Adaptivity & Control of Resources in Embedded Systems

D4c

COOPERATION

Handling Resource Constraints in Open Environments

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: **D4c 0.1**
Project Document Date: **2010-12-30**
Workpackage Contributing to the Project Document: **WP4**
Deliverable Type and Security: **R-PU**

Contents

Chapter 1

Introduction

The research on real-time open environment is mature to a point that many existing works have been proposed for the case in which a set of independently developed applications are scheduled upon a computing platform (see Deliverable D4a for an exhaustive survey on the existing approaches). Unfortunately, most of the existing works assume that applications are fully independent, meaning that they do not share any resource different from the processor. In real systems, this assumption turns out to be too constrictive and unrealistic. Even if each application accesses private memory regions, there will always be some structures that need to be shared with other applications composing the system: global memory regions, synchronization structures, I/O devices, bus, etc. Since each one of these resources might potentially be accessed concurrently by other applications executing on the same platform, mutual exclusion mechanisms need to be adopted to guarantee the logical correctness of the system.

The existing protocols for the mutually exclusive access to shared resources assume that an application is scheduled on a dedicated platform [1, 2, 3]. To adapt these protocols to the open environments considered in the ACTORS project, additional blocking effects need to be taken into account. There are recent publications proposing designs for open environments that allow for sharing other resources in addition to the preemptive processor [4, 5, 6, 7]. These designs assume that each individual application may be characterized as a collection of sporadic tasks [8, 9], distinguishing between shared resources that are *local* to an application (i.e., only shared within the application) and *global* (i.e., may be shared among different applications). However, these approaches either propose that global resources be executed with local preemptions disabled [6], potentially causing intolerable blocking inside an application; or allow applications to overrun their budget while holding a lock [6, 4, 5, 7], increasing in this way the blocking among applications. We will describe in detail such effects in Chapter 2, proposing as well a novel protocol that can reduce the schedulability penalties due to blocking, both locally — inside each application — and globally — among different applications.

A further complication on the model is needed to deal with the possibility of executing on more than one processor. When an application executing on a CPU tries to lock a resource that is held by an application executing on a different CPU, an additional blocking factor needs to be accounted for. The same is true for each other processor in which there is a task that might access the same resource. The schedulability penalty due to this "remote blocking" might be significant if the tasks locking the same global resource are spread among many different processors. We will analyze this problem, along with the existing solutions, in Chapter 4. This will allow us to design efficient open environments for multiprocessor plat-

forms, removing previous unrealistic assumptions such as the independency of the applications.

1.1 Relation to the overall project

The techniques described in this document represent an extensive description of the state of the art in the hierarchical scheduling of applications sharing global resources on single and multiprocessor platforms. Due to the generality of this work, only a subset of these techniques will be actually implemented in the Linux-based resource reservation scheme developed in Task 4.5 and described in deliverable D4e. In particular, we intend to implement the technique described in Chapter 3 for the non-preemptive execution of global critical section. The simplicity of this approach and the efficiency of the proposed solution allows avoiding complex resource sharing protocols, that are hard to support in Linux, so that locking and unlocking operations can be significantly simplified. The budget exhaustion problem will be avoided implementing each Virtual Processor by means of a BROE server, as described in Chapter 2.

Since the ACTORS methodology to allocate VPs onto the physical CPUs will be based on a partitioned solution (as will be explained in deliverable D4d), we will investigate in Chapter 4 the most suitable strategies for implementing mutual exclusion mechanisms among tasks executing on different processors. We will then propose a solution that is characterized by a small implementation overhead, allowing an easier integration in the Linux kernel mechanisms. This solution allows for a simple schedulability analysis, without needing to take into account complex blocking factors that have to be considered using alternative techniques.

To further improve the schedulability of the system, it is important to decrease the remote blocking factors due to resources accessed by tasks allocated on different processors. This can be obtained by allocating tasks that access the same global resources on the fewest possible number of processors. Optimized allocation strategies, considering the timing parameters and the requested bandwidth of each Virtual Processor, will be analyzed in Task 4.4 and extensively detailed in Deliverable D4d.

Chapter 2

Bounded delay servers for Resource sharing Open Environments

In this chapter, we describe our design of an open environment upon a computing platform composed of a single preemptive processor and additional shared resources. We assume that each application can be modeled as a collection of preemptive jobs which may access shared resources within critical sections. (Such jobs may be generated by, for example, periodic and sporadic tasks.) We require that each such application be scheduled using some local scheduling algorithm, with resource contention arbitrated using some strategy such as the Stack Resource Policy (SRP) [2]. We describe what kinds of analysis such applications must be subject to and what properties these applications must satisfy, in order for us to be able to guarantee that they will meet their deadlines in the open environment.

2.1 State of the art

The scheduling framework described in this chapter can be considered a generalization of earlier ("first-generation") open environments (see, e.g., [10, 11, 12, 13, 14, 15]), in that our results are applicable to shared platforms composed of serially reusable shared resources in addition to a preemptive processor. These projects assume that each individual application is composed of periodic implicit-deadline ("Liu and Layland") tasks that do not share resources (neither locally within each application nor globally across applications); however, the resource "supply" models considered turn out to be alternative implementations of our scheduler (in the absence of shared resources).

There are other works that instead consider the access to globally shared resources in simpler server-based environments. The servers used in these environments do not allow for the hierarchical execution of different applications, but are often limited to the execution of one single task per server. Among fixed priority scheduled servers, Kuo and Li presented in [16] a resource sharing approach to be used with sporadic servers [17]. All global resources are handled by a single dedicated server that has capacity equal to the sum of all critical sections lengths and period equal to the GCD of the periods of the tasks accessing global resources. The drawback of this approach is that it could require a large utilization to accommodate such a dedicated server.

Among EDF-scheduled task systems, Ghazalie and Baker present in [18] an overrun mechanism to be used together with dynamic deferrable, sporadic and

deadline exchange servers. Caccamo and Sha propose in [19] a modification to the CBS server [20] with a rule that allows locking a global critical section only when the CBS server has enough capacity to serve the whole critical section. Otherwise the capacity is recharged and the server deadline is postponed. Since the deadline could be arbitrarily postponed by a task monopolizing the CPU (*deadline aging problem*), other tasks sharing the same server could potentially need to wait for an arbitrarily large amount of time. Having "operative" deadline greater than the original ones can therefore violate the bounded-delay property, rendering the algorithm unsuitable for hierarchical open environments.

The Bandwidth Inheritance protocol (BWI) presented by Lipari et al. in [21] is based on a Priority Inheritance approach. A server holding a lock on a shared resource inherits the bandwidth of each server blocked on the same resource. The advantage of this protocol is that it does not require to know in advance which shared resources is accessed by each task, since it does not make use of the concept of "ceiling". However, knowing the worst-case critical section lengths is necessary to compute the blocking due to shared resources for admission control. The computation of the interference due to other components is rather complex. Moreover, particular strategies should be used for deadlock avoidance when designing an application.

Feng presents in [14] two mechanisms for the access to shared resources in a server-based environment. In the first one, there is one server for each globally shared resource; whenever a job of an application requires access to a global resource, the job is executed (in FIFO order) in the server associated with that resource, while the application can continue executing other jobs in the initial server. In the second approach, called "partition coalition", a further mechanism is added: each task that may access a global resource is assigned a share of bandwidth (subpartition); every time a task blocks other tasks on a global resource, it inherits the bandwidth assigned to the blocked tasks, together with the (potentially null) bandwidth of the server associated to the global resource. Global critical sections are therefore executed in a "coalition" of partitions. Both approaches have some analogies with the "multi-reserve PCP" presented by de Niz et al. in [22], where instead of serving the blocked tasks in FIFO order, a priority-based approach with resource ceilings is used to limit the blocking times. Also these works seem more suited for servers with one single task than for a hierarchical environment, with a rather complex schedulability analysis.

Among works that are closer in scope and ambition to this work, there are the solution proposed by Davis and Burns in [6], the work developed by the Real-Time group in Mälardalen [4, 5, 7], and the FIRST Scheduling Framework (FSF) [23]. Like these projects, our approach will model each individual application as a collection of sporadic tasks which may share resources. One major difference between our work and most of these related works concerns the case in which the budget is exhausted while an application is still holding a global lock. The works in [6, 23, 5, 7] introduce an overrun mechanism that continues executing the application until the lock is released, even if the budget is exhausted. The drawback of this approach is that in the scheduling analysis it is necessary to account for the largest overrun of each task/server, significantly reducing the available schedulable bandwidth. To reduce the overrun, Davis and Burns propose in [6] to execute each global critical section with local preemptions disabled. However, this imposes a larger interference locally on high priority jobs that do not access any global resource.

The approach we will follow, instead, will start executing a global critical section only if the remained budget is sufficiently large to execute the whole critical section. Otherwise, the acquisition of the global lock is delayed until the budget

can be safely recharged. As we will show, this delay does not cause any schedulability penalty in our framework, allowing a larger number of open environments to become schedulable. A similar design choice has been taken by the SIRAP protocol described in [4] for open environments based on Fixed Priority scheduling.

Another difference between the target of our work and the results presented in [6] concerns modularity. As in the objectives of the ACTORS project, we will adopt an approach wherein each application is evaluated in isolation, and integration of the applications into the open environment is done based upon only the (relatively simple) interfaces of the applications. By contrast, [6] presents a monolithic approach to the entire system, with top-level schedulability formulas that cite parameters of individual tasks from different applications. We expect that a monolithic approach is more accurate but does not scale, and is not really in keeping with the spirit of open environment design.

2.2 System Model

In an open environment, there is a shared processing platform upon which several independent applications $A_1, \ldots, A_q$ execute. We also assume that the shared processing platform is composed of a single preemptive processor (without loss of generality, we will assume that this processor has unit computing capacity), and m additional (global) shared resources which may be shared among the different applications. Each application may have additional "local" shared logical resources that are shared between different jobs within the application itself – the presence of these local shared resources is not relevant to the design and analysis of the open environment. We will distinguish between:

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

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 three parameters:

- A *virtual processor (VP) speed* α_k;
- A *jitter tolerance* Δ_k; and
- For each global shared resource R_ℓ, a *resource holding time* $H_k(R_\ell)$.

The intended interpretation of these interface parameters is as follows: *all jobs of the application will meet their deadlines if executing upon a processor of computing capacity*

α_k, with a service delay of at most Δ_k time-units, and will lock resource R_ℓ for no more than H_k time-units at a time during such execution.

We now provide a brief overview of the application interface parameters. Section 2.6 provides a more in depth discussion of the resource holding time parameter.

VP speed α_k Since each application A_k is assumed validated (i.e., analyzed for schedulability) 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 maximum service delay that A_k can withstand without missing any of its deadlines. 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. Let us represent a sporadic task [9, 8] by a 3-tuple: $(\textit{WCET, relative deadline, period})$. Consider the example application composed of the 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 no deadline is missed even when all jobs are released with a delay of two time units. 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 correlation 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.

Note that the validation process to derive the interface parameters for an application A_k does not require to effectively execute the application on a real processor of smaller speed. Such parameters might be derived from the worst-case execution times of the composing jobs on the (unit-speed) target platform, adopting proper schedulability tests associated to the local scheduling algorithm in use. More details on the validation process will be provided in Section 2.4.

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 [10, 11, 14] with the exception of the $H_k(R_\ell)$ parameter.

Resource holding times $H_k(R_\ell)$ For open environments which choose to execute all global resources disabling local preemptions (such as the design proposed in [6]), $H_k(R_\ell)$ is simply the worst-case execution time upon the VP of the longest critical section holding global resource R_ℓ. We have recently [24, 25, 26] derived algorithms for computing resource holding times when more general resource-access strategies such as the Stack Resource Policy (SRP) [2] and the Priority Ceiling Protocol (PCP) [1, 3] are instead used to arbitrate access to these global resources; in [24, 25, 26], we also discuss the issue of designing the specific application systems such that the resource holding times are decreased without compromising feasibil-

Symbol	Description
A_k	k-th application
α_k	VP speed of A_k
Δ_k	Jitter tolerance of A_k
R_ℓ	ℓ-th global resource
$H_k(R_\ell)$	A_k's resource holding time for R_ℓ
$\Pi(R_\ell)$	Global ceiling of resource R_ℓ
P_k	Period of the server associated to A_k
D_k	Deadline of the server
Z_k	Reactivation time of the server
V_k	Virtual time of the server
t_{cur}	Current time instant
$J_{k,i}$	i-th chunk of application A_k
$r(J_{k,i})$	Release time of $J_{k,i}$
$g(J_{k,i})$	Termination time of $J_{k,i}$
$d(J_{k,i})$	Deadline of $J_{k,i}$
n_k	Number of tasks composing the application A_k
τ_i	i-th task of the application
c_i	WCET of τ_i
d_i	Deadline of τ_i
p_i	Period of minimum inter-arrival time of τ_i
$h_i(R_\ell)$	τ_i's largest critical section on R_ℓ
$\pi(R_\ell)$	Local ceiling of resource R_ℓ

Figure 2.1: Notation used throughout the paper.

ity. We believe that our consideration of global shared resources — their abstraction by the H_k parameters in the interface, and the use we make of this information — is one of our major contributions, and serves to distinguish our work from other projects addressing similar topics. Our approach toward resource holding times is discussed in greater detail in Section 2.6.

The notation adopted in this document is summarized in Figure 2.1.

2.3 Algorithms

In this section, we present the algorithms used by our open environment to make admission-control and scheduling decisions. We assume that each application is characterized by the interface parameters described in Section 2.2 above. When a new application wishes to execute, it presents its interface to the *admission control algorithm*, which determines, based upon the interface parameters of this and previously-admitted applications, whether to admit this application or not. If admitted, each application is executed through a dedicated server. At each instant during run-time, the (system-level) *scheduling algorithm* decides which server (i.e., application) gets to run. If an application violates the contract implicit in its interface, an *enforcement algorithm* polices the application — such policing may affect the performance of the misbehaving application, but should not compromise the behavior of other applications.

We hereafter describe the global scheduling algorithm used by our open environment. A description and proof of correctness of our admission control algorithm will follow. The local schedulers that may be used by the individual applications will then be addressed in Section 2.4.

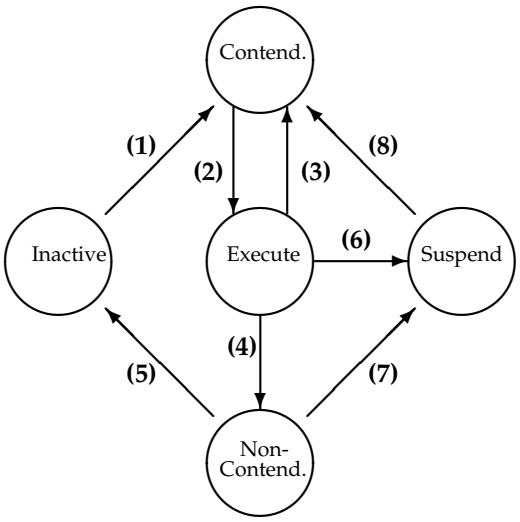

Figure 2.2: State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

System-level Scheduler

Our scheduling algorithm is essentially an application of the Constant Bandwidth Server (*CBS*) of Abeni and Buttazzo [20], enhanced to allow for the sharing of non-preemptable serially reusable resources and for the concurrent execution of different applications in an open environment. In the remainder of the paper we will refer to this server with the acronym *BROE*: Bounded-delay Resource Open Environment.

CBS-like servers have an associated *period* P_k, reflecting the time-interval at which a continuously active server replenishes its budget. For a *BROE* 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. At each instant during run-time, each server assigns a *state* to the admitted application. There are five possible states (see Figure 2.2). Let us define an application to be *backlogged* at a given time-instant if it has any active jobs awaiting execution at that instant, and *non-backlogged* otherwise.

- 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 one the *Contending*, *Executing*, or *Suspended* states[1]. While contending, it is eligible to execute; executing for more than it is eligible results in it being suspended.

These variables are updated by *BROE* according to the following rules (i)–(ix) (let t_{cur} denote the current time).

[1]Note that there is no analog of the *Suspended* state in the original definition of CBS [20].

