The Design of an EDF-scheduled Resource-sharing Open Environment[*]

Nathan Fisher Marko Bertogna Sanjoy Baruah

Abstract

We study the problem of executing a collection of independently designed and validated task systems upon a common platform comprised of a preemptive processor and additional shared resources. We present an abstract formulation of the problem and identify the major issues that must be addressed in order to solve this problem. We present (and prove the correctness of) algorithms that address these issues, and thereby obtain a design for an open real-time environment.

Keywords: Open environments; Resource-sharing systems; Sporadic tasks; Critical sections; Earliest Deadline First; Stack Resource Policy.

1 Introduction

The design and implementation of *open* real-time environments [16] is currently one of the more active research areas in the discipline of real-time computing. Such open environments aim to offer support for real-time *multiprogramming*: they permit multiple independently developed and validated real-time applications to execute concurrently upon a shared platform. That is, if an application is validated to meet its timing constraints when executing in isolation, then an open environment that accepts (or *admits*, through a process of admission control) this application guarantees that it will continue to meet its timing constraints upon the shared platform. The open environment has a run-time scheduler which arbitrates access to the platform among the various applications; each application has its own local scheduler for deciding which of its competing jobs executes each time the application is selected for execution by the "higher level" scheduler. (In recognition of this two-level scheduling hierarchy, such open environments are also often referred to as "hierarchical" real-time environments.)

In order to provide support for such real-time multiprogramming, open environments have typically found it necessary to place restrictions upon the structures of the individual applications. The first generation of such open platforms (see, e.g., [25, 18, 10, 33, 17, 13] – this list is by no means exhaustive) assumed either that each application is comprised of a finite collection of independent preemptive periodic (Liu and Layland) tasks [24], or that each application's schedule is statically precomputed and run-time scheduling is done via table look-up. Furthermore, these open environments focused primarily upon the scheduling of a single (fully preemptive) processor, ignoring the fact that run-time platforms typically include additional resources that may not be fully preemptable. The few [31, 15, 12] that do allow for such additional shared resources typically make further simplifying assumptions on the task model, e.g., by assuming that the computational demands of each application may be aggregated and represented as a single periodic task, excluding the possibility to address hierarchical systems.

More recently, researchers have begun working upon the second generation of open environments that are

[*]This research has been supported in part by the National Science Foundation (Grant Nos. CCR-0309825, CNS-0408996 and CCF-0541056).

capable of operating upon more complex platforms. Two recent publications [14, 9] propose designs for open environments that allow for sharing other resources in addition to the preemptive processor. Both designs assume that each individual application may be characterized as a collection of sporadic tasks [26, 8], 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, both approaches propose that global resources be executed non-preemptively only, potentially causing intolerable blocking among and inside the applications.

In this paper, we describe our design of such a second-generation open environment upon a computing platform comprised 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). 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.

The remainder of this paper is organized as follows. The rationale and design of our open environment is described in Sections 2 and 3. In Section 2, we provide a high-level overview of our design, and detail the manner in which we expect individual applications to be characterized — this characterization represents the *interface* specification between the open environment and individual applications running on it — and in Section 3, we present the scheduling and admission-control algorithms used by our open environment. In Section 4, we discuss the different applications that may be scheduled by our open environment. In Section 5, we relate our open environment framework to other previously-proposed frameworks. In Section 6, we discuss in more detail how applications that use global shared resources may be scheduled locally using EDF with the Stack Resource Policy [3].

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 comprised 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 complete at least* Δ_k *time units before their deadlines if executing upon a dedicated processor of computing capacity* α_k*, 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 6 provides a more in depth discussion of the resource hold time parameter.

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

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

At first glance, this characterization may seem like a severe restriction, in the sense that one will be required to "waste" a significant fraction of the VP's computing capacity in order to meet this requirement. However, this is not necessarily correct. Consider the following simple (contrived) example. Let us represent a sporadic task [26, 8] by a 3-tuple: (*WCET, relative deadline, period*). Consider the example application comprised 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 all jobs always complete two time units before their deadlines. That is, this application can be characterized by the pair of parameters $\alpha_k = \frac{1}{2}$ and $\Delta_k = 2$.

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

Our characterization of an application's processor demands by the parameters α_k and Δ_k is identical to the *bounded-delay resource partition* characterization of Feng and Mok [25, 18, 17] 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 non-preemptively (such as the designs proposed in [14, 9]), $H_k(R_\ell)$ is simply the worst-case execution time upon the VP of the longest critical section holding global resource R_ℓ. We have recently [19, 11] derived algorithms for computing resource holding times when more general resource-access strategies such as the Stack Resource Policy (SRP) [3] and the Priority Ceiling Protocol (PCP) [32, 30] are instead used to arbitrate access to these global resources; in [19, 11], we also discuss the issue of designing the specific application systems such that the resource holding times are decreased without compromising feasibility. We believe that our sophisticated 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 6.

