Adaptivity & Control of Resources in Embedded Systems

D4a

COOPERATION

Resource reservation scheme evaluation

Responsible: **Scuola Superiore Sant'Anna** (SSSA)

Marko Bertogna (SSSA)

Project Acronym: **ACTORS**
Project full title: **Adaptivity and Control of Resources in Embedded Systems**
Proposal/Contract no: **ICT-216586**
Project Document Number: **D4a 0.1**
Project Document Date: **2008-12-17**
Workpackage Contributing to the Project Document: **WP4**
Deliverable Type and Security: **R-PU**

Contents

Chapter 1

Introduction

The design and implementation of *open* real-time environments [1] is currently one of the more active research areas in the discipline of real-time computing. Such open environments aim to offer support for real-time *multiprogramming*: they permit multiple independently developed and validated real-time applications to execute concurrently upon a shared platform. That is, if an application is validated to meet its timing constraints when executing in isolation, then an open environment that accepts (or *admits*, through a process of admission control) this application guarantees that it will continue to meet its timing constraints upon the shared platform.

To guarantee temporal isolation among various components, proper bandwidth reservation mechanisms need to be adopted. In the past, many different Resource Reservation schemes have been designed. In this report, we will analyze the existing techniques for the implementation of reservations upon single and multiple processor architectures. Since a large number of different solutions is available in the literature, we will try to integrate the most interesting approaches into a standard description of the available features with a homogeneous notation. This will allow to better understand the existing relations among previously proposed resource reservation techniques.

Chapter 2

The Generic Resource Reservation Framework

We consider a set of applications A_k to be scheduled on a shared computing platform. Each application may be composed of a set of hard, soft or non real-time tasks. Each task τ_i consists of a sequence of jobs that need to receive a certain amount of execution. In order to meet the mandatory deadlines of hard real-time tasks, it is necessary to provide upper bounds on the worst-case execution times (WCET) of such tasks. For soft and non real-time tasks, instead, there is no need to specify worst-case parameters, since there are no hard deadlines. The weaker timing requirements of soft and non real-time tasks makes it inappropriate to treat such tasks as hard real-time tasks, firstly because their unpredictability could lead to an underestimation of the WCET, compromising the guarantee done on the other tasks; secondly because it would be very inefficient, since trying to guarantee a task with a WCET much greater than its mean execution time would cause a waste of the CPU resource.

This problem can be solved by a *bandwidth reservation* strategy, which assigns each soft task a maximum bandwidth, calculated using the mean execution time and the desired activation period, in order to increase CPU utilization. If a task needs more than its reserved bandwidth, it may slow down, but it will not jeopardize the schedulability of the hard real-time tasks. By isolating the effects of task overloads, hard tasks can be guaranteed using classical schedulability analysis. The enforcement of such bandwidth reservation strategy may be accomplished through the use of real-time servers. A server is an abstract entity used by a scheduler to reserve a fraction of CPU-time to a particular instance. Each server S_k is characterized by the period of the reservation P_k and by the reserved execution time per period Q_k; we define $U_k = Q_k/P_k$ the fraction of CPU-time reserved by server S_k, also called utilization factor. In addition, each server maintains its own internal variables that are updated by the scheduler depending on the server rules. One of these variables is the server priority. The servers are inserted in the priority queue of the scheduler. When a server is selected by the scheduler to execute, the corresponding instance is executed and the server budget is accordingly decremented.

Real-time servers are not only used to schedule soft and non real-time tasks together with other instances having hard timing requirements, but also to provide temporal isolation among different applications in a hierarchical environment. In such systems, different applications need to share a common computing platform, without interfering each other. By encapsulating each application in a particular real-time server, it is possible to enforce the desired isolation among the various system components. When a particular server is selected for execution by the high-level scheduler, the corresponding application is executed. The internal scheduler of the selected application will then decide which instance to execute. Note that an application could also select to execute another (lower-level) server, increasing the hierarchical structure depth.

In the following, we assume each application A_k be scheduled by means of a dedicated server S_k. Since an application may as well be composed of a single task, the above mentioned bandwidth reservation mechanism for soft and non real-time tasks can be described with the same model.

The next section will provide a more detailed description of the hierarchical system under consideration.

2.1 Open environments

In this section we resume the characteristics of the addressed environment. A more detailed description can be found in the closely related deliverable *D4b*.

We will denote as "open environment" a system in which several independent applications $A_1, \ldots, A_q$ execute upon a shared processing platform. The shared processing platform may comprise one or more preemptive processors. We will distinguish between:

- a unique *system-level scheduler* (or *global scheduler*), which is responsible for scheduling all admitted applications on the shared processing platform;
- one or more *application-level schedulers* (or *local schedulers*), that decide how to schedule the jobs of an application.

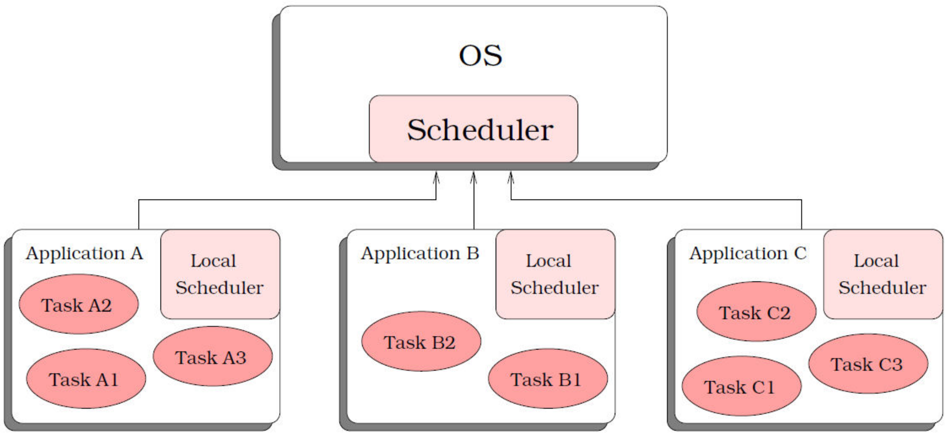

Figure 2.1: Generic structure of an open environment.

An *interface* must be specified between each application and the open environment. The goal of this interface specification is to abstract out and encapsulate the salient features of the application's resource requirements. The open environment uses this information during *admission control*, to determine whether the application can be supported concurrently with other already admitted applications; for admitted applications, this information is also used by the open environment during run-time to make scheduling decisions. If an application is admitted, the interface represents its "contract" with the open environment, which may use this information to enforce ("police") the application's run-time behavior. As long as the application behaves as specified by its interface, it is guaranteed to meet its timing constraints; if it violates its interface, it may be penalized while other applications are isolated from the effects of this misbehavior. We require that the interface for each application A_k be characterized by two parameters:

- A *virtual processor (VP) speed* α_k; and
- A *jitter tolerance* Δ_k;

The intended interpretation of these interface parameters is as follows: *all jobs of the application will complete at least* Δ_k *time units before their deadlines if executing upon a dedicated processor of computing capacity* α_k.

