasing L^{np}. In a low-latency kernel, the consistency of kernel data is enforced by using cooperative scheduling (instead of non-preemptive scheduling) when the execution flow enters the kernel. This approach is used by some real-time versions of Linux, such as RED Linux [YCL98], and by Andrew Morton's low-latency patch [Mor]. In a low-latency kernel, $\max\{L^{np}\}$ decreases to the maximum time between two rescheduling points.

Preemptable kernel: The preemptable approach, used in most real-time systems, removes the constraint of a single execution flow inside the kernel. Thus it is not necessary to disable preemption when an execution flow enters the kernel. To support full kernel preemptability, kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptable kernel patch [Lov] uses this approach and makes the kernel fully preemptable. Kernel preemption is disabled only when a spinlock is held.[1] In a preemptable kernel, $\max\{L^{np}\}$ is determined by the maximum amount of time for which a spinlock is held inside the kernel (maximum size of a kernel non-preemptable section), plus the maximum time taken by ISRs and bottom halves.

Preemptable Lock-Breaking kernel: The kernel latency can be high in Preemptable Linux when some spinlock is held for a long time. Lock breaking addresses this problem by "breaking" long spinlocks, i.e., by releasing spinlocks at strategic points. Breaking spinlocks into smaller non-preemptable sections is similar to the approach used by Low-Latency Linux. This approach reduces the size of kernel non-preemptable sections, but, of course, does not decrease the amount of time "stolen" by ISRs and bottom halves. Looking at the code, we verified that most of the Andrew Morton's preemption points are in this patch in the form of "lock breaking points".

As a final note, we would like to point out that the preemption patch has been recently accepted in the development (unstable) branch of the Linux kernel, and is now present in version 2.5.4 of the kernel.

5.2.3 Experimental Evaluation

To show the effects of the kernel structure on the real-time performance, the latency of a standard monolithic kernel, Linux 2.4.18 in particular, have

[1]There is also a different patch, from Timesys [Inc], based on mutexes and priority inheritance instead of on spinlocks.

been evaluated and compared with a low-latency, a preemptable, and a lock-breaking preemptable version of the same kernel. One method for experimentally measuring the latency is to use a task that invokes`usleep` to sleep for a specified amount of time and then measures the time that it actually slept. The latency L, as previously defined, is then the difference between these two times. Unfortunately, this approach measures the sum of all the latency components and thus does not give us an insight into the causes of latency.

The individual latency components can be measured in *isolation*, by measuring each source of latency while eliminating the others. To measure L^{timer}, L^{np} is eliminated by running the experiment on an idle system. After that, L^{np} is measured by eliminating L^{timer} through the use of high resolution timers. The following sections describe this approach in more detail.

Measuring Timer Resolution Latency

The OS non-preemptable section latency L^{np} can be reduced significantly by running experiments on a lightly-loaded system. In this case, few system calls will be invoked and a limited number of interrupts will fire and thus long non-preemptable execution paths or drivers' activations are not likely to be triggered.

The latency L^{timer} can be measured by using a typical periodic time-sensitive application, for example a process that sets up a periodic signal (using the `itimer()` system call) with a period T ranging from $100\mu s$ to $100ms$. The process measures the time when it is woken up by the signal and then immediately returns to sleep after computing the difference between two successive process activations, called *inter-activation time*. Note that in theory the inter-activation times should be equal to the period T. Hence, the deviation of the inter-activation n times from T is a measure of L^{timer}. Since Linux ensures that a timer will never fire before the correct time, this value can be expected to be $10ms$ on standard Linux kernel, and to be close to the interrupt processing time while using high resolution timers.

Measuring OS Non-Preemptable Section Latency

Once the timer resolution latency is eliminated with high resolution timers, L^{np} can be measured in isolation. Unfortunately, a periodic process is not suitable for measuring this latency. For example, to measure the effects of disabling preemption for a time S, the latency must be sampled with a period $T \ll S$ or else the non-preemptive code could execute between two consecutive measurements. More precisely, if $\mathcal{L}$ is the measured latency, then

$\mathcal{L} \leq L^{np} \leq \mathcal{L}+T$. Hence, to reliably measure L^{np}, the test task should have a period T such that $T << L^{np}$. In practice, this requirement is hard to achieve and thus we use an aperiodic test application that uses the `usleep()` call.

The test task is based on a loop that:

1. reads the current time t_1
2. sleeps for a time T
3. reads the time t_2, and computes $L^{np} = t_2 - (t_1 + T)$

Times t_1 and t_2 are read using the Pentium Time Stamp Counter (TSC), a CPU register that is increased at every CPU clock cycle and can be accessed in a few cycles. Hence, the measurements introduce very low overhead and are very accurate.

To investigate how various system activities contribute to L^{np} various load-generating tasks are were run in background. The following tasks are known to invoke long system calls or cause frequent interrupts and thus were selected as background load to trigger long non-preemptable sections:

Memory Stress: One potential way to increase L^{np} involves accessing large amounts of memory so that several page faults are generated in succession. The kernel invokes the page fault handler repeatedly and can thus execute long non-preemptable code sections.

Caps-Lock Stress: A quick inspection of the kernel code reveals that when the num-lock or caps-lock LED is switched, the keyboard driver sends a command to the keyboard controller and then spins while waiting for an acknowledgement interrupt. This process can potentially disable preemption for a long time.

Console-Switch Stress: The console driver code also seems to contain long non-preemptable paths that are triggered when switching virtual consoles.

I/O Stress: When the kernel or the drivers have to transfer chunks of data, they generally move this data inside non-preemptable sections. Hence, system calls that move large amounts of data from user space to kernel space (and vice-versa) and from kernel memory to a hardware peripheral, such as the disk, can cause large latencies.

Procfs Stress: Other potential latency problems in Linux are caused by the `/proc` file system. The `/proc` file system is a pseudo file system used by Linux to share data between the kernel and user programs. Concurrent

$T(\mu s)$	100	200	300	400	500	600	700	800	900
$L(\mu s)$	47	51	43	44	49	53	50	52	50
$T(\mu s)$	1000	2000	3000	4000	5000	6000	7000	8000	9000
$L(\mu s)$	46	47	52	48	51	49	55	50	57
$T(\mu s)$	10000	20000	30000	40000	50000	60000	70000	80000	90000
$L(\mu s)$	52	46	51	49	54	50	43	47	51

Table 5.1: The table shows L, the maximum difference between the inter-activation times and the task period, for different values of the task period T on a high resolution timer Linux.

accesses to the shared data structures in the `proc` file system must be protected by non-preemptable sections. Hence, we expect that reading large amounts of data from the `/proc` file system can increase the latency.

Fork Stress: The `fork()` system call can generate high latencies for two reasons. First, the new process is created inside a non-preemptable section and involves copying large amounts of data including page tables. Second, the overhead of the scheduler increases with increasing number of active processes in the system.

Experience and careful code analysis by various members of the Linux community (for example, see Senoner [Sen]) confirms that the above list of latency sources is comprehensive, i.e., it triggers a representative subset of long non-preemptable sections in the kernel and in the drivers.