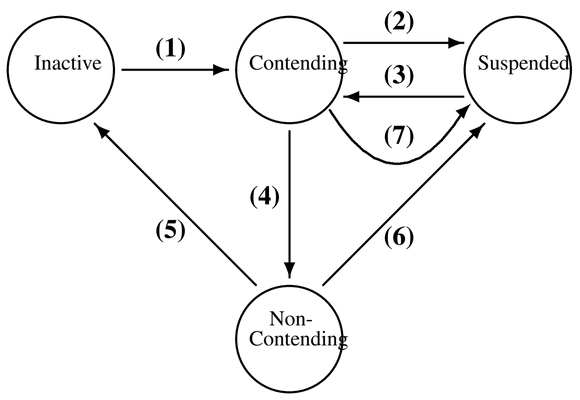


Figure 1. State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

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 above. When a new application wishes to execute, it presents its interface to the *admission control algorithm*, which determines, based upon the interface parameter 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 (ie. 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 first describe the global scheduling algorithm used by our open environment, in Section 3.1. A description and proof of correctness of our admission control algorithm follows (in Section 3.2). The local schedulers that may be used by the individual applications will be addressed in Section 4.

3.1 System-level Scheduler

Our scheduling algorithm is essentially an application of the Constant Bandwidth Server (CBS) of Abeni and Buttazzo [1], 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 remaining 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 budget replenishment tends to occur. For a BROE server, the value assigned to P_k is as follows:

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

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 four possible states (see Figure 1). 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 either the *contending* or *suspended* state[1]. While contending, it is eligible to execute; executing for more than it is eligible to results in its being suspended.

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

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

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

(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 in Section 3.1.1 below. Hence, observe that *only applications in the contending state are eligible to execute.*

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

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

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

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

(v) 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 **(3)**).

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

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

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

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

(vii) An application that wishes to gain access to global access 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 $\alpha_k(D_k - V_k) < H_k(R_\ell)$ there is insufficient budget left to complete access to resource R_ℓ by D_k. In this case, transition **(7)** is undertaken by an executing application

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

immediately prior to entering an outermost critical section locking a global resource[2] R_ℓ. This transition is accompanied by the following actions:

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

If there is sufficient budget, the server is granted access to resource R_ℓ.

Rules (i) to (vi) basically describe a bounded-delay version of the Constant Bandwidth Server, ie. a CBS in which the maximum service delay experienced by an application A_k is bounded by Δ_k. A similar server has been used in [22]. The only difference from a straightforward implementation of a bounded-delay CBS is the deadline update of rule (vi) associated to transition (6) (which has been introduced in order to guarantee that when an application resumes execution, its relative deadline is equal to the server period) and the addition of rule (vii).

Rule (vii) has been added to deal with the problem of budget exhaustion when a shared resource is locked. This problem, previously described in [12] and [14], arises when an application accesses a shared resource and runs out of budget (ie. 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 (7) right before an application A_k locks a critical section ensures that when A_k goes to the contending state (through transition (3)), 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 (2)). 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 3.2) does in fact ensure that

$$H_k(R_\ell) \le \alpha_k P_k \tag{2}$$

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 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. [12]) or implicitly, by allowing lock-holding applications to continue executing non-preemptively even when their current execution capacities are exhausted (e.g., [9, 14]).

3.1.1 Making scheduling decisions

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.

[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 (7).

[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.

3.2 Admission control

The admission control algorithm checks for three things:

1. As stated in Section 3.1 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. Finally, the effect of *inter-application blocking* must be considered – can such blocking cause any server to miss a deadline? A server-deadline miss occurs 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.

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 [23, 29, 6] 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 that requires information regarding each application's resource hold 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. In this section, we will introduce the two admission control algorithms and discuss the benefits and drawbacks of each.

Prior to introducing the admission control algorithms, Section 3.2.1 will prove that many of the desirable properties of SRP that hold for sporadic task systems [3] continue to hold for our BROE server. Section 3.2. Section 3.2.3 will describe and prove the correctness of the two admission control algorithms.

3.2.1 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 to derive 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 [12]. A new chunk is released each time the application enters the contending state and is terminated as soon as the state transitions from contending. 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 set by 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 to prove the following:

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: The proof is identical to the proof of Theorem 6 in [3]. The only difference is that in our case the items to be scheduled are application chunks instead of jobs. ■

3.2.2 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 *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 contending to non-contending or suspended. Formally,*

$$\text{DEMAND}(J_{k,\ell}, t_1, t_2) \overset{def}{=} \begin{cases} \alpha_k \left(V_k(g(J_{k,\ell})) - V_k(r(J_{k,\ell}))\right) & \text{if } (r(J_{k,\ell}) \geq t_1) \wedge (g(J_{k,\ell}) < d(J_{k,\ell}) \leq t_2) \\ \alpha_k P_k & \text{if } (r(J_{k,\ell}) \geq t_1) \wedge (d(J_{k,\ell}) \leq t_2) \wedge (g(J_{k,\ell}) \geq d(J_{k,\ell})) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

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 $[t_1, t_2]$:*

$$\text{DEMAND}(A_k, t_1, t_2) \overset{def}{=} \sum_{\ell \geq 1} \text{DEMAND}(J_{k,\ell}, t_1, t_2) \tag{4}$$

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. transitions (4) or (7)). 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,*

$$V_k(g(J_{k,\ell})) \leq d(J_{k,\ell}) \tag{5}$$

Proof: Observe that rule (iv) implies that when 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 (2), (4), or (7)) 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 virtual time does not increase between the termination of a chunk that becomes suspended, and the release of the next chunk.