We now provide a brief overview of the application interface parameters. Further details on the adopted interface are available in deliverable *D4b*.

VP speed α_k

Since each application A_k is assumed validated upon a slower virtual processor, this parameter is essentially the computing capacity of the slower processor upon which the application was validated.

Jitter tolerance Δ_k

Given a processor with computing capacity α_k upon which an application A_k is validated, this is the minimum distance between finishing time and deadline among all jobs composing the application. In other words, Δ_k is the maximum release delay that all jobs can experience without missing any deadline.

At first glance, this characterization may seem like a severe restriction, in the sense that one will be required to "waste" a significant fraction of the VP's computing capacity in order to meet this requirement. However, this is not necessarily correct. Consider the following simple (contrived) example.

Example 1 *Let us represent a sporadic task [2, 3] by a 3-tuple:* (WCET, relative deadline, period). *Consider an application comprised of two sporadic tasks* $\{(1,4,4),(1,6,4)\}$ *to be validated upon a dedicated processor of computing capacity one-half. The task set fully utilizes the VP. However, we could schedule this application such that all jobs always complete two time units before their deadlines. That is, this application can be characterized by the pair of parameters* $\alpha_k = \frac{1}{2}$ *and* $\Delta_k = 2$.

Observe that there is a trade-off between the VP speed parameter α_k and the timeliness constraint Δ_k — increasing α_k (executing an application on a faster VP) may cause an increase in the value of Δ_k. Equivalently, a lower α_k may result in a tighter jitter tolerance, with some job finishing close to its deadline. However, this relationship between α_k and Δ_k is not linear nor straightforward – by careful analysis of specific systems, a significant increase in Δ_k may sometimes be obtained for a relatively small increase in α_k.

Our characterization of an application's processor demands by the parameters α_k and Δ_k is identical to the *bounded-delay resource partition* characterization of Feng and Mok [4, 5, 6].

2.2 State Diagram

As we previously mentioned, the necessary isolation among different applications must be enforced by proper scheduling mechanisms. We will show here how existing real-time servers can be used for this purpose. First of all we will present a

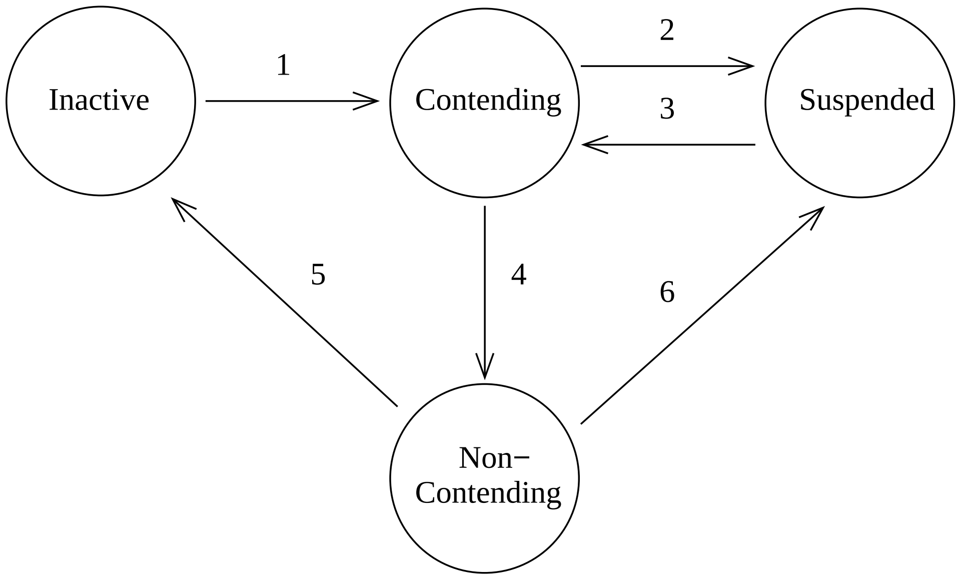

Figure 2.2: State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

basic version of an EDF-scheduled real-time server. Secondly, we will show how to enhance the basic functionalities of this server with some mechanisms for a more efficient usage of the available bandwidth. Finally, we will show how existing algorithms relate to the presented model.

Our basic server-based scheduling algorithm is essentially an application of the Constant Bandwidth Server (CBS) of Abeni and Buttazzo [7], enhanced to allow for the concurrent execution of different applications in an open environment.

CBS-like servers have an associated *period* P_k, reflecting the time-interval at which budget replenishment tends to occur. For a server, the value assigned to P_k is as follows:

$$P_k \leftarrow \frac{\Delta_k}{2(1-\alpha_k)} \,. \tag{2.1}$$

In addition, each server maintains three variables:

- a *deadline* D_k;
- a *virtual time* V_k; and
- a *reactivation time* Z_k.

Since each application has a dedicated server, we will not make any distinction between server and application parameters. Let us define an application (or server) to be *backlogged* at a given time-instant if it has any active jobs awaiting execution at that instant, and *non-backlogged* otherwise.

Definition 1 (backlogged) *A server is* backlogged *if the corresponding application has some task waiting to be executed.*

In the following, we will denote as "fair share" the amount of execution received by an ideal fluid server that executes for αt every t time-units (as a Generalized Processor Sharing (GPS) server [8] with weight α).

At each instant during run-time, each server S_k (or corresponding application A_k) is in a particular *state*. There are four possible states (see Figure 2.2). The transition from one state to another depends on the server being backlogged or not, and on the amount of execution it received.

- Each non-backlogged application is in either the *Inactive* or *Non-Contending* states. If an application has executed for more than its "fair share," then it is *Non-Contending*; else, it is *Inactive*.
- Each backlogged application is in either the *Contending* or *Suspended* state. While contending, it is eligible to execute; executing for more than it is eligible to results in its being suspended.

An *Executing* state is omitted from our description to simplify the diagram and because it is clear when the server will transition to and from the *Executing* state: there is an implicit *Executing* state with transitions to and from the *Contending* state (when a server is, respectively, de-scheduled or scheduled for execution).

Note that there is no analog of the *Suspended* state in the original definition of CBS [7]. This state has been introduced to allow our general framework to include also the description of hierarchical servers, as it will be better explained later on.

The server variables are updated according to the following rules (i)–(vii) (let t_{cur} denote the current time).