(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:

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

(ii) At each instant, the system-level scheduling algorithm selects for execution some application A_k in the *Contending* state — the specific manner in which this selection is made is discussed below. Hence, observe that *only applications in the Contending state are eligible to execute*. An application scheduled for execution undergoes transition **(2)**. The virtual time of an executing application A_k is incremented by the corresponding server at a rate $1/\alpha_k$:

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

(iii) When an application is preempted by a higher priority one, it undergoes transition **(3)** to the *Contending* state.

(iv) 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.

(v) When $t_{\text{cur}} \geq V_k$ for some such application A_k in the *Non-Contending* state, A_k transitions back to the *Inactive* state (transition **(5)**).

(vi) If the virtual time V_k of the executing application A_k becomes equal to D_k, then application A_k undergoes transition **(6)** to the *Suspended* state. This transition is accompanied by the following actions:

$$Z_k \leftarrow V_k \tag{2.2}$$

$$D_k \leftarrow V_k + P_k \tag{2.3}$$

(vii) An application A_k in *Non-Contending* state which desires to once again contend for execution (note $t_{\text{cur}} < V_k$, otherwise it would be in the *Inactive* state) transits to the *Suspended* state (transition **(7)**). This transition is accompanied by the same actions (Eq. (2.2) and (2.3)) of transition **(6)**.

(viii) An application A_k that is in the *Suspended* state necessarily satisfies $Z_k \geq t_{\text{cur}}$. As the current time t_{cur} increases, it eventually becomes the case that $Z_k = t_{\text{cur}}$. At that instant, application A_k transits back to the *Contending* state (transition **(8)**).

(ix) An application that wishes to gain access to a shared global resource R_ℓ must perform a *budget check* (i.e., is there enough execution budget to complete execution of the resource prior to D_k?). If $\alpha_k(D_k - V_k) \geq H_k(R_\ell)$, there is sufficient budget, and the server is granted access to resource R_ℓ. Otherwise, the budget is insufficient to complete access to resource R_ℓ by D_k. In this case, transition **(6)** is undertaken by an executing application immediately prior to entering an outermost critical section locking a global resource[2] R_ℓ, updating the server parameters according to Eq. (2.2) and (2.3).

[2]Each application may have additional resources that are local in the sense that are not shared outside the application. Attempting to lock such a resource does not trigger transition **(6)**.

Rules (i) to (viii) 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 [27, 28]. The only difference from a straightforward implementation of a bounded-delay *CBS* is the deadline update of rule (vii) associated to transition (7) (which has been introduced in order to guarantee that when an application resumes execution, its relative deadline is equal to the server period, so that the budget is full) and the addition of rule (ix).

Rule (ix) has been added to deal with the problem of *budget exhaustion* when a shared resource is locked. This problem, previously described in [19] and [6], arises when an application accesses a shared resource and runs out of budget (i.e., is suspended after taking Transition (2)) before being able to unlock the resource. This would cause intolerable blocking to other applications waiting for the same lock. If there is insufficient current budget, taking transition (6) right before an application A_k locks a critical section ensures that when A_k goes to the *Contending* state (through transition (8)), it will have $D_k - V_k = P_k$. This guarantees that A_k will receive $(\alpha_k P_k)$ units of execution prior to needing to be suspended (through transition (6)). Thus, *ensuring that the WCET of each critical section of A_k is no more than $\alpha_k P_k$ is sufficient to guarantee that A_k experiences no deadline-postponement within any critical section.* Our admission control algorithm (Section 2.3) does in fact ensure that

$$H_k(R_\ell) \leq \alpha_k P_k \tag{2.4}$$

for all applications A_k and all resources R_ℓ; hence, no lock-holding application experiences deadline postponement.

At first glance, requiring that applications satisfy Condition (2.4) may seem to be a severe limitation of our framework. But this restriction appears to be unavoidable if CBS-like approaches are used as the system-level scheduler: in essence, this restriction arises from a requirement that an application not get suspended (due to having exhausted its current execution capacity) whilst holding a resource lock. To our knowledge, all lock-based multi-level scheduling frameworks impose this restriction explicitly (e.g. [19]) or implicitly, by allowing lock-holding applications to continue executing non-preemptively even when their current execution capacities are exhausted (e.g., [6, 5, 7]).

We now describe how our scheduling algorithm determines which *BROE* server (i.e., which of the applications currently in the *Contending* state) to select for execution at each instant in time.

In brief, we implement EDF among the various contending applications, with the application deadlines (the D_k's) being the deadlines under comparison. Access to the global shared resources is arbitrated using SRP[3].

In greater detail:

1. Each global resource R_ℓ is assigned a ceiling $\Pi(R_\ell)$ which is equal to the minimum value from among all the period parameters P_k of A_k that use this resource. Initially, $\Pi(R_\ell) \leftarrow \infty$ for all the resources. When an application A_k is admitted that uses global resource R_ℓ, $\Pi(R_\ell) \leftarrow \min(\Pi(R_\ell), P_k)$; $\Pi(R_\ell)$ must subsequently be recomputed when such an application leaves the environment.

[3]Recall that in our scheduling scheme, *deadline postponement cannot occur for an application while it is in a critical section* — this property is essential to our being able to apply SRP for arbitrating access to shared resources.

2. At each instant, there is a *system ceiling* which is equal to the minimum ceiling of any resource that is locked at that instant.

3. At the instant that an application A_k becomes the earliest-deadline one that is in the *Contending* state, it is selected for execution if and only if its period parameter P_k is strictly less than the system ceiling at that instant. Else, it is blocked while the currently-executing application continues to execute.

As stated above, this is essentially an implementation of EDF+SRP among the applications. The SRP requires that the relative deadline of a job locking a resource be known beforehand; that is why our algorithm requires that deadline postponement not occur while an application has locked a resource.

Admission control

The admission control algorithm checks for four things:

1. As stated in Section 2.3 above, we require that each application A_k have all its resource holding times (the $H_k(R_\ell)$'s) be $\leq \alpha_k P_k$ — any application A_k whose interface does not satisfy this condition is summarily rejected. If the application is rejected, the designer may attempt to increase the α_k parameter and resubmit the application; increasing α_k will simultaneously increase $\alpha_k P_k$ while decreasing the $H_k(R_\ell)$'s.

2. The sum of the VP speeds — the α_i parameters — of all admitted tasks may not exceed the computing capacity of the shared processor (assumed to be equal to one). Hence A_k is rejected if admitting it would cause the sum of the α_i parameters of all admitted applications to exceed one.

3. The effect of *inter-application blocking* must be considered — can such blocking cause any server to miss a deadline? A server-deadline miss occurs for a backlogged server when $t_{\text{cur}} \geq D_k$ and $V_k < D_k$. The issue of inter-application blocking is discussed in the remainder of this section.

4. The admission of an application A_k (if it satisfies the above three items) must be delayed until the first time instant during which all global resources needed by A_k are available (i.e., not locked by other applications). This delay is necessary to avoid raising a resource's priority ceiling while it is currently locked by an application with larger period than A_k's. As discussed in Theorem 1, the delay will help avoid undesirable blocking within SRP. Please note that the delay of application A_k's admittance depends upon the system reaching a lock "stasis". The length of this delay is bounded by the longest response time (elapsed total time between resource lock and release) of any critical section that accesses a resource needed by A_k. Thus, A_k is potentially delayed for a longer than usual amount of time; however, this decision prevents previously-admitted applications from being penalized with additional blocking and avoids deviation from the standard SRP policy

Admission control and *feasibility* — the ability to meet all deadlines — are two sides of the same coin. As stated above, our system-level scheduling algorithm is essentially EDF, with access to shared resources arbitrated by the SRP. Hence, the admission control algorithm needs to ensure that all the admitted applications together are feasible under EDF+SRP scheduling. We therefore looked to the EDF+SRP feasibility test in [29, 30, 31] for inspiration and ideas. In designing an admission control algorithm based upon these known EDF+SRP feasibility tests there

are a series of design decisions. Based upon the available choices, we came up with two possible admission control algorithms: a more accurate one that requires information regarding each application's resource holding time for every resource, and a slightly less accurate test that reduces the amount of information required by the system to make an admission control decision.

Prior to introducing the admission control algorithms, Section 2.3 will prove that many of the desirable properties of SRP that hold for sporadic task systems [2] continue to hold for our *BROE* server. Section 2.3 will provide useful bounds on the demand of a server. Finally, Section 2.3 will describe and prove the correctness of two admission control algorithms.

Stack-Resource Policy Properties

As mentioned at the beginning of this section, $H_k(R_\ell) \leq \alpha_k P_k$ for every global resource used by application A_k. The previous considerations allow deriving some important properties for the open environment, since there won't be any deadline postponement inside a critical section, we can view each application execution as a release sequence of "chunks" (i.e., separate jobs), as suggested in [19]. A new chunk is released each time the application enters the *Contending* state (transitions (1) and (8)) and is terminated as soon as the state transitions from *Executing* (transitions (4) and (6)). We will denote the ℓ'th chunk of application A_k as $J_{k,\ell}$. The *release time* of $J_{k,\ell}$ is denoted as $r(J_{k,\ell})$. The *termination time* of $J_{k,\ell}$ is denoted by $g(J_{k,\ell})$. Finally, the *deadline* of chunk $J_{k,\ell}$ is the D_k value of the server at the time it transitioned to contending; the deadline of chunk $J_{k,\ell}$ is represented by $d(J_{k,\ell})$. Let $V_k(t)$ denote the server's value of V_k at time t.

A *priority inversion* between applications is said to occur during run-time if the earliest-deadline application that is contending — awaiting execution — at that time cannot execute because some resource needed for its execution is held by some other application. This (later-deadline) application is said to *block* the earliest-deadline application. SRP bounds the amount of time that any application chunk may be blocked. The enforcement mechanism used in our open environment allows proving the following Theorem.

Theorem 1 (SRP properties)**.** *There are no deadlocks between applications in the open environment. Moreover, all chunks $J_{k,\ell}$ of an application A_k that doesn't exceed the declared resource-holding-time have the following properties:*

- *$J_{k,\ell}$ cannot be blocked after it begins execution.*
- *$J_{k,\ell}$ may be blocked by at most one later deadline application for at most the duration of one resource-holding-time.*

Proof. Note that admission-control property 4 of Section 2.3 ensures that resource ceilings are not raised while an application is locked. Thus, our resource-arbitration protocol is identical to SRP and the proof is identical to the proof of Theorem 6 in [2]. The only difference is that in our case the items to be scheduled are application chunks instead of jobs. □

Bounding the Demand of Server Chunks

It is useful to quantify the amount of execution that a chunk of a server requires over any given time interval. We quantify the *demand* of a server chunk, and attempt to bound the total demand (over an interval of time) by a server for A_k. The

bound on demand will be useful in the next subsection which discusses our admission control algorithms. The following are formal definitions of demand for a server chunk and the total demand for a server.

Definition 1 (Demand of Server Chunk $J_{k,\ell}$)**.** *The* demand $W(J_{k,\ell}, t_1, t_2)$ *of server chunk* $J_{k,\ell}$ *over the interval* $[t_1, t_2]$ *is the amount of execution that* $J_{k,\ell}$ *(with deadline and release time in the interval* $[t_1, t_2]$*) must receive before making a transition from* Executing *to* Non-Contending *or* Suspended*. Formally,* $W(J_{k,\ell}, t_1, t_2)$ *is equal to*

$$\begin{cases} \alpha_k \left(V_k(g(J_{k,\ell})) - V_k(r(J_{k,\ell})) \right), & if\ (r(J_{k,\ell}) \geq t_1) \wedge \\ & (g(J_{k,\ell}) < d(J_{k,\ell}) \leq t_2); \\ \alpha_k P_k, & if\ (r(J_{k,\ell}) \geq t_1) \\ & \wedge (d(J_{k,\ell}) \leq t_2) \\ & \wedge (g(J_{k,\ell}) \geq d(J_{k,\ell})); \\ 0, & otherwise. \end{cases} \tag{2.5}$$

Definition 2 (Cumulative Demand of *BROE* Server for A_k)**.** *The* cumulative demand *of* A_k *over the interval* $[t_1, t_2]$ *is the total demand of all of* A_k*'s server chunks with both release times and deadlines within the interval* $[t_1, t_2]$*:*

$$W(A_k, t_1, t_2) \doteq \sum_{\ell \geq 1} W(J_{k,\ell}, t_1, t_2) \tag{2.6}$$

Different chunks of the same server may execute for different amounts of time. The reason is that some chunks may terminate early due to becoming non-contending or trying to enter a critical section (i.e., rule (iv) or (ix)). For these chunks, the execution they receive may be less than $\alpha_k P_k$. Unfortunately, there are infinitely many possible application execution scenarios over any given interval (resulting in different sequences of state transitions). With all these possibilities, how does one determine the cumulative demand of A_k over any interval? Fortunately, we may, in fact, derive upper bounds for the cumulative demand of a server for specific sequences of chunks. The upper bound for these sequences will be used in proof of correctness for the admission control algorithm. In the remainder of this subsection, we will present a series of lemmas that derives the upper bound on the cumulative demand of a sequence of server chunks.

The first lemma states that the virtual time V_k of a server cannot exceed the deadline parameter D_k.

Lemma 1. *For all chunks* $J_{k,\ell}$ *of* BROE *server of* A_k*,*

$$V_k(g(J_{k,\ell})) \leq d(J_{k,\ell}) \tag{2.7}$$

Proof. Observe that rule (vi) implies that the virtual time V_k does not exceed the current server deadline D_k. Therefore, whenever any chunk $J_{k,\ell}$ is terminated (via transitions (4), or (6)) at time $g(J_{k,\ell})$, the server's virtual time $V_k(g(J_{k,\ell}))$ does not exceed the deadline $d(J_{k,\ell})$ of the chunk. □

The next lemma formally states that, when a chunk terminates with transition (6), the virtual time does not increase until the release of the next chunk.

Lemma 2. *If* $J_{k,\ell+1}$ *was released due to transition (8) (i.e.,* Suspended *to* Contending*), then* $V_k(r(J_{k,\ell+1})) = V_k(g(J_{k,\ell}))$*.*

Proof. If $J_{k,\ell+1}$ was released due to transition (8), the transition prior to (8) must have been either (6), or the successive transitions of (4) and (7). For these transitions, one of the server rules (iv), (vi), (vii), or (ix) applies when terminating the

previous chunk $J_{k,\ell}$. However, notice that none of these rules updates V_k, and since virtual time cannot progress unless the server is contending, the virtual time at the release of $J_{k,\ell+1}$ (i.e., $V_k(r(J_{k,\ell+1}))$) must equal the virtual time at the termination of $J_{k,\ell}$ (i.e., $V_k(g(J_{k,\ell}))$). □

In the final lemma of this subsection, we consider any sequence of chunks where the server does not become inactive in between releases and the virtual time at the release of the first chunk equals actual time. For such a sequence of chunks, we show that the demand of the chunks from the release time of the first chunk of the sequence to the deadline of the last chunk of the sequence does not exceed α_k times the sequence length (i.e., the deadline of the last chunk minus release time of the first chunk).