Results

The first set of experiments measures L^{timer} and shows that it can be easily eliminated from the OS non-preemptable section latency by using high resolution timers. The high-resolution timers mechanism was evaluated and compared with the timer mechanism of a standard kernel. Figure 5.1 shows the inter-activation times on a standard Linux kernel when $T = 11ms$. Since the task period is not a multiple of T^{tick} , the difference between the inter-activation times and T is not 0: the timer will fire at the next multiple of the system tick and thus an inter-activation time is $20ms$. In fact, the inter-activation times in Figure 5.1 is close to this value, and the difference between the inter-activation times and the period is close to $20 - 11 = 9ms$. As explained, this problem is solved by the high-resolution timer kernel, which we demonstrate through experiments described below.

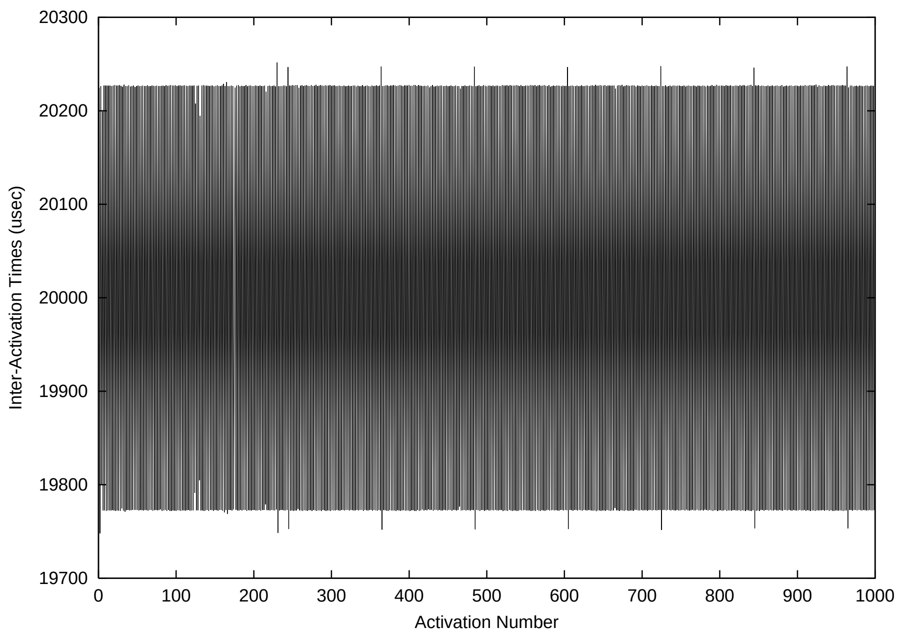

Figure 5.1: Inter-Activation times for a task that is woken up by a periodic signal with period $11ms$ on a standard Linux kernel. Note that the task period is greater than $T^{tick} = 10ms$.

Figure 5.2 shows the inter-activation times measured with period $T = 100\mu s$ on the high-resolution timer kernel. Note that after 1000 activations the maximum difference between the period and the actual inter-activation time is less that $25\mu s$. Hence, it can be conjectured that the $9ms$ latency shown in Figure 5.1 is almost completely due to the timer resolution.

Table 5.1 shows the maximum absolute value of the difference between the period and the inter-activation times for various values of T on a high resolution timer kernel. Each of these maximum values has been measured over $1,000,000$ activations. The table shows that the maximum difference does not significantly depend on the period T and its maximum value is about $57\mu s$. We hypothesise that this value is due to the OS latency L^{np}. However, we do not know the precise cause of this latency since we did not specifically control the background task set.

This experiment has been repeated with different periods where each experiment was run for $10,000,000$ activations, showing that the difference between the period and the inter-activation time does not significantly depend on the period T. Figure 5.3 plots the Probability Distribution Function (PDF) of the inter-activation times when $T = 1000\mu s$. The maximum measured inter-activation time is about $1300\mu s$, whereas the minimum is about

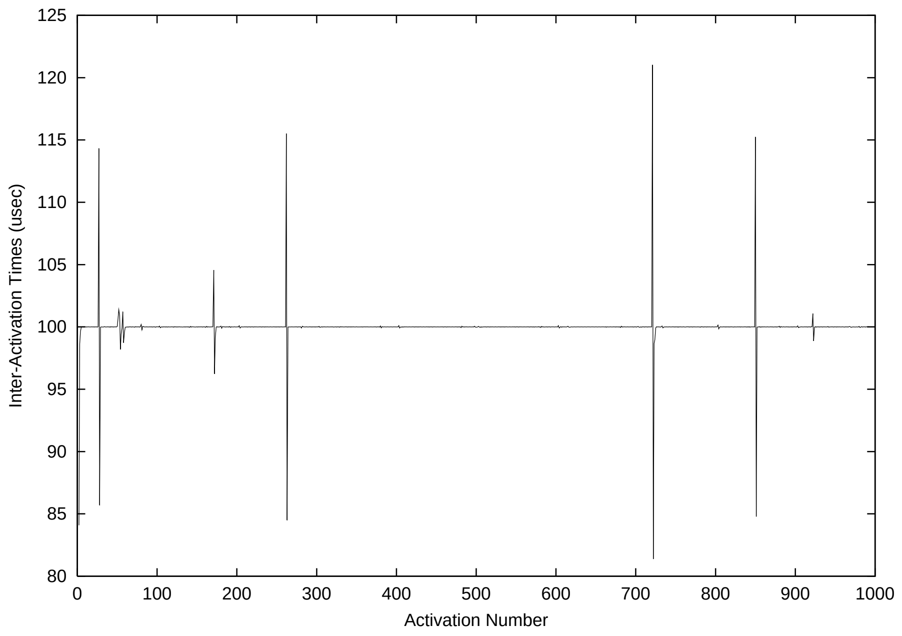

Figure 5.2: Inter-Activation times for a task that is woken up by a periodic signal with period $100\mu s$ on a high resolution timer Linux.

$630\mu s$, and this distribution does not significantly vary with increasing number of activations.

The maximum deviation between inter-activation times (about $370\mu s$) is due to the OS non-preemptable section latency L^{np}. However, the precise cause of this latency is not precisely known, since in the previous experiments there was not any specific control on the background task set.

5.2.4 Non-Preemptable Section Latency

Hence, a new set of experiments was performed to measure latencies due to the various activities that can trigger long non-preemptable paths. In this set of experiments, the `usleep()` test program described in Section 5.2.3 was run with $T = 100\mu s$ to measure and identify the causes of the non-preemptable section latency.