(i) Initially, each application is in the *Inactive* state. If application A_k wishes to contend for execution at time-instant t_{cur} then it transits to the *Contending* state (transition **(1)** in Figure 2.2). This transition is accompanied by the following actions:

$$\begin{aligned} D_k &\leftarrow t_{cur} + P_k \\ V_k, Z_k &\leftarrow t_{cur} \end{aligned}$$

(ii) At each instant, the system-level scheduling algorithm selects for execution some application A_k in the *Contending* state — we assume this selection is made according to a system-level EDF scheduler, selecting for execution the contending server S_k having the earliest absolute deadline.

(iii) The virtual time of an executing application A_k is incremented by the corresponding server at a rate $1/\alpha_k$:

$$\frac{d}{dt}V_k = \begin{cases} 1/\alpha_k, & \text{while } A_k \text{ is executing} \\ 0, & \text{the rest of the time} \end{cases}$$

(iv) If the virtual time V_k of the executing application A_k becomes equal to D_k, then application A_k undergoes transition **(2)** to the *Suspended* state. This transition is accompanied by the following actions:

$$\begin{aligned} Z_k^{\text{new}} &\leftarrow Z_k^{\text{old}} + P_k \\ D_k &\leftarrow Z_k^{\text{new}} + P_k \end{aligned}$$

(v) An application A_k that is in the *Suspended* state necessarily satisfies $Z_k \geq t_{cur}$. As the current time t_{cur} increases, it eventually becomes the case that $Z_k = t_{cur}$. At that instant, application A_k transits back to the *Contending* state (transition **(3)**).

Observe that an application may take transition **(3)** instantaneously after taking transition **(2)** – this would happen if the application were to have its virtual time become equal to its deadline at precisely the time-instant equal to its deadline.

(vi) An application A_k which no longer desires to contend for execution (i.e. the application is no longer backlogged) transits to the *Non-Contending* state (transition **(4)**), and remains there as long as V_k exceeds the current time. When $t_{cur} \geq V_k$ for some such application A_k in the *Non-Contending* state, A_k transitions back to the *Inactive* state (transition **(5)**); on the other hand, if an application A_k desires to once again contend for execution (note $t_{cur} < V_k$, otherwise it would be in the *Inactive* state), it transits to the *Suspended* state (transition **(6)**).

Observe that an application may take transition **(5)** instantaneously after taking transition **(4)** – this would happen if the application were to have its virtual time be no larger than the current time at the instant that it takes transition **(4)**.

The value of V_k is a measure of how much reserved service has been consumed by that time. At each instant in time, the server has received the same amount of service that it would have received by time V_k if executing on a dedicated processor of capacity α_k.

The system-level scheduler will simply select for execution the contending server with the earliest deadline. In brief, we implement EDF among the various contending applications, with the application deadlines (the D_k's) being the deadlines under comparison.

2.3 Design considerations

The main difference of the presented algorithm from the classic Constant Bandwidth Server presented in [7] is the insertion of a "*Suspended*" state. We will hereafter provide more details on these design decisions.

Bounded-delay server

The introduction of a *Suspended* state is needed to prevent the "deadline-aging" problem that arises with classic algorithms. With CBS, the deadline of a server that exhausts its budget is postponed, recharging the budget. In a particular situation in which there is only one backlogged server S_1, with no other contending server in the system, S_1 can continuously execute, repeatedly postponing its deadline. When other servers become backlogged, S_1 will be prevented to execute for a long time, since its deadline went too far. To avoid this problem, we introduce a *Suspended* state, preventing the server deadline to become too large. In this way, we will be able to implement a *bounded-delay server*, according to the following definition.

Definition 2 *A* bounded-delay server *is a server that implements a bounded-delay partition.*

The bounded-delay resource partition model, introduced by Mok *et al.* [4], is an abstraction that quantifies resource "supply" that an application receives from a given resource.

Definition 3 *A server implements a* bounded-delay *partition* (α_k, Δ_k) *if in any time interval of length L during which the server is continually backlogged, it receives at least*

$$(L - \Delta_k)\alpha_k$$

units of execution.

Rules (i) to (vi) basically describe a bounded-delay version of the Constant Bandwidth Server, i.e., a CBS in which the maximum service delay experienced by an application A_k is bounded by Δ_k. A similar server has also been used in [9, 10].

Reducing the preemption overhead

A simple modification to rule (vi) may reduce the number of times a server is preempted. When undertaking transition **(6)**, the following actions can be taken:

$$\begin{aligned} Z_k &\leftarrow V_k \\ D_k &\leftarrow V_k + P_k \end{aligned}$$

Such operations guarantee that when an application resumes execution, its budget is full, potentially reducing the number of preemptions. Without taking such actions when undertaking transition **(6)**, the current server budget and deadline are maintained, and it is less likely that a task will complete before the budget being exhausted. In other words, if the virtual time V_k is smaller than the deadline D_k when undertaking transition **(6)** (because A_k executed for less than expected), the original server would not increment D_k, but continue executing the application with the old deadline (and consequently, with greater priority), increasing the server responsiveness. The downside is that there are more chances for the server to be preempted due to budget exhaustion.

Therefore, in deciding whether to increment D_k as above or not, one need to consider the following tradeoff. If the deadline is not incremented and the application needs to execute for a short amount of time, then A_k is likely to complete within the current deadline, and hence to complete before it would in the presence of deadline-incrementing. On the other hand, not incrementing the deadline makes it more likely that A_k would not be able to complete within the current deadline, requiring deadline postponement and consequently, further preemptions. What could finally drive the decision is that the period parameter of a server is an indication of the granularity of the time from the perspective of the server. By not incrementing deadlines, we may obtain a response that is quicker than this granularity, but presumably this is not of much significance to the application. On the other hand, the potential drawback of additional preemptions is a very real concern, imposing a larger overhead on the system, particularly in systems where most of the job execution requirements are known to be no larger than the server budget $\alpha_k P_k$. In these cases, using the deadline updating mechanism presented in this section seems the best choice.

2.4 Reclaiming of unused bandwidth

The introduction of the *Suspended* state in the definition of the server presented in Section 2.2 has a side effect: our scheduling framework becomes non work-conserving. Basically, it can happen that the system is idle while some application is waiting to be executed, since all backlogged servers are in *Suspended* state.

To avoid this problem, as well as to improve the distribution of the unused bandwidth left by applications that execute for less than what declared, it is possible to adopt particular *bandwidth reclaiming* mechanisms.

In the last decade, many different techniques have been presented for the reclaiming of unused capacity in a server-based real-time system. The considerable attention that has recently been dedicated to this problem can be motivated with the typical issues that can arise while designing a reservation-based environment in an effective way. The need not to over-reserve the capacity dedicated to a particular task or application suggests one to assign, when possible, server parameters according to some average execution value, using worst-case parameters only for very critical instances. In order to ensure good system performances, soft real-time and best effort processes can then reclaim over-allocated capacity from servers that did not need it.