Lemma 2 *If $J_{k,\ell+1}$ was released due to transition (3) (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 (3), the transition prior to (3) must have been either (2), (7), or the successive transitions of (4) and (6). For these transitions, either the server rules (iv), (vi), or (vii) apply when terminating the previous chunk $J_{k,\ell}$. However, notice that none of these rules update 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 (3). If $J_{k,\ell}, \ldots, J_{k,s-1}$ meet their deadline, then*

$$\text{DEMAND}(A_k, r(J_{k,\ell}), d(J_{k,s})) \leq \alpha_k(d(J_{k,s}) - r(J_{k,\ell})) \tag{6}$$

Proof:

According to Definition 2, DEMAND $(A_k, r(J_{k,\ell}), d(J_{k,s}))$ is the sum of the DEMAND $(J_{k,i}, r(J_{k,\ell}), d(J_{k,s}))$ 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 $J_{k,\ell}, \ldots, J_{k,s-1}$ chunk meet their deadlines, Equation 3 implies

$$\text{DEMAND}\,(A_k, r(J_{k,\ell}), d(J_{k,s})) = \text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \sum_{i=\ell}^{s-1} \alpha_k \left(V_k(g(J_{k,i})) - V_k(r(J_{k,i}))\right). \tag{7}$$

Since chunks $J_{k,\ell+1}, \ldots, J_{k,s}$ are released due to transition (3), 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 7,

$$\text{DEMAND}\,(A_k, r(J_{k,\ell}), d(J_{k,s})) = \text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \sum_{i=\ell}^{s-1} \alpha_k \left(V_k(r(J_{k,i+1})) - V_k(r(J_{k,i}))\right). \tag{8}$$

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

$$\text{DEMAND}\left(A_k, r(J_{k,\ell}), d(J_{k,\ell})\right) = \text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s})) + \alpha_k \left(V_k(r(J_{k,s})) - r(J_{k,\ell})\right). \quad (9)$$

It remains to determine $\text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ which is dependent on whether $J_{k,s}$ meets its deadline. If $J_{k,s}$ meets its deadline, then $\text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ equals $\alpha_k(V_k(g(J_{k,s}))-V_k(r(J_{k,s}))$. Lemma 1 implies that $V_k(g(J_{k,s})) \leq d(J_{k,s})$; so, $\text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ does not exceed $\alpha_k(d(J_{k,s}) - V_k(r(J_{k,s})))$. Combining this fact and Equation 9 implies Equation 6 of the lemma. Therefore, the lemma is satisfied when $J_{k,s}$ meets its deadline.

Now consider the case where $J_{k,s}$ misses its deadline. By Definition 1, $\text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ equals $\alpha_k P_k$. Observe that by antecedent of the lemma, $J_{k,s}$ is released due to transition (3); either rule (iv), (vi), or (vii) 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 $\text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ equals $\alpha_k(d(J_{k,s}) - V_k(g(J_{k,s-1})))$. Finally, substituting the new expression for $\text{DEMAND}(J_{k,s}, r(J_{k,\ell}), d(J_{k,s}))$ into Equation 9 and canceling terms gives us Equation 6 of the lemma. Thus, the lemma is also satisfied when $J_{k,s}$ misses its deadline. ■

3.2.3 Admission Control Algorithms

Adapting the proofs from from the EDF+SRP feasibility tests in [6] and [3] 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:

$$B_k = \max_{P_j > P_k} \{H_j(R_\ell) | \exists H_x(R_\ell) \neq 0 \wedge P_x \leq P_k\} \quad (10)$$

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*

$$\forall k \in \{1, \ldots, q\} \;:\; \sum_{P_i \leq P_k} \alpha_i + \frac{B_k}{P_k} \leq 1 \quad (11)$$

where the blocking term B_k is defined in Equation 10.

Proof: This test is similar to the EDF+SRP feasibility tests in [6] and [3], 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 application is in the contending with deadline before t_{miss}; since there exists a contending server from t' to t_{miss}, the processor is continuously busy in the interval $[t', t_{\text{miss}}]$. Observe that t' is guaranteed to exist at system-start time. The total demand imposed by server chunks in $[t', t_{\text{miss}}]$ is defined as the sum of the execution costs of all chunks entirely contained in that interval, ie. $\sum_{i=1}^{q} \text{DEMAND}\left(A_i, t', t_{\text{miss}}\right)$.

We will now show that the demand of any application A_k does not exceed $\alpha_k(t_{\text{miss}}-t')$. Let $Y \overset{\text{def}}{=} \{J_{k,\ell}, \ldots, J_{k,s}\}$ be the set of server chunks that the server for A_k releases in the interval $[t', t_{\text{miss}}]$ with deadlines prior or equal to t_{miss}. If Y is empty, then the demand trivially does not exceed $\alpha_k(t_{\text{miss}} - t')$,; so, assume that Y is non-empty.