The `usleep()` test program started on an unloaded machine. Then the load-generating tasks described in Section 5.2.3 were run in the background to trigger long non-preemptable paths. To easily represent the latency results in a single plot per Linux variant, we used a background load that was generated as follows:

1. The memory stress test allocates a large integer array with a total size

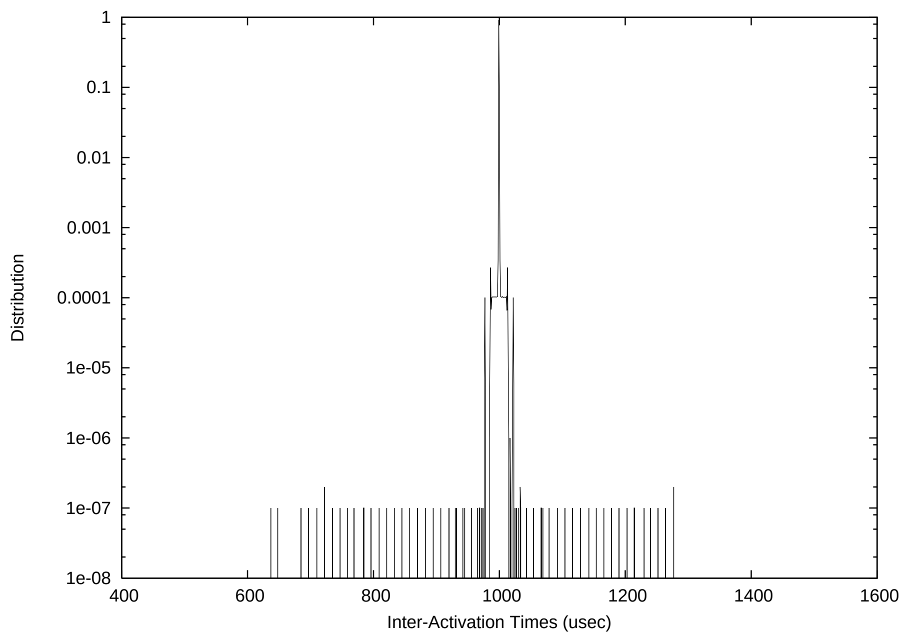

Figure 5.3: PDF of the difference between inter-activation times and period, when $T = 1000\mu s$.

of 128 MB and accesses it sequentially. This test starts at $1000ms$, and finishes around $2000ms$.

2. The caps-lock stress test runs a program that switches the caps-lock LED twice. This test turns on the LED at $7000ms$ and then turns it off at $8000ms$.

3. The console-switch stress test runs a program that switches virtual consoles on Linux twice, first at $9000ms$ and then at $10000ms$.

4. The I/O stress test uses the `read()` and `write()` system calls and accesses 2 MB of data. This test starts at $11000ms$ and finishes around $13000ms$.

5. The procfs stress test reads a 512 MB file in the `/proc` file system. It runs from $17000ms$ to around $18000ms$.

6. The fork test forks 512 processes. This test starts at $20000ms$.

Figure 5.4 shows the latency measured on a standard (monolithic) Linux kernel (version 2.4.16). Due to the implementation of the `usleep()` call on Linux, L^{timer} is around $19.9ms$ instead of $9.9ms$. The memory access

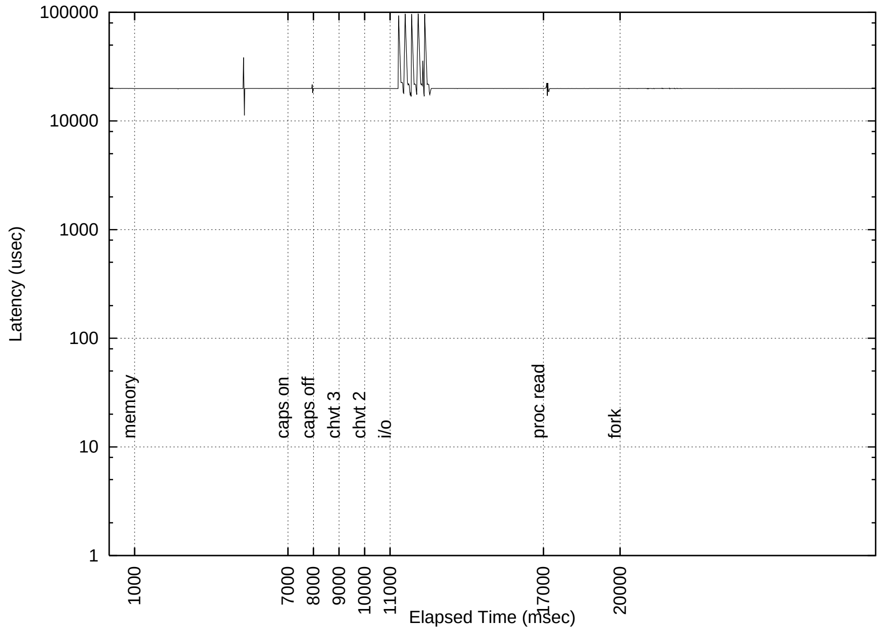

Figure 5.4: Latency measured on a standard Linux kernel. This test is performed with background load. Note that the L^{timer} component dominates the latency most of the time.

test, starting at $t = 1000ms$ does not seem to create any additional latency. However, it is possible to notice a small spike at the end of the test around $t = 5000ms$ (explained in the next experiment). In this experiment, no variation in the latency during the caps-lock stress test or the console-switch test can be noticed. On the other hand, there are some large spikes (up to $100ms$) from $t = 11000ms$ to $t = 13000ms$ during the the I/O stress test. Note that the Y axis is shown on a logarithmic scale. None of the other tests present any significant contribution to kernel latency. Hence, it can be argued that in a standard Linux kernel the timer resolution latency L^{timer} is generally larger than L^{np} and hides the effects of non-preemptable sections. This is probably one reason why latency problems have not been previously addressed by the Linux community. These results show that high resolution timers mechanism is needed to investigate L^{np}.

Figure 5.5 reports the results obtained when high resolution timers are used in the `usleep()` implementation. It shows that in this case L^{timer} is almost completely removed. Hence, the effects of long non-preemptable sections are more visible. For instance, when the system is unloaded ($t < 1000ms$) the latency lies between $4\mu s$ and $6\mu s$. This latency is due to the resolution of the timing mechanism and it matches the expected value of the

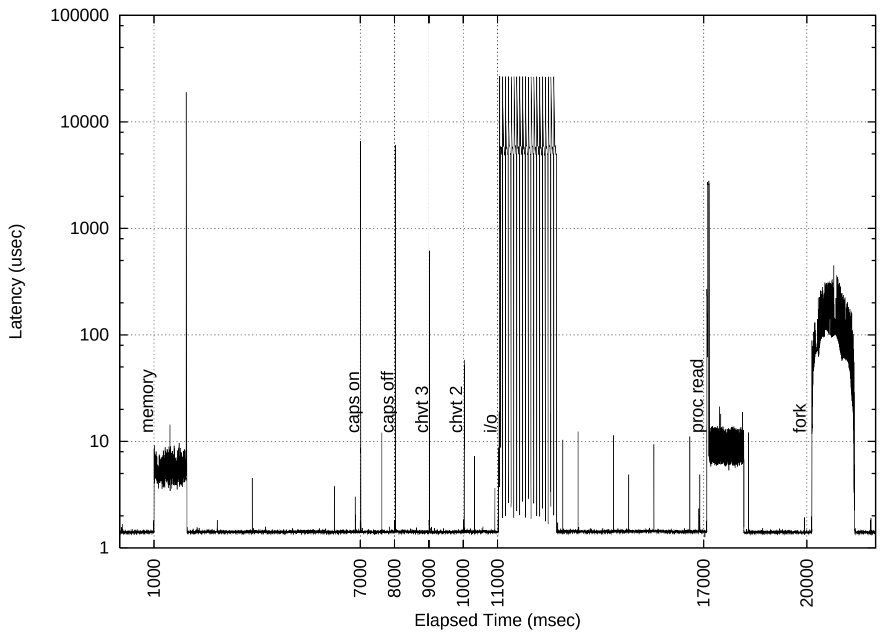

Figure 5.5: Latency measured on a Linux kernel with high resolution timers. This test is performed with background load. Now, L^{np} is visible.