Previously proposed works dealing with this problem presented a large number of different techniques to distribute the spare capacity in an effective way, allowing overrun handling and fast system responsiveness. However, it is not clear how these approaches are related, and to what extent they contribute to solve the addressed problems.

To better understand the mechanisms under the various reclaiming algorithms, we will first distinguish the unused (reclaimable) bandwidth into the following typologies.

- *Not-admitted bandwidth*: it is the share of the CPU that has not been accounted for in the admission control test. It corresponds to the capacity left when the sum of the bandwidths of the admitted servers is lower than the capacity of the computing platform.

- *Inactive capacity*: it is the capacity associated to servers that are not backlogged and that since their last activation executed for less than their fair share. In other words, this is the capacity that is left by admitted servers that temporarily don't have tasks or jobs in their ready queue. This is the bandwidth safely reclaimed by the GRUB algorithm [9].

- *Cache capacity*: it is the remaining budget of servers that have an earlier completion (an *underrun*) and that are known to activate themselves again at least after the next server deadline. This is the kind of capacity reclaimed by CASH [11] and BASH [12] algorithms.

In Section 2.5, we will show that there are also other kinds of capacities that can be reclaimed in a "less safe" way. For the moment, we will focus only on the reclaiming techniques that do not violate the temporal isolation property of the admitted servers. We hereafter detail such techniques.

- A sort of implicit reclaiming mechanism is used by any *work-conserving* server. Systems holding this property will never be idle when an application is waiting to be executed. Therefore, the bandwidth left unused by an application will always be used by some other application.

- A simple rule that can be easily added to the server presented in Section 2.2 to render it work-conserving is resetting to *Inactive* the state of all servers when the processor is idle. In this way, backlogged servers that were in *Suspended* state will immediately switch to the *Contending* state, allowing some application to be scheduled.

- Another option to add the work-conserving property to the presented server is implementing the *time-warping* mechanism used by IRIS in [13] and BEBS in [14]. According to this mechanism, whenever the system is idle because all servers are either non-backlogged or in *Suspended* state, the reactivation time Z_k of each suspended application A_k is decreased by $(Z_{\min} - t_{\text{cur}})$, where $Z_{\min}$ is the first reactivation time among all suspended servers[1]:

  $$\forall A_k : \quad Z_k \quad \leftarrow \quad Z_k - (Z_{\min} - t_{\text{cur}}).$$

  This is sufficient to avoid an idle condition when there are backlogged servers waiting to be executed. Somewhat counterintuitively, this solution shows a fairer distribution of the available bandwidth than with the previously described approach.

- For non work-conserving systems, a simple way to reclaim the not-admitted bandwidth can be provided by assigning such bandwidth to a new server that will work as a capacity tank. This server will supply additional execution to other servers that might need a further share of bandwidth to satisfy a temporary overrun.

- The *inactive capacity* may be reclaimed implementing a smart mechanism adopted by GRUB in [9]. The virtual time V_k of an executing application A_k is updated at a rate $\alpha_{\text{active}}/\alpha_k$, instead than at a rate $1/\alpha_k$, where α_{active} represents the sum of the α_k of each admitted application A_k that is either in *Contending*, *Non-Contending* or *Suspended* state, i.e. excluding all inactive applications.

- *Cache capacity* may be safely reclaimed only when there are valid reasons to believe that a non-backlogged server will not become active before a particular time-instant. The capacity that the server would have used in the considered time interval may then be safely assigned to other servers, as with CASH [11] and BASH [12] algorithms. The typical case is when a soft real-time task encapsulated into a server has an early completion. Since that task will not be activated until its next period, the unused bandwidth can be assigned to a different application, for instance to the next scheduled server. Nevertheless, the cache capacity will still be associated to the original server deadline (priority), and accordingly scheduled. Particular care must be taken when the system is idle. In this case, the cache capacity must be properly decremented (see [11, 12]) to avoid capacity overallocation. This kind of reclaiming is not particularly suitable for hierarchical systems or, in general, for servers that handle more than one task at a time. In these cases, in fact, it is very difficult to guarantee that a server will not reactivate before a certain time-instant.

2.5 Aggressive reclaiming

In the literature, there are as well more aggressive reclaiming strategies that "steal" some bandwidth from other applications. Even if this solution could result in breaking the temporal isolation of the admitted applications, it can sometimes be useful when dealing with overloaded system. In some cases, it could be better to serve an actual overrun using capacities reserved to other servers, rather than

[1]Note that IRIS decreases as well the deadline of the server(s) that first reactivates. BEBS, instead, does not decrease any deadline, obtaining a fairer reclaiming of the unused bandwidth.

waiting for unused capacities. In fact, such late capacities might appear too late in time, after the overrunning task already missed its deadline. Therefore, a stealing server may solve temporary overruns by using in advance the capacity reserved to future applications, hoping that such capacity will be left unused by some of these applications. In a certain sense, these strategies try to reclaim "future" unused bandwidth.

Another sort of aggressive strategy is the *self-reclaiming*, i.e. the reclaiming of capacity reserved to future jobs of the same tasks. This form of reclaiming has been adopted by classic CBS, postponing the deadline of the server to recharge its exhausted budget. As we previously mentioned, this could lead to the deadline aging problem, resulting in a server that does not implement a bounded-delay reservation.

Other sorts of capacities may be reclaimed when different task models are adopted. For instance, hard real-time applications may reclaim the capacities associated to firm, soft or non real-time tasks.

The selection of the most appropriate real-time server mechanisms depends on the application requirements, as well as on the particular metric one might adopt for the evaluation of server performances. Various metrics are possible. Each one is valid, depending on the addressed target:

- deadline miss ratio $\rightarrow$ for hard and firm RT tasks;
- normalized response time $\rightarrow$ for non RT tasks, or, in general, tasks without an associated deadline that nevertheless require a responsive service.
- average tardiness/lateness $\rightarrow$ for soft RT tasks;

The above list is by no means exhaustive (e.g., the actual control performance may be used for control type tasks, etc.). Greedy algorithms that tend to assign the unused bandwidth to the executing tasks show better performances if the first two metrics are used. On the other hand, if the design target is to minimize the average tardiness, fair algorithms are preferable, proportionally distributing the excess bandwidth to all tasks in the system.

Chapter 3

Existing solutions

In this chapter, we will review the major existing solutions for the bandwidth reservation in shared processing systems. The survey will be divided into the following parts:

1. Analysis of the reservation techniques for single processor platforms, with relation to previously proposed real-time servers, hierarchical schedulers, reclaiming techniques, etc.