Since the server for A_k is not in the contending state immediately prior t', it is either in the non-contending, inactive state, or suspended state for a non-zero-length time interval prior to t' (note this disallows the instantaneous transitions of (2) and (3), or (7) and (3)); therefore, the first chunk of Y must have been generated due to either transition (1) or (3), in which case either rule (i) or rule (v) 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 $\overline{\text{successively generate}}$ chunks $Y^{(1)}, Y^{(2)}, \ldots, Y^{(p)}$ where $Y^{(i)} \overset{\text{def}}{=} \{J^{(i)}_{k,\ell_i}, \ldots, J^{(i)}_{k,s_i}\}$. For each $Y^{(i)}$, $J^{(i)}_{k,\ell_i}$ has $V_k(r(J^{(i)}_{k,\ell_i}))$ equal to $r(J^{(i)}_{k,\ell_i})$, and $J^{(i)}_{k,\ell_i+1}, \ldots, J^{(i)}_{k,s_i}$ are all released due to transition (3). Observe the chunks of each subsequences $Y^{(i)}$ spans the interval $[r(J^{(i)}_{k,\ell_i}), d(J^{(i)}_{k,s_i})] \subseteq [t', t_{\text{miss}}]$. By Lemma 3, the demand of A_k over the subinterval $[r(J^{(i)}_{k,\ell_i}), d(J^{(i)}_{k,s_i})]$ does exceed $\alpha_k(d(J^{(i)}_{k,s_i}) - r(J^{(i)}_{k,\ell_i}))$. 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 $\sum_{i=1}^{p}(d(J^{(i)}_{k,s_i}) - r(J^{(i)}_{k,\ell_i})) \leq (t_{\text{miss}} - t')$. Therefore the total demand of A_k over interval $[t', t_{\text{miss}}]$ does not exceed $\alpha_k \left[\sum_{i=1}^{p}(d(J^{(i)}_{k,s_i}) - r(J^{(i)}_{k,\ell_i}))\right] \leq \alpha_k(t_{\text{miss}} - t')$.

Notice that only applications with period less than $(t_{\text{miss}} - t')$ 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 $P_k \leq (t_{\text{miss}} - t')$. 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 $[t', t_{\text{miss}}]$ is quantified by B_k.

Since some server chunk missed a deadline at time t_{miss}, the demand in $[t', t_{\text{miss}}]$, plus the blocking term B_k as defined in Equation 10, must exceed $(t_{\text{miss}} - t')$:

$$\sum_{P_i \leq P_k} (t_{\text{miss}} - t')\alpha_i + B_k \geq (t_{\text{miss}} - t') \tag{12}$$

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

$$\sum_{P_i \leq P_k} \alpha_k + \frac{B_k}{P_k} \geq 1 \tag{13}$$

which contradicts Equation 11. ■

However, such an exact 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 of the more exact approach 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 the 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 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, 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:

$$H_k \overset{\text{def}}{=} \max_{\ell=1}^{m} H_k(R_\ell) \tag{14}$$

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($A_k = (\alpha_k, P_k, H_k)$)
     ▷ Check if $A_k$ is schedulable:
1    if $\max_{P_i > P_k} H_i > P_k(1 - \sum_{P_j \le P_k} \alpha_j)$ return "reject"
     ▷ Check if already admitted applications
     remain schedulable:
2    for each $(P_i < P_k)$
3        do if $H_k > P_i(1 - \sum_{P_j \le P_i} \alpha_j)$ return "reject"
4    return "admit"
```

It follows from the properties of the SRP, (as proved in [3]) that the new application A_k, if admitted, may *block* the execution of applications A_i with period parameter $P_i < P_k$, and may itself be *subject to blocking* by applications A_i with period parameter $P_i > P_k$. Since the maximum amount by which any application A_i with $P_i > P_k$ may block application A_k is equal to H_i, line 1 of ALGORITHM ADMIT determines whether this blocking can cause A_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 2-3 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 4.

3.2.4 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., [1, 21]), 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.

However, we believe that it is rare that truly independently-developed applications share "corruptible" objects – good programming practice dictates that independently-developed applications 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.

When applications do share corruptible resources, we have argued above that isolation is not an achievable goal; however, *containment* [15] is. The objective in containment is to ensure that the only applications effected 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; however, discussion of these strategies is beyond the scope of this document.

3.3 Bounded delay property

The bounded-delay resource partition model, introduced by Mok et al. [25], 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* T *during which the server is continually backlogged, it receives at least*

$$(T - \Delta_k)\alpha_k$$

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 lemma. The first lemma quantifies the minimum virtual-time V_k for a server for application A_k that is in the contending 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 contending at time* t *(where* $r(J_{k,\ell}) \leq t \leq d(J_{k,\ell})$*), then*

$$V_k(t) \geq V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot \max\left(0, t - V_k(r(J_{k,\ell})) - P_k(1-\alpha_k)\right). \tag{15}$$

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 state at time t that has

$$V_k(t) < V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot \max\left(0, t - V_k(r(J_{k,\ell})) - P_k(1-\alpha_k)\right). \tag{16}$$

Since V_k never decreases, the above strict inequality implies that

$$t > V_k(r(J_{k,\ell})) + P_k(1-\alpha_k). \tag{17}$$

We will show that if Equation 16 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 (7) 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 16 and 17 satisfies the following inequality:

$$V_k(t) < V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot (t - V_k(r(J_{k,\ell})) - P_k(1 - \alpha_k)) \,. \tag{18}$$

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 18, 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 state throughout this scenario, the server will miss a deadline at $d(J_{k,\ell})$. This contradicts the given that the servers satisfied Theorem 2. Our original supposition of Equation 16 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* BROE *server for A_k,*

$$V_k(r(J_{k,\ell})) - r(J_{k,\ell}) \leq 0 \tag{19}$$

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 (iv) 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 application A_k admitted in the open environment, if the server for A_k is backlogged at time t, then*

$$|V_k(t) - t| \leq P_k(1 - \alpha_k) \tag{20}$$

Proof: If the server for A_k is in the suspended stated, 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; 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 time $V_k(t') - t' \leq P_k(1 - \alpha_k)$. If the server for A_k was contending at time t; let t' instead equal t. We will show in the remain 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 state at t' for application A_k.

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 $D_k = V_k(r(J_{k,\ell})) + P_k$,

$$V_k(t') \leq V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot \max\left((t' - r(J_{k,\ell}))\,, \alpha_k P_k\right) \tag{21}$$

Subtracting t' from both sides, observe that the RHS is maximized at t' equal to $r(J_{k,\ell}) + \alpha_k P_k$. Thus,

$$V_k(t') - t' \leq V_k(r(J_{k,\ell})) + P_k - r(J_{k,\ell}) - \alpha_k P_k \tag{22}$$

Lemma 5 implies that $V_k(r(J_{k,\ell})) - r(J_{k,\ell}) \leq 0$. Thus, Equation 22 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 at time t', we have an lower bound on the virtual time at t':

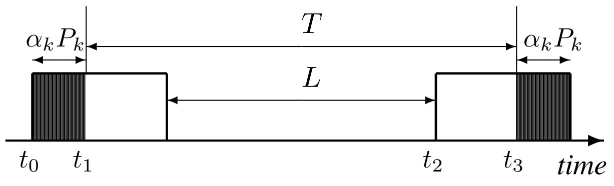


Figure 2. Worst case scenario discussed in the proof of Theorem 3. The application receives execution during the shaded intervals.

$$V_k('t) - t' \geq V_k(r(J_{k,\ell})) + \frac{1}{\alpha_k} \cdot \max\left(0, t' - V_k(r(J_{k,\ell})) - P_k(1-\alpha_k)\right) - t' \tag{23}$$

The RHS of the above inequality is minimized when t' equals $V_k(r(J_{k,\ell})) + P_k(1-\alpha_k)$. Thus, $V_k('t) - t' \geq -P_k(1-\alpha_k)$. Thus, 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.*

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 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 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) occurs when application A_k

- receives execution immediately upon entering the contending state (at time t_o in the figure), and the interval of length T begins when it completes execution and undertakes transition (2) to the suspended state (at time t_1 in the figure); and
- after having transited between the suspended and contending states an arbitrary number of times, undertakes transition (3) to enter the contending state (time t_2 in the figure) at which time it is scheduled for execution 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 L 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 $\alpha_k L$, the supply granted over interval T is $\alpha_k L = (T - 2P_k(1-\alpha_k))\alpha_k$. By the definition of P_k (Equation 1), this is equal to $(T - \Delta_k)\alpha_k$, and the lemma is proved. ■

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.

4.1 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. [25, 18] have previously addressed this problem. We restate their result, adapting it to the notation used so far.

Theorem 4 *[25, 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.

4.2 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 earlier the supply, the schedulability is preserved.*

Lemma 7 (from Feng [17]) *EDF is resource-burst-robust.*

Lemma 8 *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).* [4]