interrupt service time. It increases to $20\mu s$ during the memory stress test. This result is surprising because contrary to common belief it shows that page faults of other processes in Linux are not a serious problem for real-time performance. However, the end of the memory stress test generates a spike of about $20ms$ in kernel latency. A deeper investigation permits to discover that the source of this latency is the `munmap()` system call when large memory buffers are unmapped.

The caps-lock shift significantly increases kernel latency. During the caps-lock stress test ($t = 7000ms$ and $t = 8000ms$) the latency rises to $7ms$. On the other hand, the console switch test ($t = 9000ms$ and $t = 10000ms$) only increases the latency to $900\mu s$. Again, the longest critical paths seem to be triggered by the I/O stress test between $t = 11000ms$ and $t = 13000ms$ when the latency increases to $100ms$, similar to the previous experiment. Finally, the procfs stress test can contribute about $4ms$ to latency, whereas the fork test contributes up to about $300\mu s$. Again, note that in a standard Linux kernel, the $10ms$ resolution of the timers hides most of these values except the latency caused by file accesses.

From Figure 5.5, a expect reduction in the latency is expected if the length or granularity of the kernel non-preemptable sections is reduced. As explained in Section 5.2.2, there are several ways in which non-preemptable

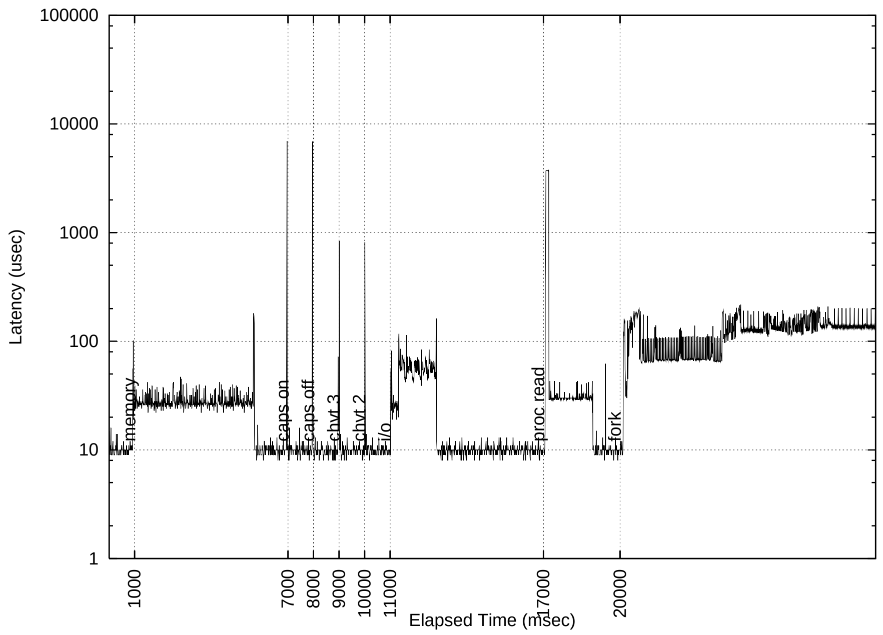

Figure 5.6: Latency measured on a Low-Latency Linux kernel with high resolution timers. The `munmap()` and I/O latencies are reduced.

kernel sections can be shortened. First, preemption points can be manually placed to break long non-preemptable paths, such as in the low-latency kernel. Second, the kernel can be made fully preemptable, where preemption is disabled only when spinlocks are held. Finally, the first technique can be used to reduce the length of spinlocks in a preemptable kernel. These techniques are explored in the next sections.

Figure 5.6 shows the latency measured on a high resolution timers kernel with the Andrew Morton low-latency patch.

First, note that the latency experienced during the memory stress test does not change significantly, but the $20ms$ spike caused by unmapping the large memory buffer has been removed. Now the `munmap()` latency is about $200\mu s$. However, the latency caused by the caps-lock and console stress tests is not changed, and in this experiment the worst latency is caused by toggling the caps-lock key! The latency spikes between $t = 11000ms$ and $t = 13000ms$ have disappeared and thus the I/O stress test does not cause serious problems for real-time performance anymore. However, the latency caused by the procfs stress test and by the fork stress test is unchanged as compared to the monolithic kernel.

In summary, the latency caused by all the activities except the procfs stress test and the caps-lock stress test is under $1ms$.

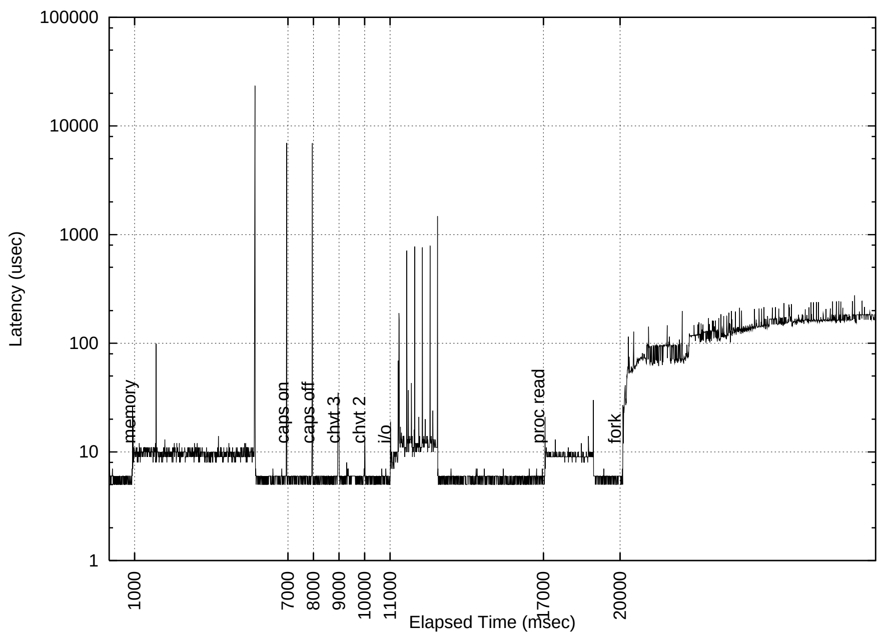

Figure 5.7: Latency measured on a Preemptable-Linux kernel with high resolution timers. The `procfs` latency is reduced, but the `munmap()` latency becomes high again.