2. Analysis of the few existing solutions for the implementation of reservations on multiprocessor platforms. Particular attention will be dedicated to the detection of the main drawbacks and bottlenecks of the existing solutions, devising possible strategies for the solution of the problems left open.

3.1 Dynamic Priority Servers for Single Core Systems

Total Bandwidth Server and related variants

Among real-time servers based on EDF scheduling, the Total Bandwidth Server (TBS) presented by Spuri *et al.* in [15] can be adopted when execution times are known upon job arrivals. Whenever a new job with execution requirement C_i needs to be scheduled, the server budget and deadline are set, respectively, to C_i and $\max(t_{cur}, D_{\text{old}}) + C_i/U$, where U is the server reserved bandwidth. The Constant Utilization Server (CUS) presented in [16, 17] has similar characteristics, except for the fact that instead of immediately replenishing the budget when a new job needs to be executed, it waits until the server deadline. TB* [18] is an improvement over TBS that obtains optimal responsiveness by properly decreasing the server deadline. Since it is necessary to know in advance the execution requirements of the scheduled entities, neither of the above servers (TBS, CUS, TB*) is suitable for open environments.

Constant Bandwidth Server

A server that does not need any information on the execution times of the scheduled entities is the Constant Bandwidth Server (CBS) presented by Abeni and Buttazzo in [7]. It is similar to the implementation of our basic server described in Section 2.2, with one main differences: a self-reclaiming mechanism that allows using in advance the capacity reserved to future jobs of the executing task. This mechanism is easily obtained by postponing the server deadline as soon as the budget is exhausted, without needing to wait until $t_{cur} \equiv Z_k$. Therefore, no *Suspended* state is needed, and the reactivation time Z_k becomes useless. The drawback of this approach is that it is prone to the deadline-aging problem, so that neither CBS can be efficiently used to serve systems in which tasks are to be executed with more complex requirements than the simple First-Come First-Served execution.

Not admitted capacity reclaiming: IRIS and BEBS

To solve this problem, Marzario *et al.* proposed IRIS [13], a server that does not suffer from the deadline-aging problem, and that can be efficiently used for the implementation of open environments. It is equivalent to our basic server, with the additional work-conserving property obtained through a time-warping mechanism, as the one described in Section 2.4: whenever the system is idle, the reactivation time of each *Suspended* server is uniformly decreased, so that the earlier reactivation time coincides with the current time. A server almost identical to IRIS has been presented by Brandt *et al.* in [14]: the Best-Effort Bandwidth Server (BEBS). It is an improvement over the Rate-Based Earliest Deadline scheduler (RBED), an earlier server presented by the same group in [19]. The only difference between IRIS and BEBS is in the actions associated to the time-warping operation. IRIS decreases the reactivation time of every server and decreases as well the deadline of the server(s) that first reactivates. BEBS instead does not decrease any deadline, obtaining a fairer reclaiming.

Inactive capacity reclaiming: GRUB and SHRUB

Lipari and Baruah presented in [9] a CBS-based approach to reclaim the reserved capacity left free by inactive servers: the Greedy Reclamation of Unused Bandwidth (GRUB). As we mentioned in Section 2.4, the virtual time V_k of an executing application A_k is updated at a rate $\alpha_{\text{active}}/\alpha_k$, where α_{active} is the sum of the α_k of each admitted application A_k that is not in *Inactive* state. In this way, in each time interval dT, an executing server reclaims the share of bandwidth that has not been reserved to servers that have backlogged work to do, i.e. the share $(1 - U_{\text{active}})dT$. Note that this includes both the inactive and the not-admitted capacity.

GRUB's reclaiming is "greedy" because this excess capacity is entirely given to the executing server. Fairer reclaiming strategies may be derived by distributing the inactive capacity according to some particular policy. The Shared Reclamation of Unused Bandwidth (SHRUB) presented in [20, 21] distributes the reclaimable bandwidth according to weights assigned to each application.

Cash capacity reclaiming: CASH and BASH

Two algorithms that are able to reclaim cache capacities have been presented by Caccamo *et al.*: CASH [11] and BASH [12] When one knows that a server will not be activated until a certain time instant, the unused bandwidth is assigned to another application. This capacity is associated to the original server deadline, and accordingly scheduled. When the system is idle, the cache capacity must be properly decremented. CASH decreases the earliest deadline CASH capacity by the amount of idle time. BASH recomputes each BASH capacity as $\alpha(D_k - T_{idle})$, where T_{idle} is the last idle time. Simplified CASH-based servers are presented in [22].

Aggressive reclaiming: SLASH, BACKSLASH and CSS

Four resource reservation algorithms have been presented in [23], each one improving over the previous one. From the simplest one to the most aggressively reclaiming one, they are: SRAND, SLAD, SLASH and BACKSLASH. While the first two servers has been designed just to show that it is better to assign the unused bandwidth to the highest priority task (as in CASH), the other two servers are more interesting. SLASH adds a self-reclaiming mechanism that recharges the capacity postponing the deadline, as with CBS, and reclaims unused capacity from other servers using its original (unextended and, therefore, earlier) deadline. This allows a greedier reclaiming than with CASH, since the executing server can use a higher priority. BACKSLASH further increases the reclaiming capabilities by retroactively allocating unused capacity to servers that used in advance their own future capacity (with self-reclaiming).

The Capacity Sharing and Stealing (CSS) server presented in [24] integrates the reclaiming mechanism used by SLASH with the possibility to steal bandwidth reserved to inactive "non-isolated" servers.

Other dynamic servers

Ghazalie and Baker introduced in [25] the Deadline Deferrable Server (DDS), Deadline Sporadic Server (DSS) and Deadline Exchange Server (DXS), adapting to the EDF case the corresponding algorithms previously defined for fixed priority systems.

Spuri and Buttazzo introduced five different servers under dynamic priorities in [26]: Dynamic Priority Exchange (DPE), Dynamic Sporadic Server (similar to the DSS in [25]), Earliest Deadline Last (EDL) and Improved Priority Exchange (IPE).

Other bandwidth isolation algorithms with rather complex implementations are the Bandwidth Sharing Server (BSS) [27], its improved version BSS-I [28], and the Processor Sharing with Earliest Deadline First (PShED) [29].

In [1, 30], Deng *et al.* introduced the analysis of open environment for real-time systems . Their two-level hierarchical implementation is based on TBS and CBS servers. Two attempts for providing the bounded-delay property to CBS and GRUB have been presented, respectively, in [31] (H-CBS) and [32] (HGRUB).

3.2 Fixed Priority Servers for Single Core Systems

The most used algorithms for the bandwidth reservation in fixed priority systems are the Deferrable Server (DS) [33] and the Sporadic Server [34, 35]. While DS has a simpler implementation, SS is able to achieve a larger schedulable utilization.