Lemma 3. *If $J_{k,\ell}, J_{k,\ell+1}, \ldots, J_{k,s}$ is a sequence of successively released chunks by the* BROE *server for A_k where $J_{k,\ell}$ satisfies $V_k(r(J_{k,\ell})) = r(J_{k,\ell})$, and $J_{k,\ell+1}, \ldots, J_{k,s}$ were all released due to transition (8); if $J_{k,\ell}, \ldots, J_{k,s-1}$ meet their deadline, then*

$$W(A_k, r(J_{k,\ell}), d(J_{k,s})) \leq \alpha_k(d(J_{k,s}) - r(J_{k,\ell})) \tag{2.8}$$

Proof. According to Definition 2, $W(A_k, r(J_{k,\ell}), d(J_{k,s}))$ represents the sum of the $W(J_{k,i}, r(J_{k,\ell}), d(J_{k,s}))$ for each chunk $J_{k,i}$ where $\ell \leq i \leq s$. Since both $r(J_{k,i})$ and $d(J_{k,i})$ must be included in the interval $[r(J_{k,\ell}), d(J_{k,s})]$ and chunks $J_{k,\ell}, \ldots, J_{k,s-1}$ meet their deadlines, Equation (2.5) implies

$$\begin{aligned} &W(A_k, r(J_{k,\ell}), d(J_{k,s})) = \\ &W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \sum_{i=\ell}^{s-1} \alpha_k \left(V_k(g(J_{k,i})) - V_k(r(J_{k,i}))\right). \end{aligned} \tag{2.9}$$

Since chunks $J_{k,\ell+1}, \ldots, J_{k,s}$ are released due to transition (8), Lemma 2 implies that $V_k(r(J_{k,i+1})) = V_k(g(J_{k,i}))$ for all $\ell \leq i < s-1$. Substituting this into Equation 2.9,

$$\begin{aligned} &W(A_k, r(J_{k,\ell}), d(J_{k,s})) = \\ &W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \sum_{i=\ell}^{s-1} \alpha_k \left(V_k(r(J_{k,i+1})) - V_k(r(J_{k,i}))\right). \end{aligned} \tag{2.10}$$

By the telescoping summation above, it may be shown that $W(A_k, r(J_{k,\ell}), d(J_{k,s}))$ equals $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \alpha_k(V_k(r(J_{k,s})) - V_k(r(J_{k,\ell})))$. By the antecedent of the lemma, $V_k(r(J_{k,\ell}))$ equals $r(J_{k,\ell})$. Thus,

$$\begin{aligned} &W(A_k, r(J_{k,\ell}), d(J_{k,\ell})) = \\ &W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \alpha_k \left(V_k(r(J_{k,s})) - r(J_{k,\ell})\right). \end{aligned} \tag{2.11}$$

It remains to determine $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ which is dependent on whether $J_{k,s}$ meets its deadline. If $J_{k,s}$ meets its deadline, then $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ equals $\alpha_k(V_k(g(J_{k,s})) - V_k(r(J_{k,s}))$. Lemma 1 implies that $V_k(g(J_{k,s})) \leq d(J_{k,s})$; therefore, $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ does not exceed $\alpha_k(d(J_{k,s}) - V_k(r(J_{k,s})))$. Combining this fact and Equation (2.11) implies Equation (2.8) of the lemma. Therefore, the lemma is satisfied when $J_{k,s}$ meets its deadline.

Now, consider the case in which chunk $J_{k,s}$ misses its deadline. By Definition 1, $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ equals $\alpha_k P_k$. Observe that by antecedent of the lemma, $J_{k,s}$ is

released due to transition (8); either rule (vi), (vii), or (ix) will be used to set $d(J_{k,s})$. Each of these rules sets $d(J_{k,s}) = V_k(g(J_{k,s-1})) + P_k \Rightarrow P_k = d(J_{k,s}) - V_k(g(J_{k,s-1}))$. Substituting the value of P_k and observing by Lemma 2 that $V_k(g(J_{k,s-1})$ equals $V_k(r(J_{k,s}))$, we derive $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ equals $\alpha_k(d(J_{k,s}) - V_k(g(J_{k,s-1})))$. Finally, substituting the new expression for $W(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ into Equation (2.11) and canceling terms gives us Equation (2.8) of the lemma. Thus, the lemma is also satisfied when $J_{k,s}$ misses its deadline. □

Admission Control Algorithms

Adapting the proofs from the EDF+SRP feasibility tests in [31] and [2] for the case where application chunks, instead of jobs, are the items to be scheduled, we can find a direct mapping relation between resource-holding-times of applications and critical section lengths of jobs. The maximum blocking experienced by $J_{k,\ell}$ is then:

$$B_k = \max_{P_j > P_k} \{H_j(R_\ell) | \exists H_x(R_\ell) \neq 0 \wedge P_x \leq P_k\} \tag{2.12}$$

In other words, the maximum amount of time for which $J_{k,\ell}$ can be blocked is equal to the maximum resource-holding-time among all applications having a server period $> P_k$ and sharing a global resource with some application having a server period $\leq P_k$. The following test may be used when the admission control algorithm has information from each application A_k on which global resources R_ℓ are accessed and what the value of $H_k(R_\ell)$ is.

Theorem 2. *Applications $A_1, \ldots, A_q$ may be composed upon a unit-capacity processor together without any server missing a deadline, if*

$$\forall k \in \{1, \ldots, q\} \; : \; \sum_{P_i \leq P_k} \alpha_i + \frac{B_k}{P_k} \leq 1 \tag{2.13}$$

where the blocking term B_k is defined in Equation (2.12).

Proof. This test is similar to the EDF+SRP feasibility tests in [31] and [2], substituting jobs and critical section lengths with, respectively, application chunks and resource-holding-times. We prove the contrapositive of the theorem.

Assume that the first deadline miss for some server chunk occurs at time t_{miss}. Let t' be the latest time prior to t_{miss} such that there is no executing (and, therefore, no contending) application with deadline before t_{miss}; since there exists an executing server from t' to t_{miss}, the processor is continuously busy in the interval $[t', t_{miss}]$. Observe that t' is guaranteed to exist at system-start time. The total demand imposed by server chunks in $[t', t_{miss}]$ is defined as the sum of the execution costs of all chunks entirely contained in that interval, i.e., $\sum_{i=1}^{q} W(A_i, t', t_{miss})$.

We will now show that the demand of any application A_k does not exceed $\alpha_k(t_{miss} - t')$. Let $Y \doteq \{J_{k,\ell}, \ldots, J_{k,s}\}$ be the set of server chunks that the server for A_k releases in the interval $[t', t_{miss}]$ with deadlines prior or equal to t_{miss}. If Y is empty, then the demand trivially does not exceed $\alpha_k(t_{miss} - t')$; so, assume that Y is non-empty. Since the server for A_k is not in the *Executing* (or *Contending*) state immediately prior t', it is either in the *Non-Contending*, *Inactive*, or *Suspended* state for a non-zero-length time interval prior to t' (note this disallows the instantaneous transition of (6) and (8)); therefore, the first chunk of Y must have been generated due to either transition (1) or (8), in which case either rule (i) or rule (viii) apply. Thus $J_{k,\ell}$ is the first chunk in Y, $V_k(r(J_{k,\ell})) = r(J_{k,\ell})$. We may thus partition Y into p disjoint subsequences of *successively generated* chunks $Y^{(1)}, Y^{(2)}, \ldots, Y^{(p)}$ where $Y^{(i)} \doteq$

$\{J_{k,\ell_i}^{(i)}, \ldots, J_{k,s_i}^{(i)}\}$. For each $Y^{(i)}$, $J_{k,\ell_i}^{(i)}$ has $V_k(r(J_{k,\ell_i}^{(i)}))$ equal to $r(J_{k,\ell_i}^{(i)})$, and $J_{k,\ell_i+1}^{(i)}, \ldots, J_{k,s_i}^{(i)}$ are all released due to transition (8). Observe the chunks of each subsequence $Y^{(i)}$ span the interval $[r(J_{k,\ell_i}^{(i)}), d(J_{k,s_i}^{(i)})] \subseteq [t', t_{miss}]$. By Lemma 3, the demand of A_k over the subinterval $[r(J_{k,\ell_i}^{(i)}), d(J_{k,s_i}^{(i)})]$ does not exceed $\alpha_k(d(J_{k,s_i}^{(i)}) - r(J_{k,\ell_i}^{(i)}))$. Furthermore, the server for A_k does not execute in intervals not covered by the chunks of some subsequence $Y^{(i)}$. Since $Y^{(1)}, Y^{(2)}, \ldots, Y^{(p)}$ is a partition of Y, the subintervals do not overlap; this implies that $\sum_{i=1}^{p}(d(J_{k,s_i}^{(i)}) - r(J_{k,\ell_i}^{(i)})) \leq (t_{miss} - t')$. Therefore the total demand of A_k over interval $[t', t_{miss}]$ does not exceed $\alpha_k \left[\sum_{i=1}^{p}(d(J_{k,s_i}^{(i)}) - r(J_{k,\ell_i}^{(i)}))\right] \leq \alpha_k(t_{miss} - t')$.

Notice that only applications with period less than $(t_{miss} - t')$ can release chunks inside the interval: since any application A_k is not backlogged at time t', the first chunk released after t' will have deadline at least $t' + P_k$. Let A_k be the application with the largest $P_k \leq (t_{miss} - t')$. For Theorem 1, at most one server chunk with deadline later than t_{miss} can execute in the considered interval. Therefore only one application with period larger than P_k can execute, for at most the length of one resource-holding-time, inside the interval. The maximum amount of time that an application with period larger than P_k can execute in $[t', t_{miss}]$ is quantified by B_k.

Since some server chunk missed a deadline at time t_{miss}, the demand in $[t', t_{miss}]$, plus the blocking term B_k as defined in Equation 2.12, must exceed $(t_{miss} - t')$:

$$\sum_{P_i \leq P_k} (t_{miss} - t')\alpha_i + B_k \geq (t_{miss} - t') \tag{2.14}$$

Dividing by $(t_{miss} - t')$, and then observing that $(t_{miss} - t') \geq P_k$, we have:

$$\sum_{P_i \leq P_k} \alpha_k + \frac{B_k}{P_k} \geq 1 \tag{2.15}$$

which contradicts Equation 2.13. □

However, such admission control test based on a policy of considering all resource usages (as the theorem above) has drawbacks. One reason is that it requires the system to keep track of each application's resource-hold times. An even more serious drawback is how to fairly account for the "cost" of admitting an application into the open environment. For example, an application that needs a VP speed twice that of another should be considered to have a greater cost (all other things being equal); considered in economic terms, the first application should be "charged" more than the second, since it is using a greater fraction of the platform resources and thus having a greater (adverse) impact on the platform's ability to admit other applications at a later point in time.

But in order to measure the impact of global resource-sharing on platform resources, we need to consider the resource usage of not just an application, but of all other applications in the systems. Consider the following scenario. If application A_1 is using a global resource that no other application chooses to use, then this resource usage has no adverse impact on the platform. Now if a new application A_2 with a very small period parameter that needs this resource seeks admission, the impact of A_1's resource-usage becomes extremely significant (since A_1 would, according to the SRP, block A_2 and also all other applications that have a period parameter between A_1's and A_2's). So how should we determine the cost of A_1's use of this resource, particularly if we do not know beforehand whether or not A_2 will request admission at a later point in time?

To sidestep the dilemma described above, we believe a good design choice is to effectively *ignore* the exact resource-usage of the applications in the online setting, instead considering only the maximum amount of time for which an application may choose to hold any resource; also, we did not consider the identity of this resource. That is, we required a simpler interface than the one discussed in Section 2.2, in that rather than requiring each application to reveal its maximum resource-holding times on all m resources, we only require each application A_k to specify a *single* resource-holding parameter H_k, which is defined as follows:

$$H_k \doteq \max_{\ell=1}^{m} H_k(R_\ell) \tag{2.16}$$

The interpretation is that A_k may hold *any* global resource for up to H_k units of execution. With such characterization of each application's usage of global resources, we ensure that we do not admit an application that would unfairly block other applications from executing due its large resource usage. This test, too, is derived directly from the EDF+SRP feasibility test of Theorem 2, and is as follows:

ALGORITHM ADMIT($A_k = (\alpha_k, P_k, H_k)$)

1 **for** each A_i: $(P_i \geq P_k)$ **do**
2 **if** $\max_{j:P_j>P_i} H_j > P_i(1 - \sum_{j:P_j \leq P_i} \alpha_j)$
return "reject"
3 **for** each A_i: $(P_i < P_k)$ **do**
4 **if** $H_k > P_i(1 - \sum_{P_j \leq P_i} \alpha_j)$
return "reject"
5 **return** "admit"

It follows from the properties of the SRP, (as proved in [2]) that the new application A_k, if admitted, may *block* the execution of applications A_i with period parameter $P_i < P_k$. Moreover, by Theorem 2, it may *interfere* with applications A_i with period parameter $P_i \geq P_k$. Since the maximum amount by which any application A_j with $P_j > P_i$ may block an application A_i is equal to H_j, lines 1-2 of Procedure ALGORITHM ADMIT determine whether this blocking can cause any application A_i with $P_i \geq P_k$ to miss its deadline. Similarly, since the maximum amount by which application A_k may block any other application is, by definition of the interface, equal to H_k, lines 3-4 of ALGORITHM ADMIT determine whether A_k's blocking causes any other application with $P_i < P_k$ to miss its deadline. If the answer in both cases is "no," then ALGORITHM ADMIT admits application A_k in line 5.

Enforcement

One of the major goals in designing open environments is to provide inter-application *isolation* — all other applications should remain unaffected by the behavior of a misbehaving application. By encapsulating each application into a *BROE* server, we provide the required isolation, enforcing a correct behavior for every application.

Using techniques similar to those used to prove isolation properties in *CBS*-like environments (see, e.g., [20, 27]), it can be shown that our open environment does indeed guarantee inter-application isolation in the absence of resource-sharing. It remains to study the effect of resource-sharing on inter-application isolation.

Clearly, applications that share certain kinds of resources cannot be completely isolated from each other: for example if one application corrupts a shared data-structure then all the applications sharing that data structure are affected. When a

resource is left in an inconsistent state, one option could be to inflate the resource-holding time parameters with the time needed to reset the shared object to a consistent state, when there is such a possibility.

However, we believe that it is rare that truly independently-developed applications share "corruptible" objects — good programming practice dictates that independently-developed applications do not depend upon proper behavior of other applications (and in fact this is often enforced by operating systems). Hence the kinds of resources we expect to see shared between different applications are those that the individual applications cannot corrupt. In that case, the only misbehavior of an application A_k that may affect other applications is if it holds on to a global resource for greater than $\alpha_k P_k$, or than the H_k time units of execution that it had specified in its interface. To prevent this, we assume that our enforcement algorithm simply preempts A_k after it has held a global resource for $\min\{H_k, \alpha_k P_k\}$, and ejects it from the shared resource. This may result in A_k's internal state getting compromised, but the rest of the applications are not affected. Note that such an enforcement algorithm might require to set a timer each time a resource is locked, increasing the system overhead. We believe this is the minimum price to pay for guaranteeing temporal isolation among applications that share global resources. Applications that repeatedly access global resources will need to be charged with a larger overhead, unless performing multiple accesses within a single lock.

When applications do share corruptible resources, we have argued above that isolation is not an achievable goal; however, *containment* [22] is. The objective in containment is to ensure that the only applications affected by a misbehaving application are those that share corruptible global resources with it — the intuition is that such applications are not truly independent of each other. We have strategies for achieving some degree of containment; for instance, one option could be to donate to an application locking a corruptible resource the bandwidth of applications that share the same resource. However, discussion of these strategies is beyond the scope of this document.

Bounded delay property

The bounded-delay resource partition model, introduced by Mok et al. [10], 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.

Definition 4. *A* bounded-delay server *is a server that implements a bounded-delay partition.*

We will show that when every application is admitted through a proper admission control test, *BROE* implements a bounded delay partition. Before proving this property, we need some intermediate lemmas. The first lemma quantifies the minimum virtual-time V_k for a server for application A_k that is in the *Contending* or *Executing* state.

Lemma 4. *Given* BROE *servers of applications $A_1, \ldots, A_q$ satisfying Theorem 2, if server chunk $J_{k,\ell}$ of server A_k is executing or contending at time t (where $r(J_{k,\ell}) \leq t \leq d(J_{k,\ell})$), then*

$$V_k(t) \geq V_k(r(J_{k,\ell})) + \frac{\max\left(0, t - V_k(r(J_{k,\ell})) - P_k(1-\alpha_k)\right)}{\alpha_k}. \tag{2.17}$$

Proof. The proof is by contradiction. Assume that all servers have been admitted to the open environment via Theorem 2, but there exists a server A_k in the *Contending* or *Executing* state at time t that has

$$V_k(t) < V_k(r(J_{k,\ell})) + \frac{\max\left(0, t - V_k(r(J_{k,\ell})) - P_k(1-\alpha_k)\right)}{\alpha_k}. \tag{2.18}$$

Since V_k never decreases, the above strict inequality implies that

$$t > V_k(r(J_{k,\ell})) + P_k(1-\alpha_k). \tag{2.19}$$

We will show that if Equation (2.18) holds, there exist a legal scenario under which A_k will miss a server deadline. Assume that application A_k has $\alpha_k P_k$ units of execution backlogged at time $r(J_{k,\ell})$ (the server can be in any state immediately prior to $r(J_{k,\ell})$); also assume that no job of application A_k requests any global resources during the next $\alpha_k P_k$ units of A_k's execution (i.e., transition (6) will not be used). The described scenario is a legal scenario for application A_k with parameter α_k and Δ_k.