Proof: We prove the contrapositive. Assume a collection of jobs of an application A_k complete 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 $\leq t_{\text{miss}}$ awaiting execution in the partition schedule ($t_s \leftarrow 0$ if there was no such instant). Hence over $[t_s, t_{\text{miss}})$, the partition is only executing jobs with deadline $\leq t_{\text{miss}}$, or jobs that were *blocking* the execution of jobs with deadline $\leq t_{\text{miss}}$. Let Y be the set of such jobs.

Since a deadline is missed, the total amount of demand of jobs in Y during $[t_s, t_{\text{miss}})$ upon on 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 $[t_s, t_{\text{miss}})$, is $\alpha_k((t_{\text{miss}} - t_s) - \Delta_k)$.

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 $t_{\text{miss}} - \Delta_k$ in the VP schedule. Moreover, since EDF always schedules tasks according to their absolute deadline, no jobs in Y will be scheduled in interval $[t_{\text{miss}} - \Delta_k, t_{\text{miss}}]$. Therefore, the total demand of jobs in Y during $[t_s, t_{\text{miss}})$ does not exceed $\leq \alpha_k((t_{\text{miss}} - \Delta_k) - t_s)$. However, this contradicts the fact that the minimum amount of execution that is provided by the BROE server over this interval is $\alpha_k((t_{\text{miss}} - \Delta_k) - t_s)$. ■

This is a stronger result than the one in [25, Corollary 4], where applications needed to be scheduled according to VP-EDF.

Moreover, notice that since the proof doesn't 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 [3] 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 [6], the next theorem follows.

[4] In [17, 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 an application composed by two periodic tasks $\tau_1 = (1, 6, 4)$ and $\tau_2 = (1, 6, 6)$. If it is validated on a processor of speed $\alpha_k = 1/2$ then each job would finish at least $\Delta_k = 2$ time units prior to its deadline. However, if scheduled on a bounded-delay partition $(\alpha_k, \Delta_k) = (1/2, 2)$, it could potential miss a deadline when both τ_1 and τ_2 release jobs at time t and the server exhausts its budget simultaneously at t. The application may have to wait until time $t + 2$ to receive service. At which point τ_1 would execute in $[t+2, t+3)$ (exhausting the budget). The next service interval could be at latest $[t+4, t+5)$, but at that point τ_1 could release its next job and require the service. The next service time could be $[t+6, t+7)$, but at that point τ_2 would miss its deadline.

Theorem 5 *A collection of jobs is schedulable with EDF+SRP on a partition (α_k, Δ_k) if and only it is schedulable with some scheduling algorithm on an α_k-speed VP with a jitter tolerance of Δ_k, ie. 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 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 4.1 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.

4.3 Application-Level Scheduling with Other Algorithms

The application may require that a scheduler other than EDF + SRP be used to 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 sophisticate schedulability analysis instead of the validation process on a dedicated VP. This requires 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 [25, 33, 34, 22] 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 to a future paper.

5 Related work

We consider this paper to be a generalization of earlier ("first-generation") open environments (see, e.g., [25, 18, 10, 33, 17, 13]), in that our results are applicable to shared platforms comprised of serially reusable shared resources in addition to a preemptive processor.

Our work is closest in scope and ambition to the work from York described in [14], the work in progress at Malardalen outlined in the work-in-progress paper [9], and the First Scheduling Framework (FSF) [2]. Like these projects, our approach models each individual application as a collection of sporadic tasks which may share resources. One major difference between our work and both these pieces of related work concerns the approach towards sharing global resources — while both [14, 9, 2] have made the design decision that global resources will be analyzed and executed non-preemptively, we believe that this is unnecessarily restrictive[5]. The issue of scheduling global resources is explored further in Section 6 below.

Another difference between our work and the results presented in [14] concerns modularity. We have adopted 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, [14] 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.

[5]We should point out that [14] considers static-priority scheduling while we adopt an EDF-based approach.

The brief presentation in the work-in-progress paper [9] did not provide sufficient detail for us to determine whether they adopt a modular or a monolithic view of a composite open system.

Although they do not consider additional shared resources, two other projects bear similarities to our work. One is the bounded-delay resource partition work out of Texas [25, 18, 17], and the other the compositional framework studied by Shin and Lee [33]. Both these projects assume that each individual application is comprised 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).

6 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, most related work that allows global resource sharing (e.g. [14, 9]) mandates that global resources be accessed non-preemptively. 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 [6] 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 meet 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 $\xi_k(R_\ell)$ be the largest critical section of any job of A_k that accesses global resource R_ℓ. If and $\xi_k(R_\ell) \leq \Delta_k/2$ (in addition to the previously-stated constraint that the resource-hold time of $H_k(R_\ell) \leq \alpha_k P_k$), 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 $\xi_k(R_\ell) > \Delta_k/2$ 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 discussed at the end of this section.

Executing Global Critical Sections Without Local Preemptions In this section, we will assume that A_k is comprised of sporadic tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Also, we will assume that for each global resource there is a application-level ceiling $\Pi_k(R_\ell)$ for resource R_ℓ. We set $\Pi_k(R_\ell)$ to be the minimum deadline parameter of any task τ_i of A_k that access R_ℓ. Additionally, there is an application-wide ceiling that is the minimum ceiling of any global resource that is currently being accessed by some task of A_k; the scheduling decisions are identical to the rules for SRP described in Section 3.1.1. We now formally show that even if application A_k was validated upon a dedicated virtual processor of speed α_k using EDF+SRP, some critical sections may executed without local preemptions under a BROE server.