Both algorithms are able to solve the poor performance of the Periodic or Polling server previously presented in [36]. The processor reservation approach presented by Mercer *et al.* in [37] is similar to a polling server.

The Priority Exchange Server (PE) was introduced in [38, 39]. However, it has no advantage over a Sporadic Server, but requires a more complex implementation [34, 35].

Fixed priority algorithms that implement stealing mechanisms have been proposed in [40], [41], and [42].

Two complementary schemes for the reclaiming of unused bandwidth are the Capacity Sharing server presented in [43] and the HisReWri algorithm proposed in [14].

The Dual Priority mechanism has been first introduced in [44] and later extended in[45].

3.3 Other Servers for Single Core systems

Hierarchical Loadable Schedulers (HLS) [46] is a framework for the composition of existing scheduling algorithms using hierarchical scheduling, providing a guaranteed scheduling behavior to the applications. Another framework that supports a hierarchy of arbitrary schedulers, without providing compositional guarantees, is the CPU inheritance scheduling proposed by Ford and Susarla in [47].

Offline strategies to deal with aperiodic workloads have been presented by Fohler *et al.* in [48, 49].

Overrun handling mechanisms for different task models

When different task models are adopted, other mechanisms have been designed for the bandwidth reclaiming in overrun conditions. For instance, Buttazzo and Stankovic proposed in [50] the Robust Earliest Deadline (RED) scheduling algorithm for applications composed of firm real-time tasks. In [51], a related algorithm is described: Robust High Density (RHD). Koren and Shasha presented in [52] an on-line scheduling algorithm, called D^{over}, that has an optimal competitive factor. Baruah and Haritsa [53] proposed an on-line scheduling algorithm (ROBUST) that maximizes processor utilization during overload conditions, given a minimum slack factor for all tasks. Thomas *et al.* proposed Spare CASH, an algorithm that adapts the reclaiming mechanism of CASH for a different model of firm real-time tasks [54]. For adaptive tasks that may change their rate, Buttazzo et al. formulated an algorithm in which rate changes are modeled using spring coefficients [55]. The Variable Rate Execution Model (VRE) in [56] is a similar (broader) model that implements and provides schedulability conditions for systems in which task execution rates change dynamically.

Networking algorithms

There are also various algorithms that have been proposed in the networking literature. Weighted Fair Queueing (WFQ) (also known as packet-by-packet Generalized Processor Sharing (GPS)) is a well-known proportional-share scheduling algorithm. The WFQ scheduler associates a weight to each connection session; all connection sessions share the routerŠs bandwidth in proportion to their weights. The transmission rate of each session depends on the combination of its weight and the summation of all weights. The virtual time V(t) is defined as follows:

$$V(t) = \int_0^t \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau$$

where w_j is the weight of task j and $A(\tau)$ is the set of active tasks at time τ. Thus, virtual time progresses at a rate inversely proportional to the summation of all weights. That is, the more sessions in the system, the slower transmission rate each session gets. Worst-case Fair Weighted Fair Queuing (WF^2Q) [8] is an extension of WFQ that prevents a task from getting executed faster than expected in a perfect fair share scheduler. WF^2Q+ [57] is an improved version of WF^2Q, having a lower complexity. Straightforward extensions are derived for hierarchical systems [57]: H-GPS, H-WF^2Q+, etc.

Proportional Share algorithms

Proportional-share algorithms have been adapted to solve specific scheduling latency problems facing soft real-time applications such as multimedia. Proportional-sharing of CPU is similar to flow-based packet schedulers such as WF^{2}Q, because awareness of throughput is used to make scheduling decisions. While the goal of most proportional-share algorithms is maintaining constant rate (i.e. a fluid model) over any interval, the CBS algorithm relaxes the fairness constraint, only ensuring that enough proportion is received at deadlines.

Two multimedia schedulers are built on WFQ: SMART [58] and BERT [59]. SMART prioritizes a task by two parameters: priority and biased virtual finishing time (BVFT). The scheduler always chooses the task with the highest priority. When multiple tasks are at the same priority level, the scheduler tries to satisfy as many BVFTs as possible. BERT is an implementation of WF^{2}Q plus a cycle stealing mechanism While WF^{2}Q provides proportional sharing, the cycle stealing mechanism provides a flexible way for urgent tasks to meet their deadlines when their demands exceed their shares.

The Earliest Eligible Virtual Deadline First (EEVDF) [60] algorithm is another proportional-share algorithm that employs virtual time. The EEVDF algorithm puts all aperiodic jobs into the same queue and assigns a deadline for each job. According to weight w_i, release time t_0^i and execution time r_k, the virtual eligible time Z_k and virtual deadline D_k of a task are computed using equations presented in [60] and summarized as follows: $Z_1 = V(t_0^i); D_k = Z_k + r(k)/w_i; Z_{k+1} = D_k$. The virtual time in EEVDF is identical to the definition in WFQ.

Further proportional-share algorithms are Borrowed-virtual-time (BVT) [61], Start-Time Fair Queuing (SFQ) [62], Lottery scheduling [63], Stride scheduling [64].

The Completely Fair Scheduler (CFS) is a variant of SFQ that has been adopted as the default scheduler in the Linux kernel. Further detail on this scheduler are contained in the deliverable *D4b*.

3.4 Servers for Multi Core Systems

While there are many existing works addressing the uni-processor case, only a few results deal with multiprocessor case.

Adaptations of uniprocessor servers: M-CBS, M-TBS and M-CASH

Baruah *et al.* presented in [65, 66] the M-CBS algorithm, extending CBS to support identical multicore systems. Each server is characterized by a period P_i and a share U_i, which is required to be less than 1. An application is guaranteed to complete within a margin of P_i from the time it would complete on a fully dedicated processor. The main drawback of this approach is that it is designed for applications composed by a single task contained in a single server.

Baruah and Lipari also proposed in [67] a multiprocessor version of the TBS algorithm (M-TBS). This server is able to fully exploit a multiprocessor system, being able to reclaim for the service of aperiodic jobs, the bandwidth that can not be used to satisfy hard real-time requirements of periodic tasks under global EDF scheduling. It suffers from the same design issues described before for the M-CBS algorithm. In [68], Kato and Yamasaki considered TBS too, with the aim of improving the response time of aperiodic activities in partitioned EDF scheduled multiprocessor systems.

In [69], Pellizzoni and Caccamo extended the CASH algorithm to uniform multiprocessor platforms. The M-CASH algorithm provides M-CBS with the capability of exploiting the computational resources left unused by the various servers. The reclaiming mechanism is based on a queue of unused capacity chunks and deadlines.