Figure 5.7 shows the results obtained using a Preemptable-Linux kernel. The big difference that can be noticed as compared to the Low-Latency kernel is that the `munmap()` system call causes high latency once again (about $20ms$ around time $t = 5000ms$). The latency caused by the I/O stress test is also increased with spikes up to $1ms$. On the other hand, the procfs stress test does not cause significant latency. In particular, the big spike in latency at time $t = 17000ms$ has been removed. In this experiment, the worst latency is caused by the `munmap()` system call and is due to the kernel holding a spinlock for a long time.

Figure 5.8 shows the results obtained when the lock-breaking preemptable kernel is used. Note that breaking long spinlocks solves the `munmap()` problem. The kernel behaviour during the memory stress (and during the final `unmap()`) is similar to the behaviour of the low-latency kernel. Moreover, this kernel also has the benefits of the preemptive kernel. For instance, compared to the low-latency kernel, there are improvements in the latency caused by the console switch stress test and by the procfs stress test.

In summary, the largest latency is caused by the caps-lock stress test and all other latencies are within $1ms$. File accesses are still not as low as in Figure 5.6. This latency is caused by heavy interrupt loads and long

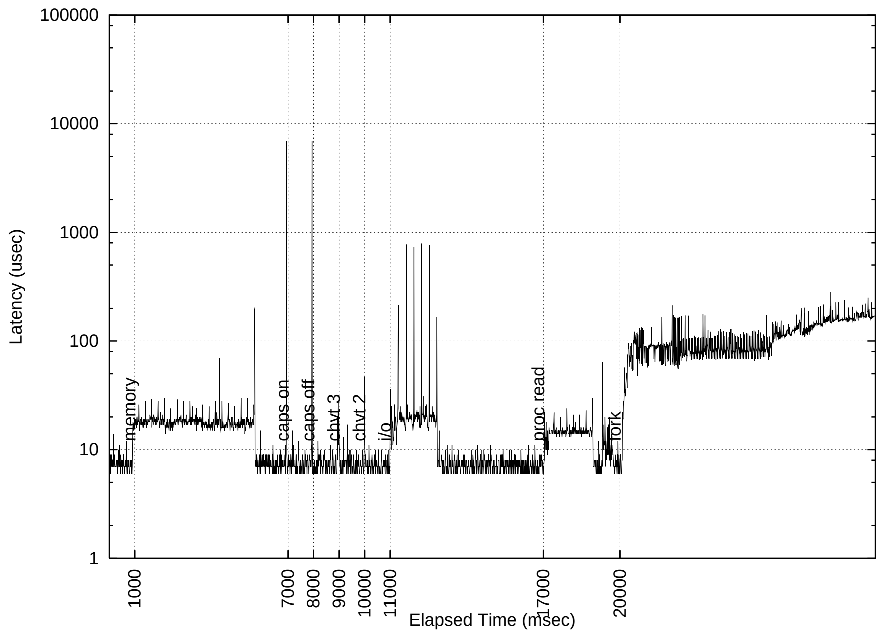

Figure 5.8: Latency measured on a Lock-Breaking Preemptable-Linux kernel with high resolution timers. Note that most of the latencies are under $1ms$.

non-preemptable interrupt processing times inside BHs. In fact, BHs are serialised using a spinlock, that can disable preemption for a long time.

5.3 Interrupt Processing Time

Until this point, the CPU as been considered as the only hardware resource in the system (hence, as the only resource that has to be scheduled). However, a modern PC is connected to a lot of peripherals, that can be considered as hardware resources that the OS kernel has to manage. In most case, these resources can produce events (in the form of hardware interrupts), and the kernel properly manages them, or forwards them to an appropriate handler task. In a μkernel based system, an hardware interrupt can be converted in an IPC to a server task, that will properly handle the hardware device; in a multithreaded kernel, a kernel thread can be used to properly serve the interrupt, whereas in a monolithic kernel a bottom half (or a DPC) is generally used to this task.

5.3.1 The Problem

Independently from the kernel structure, a hardware interrupt will be generally served in two phases:

- a short **Interrupt Service Routine (ISR)** generally executes with interrupts disabled, and is responsible for acknowledging the hardware interrupt mechanism and activating a proper DPC, bottom half, kernel thread, or server task.
- a longer routine (running in a kernel thread, server task, bottom half, or DPC) is responsible for correctly manage the device. Note that kernel threads and server tasks are generally scheduled, whereas DPCs and bottom halves are not.

As noted above, if a μkernel or a multithreaded kernel is used, the code handling the device can be scheduled like all the other tasks in the system. This solution can present a slightly higher overhead, and requires a more careful synchronisation, but permits to correctly account the handler code in a time sensitive system. In facts, the handler code requires some CPU time to execute, and it must be correctly accounted in order not to break the system's guarantees.

To better understand this fact, let's consider a monolithic kernel: as explained before, the handler code runs in a bottom half, that is invoked by the CPU scheduler before selecting the next application task and is not preemptable with respect to the application tasks. This fact can introduce two sources of unpredictability:

- the handler code is execute at apparently random times (depending on the interrupts' arrival pattern) and *is not scheduled*, introducing anomalies in the CPU scheduling that can be seen as *stolen time*
- bottom halves are not preemptable, violating one of the assumptions of a priority based scheduler (at each time, the task having the higher priority is scheduled).

As a result of these scheduling anomalies, the real-time guarantees provided by the system may be broken. From a practical point of view, the system behaves like if some execution time has been stolen to application tasks, hence this problem will be referred as the *stolen time* problem.

Some solutions to the stolen time problem have been proposed, ranging from accounting the interrupt and bottom half time in the schedulability guarantee [JS93] to scheduling the bottom half code [JSMA98, DB96] or temporally disabling the hardware interrupts [MR97, IMS97]. However, none of those solution can be easily and practically implemented in an usable OS kernel.

In order to show the impact of this problem, some experiments have been performed using Linux/RK. The version of Linux/RK used for these experiments provides predictable guarantees for CPU reservations, outgoing network reservations, and disk reservations, but does not account properly the time stolen by network bottom halves.

5.3.2 A Possible Way Out

If a reservation based scheduler is used, another possible solution to the stolen time problem could be to use the *augmented reservation* abstraction [RS01], that permits to resize the system reservations in order to compensate the effects of the stolen time.

The augmented reservations approach results to be very effective and easy to implement, but it requires to monitor the time stolen by DPCs or bottom halves (hence, it requires additional modifications to the OS kernel), in order to sum it to the reserved time. This requirement is due to the fact that augmented reservations have been designed to support a generic task model; if, on the other hand, the real-time task model is used, then the requirement of instrumenting the kernel can be relaxed. In fact, using the real-time task model each task is divided in jobs, and each job is characterised by an absolute deadline that can be used to monitor application performance. In this way, an implicit monitoring of the effects of the interrupt handlers execution can performed by simply measuring the number of missed deadlines, and the DPC or bottom half time does not have to be explicitly monitored.