Note that each of the server deadline update rules essentially sets D_k equal to $V_k + P_k$; therefore, $d(J_{k,\ell})$ equals $V_k(r(J_{k,\ell})) + P_k$. The current time remaining until $J_{k,\ell}$'s deadline is $V_k(r(J_{k,\ell})) + P_k - t$. The virtual time of A_k at time t by Equations (2.18) and (2.19) satisfies the following inequality:

$$V_k(t) < V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot \left(t - V_k(r(J_{k,\ell})) - P_k(1-\alpha_k)\right). \tag{2.20}$$

The remaining amount of time at time t that the server for A_k must execute for V_k to equal $d(J_{k,\ell})$ (i.e., complete its execution) is $\alpha_k(V_k(r(J_{k,\ell})) + P_k - V_k(t))$. Combining this expression with Equation (2.20), the remaining execution time is strictly greater than $V_k(r(J_{k,\ell})) + P_k - t$. However, this exceeds the remaining time to the deadline; since the server for A_k is continuously in the *Contending* or *Executing* state throughout this scenario, the server will miss a deadline at $d(J_{k,\ell})$. This contradicts the lemma, given that the servers satisfied Theorem 2. Our original supposition of Equation (2.18) is falsified and the lemma follows. □

We next show that at any time a server chunk is released for A_k, the actual time must exceed the virtual time.

Lemma 5. *For any server chunk $J_{k,\ell}$ of the* BROE *server for A_k,*

$$V_k(r(J_{k,\ell})) - r(J_{k,\ell}) \leq 0 \tag{2.21}$$

Proof. The lemma may be proved by analyzing each of the server rules involved in moving the server state to *Contending*. If the server state for A_k is inactive prior to the release of chunk $J_{k,\ell}$, then rule (i) sets V_k to current time, and the lemma is satisfied. If the server was suspended immediately prior to the release of $J_{k,\ell}$, rule (viii) releases $J_{k,\ell}$ only when Z_k equals t_{cur}. Observe that all the rules of the server set Z_k to a value greater than or equal to V_k. Thus, $V_k(r(J_{k,\ell})) - r(J_{k,\ell})$ is either zero or negative. □

The final lemma before proving that *BROE* is a bound-delay server, shows that for any server the absolute difference between virtual time and actual time is bounded in terms of the server parameters.

Lemma 6. *For an application A_k admitted in the open environment, if the server for A_k is backlogged at time t, then*

$$|V_k(t) - t| \leq P_k(1 - \alpha_k) \tag{2.22}$$

Proof. If the server for A_k is in the *Suspended* state, then because the server is backlogged this implies that $V_k(t) - t > 0$; so, the server will not become contending until time V_k. Let t' be the last time prior to t that the server was contending or executing; it's easy to see that $V_k(t') - t' > V_k(t) - t$. So, we will reason about t' and show for any such contending or executing time $V_k(t') - t' \leq P_k(1 - \alpha_k)$. If the server for A_k was contending or executing at time t, let t' instead equal t. We will show in the remainder of the proof that $-P_k(1 - \alpha_k) \leq V(t') - t' \leq P_k(1 - \alpha_k)$. Let $J_{k,\ell}$ be the server chunk corresponding to the last contending or executing state at t' for application A_k. Observe that this implies that $t' \geq r(J_{k,\ell})$.

Let us first show that $V(t') - t' \leq P_k(1 - \alpha_k)$. Observe that because virtual time progresses at a rate equal to $1/\alpha_k$ and cannot exceed $D_k = V_k(r(J_{k,\ell})) + P_k = V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot (\alpha_k P_k)$,

$$V_k(t') \leq V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot \min\left(t' - r(J_{k,\ell}), \alpha_k P_k\right). \tag{2.23}$$

Subtracting t' from both sides, and observing that the RHS is maximized for $t' = r(J_{k,\ell}) + \alpha_k P_k$ (since $t' \geq r(J_{k,\ell})$, the "min" term increases at a rate of $\frac{1}{\alpha_k}$ in $[r(J_{k,\ell}), r(J_{k,\ell}) + \alpha_k P_k]$, and remains constant for all $t > r(J_{k,\ell}) + \alpha_k P_k$) implies

$$V_k(t') - t' \leq V_k(r(J_{k,\ell})) + P_k - r(J_{k,\ell}) - \alpha_k P_k. \tag{2.24}$$

Lemma 5 implies that $V_k(r(J_{k,\ell})) - r(J_{k,\ell}) \leq 0$. Thus, Equation (2.24) may be written as $V_k('t) - t' \leq P_k - \alpha_k P_k = P_k(1 - \alpha_k)$, proving the upper bound on $V_k(t') - t'$ (and thus an upper bound on $V(t) - t$).

We will now prove a lower bound on $V(t') - t'$. By Lemma 4 and the fact that the server is contending or executing at time t', we have a lower bound on the virtual time at t':

$$\begin{aligned} V_k(t') - t' \geq\ & V_k(r(J_{k,\ell})) + \\ & \frac{1}{\alpha_k} \cdot \max\left(0, t' - V_k(r(J_{k,\ell})) - P_k(1 - \alpha_k)\right) - t'. \end{aligned} \tag{2.25}$$

Observe that for all $t' : r(J_{k,\ell}) \leq t' \leq V_k(r(J_{k,\ell})) + P_k(1 - \alpha_k)$, the "max" term in the RHS of the above expression is zero; thus, the RHS decreases in t' from $r(J_{k,\ell})$ to $V_k(r(J_{k,\ell})) + P_k(1 - \alpha_k)$. For t' greater than $V_k(r(J_{k,\ell})) + P_k(1 - \alpha_k)$, the "max" term increases at a rate of $\frac{1}{\alpha_k}$. Therefore, the RHS of the above inequality is minimized when t' equals $V_k(r(J_{k,\ell})) + P_k(1 - \alpha_k)$. Thus, $V_k('t) - t' \geq -P_k(1 - \alpha_k)$, proving the lower bound on $V(t) - t$; the lemma follows.

□

We are now ready to prove that *BROE* implements a bounded-delay partition.

Theorem 3 (Bounded-delay property)**.** BROE *is a bounded-delay server.*

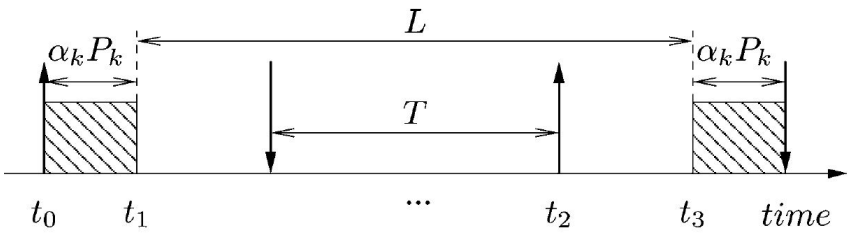

Figure 2.3: Worst case scenario discussed in the proof of Theorem 3. The application receives execution during the shaded intervals.

Proof. From the definition of the *BROE* server, it can be seen that the virtual time V_k is updated only when an inactive application goes active or whenever subsequently it is executing. In the latter case, V_k is incremented at a $1/\alpha_k$ rate. Thus, there is a direct relation between the execution time allocated to the application through the *BROE* server and the supply the application would have received if scheduled on a Virtual Processor of speed α_k. The quantity $V_k(t) - t$ indicates the advantage the application A_k executing through the *BROE* server has compared with VP in terms of supply. If the above term is positive, the application received more execution time than the VP would have by time t. If it is negative, the *BROE* server is "late".

From Lemma 6, the execution time supplied to an application through a dedicated *BROE* server never exceeds nor is exceeded by the execution time it would have received on a dedicated VP for more than $P_k(1 - \alpha_k)$ time units. The "worst case" is when both displacements happen together, i.e., interval T starts when $V_k(t) - t = P_k(1 - \alpha_k)$ and ends when $V_k(t) - t = -P_k(1 - \alpha_k)$. This interval in which the *BROE* server can be delayed from executing while still satisfying the bound on $|V(t) - t|$ from Lemma 6 is of length at most twice $P_k(1 - \alpha_k)$. By the definition of P_k (Equation 2.1), this is equal to Δ_k. Thus, the maximum delay that an application executing on a *BROE* server may experience is Δ_k.

In other words, it can be shown that the "worst case" (see Figure 2.3) occurs when application A_k

- receives execution immediately upon entering the *Contending* state (at time t_0 in the figure), and the interval of length L begins when it completes execution and undertakes transition (6) to the *Suspended* state (at time t_1 in the figure); and

- after having transited between the *Suspended*, *Contending* and *Executing* states an arbitrary number of times, undertakes transition (8) to enter the *Contending* state (time t_2 in the figure) at which time it is scheduled for execution (transition (2)) as late as possible; the interval ends just prior to A_k being selected for execution (time t_3 in the figure).

A job arriving at time t_1 will be served by the *BROE* with the maximum delay of Δ_k from the supply granted by a VP of speed α_k. Since the execution received in an interval T going from the deadline of the first chunk (released at t_0) until the release time of the last one at t_2 cannot be higher than $\alpha_k T$, the supply granted over interval L is $\alpha_k T = (L - 2P_k(1 - \alpha_k))\alpha_k$. By the definition of P_k (Equation (2.1)), this is equal to $(L - \Delta_k)\alpha_k$. □

2.4 Application-level schedulers

In the previous section we analyzed how to compose multiple servers on the same processor without violating the bounded-delay server constraints. Provided that these *global* constraints are met, we now address the *local* schedulability problem, to verify if a collection of jobs composing an application can be scheduled on a bounded delay server with given α_k and Δ_k, when jobs can share exclusive resources with other applications.

To do this, we have three options on how to schedule and validate the considered collection of jobs:

1. Validate the application on a dedicated Virtual Processor with speed α_k using a given scheduling algorithm. If every job is completed at least Δ_k time-units before its deadline, then the application is schedulable on a bounded-delay partition $(\alpha_k \Delta_k)$ when jobs are scheduled according to the same order as they would on a dedicated VP schedule.

2. Validate the application on a dedicated Virtual Processor with speed α_k using EDF. If every job is completed at least Δ_k time-units before its deadline, then the application is schedulable with EDF on a bounded-delay partition $(\alpha_k \Delta_k)$, without needing to "copy" the VP schedule.

3. Validate the application by analyzing the execution time effectively supplied by the partition in the worst-case and the demand imposed by the jobs scheduled with any scheduling algorithm, avoiding validation on a VP.

These options are hereafter explained in more detail.

Replicating the Virtual Processor scheduling

When scheduling a set of applications on a shared processor, there is sometimes the need to preserve the original scheduling algorithm with which an application has been conceived and validated on a slower processor. If this is the case, we need to guarantee that all jobs composing the application will still be schedulable on the bounded-delay partition provided by the open environment through the associated *BROE* server. Mok et al. [10, 11] have previously addressed this problem. We restate their result, adapting it to the notation used so far.

Theorem 4. *[10, Theorem 6] Given an application A_k and a Bounded Delay Partition (α_k, Δ_k), let S_n denote a valid schedule on a Virtual Processor with speed α_k, and S_p the schedule of A_k on Partition (α_k, Δ_k)* according to the same execution order and amount *as S_n. Also let $\bar{\Delta}_k$ denote the largest amount of time such that any job of S_n is completed at least $\bar{\Delta}_k$ time units before its deadline. S_p is a valid schedule if and only if $\bar{\Delta}_k \geq \Delta_k$.*

The theorem states that all jobs composing an application are schedulable on a *BROE* server having α_k equal to the VP speed and Δ_k equal to the jitter tolerance of the VP schedule, provided that jobs are executed in the *same execution order* of the VP schedule.

In order to be applicable to general systems, this approach would require that each individual application's scheduling event (job arrivals and completions) be "buffered" during the delay bound Δ_k — essentially, an event at time t_o is ignored until the earliest time-instant when $V(t) \geq t$ — so that events are processed in the same order in the open environment as they would be if each application were running upon its dedicated virtual processor. However we will see that such buffering

is unnecessary when the individual application can be EDF-scheduled in the open environment.

Application-Level Scheduling using EDF

To avoid the complexity of using buffers to keep track of the scheduling events, it is possible to use a simplified approach. When an application doesn't mandate to be scheduled with a particular scheduling algorithm, we show that EDF can be optimally used as application-level scheduler for the partition, without needing to "copy" the virtual processor behavior. To distinguish the buffered from the native version of the partition local scheduler, we will call VP-EDF the application-level scheduler reproducing the virtual processor behavior, while the normal local scheduler using only jobs earliest deadlines will be simply called EDF.

Definition 5. *A scheduling algorithm is* resource-burst-robust *if advancing the supply, the schedulability is preserved.*

Lemma 7 (from Feng [14])**.** *EDF is resource-burst-robust.*

Lemma 8. *Consider an application A_k, composed by a set of jobs with fixed release times and deadlines. If all jobs of application A_k always complete execution at least Δ_k time units prior to their deadlines when scheduled with EDF upon a dedicated VP of computing capacity α_k, then all jobs of A_k are schedulable with EDF on a partition (α_k, Δ_k).*

Proof. We prove the contrapositive. Assume a collection of jobs of an application A_k completes execution at least Δ_k time units prior to their deadlines when scheduled with EDF on a dedicated α_k-speed VP, but some of these jobs miss a deadline when A_k is scheduled with EDF on a partition (α_k, Δ_k). Let t_{miss} be the first time a deadline is missed and let t_s denote the latest time-instant prior to t_{miss} at which there are no jobs with deadline $\leq t_{miss}$ awaiting execution in the partition schedule ($t_s \leftarrow 0$ if there was no such instant). Hence over $[t_s, t_{miss})$, the partition is only executing jobs with deadline $\leq t_{miss}$, or jobs that were *blocking* the execution of jobs with deadline $\leq t_{miss}$. Let Y be the set of such jobs.

Since a deadline is missed, the total amount of demand of jobs in Y during $[t_s, t_{miss})$ upon the *BROE* server is greater than the execution time supplied in the same interval. From Lemma 7, we know that the minimum amount of execution A_k would receive in interval $[t_s, t_{miss})$, is $\alpha_k((t_{miss} - t_s) - \Delta_k)$.

Consider now the VP schedule. Since every job completes at least Δ_k time-units before its deadline, the job that misses its deadline in the partition schedule will complete before instant $t_{miss} - \Delta_k$ in the VP schedule. Moreover, since EDF always schedules tasks according to their absolute deadline, no jobs in Y will be scheduled in interval $[t_{miss} - \Delta_k, t_{miss}]$. Therefore, the total demand of jobs in Y during $[t_s, t_{miss})$ does not exceed $\alpha_k((t_{miss} - \Delta_k) - t_s)$. However, this contradicts the fact that the minimum amount of execution that is provided by the *BROE* server over this interval is $\alpha_k((t_{miss} - \Delta_k) - t_s)$. □

Lemma 8 is a stronger result than the one in [10, Corollary 4], where applications needed to be scheduled according to VP-EDF. In [14, Theorem 2.7] a more general result is proved, saying that any resource-burst-robust scheduler can be used without needing to reproduce the VP schedule. However there is a flaw in this result; for instance, even though DM is a resource-burst-robust scheduler, it cannot be used without buffering events. To see this, consider the following example.

Example 1. *An application composed of two periodic tasks* $\tau_1 = (1, 6, 4)$ *and* $\tau_2 = (1, 6, 6)$ *is validated on a processor of speed* $\alpha_k = 1/2$*. Each job is completed at least* $\Delta_k = 2$ *time units prior to its deadline. However, when* A_k *is scheduled on a bounded-delay partition* $(\alpha_k, \Delta_k) = (1/2, 2)$*, it can miss a deadline when both* τ_1 *and* τ_2 *release jobs at time t and the server contemporarily exhausts its budget. The application may have to wait until time* $t + \Delta = t + 2$ *to receive service, at which point* τ_1 *executes in* $[t + 2, t + 3)$ *(exhausting the budget). The next service interval is* $[t + 4, t + 5)$*, when the next job of* τ_1 *is scheduled. Then,* τ_2 *has to wait for the next service interval* $[t + 6, t + 7)$*, but at that point it would miss its deadline.*

Notice that since the proof of Lemma 8 does not rely on any particular protocol for the access to shared resources, the validity of the result can be extended to every reasonable policy, like SRP [2] or others, provided that the same mechanism is used for both the VP and the partition schedule. Since EDF+SRP is an optimal scheduling algorithm for virtual processors [31], the next theorem follows.