Multiprocessor periodic resource models

Shin et *al.* proposed in [70] a multiprocessor periodic resource model to describe the computational power supplied by a parallel machine. Their model is represented by the triplet $\langle \Pi, \Theta, m' \rangle$ meaning that the computational resource provided by a multiprocessor constituted by m' processors is Θ every period Π. Indeed this interface is extremely simple. Nonetheless we highlight two drawbacks of this abstraction.

1. The same periodicity Π is provided to all the tasks scheduled on the same virtual multiprocessor. This can lead to a quite pessimistic interface design. In fact the period of the interface will tipycally be imposed by the shortest period task, with a resulting waste of computational capacity due to the overhead. We believe that an approach that reserves time with different periodicity is better capable to answer to the needs of an application.

2. Considering the supplied resource Θ cumulatively among all the processors leads to a more pessimistic analysis than considering separately the contribution of each VP. This is due to the consideration that the worst-case analysis in multiprocessor systems occurs when the available processors allocate resource at the same time.

Leontyev and Anderson proposed in [71] a very simple, though effective, interface for the multiprocessor platform based on the only *bandwidth*. The authors suggest that a bandwidth requirement $w > 1$ is best allocated by an integer number $\lfloor w \rfloor$ of dedicated processors plus a fraction of $w - \lfloor w \rfloor$ allocated onto the other processors. This choice is supported by the evidence that a given amount of computing speed is better exploited on the minimum possible number of processors. However there are some circumstances where this approach is not the best.

1. As the authors themselves show by an example [71], there are some hard real-time tasks sets that can miss a deadline if the suggested policy is adopted, whereas they can meet the deadline if a different bandwidth allocation strategy is used.

2. If the physical platform already accommodated other applications, then a whole processor may be unavailable. Hence, in this case, we cannot allocate a dedicated processor to an application, but only a fraction of it that cannot possibly be expressed just by an unused bandwidth.

Other servers

Two server implementations based on, respectively, Pfair and ERfair scheduling have been presented in [72].

A proportional-share algorithm for multiprocessor systems is the Surplus Fair Scheduling (SFS) proposed by Chandra *et al.* in [73].

Bibliography

[1] Z. Deng and J. Liu, "Scheduling real-time applications in an Open environment," in *Proceedings of the Eighteenth Real-Time Systems Symposium*, (San Francisco, CA), pp. 308–319, IEEE Computer Society Press, December 1997.

[2] A. K. Mok, *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[3] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the 11th Real-Time Systems Symposium*, (Orlando, Florida), pp. 182–190, IEEE Computer Society Press, 1990.

[4] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *7th IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, pp. 75–84, IEEE, May 2001.

[5] X. A. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 26–35, IEEE Computer Society, 2002.

[6] X. Feng, *Design of Real-Time Virtual Resource Architecture for Large-Scale Embedded Systems*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 2004.

[7] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the Real-Time Systems Symposium*, (Madrid, Spain), pp. 3–13, IEEE Computer Society Press, December 1998.

[8] J. Bennett and H. Zhang, "WF^{2}Q: Worst-case fair queueing," in *Proceedings of IEEE INFOCOM'96*, pp. 120–128, March 1996.

[9] G. Lipari and S. Baruah, "Greedy reclaimation of unused bandwidth in constant-bandwidth servers," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, (Stockholm, Sweden), pp. 193–200, IEEE Computer Society Press, June 2000.

[10] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proceedings of the EuroMicro Conference on Real-time Systems*, (Porto, Portugal), pp. 151–160, IEEE Computer Society, 2003.

[11] G. B. M. Caccamo and L. Sha, "Capacity sharing for overrun control," in *Proceedings of 21th IEEE RTSS*, (Orlando, Florida), pp. 295–304, 2000.

[12] G. B. M. Caccamo and D. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Transactions on Computers*, vol. 54, February 2005.

[13] P. B. L. Marzario, G. Lipari and A. Crespo, "Iris: A new reclaiming algorithm for server-based real-time systems," in *Proceedings of the 10th IEEE RTAS*, (Toronto, Canada), 2004.

[14] T. B. S. Banachowski and S. A. Brandt, "Integrating best-effort scheduling into a real-time system," in *Proceedings of the 25th IEEE Real-Time Systems Symposium*, December 2004.

[15] M. Spuri, G. Buttazzo, and F. Sensini, "Robust aperiodic scheduling under dynamic priority systems," in *Proceedings of the Real-Time Systems Symposium*, (Pisa, Italy), pp. 210–221, IEEE Computer Society Press, 1995.

[16] J. S. Z. Deng, J.W.-S. Liu, "A scheme for scheduling hard real-time applications in open system environment," in *Ninth Euromicro Workshop on Real-Time Systems*, (Toledo, Spain), November 1997.

[17] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, New Jersey 07458: Prentice-Hall, Inc., 2000.

[18] G. Buttazzo and F. Sensini, "Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments," *IEEE Transactions on Computers*, vol. 48, pp. 1035–1052, October 1999.

[19] C. L. S. A. Brandt, S. Banachowski and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pp. 396–407, December 2003.

[20] T. C. G. L. S. B. Luigi Palopoli, Luca Abeni, "Weighted feedback reclaiming for multimedia applications," in *6th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, October 2008.

[21] L. A. Sanjoy Baruah, Giuseppe Lipari, "Shrub: Shared reclamation of unused bandwidth," tech. rep., Scuola Superiore Sant'Anna, http://retis.sssup.it/ lipari/papers/shrub_tech_report_jul_08.pdf, 2008.

[22] L. A. M. C. Giorgio Buttazzo, Giuseppe Lipari, *Soft Real-Time Systems: Predictability vs. Efficiency*. Plenum Publishing Co. (Series in Computer Science), 2005.

[23] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *Proceedings of the 26th IEEE RTSS*, pp. 410–421, 2005.

[24] L. Nogueira and L. M. Pinho, "Capacity sharing and stealing in dynamic server-based real-time systems," in *15th International Workshop on Parallel and Distributed Real-Time Systems*, March 2007.

[25] T. M. Ghazalie and T. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 9, 1995.

[26] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 10, no. 2, 1996.

[27] G. Lipari and G. Buttazzo, "Scheduling real-time multi-task applications in an open system," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, (York, UK), IEEE Computer Society Press, June 1999.

[28] G. Lipari and S. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in *Proceedings of the Real-Time Technology and Applications Symposium*, (Washington, DC), pp. 166–175, IEEE Computer Society Press, May–June 2000.

[29] G. Lipari, J. Carpenter, and S. Baruah, "A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments," in *Proceedings of the Real-Time Systems Symposium*, (Orlando, FL), IEEE Computer Society Press, November 2000.

[30] Z. Deng, J. Liu, L. Zhang, M. Seri, and A. Frei, "An Open environment for real-time applications," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 16, pp. 155–186, May 1999.