This idea is used by *Adaptive Reservations*, presented in Section 4.1: the adaptive reservation abstraction was originally developed in order to cope with tasks characterised by unknown or highly variable execution times, but it can be successfully used to mitigate the effects of stolen time. In fact, the time stolen by ISRs and bottom halves can be modelled as a variance in tasks' execution times, and adaptive reservations will properly cope with it.

This is a simple explanation of how adaptive reservations compensate the effect of the time stolen by interrupt processing: when the network load increases, the bottom halves begin to consume a significant amount of CPU time, stealing it to reserved processes. Hence, a reserved process will miss some deadline, and if the process is attached to an adaptive reservation its reserved time will be increased. In this way, the amount of time reserved to a process increases when the network traffic increases, compensating the effects of the bottom halves execution.

```
init = rdtsc();
for (i = 0; i < MAX; i++) {
    for (j = 0; j < COUNT; j++) {
        for (k = 0; k < 100; k++) {
            /* Just to spend some time...  */
            time = rdtsc();
        }
    }
    timevect[i] = CLOCK2USEC(time - init);
    task_endcycle(); /* Blocks until the next period */
}
```

Figure 5.9: The test process

Note that, in contrast with augmented CPU reservations, adaptive reservations can be implemented in user space, without requiring modifications to the kernel. The only requirement is that the kernel provides temporal protection in the CPU scheduler; as a proof of concept, adaptive reservations have been implemented through a portable QoS Manager, that has been ported on the HARTIK kernel [AB00] and Linux/RK [RAdN$^+$00] (as already explained in Section 4.3); in this work, the RK version has been used.

In order to prove the effectiveness of Adaptive Reservations in compensating the effects of the stolen time, some experiments have been run in Linux/RK, a Resource Kernel based on Linux. A Resource Kernel in general permits to reserve an hardware resource to a process: based on some *reserve* abstractions, a process can be guaranteed to receive the resource for a time Q each period T. The version of Linux/RK used for these experiments provides predictable guarantees for CPU reservations, outgoing network reservations, and disk reservations, but does not properly account the time stolen by the bottom halves.

The influence of the bottom halves on the CPU scheduling can be easily seen by simply causing a lot of bottom halves execution and measuring the impact on the execution of a reserved process, as shown by the following experiments. First of all, the periodic process shown in Figure 5.9 has been run with period $T = 20ms$ on an AMD-K6 at 333 MHz, attached to a proper CPU reservation. Since the `COUNT` constant is tuned so that the `j` loop takes about $4ms$, when attached to a $(4ms, 20ms)$ reservation this process does not miss any deadline. In fact, the difference between two consecutive values

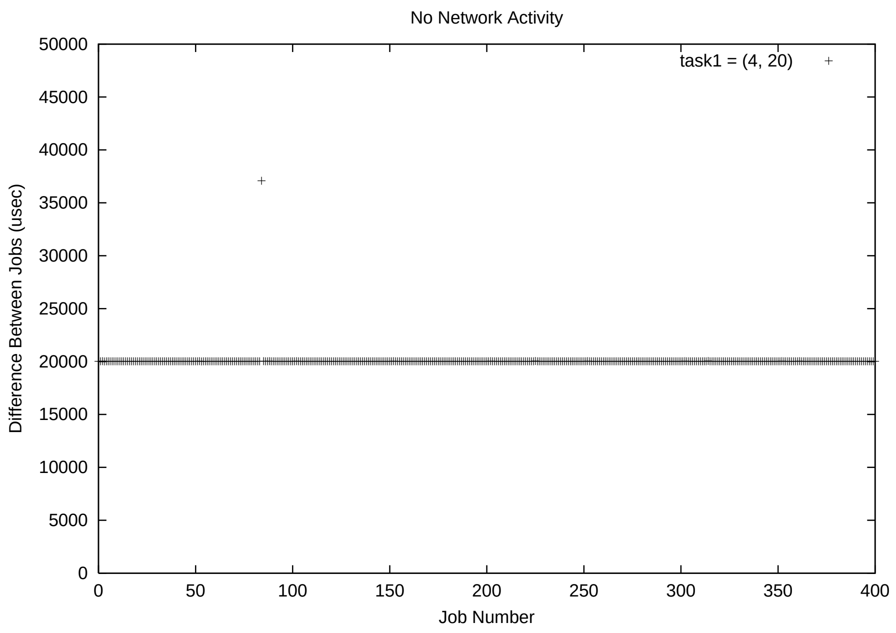

Figure 5.10: Reserved process running in regular network load conditions: job inter finishing times.

of `timevect` (referred as *job inter finishing time* in this dissertation) is about constant, and equal to $20ms$ (the process & reserve period), as shown in Figure 5.10. Hence, as forecasted, all the jobs finishes within their deadlines (in facts, Linux/RK is able to provide a reliable real-time or reservation guarantee if bottom halves do not steal too much time). After that, a heavy network traffic has been sent to the test machine, in order to increase the CPU time consumed by bottom halves. When the network load is increased, the bottom halves steal execution time to the reserved process, hence the difference between two consecutive values of the `timevect` array increases, and the process starts to miss deadlines, as shown in Figure 5.11.

Adaptive reservations can nicely solve this problem: in order to prove the effectiveness of such a solution, the previous experiment has been repeated attaching an adaptive reservation with period $20ms$ to the user process. As a result, the process parameters were adapted so that the number of missed deadlines resulted to be controlled to 0 after a short transient. In fact, the difference between two consecutive values of `timevect` resulted to be controlled below $40ms$, as shown in Figure 5.12.

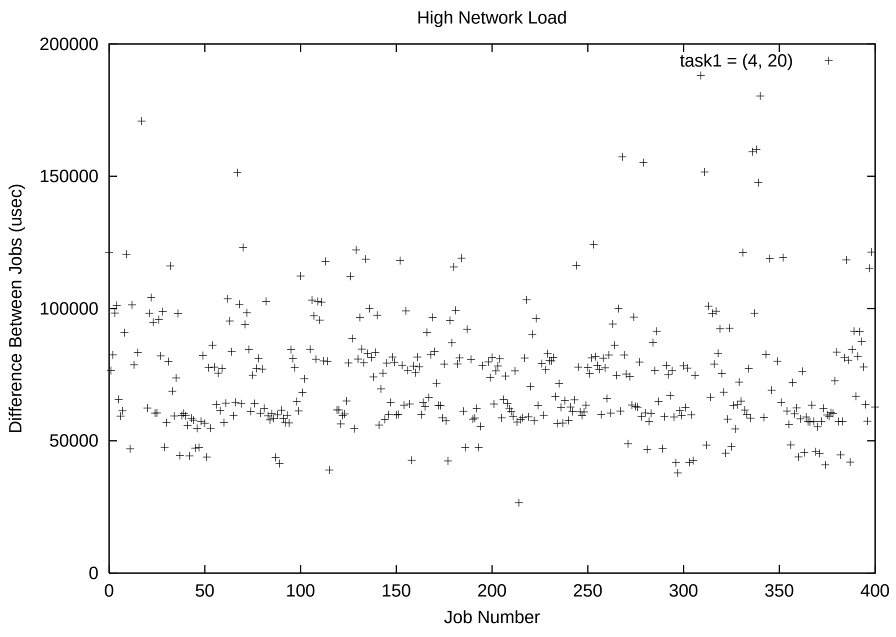

Figure 5.11: Reserved process running in high network load conditions: job inter finishing times — the process misses deadlines.