Theorem 5. *A collection of jobs is schedulable with EDF+SRP on a partition* (α_k, Δ_k) *if and only if it is schedulable with some scheduling algorithm on an* α_k*-speed VP with a jitter tolerance of* Δ_k*, i.e., all jobs finish at least* Δ_k *time units before their deadline.*

Therefore, when there is no limit on the algorithm to be used to schedule the application jobs on a partition, using EDF+SRP is an optimal choice, since it guarantees that all deadlines are met independently from the algorithm that has been used for the validation on the dedicated virtual processor. This also explains the meaning of the names we gave in Section 2.2 to α_k and Δ_k parameters.

On the contrary, when the scheduling algorithm cannot be freely chosen, for instance when a fixed priority order among tasks composing an application has to be enforced, we showed in Section 2.4 that a buffered version of the VP schedule can be used. However, to avoid the computational effort of reproducing the VP scheduling order at run-time, some more expense can be paid off-line by analyzing the execution time supplied by the partition together with the demand imposed by the jobs of the application. The next section addresses this problem.

Application-Level Scheduling with Other Algorithms

The application may require that a scheduler other than EDF + SRP be used as an application-level scheduler. When a buffered version of the VP schedule is not feasible due to the associated run-time complexity, an alternative could be to use a more sophisticated schedulability analysis instead of the validation process on a dedicated VP. This requires one to consider the service effectively supplied by the open environment in relation to the amount of execution requested by the application. Our *BROE* server implements a bounded-delay server in the presence of shared resources. Examples of analysis for the fixed-priority case under servers implementing bounded-delay partitions or related partitions, in absence of shared resources, can be found in [10, 13, 32, 28], and easily applied to our open environment. We conjecture that the results for local fixed-priority schedulability analysis on resource partitions can be easily extended to include local and global resources, and be scheduled by *BROE* without modification to the server. We leave the exploration of this conjecture as a future work.

2.5 Local schedulability analysis

Since we showed (Theorem 5) that EDF+SRP can be used to optimally schedule the jobs composing an application on a bounded-delay partition, we choose this algorithm as the default local scheduling algorithm for our open environment. This means that whenever an application does not require a different scheduling policy, the open environment will schedule the jobs of an admitted executing application using EDF with resource access arbitrated through the SRP protocol, since this choice allows optimizing system performances. We will hereafter derive local schedulability tests for this case.

Since we are moving the detail of our analysis from server chunks to the jobs composing an application, we need to extend the notational model used. In the rest of the paper, we will consider an application A_k to be composed of n_k sporadic tasks [8] [4]. We will indicate with c_i, d_i, p_i, the WCET, relative deadline, and period or minimum inter-arrival time of task τ_i. The maximum size of a critical section on resource R_ℓ accessed by a task τ_i is denoted with $h_i(R_\ell)$. To avoid confusion with the shared resource policy adopted at system-level and described in Section 2.3, we will distinguish between *local* and *global* ceiling of a shared resource R_ℓ. Global ceiling $\Pi(R_\ell)$ is given by the minimum value from among all the period parameters P_k of applications A_k that use resource R_ℓ; local ceiling $\pi_k(R_\ell)$ is given, locally to each application A_k, by the minimum value from among all relative deadlines d_i of tasks $\tau_i \in A_k$ that can lock resource R_ℓ. Note that the *system ceiling* used for global SRP — equal to the minimum global ceiling of any resource that is locked at a given instant — corresponds at the local level with the *application ceiling* for local SRP — defined as the minimum local ceiling of any resource that is locked at a given instant by tasks of the considered application.

For any collection of jobs released by an application A_k and any real number $t \geq 0$, the *demand bound function* $\mathrm{DBF}(A_k, t)$ is defined as the largest cumulative execution requirement of all jobs that can be generated by A_k to have both their arrival times and their deadlines within a contiguous interval of length t. For instance, for the sporadic task model it has been shown [8] that the cumulative execution requirement of jobs over an interval $[t_o, t_o + t)$ is maximized if all tasks arrive at the start of the interval — i.e., at time-instant t_o — and subsequent jobs arrive as rapidly as permitted — i.e., at instants $t_o + p_i$, $t_o + 2p_i$, $t_o + 3p_i$, ...:

$$\mathrm{DBF}(A_k, t) \doteq \sum_{\tau_i \in A_k} \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) c_i\right) \tag{2.26}$$

Algorithms are described in [8] for efficiently computing $\mathrm{DBF}(A_k, t)$ in constant time for the sporadic task model, for any $t \geq 0$.

Shin and Lee [32] showed that $\mathrm{DBF}(A_k, t)$, together with a bound on the provided supply, can be used to derive a schedulability condition for periodic task systems scheduled on a bounded-delay resource partition (α_k, Δ_k). We restate their result in terms of general collection of jobs released by an application A_k, whose execution requirements are captured through a demand bound function $\mathrm{DBF}(A_k, t)$.

Theorem 6 (adapted from Shin and Lee [32])**.** *An application A_k is EDF-schedulable on a bounded-delay partition (α_k, Δ_k) if for all $t \geq 0$,*

$$DBF(A_k, t) \leq \alpha_k(t - \Delta_k). \tag{2.27}$$

[4] We believe that the following analysis can be easily adapted also for more general models, such as arbitrary collections of jobs.

Baruah proposed in [31] a technique for analyzing systems scheduled with EDF+SRP using the demand-bound function. The approach is to calculate, for all $L > 0$, the maximum amount of time that a job with relative deadline at most L could be "blocked" by a job with relative deadline greater than L. The following function [31] quantifies this maximum blocking over an interval of length L in an application A_k:

$$b_k(L) \doteq \max\{h_i(R_j) | \exists \ell : (\tau_i, \tau_\ell \in A_k) \wedge (d_i > L) \\ \wedge (d_\ell \leq L) \wedge (\tau_i \text{ and } \tau_\ell \text{ access } R_j)\}. \tag{2.28}$$

The next corollary follows from the application of Theorem 1 in [31] and Theorem 6 above:

Corollary 1. *An application A_k is EDF+SRP-schedulable on a bounded-delay partition (α_k, Δ_k) if for all $L > 0$,*

$$b_k(L) + DBF(A_k, L) \leq \alpha_k(L - \Delta_k). \tag{2.29}$$

Using techniques from [8], it is possible to bound the number of points at which inequality (2.29) should be checked. It is sufficient to check only the values of L satisfying $L \equiv (d_i + kp_i)$, for some i,$1 \leq i \leq n$, and some integer $k \geq 0$. The time complexity of determining whether an application A_k is EDF+SRP-schedulable on a bounded delay resource partition cannot be larger than the complexity of the feasibility analysis for independent applications (see [31] for details). For a resource partition (α_k, Δ_k) and an application A_k, an upper bound on the values of L to be checked is given by the least of (i) the least common multiple (lcm) of $p_1, p_2, \ldots, p_{n_k}$, and (ii) the following expression

$$\max\left(d_{\max}, \frac{1}{\alpha_i - U} \cdot \left(\alpha_i \Delta_i + \sum_{i=1}^{n_k} U_i \cdot \max\left(0, p_i - d_i\right)\right)\right)$$

where $d_{\max} \doteq \max_{i=1}^{n}\{d_i\}$; U_i denotes the utilization e_i / p_i of τ_i; and U denotes the application utilization: $U \doteq U_1 + U_2 + \cdots + U_{n_k}$. This bound may in general be exponential in the parameters of A_k; however, it is pseudo-polynomial if the application utilization is a priori bounded from above by a constant less than α_i.

2.6 Resource Holding Times

In the previous sections, the interface parameters α_k and Δ_k have been exhaustively characterized for the considered open environment. This section will instead give further details on the third parameter, the resource holding time — i.e., the maximum time $H_k(R_\ell)$ for which an application A_k can lock a global resource R_ℓ — and on the methods that can be used to compute it for a given application.

Computing Resource Holding Times

When a resource is locked by an application executing on a bounded delay server, it may happen that during the time a resource is locked, the server upon which the locking task executes is preempted by higher priority servers. Even if this time has not to be accounted inside H_k (not being part of the blocking term in the test for the server admission control), higher priority tasks with period lower than the application ceiling can meanwhile arrive in the blocked server, increasing the resource holding time. Therefore, this delay in the execution supplied to the preempted application has to be properly considered when computing H_k.

Examining rule (ix) of the *BROE* server in Section 2.3, it is easy to see that when an application A_k locks a resource R_ℓ, it can execute for the duration of the whole resource holding time $H_k \leq \alpha_k P_k$ before needing to be suspended. This execution time will be supplied in the worst case after $(P_k - \alpha_k P_k) = \Delta_k/2$ time units. On the other hand, if an application holds a resource for more than $\alpha_k P_k$, the enforcement mechanism described in Section 2.3 will release the lock.

To compute the resource holding time under EDF+SRP, we will adapt the technique described in [24] (which is valid for the case in which a dedicated computing resource is used) to the case in which the execution is supplied after $\Delta_k/2$ time units. (Note that $\Delta_k/2$ is the maximum delay in execution that an application can experience while it executing within a server chunk; otherwise, the server will miss a deadline). The supply function to consider is null for $\Delta_k/2$ time-units and than grows with unitary slope until the resource is unlocked (see Figure 2.4). The algorithms to compute the resource holding times in [24, 25] assume instead the availability of a dedicated unit-capacity processor, having full supply without any delay. To adapt these algorithms to the case considered in this paper, it is necessary to add a term $\Delta_k/2$ in the fixed point iteration formula used to derive the resource holding times. The resource holding time $H_k(R_\ell)$ of an application A_k is defined as the maximum time $\mathrm{RHT}_i(R_\ell)$ any task $\tau_i \in A_k$ may hold a resource R_ℓ:

$$H_k(R_\ell) \doteq \max_{\tau_i \in A_k} \{\mathrm{RHT}_i(R_\ell)\}.$$

The cumulative execution requests of jobs that can preempt τ_i while it is holding a resource R_j for t units of time, along with the maximum amount τ_i can execute on resource R_j, and the maximum supply delay $\frac{\Delta_k}{2}$, is given by (see [24])

$$F_i(t) \doteq \frac{\Delta_k}{2} + h_i(R_\ell) + \sum_{j=1}^{\Pi(R_\ell)-1} \left\lceil \frac{\min\left(t, D_i - D_j\right)}{T_j} \right\rceil \cdot C_j. \tag{2.30}$$

Let t_i^* be the smallest fixed point of function $F_i(t)$ (i.e., $F_i(t_i^*) = t_i^*$). The resource holding time $\mathrm{RHT}_i(R_\ell)$ of a task τ_i is given by

$$\mathrm{RHT}_i(R_\ell) \doteq t_i^* - \frac{\Delta_k}{2}. \tag{2.31}$$

The iteration can be aborted when $F_i(t)$ exceeds $\min(\alpha_k P_k, d_i)$, since in that case a deadline could be missed, and the application is rejected. More details on the above technique can be found in [24], where it is proved for a similar case that a fixed point is reached in a finite (pseudo-polynomial) number of steps.

Note that the resource holding time represents the maximum supply needed by an application to release a lock. It is therefore a measure of the *execution time* needed rather than a measure of the *real time* elapsed between the lock and release of the resource[5].

Decreasing Resource Holding Times

Since the resource holding time determines the time for which other servers below the global ceiling of the locked resource can be blocked, it is very important to keep this value as small as possible. In this way, it is possible to compose more servers on the same system without having to account for a large server blocking

[5]We believe the resource holding time to be similar to the *maximum critical section execution time* defined in [7].

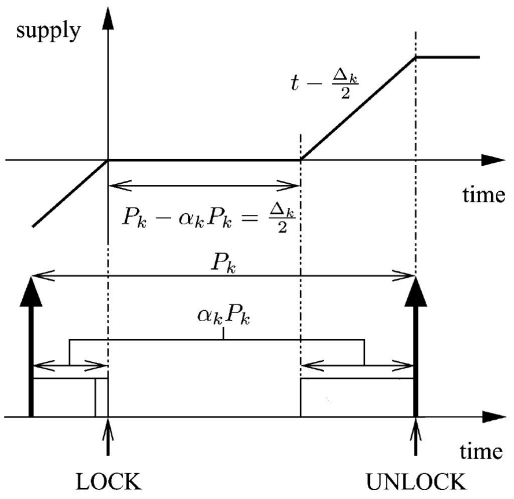

Figure 2.4: Worst case supply when a global resource is locked.

time. One way to do that, is using techniques from [25, 24] and [26]. Basically, these techniques artificially decrease the ceiling of a resource by adding a dummy critical section to tasks with a deadline lower than the resource ceiling, when this does not affect the schedulability of the application. The semantics of the application do not change, but the resulting resource holding time decreases due to the reduced number of preemptions on a task holding the resource. The procedure to decrease the ceiling to the minimum possible value depends on the scheduling algorithm used. The EDF case is treated in [25, 24], while [26] deals with RM/DM. Both works assume a dedicated computing resource is used. We need to adapt them to the case in which a task set is to be scheduled on a bounded delay server. We will consider here only the EDF case.

When the application is locally scheduled with EDF, the only modification needed to the algorithm presented in [25] is in the schedulability test used to verify if the ceiling can be decreased. Instead of the test used at line 2 of Procedure REDUCECEILING in [25], we will use the test given by Corollary 1. Since this is just a trivial modification, we do not include here a detailed description of the algorithm, which can be found in [25, 24].

In the following section, we will provide an alternative and efficient way to decrease the resource holding time of an application, without requiring any additional computation.

Executing Global Critical Sections Without Local Preemptions

In this section, we will formally show that even if application A_k was validated upon a dedicated virtual processor of speed α_k using EDF+SRP or any other policy, some critical sections may be executed without local preemptions under a *BROE* server.

Theorem 7. *Given an application A_k accessing a globally shared resource R_ℓ can be scheduled upon a dedicated virtual processor of speed-α_k, where each job completes at least Δ_k time units prior to its deadline: if $h_k(R_\ell) \leq \frac{\Delta_k}{2}$ and $H_k(R_\ell) \leq \alpha_k P_k$ then A_k can be EDF-scheduled on a* BROE *server with parameters (α_k, Δ_k), executing each critical section of R_ℓ* with local preemptions disabled.

Proof. The proof is by contradiction. Assume the antecedent of the theorem holds, but the application A_k misses a deadline on a *BROE* server with parameters (α_k, Δ_k), when executing a critical section of a global resource R_ℓ with local preemptions disabled. For Theorem 5, A_k is schedulable on a partition (α_k, Δ_k) with EDF+SRP.

Let t_{miss} be the first missed deadline with non-preemptive locking, and t_s the latest time-instant prior to t_{miss} at which there are no jobs with deadline $\leq t_{miss}$ awaiting execution ($t_s \leftarrow 0$ if there was no such instant). With non-preemptive locking, as with SRP, there is at most one blocking job over $[t_s, t_{miss}]$ (see [31]). The blocking job must have acquired the lock before t_s and have a deadline after t_{miss}. Let Y be the set of jobs that have both release time and deadline in $[t_s, t_{miss}]$. Over $[t_s, t_{miss}]$, the application executes only jobs in Y, or a blocking job. Let t_l and t_u be, respectively, the time at which the blocking job acquires and releases the lock on R_ℓ, when EDF+SRP is used as a local scheduling algorithm. We hereafter prove that $t_{miss} \leq t_u$. Suppose, by contradiction, $t_{miss} > t_u$. The total demand in $[t_s, t_{miss}]$ is given by the contributions of the jobs in Y, plus the remaining part of the blocking critical section that has still to be executed at t_s. The demand due to jobs in Y is the same whether SRP or non-preemptive locking is used. Moreover, since with non-preemptive locking a job cannot be interfered with while holding a global lock, the contribution of the blocking job in $[t_s, t_{miss}]$ cannot be larger than with SRP. Therefore, the total amount of execution requested in $[t_s, t_{miss}]$ with SRP is at least as large as with non-preemptive locking. Since a deadline is missed with non-preemptive locking, a deadline will be missed even with SRP, contradicting the hypothesis. Then, $t_{miss} \leq t_u$.