[31] G. Lipari and S. Baruah, "A hierarchical extension to the constant bandwidth server framework," in *Proceedings of the Real-Time Technology and Applications Symposium*, (Taipei, Taiwan), IEEE Computer Society Press, May–June 2001.

[32] G. L. Luca Abeni, Claudio Scordino, "Serving non real-time tasks in a reservation environment," in *Real-Time Linux Workshop*, Nov. 2008.

[33] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-teime environments," *IEEE Transactions on Computers*, vol. 44, January 1995.

[34] B. Sprunt, L. Sha, and J. P. Lehoczky, "Scheduling sporadic and aperiodic events in a hard real-time system," Tech. Rep. ESD-TR-89-19, Carnegie Mellon University, 1989.

[35] B. Sprunt, *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[36] L. Sha, J.P.Lehoczky, and R. Rajkumar, "Solutions for some parctical problems in prioritised preemptive scheduling," in *Proceedings IEEE Real-Time Systems Symposium*, pp. 181–191, 1986.

[37] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: operating system support for multimedia applications," in *Proceedings of the International Conference on Multimedia Computing and Systems, Boston, MA, USA, May 15–19, 1994* (IEEE, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. 90–99, IEEE Computer Society Press, 1994.

[38] J. Lehoczky, L. Sha, and J. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. of Real-Time Systems Symposium*, pp. 261–270, 1987.

[39] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm," in *Proceedings of the Real-Time Systems Symposium*, (Huntsville, Alabama), pp. 251–258, IEEE, December 1988.

[40] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems," in *Proceedings of the 13th Real-Time Systems Symposium*, pp. 110–123, December 1992.

[41] K. W. T. R. I. Davis and A. Burns, "Scheduling slack time in fixed priority preemptive systems," in *Proceedings of the 14th Real-Time Systems Symposium*, pp. 222–231, 1993.

[42] J. L. T.S. Tia and M.Shankar, "Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems," *Journal of Real-Time Systems*, 1995.

[43] G. Bernat and A. Burns, "Multiple servers and capacity sharing for implementing flexible scheduling journal = Journal of Real-Time Systems, year = 2002, volume = 22, number = 1-2, pages = 49–75,,"

[44] R. Davis and A. Wellings, "Dual priority scheduling," in *IEEE Real-Time Systems Symposium*, pp. 100–109, Dec. 1995.

[45] G. Bernat and A. Burns, "Combining (n, m)-hard deadlines and dual priority scheduling," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 46–57, 1997.

[46] J. Regehr and J. A. Stankovic, "A framework for composing soft real-time schedulers," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, (London, UK), pp. 3–14, December 2001.

[47] B. Ford and S. Susarla, "Cpu inheritance scheduling," in *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, (Seattle, WA), October 1996.

[48] D. Isovic and G. Fohler, "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints," in *Proceedings of the Real-Time Systems Symposium*, (Orlando, FL), IEEE Computer Society Press, November 2000.

[49] G. Fohler, T. Lennvall, and G. Buttazzo, "Improved handling of soft aperiodic tasks in offline scheduled real-time systems using total bandwidth server," in *In 8th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, (Nice, France), October 2001.

[50] G. Buttazzo and J. Stankovic, "Red: Robust earliest deadline scheduling," in *Third International Workshop on Responsive Computing Systems*, (New Hampshire, USA), September 1993.

[51] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. second ed., 2005.

[52] G. Koren and D. Shasha, "D^{over}: An optimal on-line scheduling algorithm for overloaded real-time systems," 1992.

[53] S. Baruah and J. Haritsa, "Scheduling for overload in real-time systems," *IEEE Transactions on Computers*, vol. 46, pp. 1034–1039, September 1997.

[54] M. C. D.C. Thomas, S. Gopalakrishnan and C. Lee, "Spare cash: Reclaiming holes to minimize aperiodic response times in a firm real-time environment," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, (Palma de Mallorca, Spain), July 2005.

[55] M. C. G. C. Buttazzo, G. Lipari and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, pp. 289–302, March 2002.

[56] S. Goddard and L. Xu, "A variable rate execution model," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pp. 135–143, July 2004.

[57] J. C. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 675–689, Oct 1997.

[58] J. Nieh and M. Lam, "The design, implementation and evaluation of smart: A scheduler for multimedia applications," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[59] P. L. Bavier, A. and M. D., "Bert: A scheduler for best effort and real-time tasks," tech. rep., Department of Computer Science, Princeton University, Princeton, NJ, 1999.

[60] I. Stoica, H. Abdel-Wahab, K. Jeffay, J. Gherke, G. Plaxton, and S. Baruah, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of the Real-Time Systems Symposium*, (Washington, DC), pp. 288–299, December 1996.

[61] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[62] P. Goyal, X. Guo, and H. Vin, "A hierarchical cpu scheduler for multimedia operating systems," in *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, (Seattle, Washington), pp. 107–122, October 1996.

[63] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proceedings of the First Symposium on Operating System Design and Implementation*, 1994.

[64] C. A. Waldspurger and W. E. Weihl, "Stride scheduling: Deterministic proportional-share resource management," Tech. Rep. Technical Memorandum, MIT/LCS/TM–528, Laboratory for Computer Science, Massachusetts Institute of Technology, 1995.

[65] S. Baruah and G. Lipari, "Executing periodic jobs in a multiprocessor Constant-Bandwidth Server implementation," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, (Catania, Sicily), pp. 109–117, IEEE Computer Society Press, July 2004.

[66] S. Baruah, J. Goossens, and G. Lipari, "Implementing constant-bandwidth servers upon multiprocessor platforms," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, (San Jose, California), pp. 154–163, IEEE Computer Society Press, September 2002.

[67] S. Baruah and G. Lipari, "A multiprocessor implementation of the Total Bandwidth Server," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, (Santa Fe, New Mexico), IEEE Computer Society Press, April 2004.

[68] S. Kato and N. Yamasaki, "Scheduling aperiodic tasks using total bandwidth server on multiprocessors," in *Proceedings of the 6th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC2008)*, 2008.

[69] R. Pellizzoni and M. Caccamo, "M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms," *Real-Time Systems*, vol. 40, no. 1, pp. 117–147, 2008.

[70] I. Shin, , A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering multiprocessors," in *Proceedings of the* 20^{th} *Euromicro Conference on Real-Time Systems*, (Prague, Czech Republic), pp. 181–190, July 2008.

[71] H. Leontyev and J. Anderson, "A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, (Prague, Czech Republic), pp. 191–200, July 2008.

[72] A. Srinivasan, P. Holman, and J. Anderson, "Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, (Vienna, Austria), pp. 19–28, IEEE Computer Society Press, June 2002.

[73] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors," in *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, CA*, October 2000.