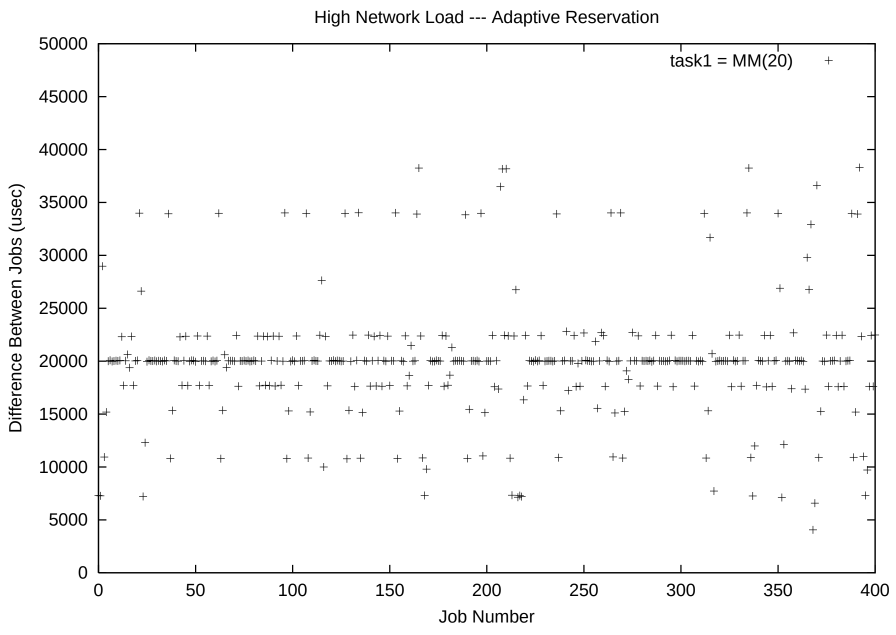

Figure 5.12: Adaptive Reservations in high network load conditions: job inter finishing times — the number of missed deadlines is controlled.

Chapter 6

Conclusions

I've seen things you people wouldn't believe.
Attack ships on fire off the shoulder of Orion.
I watched C-beams glitter in the dark near the Tannhauser gate.
All those moments will be lost in time, like tears in rain.
Time to die.
Blade Runner

This dissertation showed how to support time sensitive activities in a general purpose operating system. In particular, it was argued that the use of appropriate kernel techniques enables advanced scheduling and resource allocation to better exploit system resources and to provide more predictable QoS for time sensitive applications.

The thesis supported in this dissertation is that three different requirements can be identified:

1. *low kernel latencies* and *high-resolution timers* are needed to implement a correct and accurate scheduler;

2. *temporal protection* must be provided by the scheduler so that a precise and effective resource allocation can be implemented;

3. *dynamic adaptation* of the amount of reserved resources is needed to cope with varying and unpredictable workloads.

6.1 OS Support

An evaluation of the latencies of a general-purpose kernel such as Linux showed that the traditional monolithic design on which traditional OSs are based can introduce big errors in resource allocation. This is due to various factors, such as:

- the non-preemptive sections used by the kernel to ensure the consistency of internal structures;
- the low temporal resolution provided by traditional kernel timers, based on a periodic interrupt source;
- the inaccurate resource accounting provided by traditional OSs;
- the anomalies produced by interrupt service.

The kernel latency can be reduced by using preemptable kernels, by introducing preemption points in the kernel, and by using high-resolution timers, based on an aperiodic interrupt source. When these solutions are used to reduce the latency, the scheduler can be precise enough to properly allocate system resources so that each application can achieve the desired QoS.

To prove that accurate scheduling is possible on Linux, the influence of the kernel latency on the scheduler accuracy has been measured through an extensive set of experiments. Then, the kernel latency has been accurately analysed and evaluated, showing the effectiveness of kernel preemption in reducing it.

6.2 Scheduling

Once the kernel provide low latencies, the scheduling algorithm becomes important. However, the fixed priority algorithm implemented in the standard Linux scheduler is not suitable for scheduling generic time-sensitive activities, because it does not provide *Temporal Isolation*: temporal isolation (also known as temporal protection) is important for ensuring that the temporal behaviour of a task does not affect the schedulability of the other tasks in the system.

In other words, the isolation property is necessary to protect applications from the misbehaviours of the other applications: the net effect is that each application executes as it were on a slower dedicated processor. The scheduling technique chosen in this paper to provide temporal isolation is based on resource reservation techniques, hence an efficient and effective scheduling algorithm implementing resource reservations has been proposed. The service mechanism proposed in this dissertation is the Constant Bandwidth Server (CBS), a work conserving server (implementing soft reservations) that has been inspired by the Total Bandwidth Server and by the Dynamic Sporadic Server.

Together with temporal isolation, the CBS provides some other interesting properties, such as reclaiming of unused time, some kind of fairness, hard schedulability for tasks with known parameters, and the possibility to perform a probabilistic guarantee for soft real-time tasks.

6.3 Adaptive Resource Allocation

Resource reservations can be used to implement an adaptive mechanism which reserves the correct amount of resources to each task. This feedback mechanism, which can dynamically adapt the reservation parameters, is particularly useful to achieve the desired QoS when some tasks parameters (such as the WCET) are not known in advance.

The adaptive reservation abstraction, obtained combining the reservation and the feedback mechanisms, uses a *control function* $f()$ to compute the amount of CPU time reserved to a task based on its *scheduling error*. Control theory can be used to design the feedback function, and to prove that the closed loop scheduler is stable (meaning that it is able to control the scheduling error to a desired value), and can provide the desired QoS.

After introducing a formal definition of the adaptive reservations mechanism, an accurate formal model of a reservation-based scheduler was developed and presented. Based on this model, control theory has been used to develop a feedback function and the performance of the closed-loop system has been evaluated. According to our model and to the control theoretical analysis that we performed, a simple PI controller resulted to be the correct choice for controlling the amount of time reserved to a task.

The proposed feedback scheme has been initially implemented by using a simulator and a synthetic workload, then by using a realistic workload obtained by profiling an MPEG player. After that, adaptive reservations have been implemented on a real system (using Linux/RK), and the effectiveness of the proposed scheme has been validated by performing experiments on a real system.

6.4 Final Remarks

Well, this is the end of the dissertation. I hope you all enjoyed reading it. I also hope that the contents of this dissertation will be useful for someone, and could help the development of future research.