When there is a blocking contribution in $[t_s, t_{miss}]$, it means that the blocking job is still holding the lock on R_ℓ at time t_s. Therefore, with non-preemptive locking, the blocking job is the only one executing in $[t_l, t_s]$. We now prove that no job with deadline $\leq t_{miss}$ of A_k is released in $[t_l, t_s]$. Suppose that a job J is released in that interval. Since the blocking job is executing non-preemptively in $[t_l, t_s]$, J is still awaiting execution at time t_s. Since t_{miss} is the first missed deadline, J's deadline cannot be $< t_s$. Moreover, for the definition of t_s, J's deadline cannot even be between t_s and t_{miss}. Thus, J's deadline must be later than t_{miss}.

Let $W(Y)$ be the total amount of execution requested in $[t_s, t_{miss}]$ by jobs in Y, and let $\xi^{\text{NPL}}(t_1, t_2)$ (resp. $\xi^{\text{SRP}}(t_1, t_2)$) be the amount of execution received by the blocking job in $[t_1, t_2]$ when non-preemptive locking (resp. SRP) is used. Finally, let $s(t_1, t_2)$ be the amount of execution supplied to A_k in interval $[t_1, t_2]$. Since a deadline is missed with non-preemptive blocking, the following relation holds,

$$W(Y) + \xi^{\text{NPL}}(t_s, t_{miss}) > s(t_s, t_{miss}) = s(t_l, t_{miss}) - s(t_l, t_s). \tag{2.32}$$

Consider the EDF+SRP schedule. The *BROE* server is never suspended in $[t_l, t_u]$ (remember t_u is the unlocking time when SRP is used), and A_k has at least one pending job throughout the same interval. Rule (ix) of the *BROE* server guarantees that the total execution needed by A_k to release the global lock will be granted with a delay of at most $(P_k - \alpha_k P_k) = \frac{\Delta_k}{2}$ time-units. Therefore, the execution supplied to A_k in $[t_l, t_{miss}]$ is at least $(t_{miss} - t_l - \frac{\Delta_k}{2})$. Note that the execution supplied to an application does not depend on the particular local scheduling algorithm, but only on the global scheduling policy. Therefore for both SRP and non-preemptive locking, $s(t_l, t_{miss}) \geq t_{miss} - t_l - \frac{\Delta_k}{2}$. Equation (2.32) then becomes

$$W(Y) + \xi^{\text{NPL}}(t_s, t_{miss}) > t_{miss} - t_l - \frac{\Delta_k}{2} - s(t_l, t_s).$$

Using $h_k(R_\ell) \geq \xi^{\text{NPL}}(t_l, t_s) + \xi^{\text{NPL}}(t_s, t_{miss})$, it follows

$$W(Y) + h_k(R_\ell) - \xi^{\text{NPL}}(t_l, t_s) > t_{miss} - t_l - \frac{\Delta_k}{2} - s(t_l, t_s).$$

Since the only job of A_k that is scheduled in $[t_l, t_s]$ is the blocking job, $\xi^{\text{NPL}}(t_l, t_s) \equiv s(t_l, t_s)$. Then,

$$W(Y) + h_k(R_\ell) > t_{miss} - t_l - \frac{\Delta_k}{2}.$$

Being $h_k(R_\ell) \leq \frac{\Delta_k}{2}$, the total amount of execution requested in $[t_s, t_{miss}]$ by jobs in Y is

$$W(Y) > t_{miss} - t_l - \Delta_k.$$

Since we assumed that when A_k is scheduled on a dedicated processor of speed $\alpha_k \leq 1$, each job completes at least Δ_k time-units before its deadline, the total demand of jobs in Y cannot be larger than the RHS of the above equation, reaching a contradiction and proving the theorem. □

We believe Theorem 7 represents a very interesting result, since it allows improving the schedulability of the system both locally and globally. Notice, if the above theorem is satisfied for some A_k and R_ℓ, then we may use $h_k(R_\ell)$ instead of $H_k(R_\ell)$ in the admission control tests of Section 2.3. This increases the likelihood of A_k being admitted because the amount A_k could block applications A_i with $P_i < P_k$ is decreased. Moreover, it allows using a very simple non-preemptive locking protocol, decreasing the number of preemptions experienced by an application while holding a lock, and avoiding the use of more complex protocols for the access to shared resources.

2.7 Sharing global resources

One of the features of our open environment that distinguishes it from other work that also considers resource-sharing is our approach towards the sharing of global resources across applications.

As stated above, there are works [6] mandating that global resources be accessed with local preemptions disabled. The rationale behind this approach is sound: by holding global resources for the least possible amount of time, each application minimizes the blocking interference to which it subjects other applications. However, the downside of such non-preemptive execution is felt *within* each application — by requiring certain critical sections to execute non-preemptively, it is more likely that an application when evaluated in isolation upon its slower-speed VP will be deemed infeasible. The server framework and analysis described in this paper allows for several possible execution modes for critical sections. We now analyze when each mode may be used.

More specifically, in extracting the interface for an application A_k that uses global resources, we can distinguish between three different cases:

- If the application is feasible on its VP when it executes a global resource R_ℓ non-preemptively, then have it execute R_ℓ non-preemptively.

- If an application is infeasible on its VP of speed α_k when scheduled using EDF+SRP for R_ℓ, it follows from the optimality of EDF+SRP [31] that no (work-conserving) scheduling strategy can result in this application being

feasible upon a VP of the specified speed. Thus, by Theorem 5, no application-level scheduler can guarantee deadlines will be met for the application on any *BROE* server with parameter α_k.

- The interesting case is when neither of the two above holds: the system is infeasible when R_ℓ executes non-preemptively but feasible when access to R_ℓ is arbitrated using the SRP. In that case, the objective should be to *devise a local scheduling algorithm for the application that retains feasibility while minimizing the resource holding times*. There are two possibilities:

  a) Let $h_k(R_\ell)$ be the largest critical section of any job of A_k that accesses global resource R_ℓ. If $h_k(R_\ell) \leq \Delta_k/2$ (in addition to the previously-stated constraint on the resource-hold time $H_k(R_\ell) \leq \alpha_k P_k$), then A_k may disable (local) preemptions when executing global resource R_ℓ on its *BROE* server. In some cases, it may be advantageous to reduce $H_k(R_\ell)$ to increase the chances that the constraint $H_k(R_\ell) \leq \alpha_k P_k$ is satisfied.

  b) If $h_k(R_\ell) > \Delta_k/2$ but $H_k(R_\ell) \leq \alpha_k P_k$ still holds, R_ℓ may be executed using SRP. The resource-hold time could potentially be reduced by using techniques from [24], as discussed in Section 2.6.

2.8 Observations

As a final remark, it is possible to integrate our server with some previously proposed reclaiming mechanism to exploit the unused bandwidth (see Deliverable D4a). A simple rule can be easily added by setting to *Inactive* the state of all servers when the processor is idle. Moreover, the reclaiming mechanism used by GRUB in [27] may be implemented by updating the virtual time V_k of an executing application A_k 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. We believe that other mechanisms, like the ones used in [33] and [34], can be adapted to the presented framework. We leave these problems for future works.

2.9 Conclusion

In this chapter, we have presented a design for an open environment that allows for multiple independently developed and validated applications to be multi-programmed onto a single shared platform. We believe that our design contains many significant innovations.

- We have defined a clean interface between applications and the environment, which encapsulates the important information while abstracting away unimportant details.

- The simplicity of the interface allows for efficient run-time admission control, and helps avoid combinatorial explosion as the number of applications increases.

- We have addressed the issue of inter-application resource sharing in great detail, moving beyond the ad hoc strategy of always executing shared global resources non-preemptively, we have instead formalized the desired property of such resource-sharing strategies as *minimizing resource holding times*.

- We have studied a variety of strategies for performing arbitration for access to shared global resources within individual applications such that resource holding times are indeed minimized.

For the sake of concreteness, we have analyzed the local schedulability of individual applications that are executed upon our open environment using EDF and some protocol for arbitrating access to shared resources. This is somewhat constraining — ideally, we would like to be able to have each application scheduled using *any* local scheduling algorithm[6].

Let us first address the issue of the task models that may be used in our approach. The presented results have assumed that each application is composed of a collection of jobs that share resources. Therefore, the results contained in the paper extend in a straightforward manner to the situation where individual applications are represented using more general task models such as the multiframe [35, 36], generalized multiframe [37], or recurring [38, 39] task models — in essence, any formal model satisfying the *task independence assumptions* [37] may be used.

We conjecture that our framework can also handle applications modeled using task models not satisfying the task independence assumptions, provided the resource sharing mechanism used is independent of the absolute deadlines of the jobs, and only depends upon the relative priorities of the jobs according to EDF. This result allows the techniques developed in this document to be easily integrated with the workload model coming from the upper levels of the ACTORS framework, where sets of jobs with precedence constraints have to be scheduled on particular Virtual Processors.

Next, let us consider local scheduling algorithms. We expect that analysis similar to ours could be conducted if a local application were to instead use (say) the deadline-monotonic scheduling algorithm [40, 41] with sporadic tasks, or some other fixed priority assignment with some more general task model (again, satisfying the task independence assumption). As discussed in Section 2.4, prior work on scheduling on resource partitions has assumed the local tasks do not share resources; we believe these results could be easily extended to include resource sharing and used within our server framework.

A final note concerning generalizations. Our approach may also be applied to applications which are scheduled using *table-driven scheduling*, in which the entire sequence of jobs to be executed is pre-computed and stored in a lookup table prior to run-time. Local scheduling for such systems reduces to dispatch based on table-lookup: such applications are also successfully scheduled by our open environment.

[6]Shin and Lee [13] refer to this property as *universality*.

Chapter 3

Non-preemptive locking in Open Environments

In this chapter, we present an alternative strategy to arbitrate the access to globally shared resources in hierarchical EDF scheduled real-time systems. The advantage of this strategy over the one described in Chapter 2 is that no information is needed a priori on the duration of each critical section. Previous works addressing this problem assumed each task worst-case critical section length be known in advance. However, this assumption is valid only in restricted system domains, and is definitely inadequate for general purpose real-time operating systems. To sidestep this problem, we will instead measure at run-time the amount of time for which a task keeps a resource locked, assuring that there is enough bandwidth to tolerate the interferences associated to such measured blocking times. The protocol will execute each critical section non-preemptively, exploiting the BROE server described in Chapter 2. Two methods with different complexities will be derived to compute upper-bounds on the maximum time for which a critical section may be executed non-preemptively in a given hierarchical system.

3.1 Introduction

One of the main problems in composing different applications on a dedicated processor is related to the existing inter-dependencies due to the concurrent access to shared resources. As previously mentioned, particular care should be taken in designing a proper shared resource protocol that could avoid the so-called "budget exhaustion" problem. This problem arises when the same resource is accessed by two tasks standing at different hierarchy levels. When a server finishes its budget while a task is still inside a critical section, and is preempted by other servers sharing the same locked resource, no progress can be made in the system without losing consistency, until the preempted server resumes its execution and unlocks the shared resource.

In order to avoid complex protocols to arbitrate the access to shared resources, a good programming practice is to keep the length of every critical section short [42]. If this is the case, preemptions may be disabled while a task is holding a lock, without incurring significant schedulability penalties. In Section 3.2, we will show how to derive safe upper bounds on the maximum time for which a critical section may be executed non-preemptively. In Section 3.3, we will then explain how to efficiently use such bounds.

General purpose operating systems, like Linux, complicate, sometimes making it impossible, the task of providing tight estimations of the worst-case critical sec-

tion lengths for the general workloads they support. These systems often execute a small number of real-time tasks in parallel with many other best effort tasks, and there may be shared resources among the two classes of applications; thus, a shared resource protocol has to be transparent as much as possible, to avoid changes to existing applications.

Our approach tries to deal with such situations, using a runtime estimate for the critical section lengths and adapting the admission parameters to the behavior shown by the already admitted tasks. When an estimate is proven to be too optimistic (i.e., when a task executes inside a critical section for more than what it has been estimated when admitting it) there may be deadline misses. The system will nevertheless react to the overload condition by taking appropriate scheduling decisions and accordingly updating the estimations on the critical section lengths.

The computational complexity of the adopted mechanism is a key factor. For the admission control to be really useful in a highly dynamic system, as general purpose operating systems usually are, it has to be implemented inline and it must be able to support a very large number of tasks.

3.2 Scheduling analysis

Assume that each individual application may be characterized as a collection of sporadic tasks [9, 8], having deadlines equal to periods. Tasks may share resources that are *local* to an application (i.e., only shared within the application) or *global* (i.e., may be shared among different applications). Each application is scheduled in a dedicated server, which can potentially include other servers, forming a hierarchy of "entities". The term entity is introduced to unify the analysis of tasks and servers, according to the following definition.

Definition 6 (Entity)**.** *We recursively define an* entity η_i *as either a sporadic task, or a server in which a set of entities is scheduled. The period T_i of an entity is accordingly identified either by the minimum interarrival time of the sporadic task, or by the period of the server. Similarly, the budget C_i of an entity is either the worst-case computation time of the task, or the server budget.*

An example scenario is depicted in Figure 3.1, with η_0 being the root server, η_1 and η_2 its child entities, and η_3 being the only child entity of η_1. The leaf entities η_2 and η_3 are both tasks, while η_1 is a non-root server, so it acts both as an entity scheduled by its parent server (i.e., η_0), and a server itself, scheduling its own child η_3.

To avoid the implementation of complex shared resource protocols, as well as to limit the budget exhaustion problem, we decided to execute each critical section *with system preemptions disabled*. However, to preserve the system schedulability, we will compute for each entity the total amount of time for which preemptions may be safely disabled, exploiting this information during the admission control. If the admission of a new entity would leave enough bandwidth to execute each critical section non-preemptively, the entity is admitted. Otherwise it is rejected.

As previously mentioned, one of the main problems in designing an efficient shared resource protocol for a hierarchical system is given by the difficulties in deriving tight upper bounds on the time spent in each critical section. Since we do not want to charge the user with the burden of providing such upper bounds, we developed an alternative strategy that is able to efficiently solve this problem.

- First of all, we define a parameter h_i specifying the maximal length for which an entity η_i can execute non-preemptively without missing any deadline.

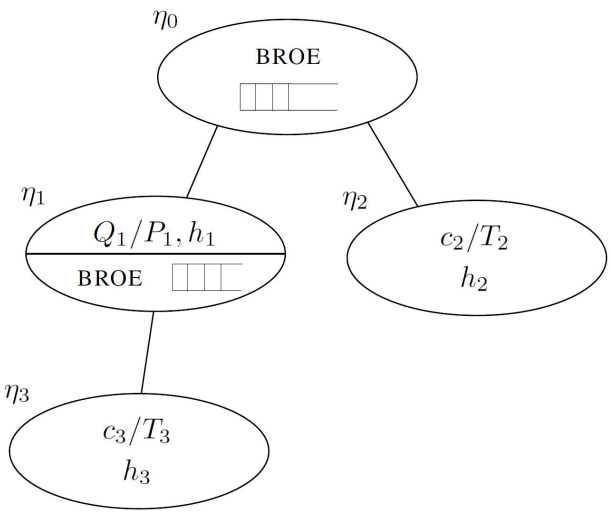

Figure 3.1: Tree of entities.

- We then compute a safe lower bound on h_i for each admitted entity.
- We measure the amount of time spent inside a critical section by each executing entity, storing the worst-case critical section length of each entity.
- New entities are admitted at a particular hierarchy level as long as each resulting non-preemptive chunk length is not lower than the corresponding maximal critical section length measured, as we will better describe in Section 3.3.

We hereafter provide a method to compute a safe lower bound on the maximal non-preemptive chunk length of an entity.

Assume a set η of entities $\eta_1, \eta_2, \ldots, \eta_n$ at a particular hierarchy level is scheduled with EDF on a *CBS*-like server with budget Q and period P. Entities are indexed in increasing period order, with $T_i \leq T_{i+1}$. The utilization of an entity is defined as $U_k = \frac{C_k}{T_k}$. Let T_{min} be the minimum period among all entities, and U_{tot} be the sum of the utilizations of all entities.