Theorem 6 *Given an application A_k (comprised of sporadic tasks) accessing globally shared resource R_ℓ can be* EDF + SRP *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 $H_k(R_\ell) \leq \alpha_k P_k$ and $\xi_k(R_\ell) \leq \Delta_k/2$ then A_k may execute any critical section accessing R_ℓ <u>with local preemptions disabled</u> on a* BROE *server with parameter (α_k, Δ_k).*

Proof: The proof is by contradiction; assume the antecedent of the theorem holds, but the application misses a deadline on a BROE server with parameters (α_k, Δ_k) when executing R_ℓ with local preemptions disabled. Let t_{miss} be the first deadline miss for A_k on its BROE server. By Lemma 8, the deadline miss must be due to executing R_ℓ without local preemptions. Let t_s be the latest time prior to t_{miss} at which there were no jobs with deadline $\leq t_{\text{miss}}$ that are awaiting execution. Thus, the server must be continuously backlogged over $[t_s, t_{\text{miss}}]$. Let Y be the set of jobs of A_k that are released after or at t_s but have deadline prior or equal to t_{miss}. The execution requirement of the jobs of Y plus the non-preemptable critical section execution on R_ℓ must exceed the execution provided to A_k over the interval $[t_s, t_{\text{miss}}]$ for the deadline miss to have occurred. Let J_{block} be the job with deadline $\geq t_{\text{miss}}$ that "blocks" jobs of Y. Let $\xi_k(R_\ell, J_{\text{block}})$ be the execution requirement of J_{block}'s critical section on R_ℓ (note $\xi_k(R_\ell, J_{\text{block}}) \leq \xi_k(R_\ell)$).

A remark about the jobs of Y: each of these jobs is generated by a task that has deadline less than $\Pi_k(R_\ell)$. Consider if this were not the case and some jobs are generated by tasks with deadline $\geq \Pi_k(R_\ell)$. In this case, these jobs could be blocked by the critical section of J_{block} when using EDF + SRP, but the absolute deadline of each of these jobs is still at most t_{miss}. Thus, the demand of Y over $[t_s, t_{\text{miss}}]$ plus the critical section of J_{block} would still exceed the execution provided to A_k over this interval and a deadline miss would still occur even under EDF + SRP (contradicting Lemma 8); so, any job of Y could not have been generated by a task with deadline $\geq \Pi_k(R_\ell)$. By definition, the resource-hold time, $H_k(R_\ell)$, accounts for the execution requirement for all jobs of tasks with deadlines $\leq \Pi_k(R_\ell)$ that may preempt J_{block} while it is holding resource R_ℓ under EDF + SRP plus the execution of $\xi_k(R_\ell, J_{\text{block}})$ (see [19] for further details on resource-hold times). So, the demand of Y does not exceed $H_k(R_\ell) - \xi_k(R_\ell, J_{\text{block}})$. Furthermore, in the dedicated VP, since each job of Y completes Δ_k prior to its deadline, no job of Y executes on the VP in the interval $[t_{\text{miss}} - \Delta_k, t_{\text{miss}}]$, which implies that $H_k(R_\ell) - \xi_k(R_\ell, J_{\text{block}}) \leq \alpha_k((t_{\text{miss}} - \Delta_k) - t_s)$.

We will now show that the BROE server will provide at least $H_k(R_\ell) - \xi_k(R_\ell, J_{\text{block}})$ units of execution over $[t_s, t_{\text{miss}}]$ to jobs of Y contradicting the assumption that A_k missed a deadline at t_{miss}. Observe that when J_{block} obtains access to global resource R_ℓ, rule (vii) implies that there is enough budget to execute $H_k(R_\ell)$ within the current chunk. If t_{miss} is greater than the current server deadline D_k, Theorem 2 and the fact there is at $H_k(R_\ell)$ budget left for the current server chunk imply that A_k will receive at least $H_k(R_\ell)$ units of execution over from the start of J_{block}'s execution of R_ℓ to t_{miss}. Thus, jobs of Y receive $H_k(R_\ell) - \xi_k(R_\ell, J_{\text{block}})$ units of execution over $[t_s, t_{\text{miss}}]$.

If t_{miss} is at most the current server deadline D_k, observe that the server is contending during the entire interval $[t_s, t_{\text{miss}}]$; the reason is there is sufficient budget upon J_{block} accessing the resource to accommodate any global resource access of jobs of Y without taking transition (vii), and the server cannot be suspending during $[t_s, t_{\text{miss}}]$ because it is continuously backlogged and will not exhaust its budget. Thus, the BROE server must not have provided sufficient execution over $[t_s, t_{\text{miss}}]$ because it was prevented from executing Y for $H_k(R_\ell) - \xi_k(R_\ell, J_{\text{block}})$

time over this interval. However, notice the least that the server A_k could be executing over this interval is $P_k - \alpha_k P_k = \Delta_k/2$ (otherwise, it would miss the server deadline and contradict Theorem 2). So, the jobs of Y could be blocked for $\xi_k(R_\ell, J_{\text{block}})$ by the execution of J_{block}'s critical section, and further prevented executing by at $\Delta_k/2$ time by other server's execution. This implies the minimum execution jobs of Y receive in $[t_s, t_{\text{miss}}]$ is $(t_{\text{miss}} - t_s - \Delta_k/2 - \xi_k(R_\ell, J_{\text{block}}))$. Since $\xi_k(R_\ell, J_{\text{block}}) \leq \xi_k(R_\ell) \leq \Delta_k/2$, the total execution received must be greater than $(t_{\text{miss}} - t_s - \Delta_k)$ which is greater than $\alpha_k(t_{\text{miss}} - t_s - \Delta_k)$, and thus greater than $H_k(R_\ell) - \xi(R_\ell, J_{\text{block}})$. ■

Notice, if the above theorem is satisfied for some A_k and R_ℓ, then we may use $\xi_k(R_\ell)$ instead of $H_k(R_\ell)$ in the admission control tests of Section 3.2.3. 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.

Reducing Resource-Hold Times $H_k(R_\ell)$. We illustrate a resource-hold-time reduction by an example. Consider the application comprised of the following three sporadic tasks, executing upon a VP of unit computing capacity[6]:

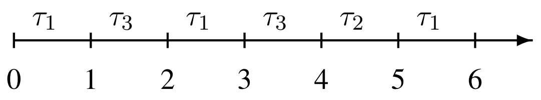
	c_i	d_i	p_i
τ_1	1	2	2
τ_2	1	5	5
τ_3	2	8	8

There is one shared resource, which is accessed by both τ_2 and τ_3 within critical sections for the entire duration of their executions.

If the shared resource is accessed non-preemptively by τ_3, then τ_1's jobs may miss their deadlines.

Now let us consider the situation if the local scheduling algorithm used is EDF+SRP (the interested reader may refer to [6] for details of the feasibility test). Under such scheduling, τ_3's execution of the shared resource may be preempted by τ_1 though not by τ_2. Consequently, τ_1's jobs all meet their deadline, as do τ_2's and τ_3's, and the system is feasible.

What is the resource holding time? The test of [6] reveals that the worst-case blocking scenario occurs when τ_1 and τ_2 both begin releasing jobs as frequently as possible immediately after τ_3 has entered its critical section. Assuming that τ_3 locks the CS at time-instant zero, the resulting EDF+SRP scheduling looks as follows:



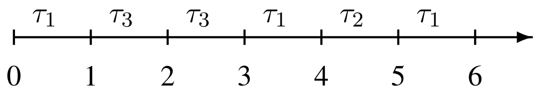