If you have comments, ideas, or questions about this dissertation and the algorithms presented in it, feel free to write me at `lucabe72@gmail.com`

Luca

Appendix A

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF,

A.4 Modifications

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History";

likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections

Appendix B

Miscellaneous

Ok, since Appendix titles go into the index, I could not put the correct title here, but this is the recipe that I promised to a lot of people: “Spaghetti al Pomodoro”.

To prepare good spaghetti al pomodoro, you will need:

- about 1/2 Kg of spaghetti (translation to lb is left as a simple exercise for the reader). Pasta by “Barilla” can be easily found even in the US, and is fairly good, hence I suggest it.
- 1 can of diced tomatoes
- a small onion
- oil
- salt, pepper, oregano, and similar stuff

First of all, put about 3 litres of water in a pot, and put it on the stove. When the water boils, add some salt and the spaghetti. At the same time, put some oil in a pan, together with the onion cut in small pieces. Cook it for four/five minutes, and then add the tomatoes. Add salt, pepper, oregano, red pepper, and whatever else you like, according to your preference.

After about 8 minutes that spaghetti are cooking in the boiling water, remove them from the pot, and put them in the pan containing the tomatoes. Also add 3 or 4 table spoons of the cooking water. Finish to cook the pasta for about 4 minutes, and serve. Enjoy!!!

Remember, if you miss this deadline and you cook the spaghetti too much, they will result to be overcooked, and will not be good. However, the criticality of this deadline depends on the Quality of the Pasta (QoP?). If you use good-quality spaghetti (such as Barilla) you can have a 1 or 2 minutes tolerance on the deadline, otherwise the deadline is hard!!! (ok, this is just to maintain the appendix on-topic).

Bibliography

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[AB99] Luca Abeni and Giorgio Buttazzo. Qos guarantee using probabilistic dealines. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, York, England, June 1999.

[AB00] Luca Abeni and Giorgio Buttazzo. Support for dynamic QoS in the HARTIK kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

[AB01] Luca Abeni and Giorgio Buttazzo. Hierarchical qos management for time sensitive applications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.

[Abe98] Luca Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[ABRT93] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[But93] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1993.

[BY96] Michael Barabanov and Victor Yodaiken. Real-time linux. *Linux Journal*, March 1996.

[CT94] Charles L. Compton and David L. Tennenhouse. Collaborative load shedding for media-based applications. In *Proceedings of the International Conference on Multimedia Computing and Systems*, 1994.

[DB96] Peter Druschel and Gaurav Banga. Lazy reciver processing (LRP): A network subsystem architecture for server systems. In *Proceding of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, Oct 1996.

[DL97] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[DLS97] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems*, 1997.

[dPSR96] F. B. des Places, N. Stephen, and F. D. Reynolds. Linux on the osf mach3 microkernel. In *Proceedings of the Conference on Freely Distributable Software*, Boston, MA, February 1996.

[fre] The freedos project. http://www.freedos.org/.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems developmet. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[GGV96] Pawan Goyal, Xingang Guo, and Harrik M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of the 2nd OSDI Symposium*, October 1996.

[GJP$^+$00] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In *Proceedings of the 9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[Goe02] Ashvin Goel. A chickenitarian operation. Private Communication, May 2002.

[HBB$^+$98] H. Hartig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schonberg, and

J. Wolter. DROPS - os support for distributed multimedia applications. In *Proceedings of the Eigth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Hel94] J. Helander. Unix under mach: The lites server. Master's thesis, Helsinki University of Technology, 1994.

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, April 1992.

[IMS97] A. Indiresan, A. Mehra, and K. G. Shin. Receive livelock elimination via dynamic interrupt rate control. Technical report, University of Michigan, June 1997.

[Inc] TimeSys Inc. Timesys linux. http://www.timesys.com.

[JB95] K. Jeffay and D. Bennet. A rate-based execution abstraction for multimedia computing. In *Proceedings of Network and Operating System Support for Digital Audio and Video*, 1995.

[JS93] Kevin Jeffay and Donald L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *IEEE Real Time System Symposium*, pages 212–221, 1993.

[JSMA98] Kevin Jeffay, F.D. SMith, A. Moorthy, and J.H. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[LL73] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[LLB$^+$97] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Proceedings of the IEEE Conference on Real-Time Computing Systems and Applications*, October 1997.

[Lov] Robert Love. The linux kernel preemption project. http://kpreempt.sourceforge.net/.

[LSA$^+$00] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21th IEEE Real-Time Systems Symposium*, Orlando, FL, December 2000.

[LSTS99] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.

[Luc99] George Lucas. Star wars episode I: The phantom menace, May 1999.

[MBDP00] P. Mantegazza, E. Bianchi, L. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 72, 2000.

[Meh99] F. Mehnert. L4rtl - porting rtlinux api to l4/fiasco. In *Proceedings of the Workshop on Common Microkernel System Platforms*, Kiawah Island, December 1999.

[MHSH01] F. Mehnert, M. Hohmuth, S. Schonberg, and H. Hartig. Rtlinux with address spaces. In *Proceedings of the 3rd Real-Time Linux Workshop*, Milano, Italy, November 2001.

[Mor] Andrew Morton. Linux scheduling latency. http://www.zip.com.au/ akpm/linux/schedlat.html.

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3), August 1997.

[MRT93] Clifford W. Mercer, Raguanathan Rajkumar, and Hideyuki Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[PG93] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[PG94] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in intergrated services networks: the multiple node case. *IEEE/ACM Transanctions on Networking*, 2:137–150, April 1994.

[RAdN+00] Ragunathan (Raj) Rajkumar, Luca Abeni, Dionisio de Niz, Sourav Ghosh, Akihiko Miyoshi, and Saowanee Saewong. Recent developments with linux/rk. In *Proceedings of the Second Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[Re97] Dickson Reed and Robin Fairbairns (eds.). Nemesis, the kernel – overview, May 1997.

[RS01] John Regehr and John A. Stankovic. Augmented CPU Reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.

[rte] The rtems real-time open-source operating system. http://www.rtems.com/RTEMS/rtems.html.

[SAWJ+96] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[SAWJ97] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Proceedings of the SPIE Conference on Multimedia Computing and Networking*, volume 3020, pages 207–214, San Jose, CA, February 1997.

[Sen] Benno Senoner. Audio latency benchmark. http://www.gardena.net/benno/linux/audio/.

[Sla64] J. Slaughter. Quantization errors in digital control systems. *IEEE Transactions on Automatic Control*, 1964.

[ST93] S. Savage and H. Tokuda. Rt-mach timers: Exporting time to the user. In *In Proceedings of USENIX 3rd Mach Symposium*, April 1993.

[TB] BSG Thomas Bushnell. Towards a new strategy of os design. http://www.gnu.org/software/hurd/hurd-paper.html.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *USENIX Mach Workshop*, pages 73–82, October 1990.

[YCL98] Yu-Chung and Kwei-Jay Lin. Enhancing the Real-Time Capability of the Linux Kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.