When entities may share common resources with other tasks or groups, it is important to avoid the situation in which the budget is exhausted while an entity is still inside a critical section. To avoid this problem, a budget check is performed before each locking operation, as in the *BROE* server described in Chapter 2. If the remaining budget is not sufficient to serve at least the maximum non-preemptive chunk length, the server is suspended, and reactivated as soon as the server capacity can be safely recharged. Otherwise, the critical section is served with the current server parameters.

Before stating our first schedulability result, we need to impose two constraints on each non-preemptive chunk length h_k, in order to avoid interfering with other entities in the system. In particular,

- *(i) h_k should not exceed the maximum budget Q,*
- *(ii) the computed maximum non-preemptive chunk length at a given hierarchy level should not exceed the same parameter at the parent level.*

Both constraints are needed to avoid the budget exhaustion problem while holding a lock, as well as to preserve the bandwidth isolation properties of the server-based open environment.

The following theorem presents a method to compute a safe bound on the maximum time length h_k for which an entity η_k may execute non-preemptively, preserving system schedulability.

Theorem 8 ($O(n)$)**.** *A set of entities that is schedulable with preemptive EDF on a server with budget Q and period P remains schedulable if every entity executes non-preemptively for at most h_k time units, where h_k is defined as follows, with $h_0 = \infty$:*

$$h_k = \min \left\{ h_{k-1}, \left(\frac{Q}{P} - \sum_{i=1}^{k} U_i \right) T_k - 2(P - Q) \right\}. \tag{3.1}$$

Proof. The proof is by contradiction. Assume a set of entities η misses a deadline when scheduled with EDF on a server having budget Q and period P, executing every entity η_k non-preemptively for at most h_k time-units, with h_k as defined by Equation (3.1). Let t_2 be the first missed deadline. Let t_1 be the latest time before t_2 in which there is no pending entity with deadline $\leq t_2$. Consider interval $[t_1, t_2]$. Since at start time there are no active entities, the interval is correctly defined, and the processor is never idled in $[t_1, t_2]$. Due to the adopted policy, at most one job with deadline $> t_2$ might execute in the considered interval: this happens if such job is executing in non-preemptive mode at time t_1. Let η_{np} be the entity which such job, if any, belongs to. The demand of η_{np} in $[t_1, t_2]$ is bounded by h_{np}. Moreover, $T_{\text{np}} > t_2 - t_1$. Every other entity executing in $[t_1, t_2]$ has instead $T_i \leq (t_2 - t_1)$. Let η_k be the entity with the largest period among such entities. Then, $T_k \leq t_2 - t_1 < T_{\text{np}}$, and $h_k \geq h_{\text{np}}$.

Since there is a deadline miss, the total demand in interval $[t_1, t_2]$ must exceed the capacity supplied by the server throughout the same interval. Such capacity[1] cannot be lower than $\frac{Q}{P}(t_2 - t_1) - 2(P - Q)$. Then,

$$h_{\text{np}} + \sum_{i=1}^{k} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i > \frac{Q}{P}(t_2 - t_1) - 2(P - Q).$$

Using $x \geq \lfloor x \rfloor$ and $h_k \geq h_{\text{np}}$, we get

$$h_k + (t_2 - t_1) \sum_{i=1}^{k} U_i > \frac{Q}{P}(t_2 - t_1) - 2(P - Q), \tag{3.2}$$

$$h_k > (\frac{Q}{P} - \sum_{i=1}^{k} U_i)(t_2 - t_1) - 2(P - Q). \tag{3.3}$$

And, since $t_2 - t_1 \geq T_k$,

$$h_k > \left(\frac{Q}{P} - \sum_{i=1}^{k} U_i \right) T_k - 2(P - Q),$$

reaching a contradiction. □

The above theorem provides a method to compute an upper bound on the time for which an entity can be executed non-preemptively, with a complexity that is

[1]The capacity supplied by the server can be bounded from below by a function that is null for $2(P - Q)$ time-units and than increases with slope $\frac{Q}{P}$, as shown in [13].

linear in the number of entities at the same hierarchy level. This result can be used for the arbitration of the access to shared resources in a hierarchical system: it is possible to execute each critical section non-preemptively, as long as no critical section is longer than the derived bound.

A weaker upper bound on the available non-preemptive chunk length can be computed with a reduced (constant) complexity, as shown by the following corollary.

Corollary 2 ($O(1)$)**.** *A set of entities that is schedulable with preemptive EDF on a server with budget Q and period P remains schedulable if every entity executes non-preemptively for at most*

$$\left(\frac{Q}{P} - U_{tot}\right) T_{\min} - 2(P - Q)$$

time units.

The above theorem allows computing a single value h for the allowed maximum non-preemptive chunk length *of all entities* at a given hierarchy level, in a constant time. This lighter tests is a valid option whenever it is important to limit the overhead imposed on the system.

In the following sections, we will compare the solutions given by Theorem 8 and Corollary 2, in terms of schedulability performances and system overhead. Other more complex methods may be used to derive tighter values for the allowed lengths of non-preemptive chunks (see, for instance, the work presented by Baruah in [43]); nevertheless, we chose not to implement such methods due to their larger (pseudo-polynomial) complexity. Having a fast, $O(n)$ method to calculate a global value for h is, in our opinion, really important, as in a highly dynamical system, with thousands of tasks, as Linux can be, it allows our method to be used without significant overhead. Using a global value for h simplifies the implementation and reduces the runtime overhead of the enforcing mechanism, that has not to keep track of the per-task values.

It is worth noting that more sophisticate shared resource protocols like the Stack Resource Policy (SRP) [2] are not so suitable for the target architecture, since they are based on the concept of ceiling of a resource. To properly compute such parameter, it would be necessary to know a priori which task will lock each resource and, in a real operating system, this is definitely not a viable approach from a system design point of view.

3.3 Admission Control

One of the key points of our approach is that there is no need for the user to specify a safe upper bound on the worst-case length of each critical section, something that is very problematic in non-trivial architectures. The system will use all the available bandwidth left by the admitted entities to serve critical sections, automatically detecting the length of each executed critical section, by means of a dedicated timer. If some entity holds a lock for more than the current corresponding non-preemptive chunk length, it means that some deadline may be missed, and the system is overloaded. In this case, some decision should be taken to reduce the system load.

There are many possible heuristics that can be used to remove some entities from the system to solve the overload condition, the choice of which depends on the particular application. For instance, the system may reject entities with heavier utilizations or longer critical sections, leaving enough bandwidth for the admission

of lighter entities; it can penalize less critical entities, if such information is available, or the most recently admitted one; or it can simply ask the user what to do. We chose to reject the entity with the largest critical section length, which is the one that triggered such scheduling decision executing for more than the current non-preemptive chunk length.

The system keeps track of the largest critical section at each hierarchy level. The maximum critical section length of a group is recursively defined as the maximum critical section length among all entities belonging to that group. For all deadlines to be met, this value should always be lower than the corresponding non-preemptive chunk length.

The admission control algorithm changes depending on the complexity of the adopted method to compute the time for which a task may execute with preemptions disabled. We distinguish into two cases: (i) using for all entities in the same hierarchy level a single value h given by Corollary 2; or (ii) using for each entity η_i a different value h_i given by Theorem 8.

In the first case the system keeps track of the largest critical section among all entities belonging to the same group: $R_{\max} = \max_{\text{group}}\{R_i\}$. For all deadlines to be met, this value should always be lower than the non-preemptive chunk length at the corresponding level:

$$R_{\max} \leq h. \tag{3.4}$$

When an entity η_k asks to be admitted into the system at a particular hierarchy level, the following operations are performed:

- the new allowed non-preemptive chunk length h' for the considered group of entities after the insertion of the new element is computed using Theorem 2.
- If such value is lower than the maximum critical section length $R_{\max}$ among the entities already admitted in the group, the candidate entity is rejected, since it means that there would not be enough space available to allocate the blocking time of some entity.
- Otherwise, η_k is admitted into the system, updating h to h'. Note that $R_{\max}$ does not need to be updated, since there is no available estimation of the maximum critical section length of η_k (initially, $R_k = 0$).

When instead an entity η_k leaves the system, the new (larger) value of h is computed and accordingly updated for the considered group of entities. Moreover, if $R_k = R_{\max}$, $R_{\max}$ may as well be updated (decreased).

The slightly more complex case in which different non-preemptive chunk values h_i are used for each entity η_i, we will instead proceed as follows. In order to guarantee all deadlines be met, we will check that every entity η_i has a non-preemptive chunk length h_i sufficiently large to accommodate the maximum critical section of that entity:

$$\forall i, \quad R_i \leq h_i. \tag{3.5}$$

When an entity η_k asks to be admitted into the system at a particular hierarchy level, the following operations are performed within that level:

- using Theorem 8, we compute the allowed non-preemptive chunk length h'_i after the insertion of the new entity, for all entities η_i having a period at least as large as η_k's: $T_i \geq T_k$.
- If there is at least one value h'_i that is lower than the maximum critical section length of the corresponding entity η_i — i.e., $h'_i < R_i$ — the candidate entity η_k is rejected.

- Otherwise, η_k is admitted into the system, updating each h_i to h'_i.

When an entity η_k leaves the system, we simply recompute the h_i values of the entities with period greater than T_k.

Independently from the adopted strategy, the system will check if an invariant condition (given by Equation (3.4) or Equation (3.5)) is maintained. When it is not, some decision should be taken to solve the overload condition.

In a certain sense, we can say that an entity is *conditionally* admitted into the system, and it will remain so as long as it does not show any critical section that is longer than the maximum non-preemptive chunk length allowed, in which case the task is rejected from the system. As we previously mentioned, alternative strategies may instead trigger different scheduling decision when $R_{\max}$ exceeds h, for instance creating room for an entity with a long critical section by rejecting different entities.

One last question needs to be answered: what to do if some entity holds a lock for more than the available non-preemptive chunk length. It is worth noting that there is no way to preventively reject an entity that will hold a lock for more than the allowed non-preemptive chunk length, since there is no way to know in advance for how long each lock will be held. We may notice that an entity is executing a critical section for longer than the allowed non-preemptive time interval only at run-time, in which case we can decide to (i) *suspend the entity*, (ii) *abort it*, or (iii) *continue executing it until the lock is released*. Each one of these methods has its advantages and drawbacks. Suspending an entity may increase the blocking time on other entities that share the same locked resource. Aborting a task while inside a critical section may leave the system in an inconsistent state. Continuing to execute the overloading entity non-preemptively may delay other tasks, leading to a deadline miss.

The choice of the adopted methods depends on the addressed application and on the characteristics of the shared resources that are accessed. When no information is available, we chose to continue executing the entity with system preemptions disabled until the lock is released (case (i)). Of course the compositional guarantees will be temporarily violated, but this appears to be the minimum price to pay for the indeterminism of the considered system model. We believe it is better to miss some deadline executing an overloaded entity, rather than leave the shared resource in an unpredictable state (as in case (ii)), or than stall the system due to the budget exhaustion problem (as in case (iii)). This consideration appears to be particularly true when considering nested resources, since a crossed access of nested resources by two different entities could lead to a deadlock condition when an entity is preempted before releasing a lock (case (iii)). Particular measures to avoid deadlocks need therefore to be taken when choosing this method.

3.4 Experimental Results

To evaluate the effectiveness of the proposed method, we performed a set of experiments with various different kinds of randomly generated task sets. We observed the values obtained for the maximum non-preemptiveness intervals given by our tests, trying to understand if they can be employed in real-world scenarios.

Experiment Setup

We focused on a single node of the entity hierarchy, and we analyzed the behavior of the tests when changing the entity's server parameters and the child entities generation parameters. With respect to the server, we considered different values

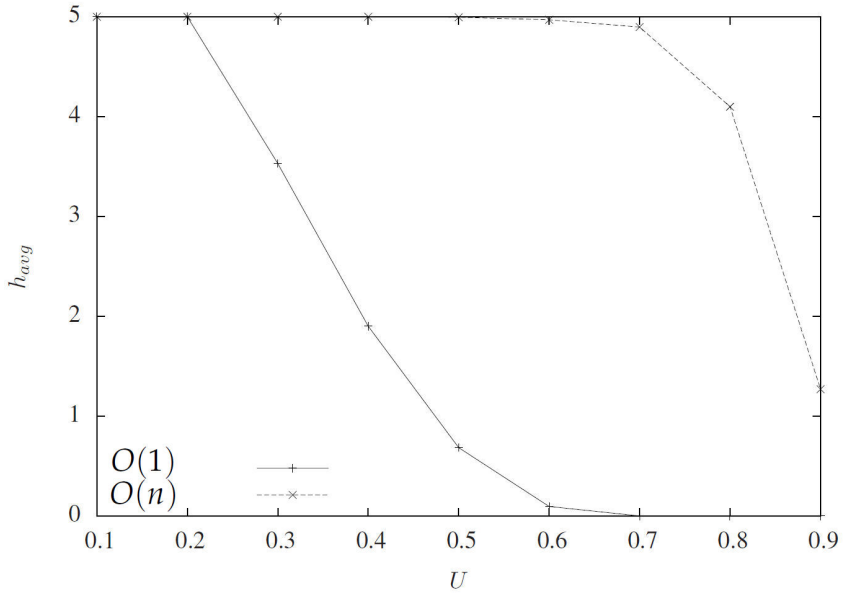

Figure 3.2: Large periods, small utilization.

for its P and Q, ranging from low utilization over short periods (e.g., 5ms every 100ms), to medium utilization over long periods (e.g., 500ms every 1s). The child entity set parameters we considered were the total utilization U, given as a fraction of the parent's Q/P ratio, the number of child entities, and for each entity, the period T_k and the computation time C_k.

We considered entity sets composed by 3 and 10 elements, and we varied the entity set utilization from 0.1 to 0.9, in steps of size 0.1. For each configuration we generated the entities periods T_k from a uniform distribution over intervals $[10P, 100P]$ or $[20P, 500P]$, and the entities utilizations U_k using the method described in [44]. The values for the entities computation times where then derived as $C_k = U_k * T_k$.

Evaluation of Experiments

First of all, here we report only a fraction of the results, as the conclusions that may be obtained from the ones we show are confirmed by the remaining ones.

We report the results only for entity sets of 10 elements; with a lower number of entities, results are better, since T_{min} tends to be larger, resulting in larger values of h.

Figure 3.2 shows the average h value obtained generating child entities with large periods (in the $[20P, 500P]$ range), inside a server with small utilization, with $Q = 5$ and $P = 100$. For the $O(n)$ test the plotted average is taken on the minimum h_k obtained.

From the figure it is clear that the $O(1)$ test starts degrading very early when utilization increases, giving pretty soon small values of h. On the other hand, the $O(n)$ test shows a good behavior in this scenario, giving a worst-case h_k that is close to the upper bound Q for medium-to-high utilizations (up to 0.7).

Figure 3.3 shows a scenario with a server having a large utilization, with $Q = 50$ and $P = 100$; as we could expect, increasing the server budget produces higher values for h, with both tests. It is worth noting that the above results were obtained with small values for T_{min}, as T_k were generated according to a uniform distribution

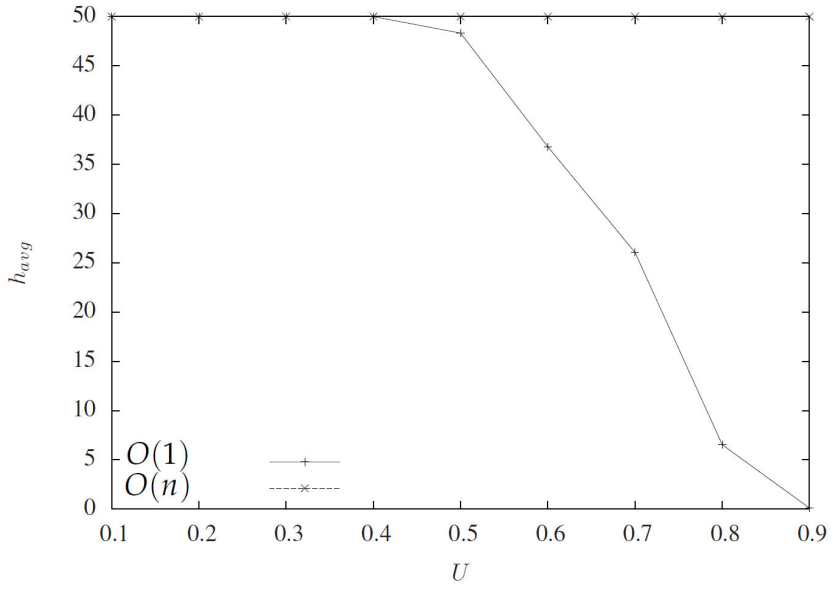

Figure 3.3: Small periods, high utilization.

in $[5P, 50P]$; bigger values for T_{min} resulted in even bigger values for h.