with the critical section executing over $[1, 2)$ and $[3, 4)$ and released at time-instant 4; hence, the worst case resource hold time is equal to 4.

In [19], we present an algorithm for computing resource hold times when applications are scheduled using EDF+SRP; this algorithm essentially identifies the worst-case (as we did in our example above) and computes the resource hold time for this case. We also presented an algorithm for sometimes reducing the resource hold times by introducing "dummy" critical sections and thereby changing the preemption ceilings of resources. Both algorithms from [19] may be modified for use in computing/ reducing the resource hold times that are needed to specify the application interfaces for our open environment. The straightforward modifications to these algorithms

[6]Note that this example as constructed is not really appropriate for executing upon an open environment on a unit-capacity processor, since it is itself of unit computing capacity. However, it suffice to illustrate the point regarding [non-]preemptive execution of shared resources.

that allow for the computation of resource hold times on a bounded-delay server will be presented in a journal version of this paper.

Depending upon how much one is willing to modify the local resource-access algorithm from "standard" SRP, further reduction in resource holding times may be possible. With respect to our example above, we notice that while task τ_1's jobs cannot be blocked by the entire CS (which has a WCET of 2 time units), each job of τ_1 can however tolerate one unit of blocking. This fact can be incorporated into the local algorithm which would then recognize that the job of τ_1 arriving at time-instant 2 in the "worst-case" scenario described above needn't preempt the critical section of τ_3, yielding the following schedule:



with the critical section executing over $[1, 3)$; hence, the worst case resource hold time is now reduced to 3.

That concludes our discussion of the example. Details on the algorithm for reducing resource holding times by permitting partial blocking will be presented in an extended version of [19], currently under review.

7 Discussion and Conclusions

In this paper, we have presented a design for an open environment that allows for multiple independently developed and validated applications to be multi-programmed on to 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 assumed that each individual application to be executed upon our open environment scheduled 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[7].

Let us first address the issue of task models may be used in our approach. The results obtained in this paper have assumed that each application is comprised of a collection of jobs that share resources (with the exception of Theorem 6 that assumes the sporadic task model). 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 [27, 28], generalized multiframe [7], or recurring [4, 5] task models – in essence, any formal model satisfying the *task independence assumptions* [7] 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 up on the relative priorities of the jobs according to EDF. We believe that our approach is general enough to successfully schedule such applications that have been validated by hand on a slower processor; we are currently working on proving this conjecture.

[7]Shin and Lee [33] refer to this property as *universality*.

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 [24, 20] 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 4.3, 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 local 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.

References

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

[2] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. Gutirrez, T. Lennvall, G. Lipari, J. Martnez, 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)*, pages 113–124, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

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

[4] S. Baruah. A general model for recurring real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 114–122, Madrid, Spain, December 1998. IEEE Computer Society Press.

[5] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 24(1):99–128, 2003.

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

[7] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 17(1):5–22, July 1999.

[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*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[9] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Real-time subsystem integration in the presence of shared resource. In *Proceedings of the Real-Time Systems Symposium – Work-In-Progress Session*, pages 9–12, Rio de Janerio, December 2006.

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

[11] M. Bertogna, N. Fisher, and S. Baruah. Resource-locking durations in static-priority systems. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, April 2007.

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

[13] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 389–398, Miami, Florida, 2005. IEEE Computer Society.

[14] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 257–267, Rio de Janeiro, December 2006. IEEE Computer Society Press.

[15] 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, December 2001. IEEE Computer Society Press.

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

[17] 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.

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

[19] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in edf-scheduled systems. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, April 2007.

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

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

[22] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the EuroMicro Conference on Real-time Systems*, pages 151–160, Porto, Portugal, 2003. IEEE Computer Society.

[23] G. Lipari and G. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal Of Systems Architecture*, 46(4):327–338, 2000.

[24] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

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

[26] 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.

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

[28] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, October 1997.

[29] R. Pellizzoni and G. Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems: The International Journal of Time-Critical Computing*, 30(1–2):105–128, May 2005.

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

[31] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. In *Readings in multimedia computing and networking*, pages 476–490. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[32] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

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

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