In this scenario the h_k values obtained with the $O(n)$ test are always close to the Q value, and even the $O(1)$ test gives large h values until $U = 0.6$.

3.5 Conclusions

In this chapter, we presented a combined method to arbitrate the access to shared resources in a hierarchical system composed by EDF scheduled sporadic tasks. The advantage of our approach are manyfold. There is no need for the user to specify the length of the critical sections that are accessed. The protocol for the arbitration of the access to shared resources is very simple, executing each critical section non-preemptively and limiting the overhead to a few bookkeeping operation. A simple *CBS*-like server is used to solve the budget exhaustion problem, limiting the interference imposed on each entity, improving the system schedulability. We presented two methods with different complexities for the computation of the available non-preemptive chunk length to accommodate the non-preemptive execution of the critical sections. The reported simulations show the effectiveness of both methods.

Chapter 4

Multiprocessors

In the previous chapters, we analyzed the scheduling of hierarchical open environments implemented on a single processor platform. The similar problem of executing hierarchical entities with shared resources on a multiprocessor is significantly more complex, since it is necessary to consider that, at each moment, more than one entity can contemporarily execute. Unfortunately, this might increase the blocking experienced by each application, deteriorating the schedulability of the system, as will be explained in the next sections. Solutions to limit as much as possible this blocking have been proposed in the literature. We will show how to integrate them with the task and resource models adopted in ACTORS, as well as with the proposed solutions developed in Chapters 2 and 3.

4.1 Problem description

We consider the same task model described in the previous chapters, with one single main difference: the computing platform upon which the applications execute may contain more than one processor. We will assume that an application A_i be composed by a set of actions, that are scheduled using one or more dedicated servers with budget Q and period P — or, with the equivalent parameters adopted in Chapter 2, on one or more Virtual Processors with speed α and jitter tolerance Δ_i. Each server/VP has a utilization/speed smaller than or equal to 1. This assumption has been made in accordance to the resource model adopted in ACTORS, which mandates that each VP be scheduled on at most one real processor. Virtual Processors with a speed α larger than one cannot be implemented on one single CPU, and are, therefore, not considered in the project. Note that, instead, an application with heavy bandwidth requirements can still be scheduled on two or more (dedicated) Virtual Processors.

The problem is further complicated by the presence of resources that are shared among different applications, and that need to be accessed in mutual exclusion. A resource that is shared exclusively among actions scheduled on the same VP will be called "local". A resource that is instead shared among different VPs — independently of whether they belong to the same application or to different ones — will be called "global".

Remote blocking

Whenever a global resource is locked by an action executing on a processor, any other entity needing to access the same resource on a different processor will have to wait until the resource is unlocked. This waiting time is called "remote block-

ing", since the blocking entity resides on a different (remote) processor. The remote blocking can be significantly large when the VPs upon which are scheduled entities needing the same global resource are spread on a large number of physical processors. In this case, a task might need to wait until the global resource is locked/unlocked multiple times in each one of the different processors, depending on the adopted locking protocol. In the next sections, we will summarize the main existing techniques that deal with this problem, highlighting the advantages and drawbacks of each one of them, with particular attention on the architecture targeted in the ACTORS project.

4.2 Related work

The problem of designing shared resource protocols for real-time systems scheduled on more than one processor has been analyzed in different papers. Most of them [45, 46, 48, 49, 50, 47, 54] are extensions of well-known approaches that have been developed for uniprocessor systems. Others, instead, mandate that each short global critical section be executed non-preemptively, reducing the remote blocking [52, 53]. We will hereafter summarize the main characteristics of these solutions, distinguishing between solutions for partitioned and global scheduling. Global schedulers have a single system-wide queue from which tasks are extracted to be scheduled on the available processors. Note that, with a global scheduler, a task might be preempted by a higher priority job, and later resume its execution on a different processor. This process is called "migration" of the task. With a partitioned scheduler, instead, tasks are statically assigned to processors, so that they are bound to execute on that processor and cannot migrate.

Protocols for partitioned scheduling

One of the first attempts of designing a shared resource protocol for multiprocessor systems is the Multiprocessor Priority Ceiling Protocol (MPCP) proposed by Rajkumar et al. in [45] for Fixed Priority scheduling. It is basically an extension for partitioned systems of the uniprocessor PCP protocol described in [1, 3]. Since each global resource is controlled by one particular processor, the solution proposed in [45] is not so suitable for shared-memory multiprocessors. For this reason, an improved MPCP version was proposed in [46]. According to this policy, local resources — i.e., resources that are shared only within tasks assigned to one specific processor — are arbitrated using the classic uniprocessor PCP. Instead, global critical sections — accessing resources that are shared among more than one processor — are executed at a priority higher than any non-critical section of code. This allows reducing the remote blocking experienced by the system, making it a function of critical sections only. In particular, the execution priority of a critical section accessing a global resource R is set to $P_G + \pi(R)$, where P_G is the maximum static priority among the tasks in the system, and $\pi(R)$ is the ceiling of R, i.e., the maximum priority among tasks that access R. Whenever a task cannot lock a global resource since it is locked by a different task, the blocked task is inserted in a prioritized queue, using the normal priority as the key for the queue insertion. When the blocking task releases the lock, the highest priority task in the queue can acquire the lock. Since a task is suspended whenever it is blocked on a shared resource, lower priority tasks can execute and lock some other shared resource. When the blocked task will resume execution, it might experience additional blocking due to the (global or local) shared resources meanwhile accessed by lower priority tasks. The blocking factors to be accounted for are therefore multiple and can be signifi-

cantly large. Moreover, deadlock conditions are possible in case of nested resources without any partial ordering.

To avoid these problems, an alternative approach is to avoid lower priority tasks to lock any resource while a higher priority task is blocked in the same processor. Alternatively, the blocked task might "busy wait" until the resource is granted, forbidding the execution of lower priority tasks. However, even if the blocking factors are significantly reduced, both approaches introduce some inefficiency, in that the processor is left unused whenever a high priority task is blocked on a global resource, increasing the interference on lower priority tasks. Schedulability tests and partitioning algorithms for MPCP with suspending or spinning have been recently presented in [47].

Extensions for partitioned EDF scheduling have been proposed by Chen and Tripathi in [48]. Global critical sections were executed non-preemptively and they were not allowed to be nested with local critical sections. However, their method is valid only for the more restrictive periodic task model, and not for the sporadic one.

Among busy-waiting approaches, Gai et al. presented in [50] the Multiprocessor Stack Resource Policy (MSRP), generalizing to multiprocessor systems the SRP protocol proposed in [2]. Global critical sections are executed non-preemptively, and access to global resources is implemented with FIFO-based spin-locks: tasks that are waiting for a global resource insert their requests on a global FIFO queue, without surrendering the processor until the resource is granted. To avoid deadlocks, global critical sections cannot be nested. A less efficient solution based on SRP for partitioned EDF scheduling has been proposed by Lopez et al. in [49], assigning to the same processor all tasks sharing the same resource.

Protocols for global scheduling

The problem of mutually exclusive access to shared resources for global scheduling algorithms has only recently been analyzed [51, 52, 53, 54]. Devi et al. proposed in [52] a straightforward modification to the global EDF scheduler, with a locking protocol based on the non-preemptive execution of global critical sections and with FIFO-based wait queues. Holman and Anderson designed locking protocols for Pfair schedulers in [51]. Block et al. implemented the FMLP protocol in [53], validating it for different scheduling strategies, namely, partitioned EDF, global EDF, and Pfair scheduling. Short critical sections are handled using spin-locks with FIFO-based wait queues, while long critical sections are arbitrated using a priority inheritance protocol similar to PCP [1, 3]. In all above approaches, the problem of accessing nested shared resources is either avoided — forbidding the nesting of critical sections — or is solved limiting the degree of locking parallelism — i.e., protecting nested critical sections with group locks. Very recently, Easwaran and Andersson proposed in [54] two different protocols for the arbitration of the access to shared resources in multiprocessor systems globally scheduled with Fixed Priority. The first one (PIP) is a generalization to global scheduling of the Priority Inheritance Protocol [1, 3]. The second one (P-PCP) implements a tunable trade-off between PIP and a ceiling-based protocol that limits the number of times a job can be blocked by lower priority tasks. In the same paper, schedulability tests based on response time analysis were presented for both protocols.

4.3 Conclusion

As mentioned in the introduction, only a subset of the techniques described in this document will be actually implemented in the Linux-based resource reservation scheme developed in Task 4.5 and described in deliverable D4e. For the simplicity of the approach and the efficiency of the solution, we chose to focus on the non-preemptive execution of global critical section, as described in Chapter 3. In this way, there is no need of complex resource sharing protocols, that are hard to support in Linux, so that locking and unlocking operations can be significantly simplified. The budget exhaustion problem will be avoided implementing each Virtual Processor by means of a BROE server, as described in Chapter 2.

As will be explained in deliverable D4d, the ACTORS methodology to allocate VPs onto the physical CPUs will be based on a partitioned solution. Therefore, the most suitable strategy for implementing mutual exclusion mechanisms among tasks executing on different processors appears to be using FIFO-based spin-locks. According to this policy, every time a task needs to access a global resource currently locked on a different CPU, the blocked task inserts its request on a FIFO queue, without surrendering the processor (i.e., "spinning") until the request is served. This solution presents the smallest implementation overhead among the methods in the literature, allowing an easier integration in the Linux kernel mechanisms. Furthermore, it allows for a simpler schedulability analysis, without needing to take into account the different blocking factors that take place when suspending blocked tasks.

To decrease the remote blocking factors that can significantly limit the schedulability of the system, tasks that access the same global resources will be allocated on the fewest possible number of processors. The best way to do that, considering the timing parameters and the requested bandwidth of each Virtual Processor, will be analyzed in Task 4.4 and extensively detailed in Deliverable D4d.

Bibliography

[1] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[2] T. P. Baker, "Stack-based scheduling of real-time processes," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 3, 1991.

[3] R. Rajkumar, *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Boston: Kluwer Academic Publishers, 1991.

[4] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharingin real-time open systems," in *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007.

[5] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Scheduling of semi-independent real-time components: Overrun methods and resource holding times," in *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08)*, (Hamburg, Germany), pp. 575–582, September 2008.

[6] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority preemptive systems," in *Proceedings of the IEEE Real-time Systems Symposium*, (Rio de Janeiro), pp. 257–267, IEEE Computer Society Press, December 2006.

[7] I. Shin, M. Behnam, T. Nolte, and M. Nolin, "Synthesis of optimal interfaces for hierarchical scheduling with resources," in *In Proceedings of the 29th IEEE International Real-Time Systems Symposium (RTSS'08)*, (Barcelona, Spain), pp. 209–220, December 2008.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] G. Bernat and A. Burns, "Multiple servers and capacity sharing for implementing flexible scheduling," *Real- Time Systems*, vol. 22, pp. 49–75, 2002.

[13] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 2–13, IEEE Computer Society, 2003.

[14] 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.

[15] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proceedings of the IEEE Real-time Systems Symposium*, (Miami, Florida), pp. 389–398, IEEE Computer Society, 2005.

[16] T.-W. Kuo and C.-H. Li, "A fixed prioriy driven open environment for real-time applications," in *Proceedings of the IEEE Real-Time Systems Symposium*, (Madrid, Spain), p. 256, IEEE Computer Society Press, December 1999.

[17] 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.

[18] 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.

[19] M. Caccamo and L. Sha, "Aperiodic servers with resource constraints," in *Proceedings of the IEEE Real-Time Systems Symposium*, (London, UK), IEEE Computer Society Press, December 2001.

[20] 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.

[21] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Trans. Computers*, vol. 53, no. 12, pp. 1591–1601, 2004.

[22] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, (London), IEEE Computer Society Press, December 2001.

[23] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. Gutiérrez, T. Lennvall, G. Lipari, J. Martínez, J. Medina, J. Palencia, and M. Trimarchi, "Fsf: A real-time scheduling architecture framework," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, (Los Alamitos, CA, USA), pp. 113–124, IEEE Computer Society, 2006.

[24] M. Bertogna, N. Fisher, and S. Baruah, "Resource holding times: Computation and optimization," *Real-Time Systems*, vol. 41, pp. 87–117, February 2009.

[25] N. Fisher, M. Bertogna, and S. Baruah, "Resource-locking durations in EDF-scheduled systems," in *13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (Bellevue, WA. (USA)), pp. 91–100, 2007.

[26] M. Bertogna, N. Fisher, and S. Baruah, "Static-priority scheduling and resource hold times," in *International Workshop on Parallel and Distributed Real-Time Systems (IPDPS)*, (Long Beach, CA, USA), pp. 1–8, March 2007.

[27] 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.

[28] 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.

[29] G. Lipari and G. Buttazzo, "Schedulability analysis of periodic and aperiodic tasks with resource constraints," *Journal Of Systems Architecture*, vol. 46, no. 4, pp. 327–338, 2000.

[30] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 30, pp. 105–128, May 2005.

[31] S. Baruah, "Resource sharing in EDF-scheduled systems: A closer look," in *Proceedings of the IEEE Real-time Systems Symposium*, (Rio de Janeiro), pp. 379–387, IEEE Computer Society Press, December 2006.

[32] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 57–67, IEEE Computer Society, 2004.

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

[34] 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.

[35] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," in *Proceedings of the 17th Real-Time Systems Symposium*, (Washington, DC), IEEE Computer Society Press, 1996.

[36] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," *IEEE Transactions on Software Engineering*, vol. 23, pp. 635–645, October 1997.

[37] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 17, pp. 5–22, July 1999.

[38] S. Baruah, "A general model for recurring real-time tasks," in *Proceedings of the Real-Time Systems Symposium*, (Madrid, Spain), pp. 114–122, IEEE Computer Society Press, December 1998.

[39] S. Baruah, "Dynamic- and static-priority scheduling of recurring real-time tasks," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 24, no. 1, pp. 99–128, 2003.

[40] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[41] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[42] V. Yodaiken, "Against priority inheritance," tech. rep., Finite State Machine Labs, June 2002.

[43] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, (Palma de Mallorca, Balearic Islands, Spain), pp. 137–144, IEEE Computer Society Press, July 2005.

[44] E. Bini and G. C. Buttazzo, "Biasing effects in schedulability measures," in *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, (Catania, Italy), July 2004.

[45] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proceedings of the Ninth IEEE Real-Time Systems Symposium*, pp. 259–269, IEEE, 1988.

[46] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proceedings of the International Conference on Distributed Computing Systems*, pp. 116–123, 1990.

[47] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proceedings of IEEE Real-Time Systems Symposium*, (Washington, DC, USA), 2009.

[48] C.-M. Chen and S. K. Tripathi, "Multiprocessor priority ceiling based protocols," tech. rep., College Park, MD, USA, 1994.

[49] J. M. Lopez, J. L. Diaz, and D. F. Garcia, "Utilization bounds for EDF scheduling on real-time multiprocessor systems," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 28, no. 1, pp. 39–68, 2004.

[50] P. Gai, G. Lipari, and M. di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, December 2001.

[51] P. Holman and J. H. Anderson, "Locking under pfair scheduling," *ACM Trans. Comput. Syst.*, vol. 24, no. 2, pp. 140–174, 2006.

[52] U. C. Devi, H. Leontyev, and J. H. Anderson, "Efficient synchronization under global edf scheduling on multiprocessors," in *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, (Washington, DC, USA), pp. 75–84, 2006.

[53] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, (Washington, DC, USA), pp. 47–56, 2007.

[54] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proceedings of IEEE Real-Time Systems Symposium*, (Washington, DC, USA), 2009.