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 in the presence of shared global resources.

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 [13] 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., [22, 15, 7, 28, 14, 10] – 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 [21], 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 [26, 12, 9] 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 capable of operating upon more complex platforms. Two recent publications [11, 6] 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 [23, 5], 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

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

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 relate our open environment framework to other previously-proposed frameworks. In Section 5, we discuss in more detail how applications that use global shared resources may be scheduled locally using EDF with the Stack Resource Policy [3]. The companion technical report [16] to this paper discusses each of the issues raised in these sections in far greater detail.

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 $H_k(R_\ell)$ time-units at a time during such execution.*

We now provide a brief overview of the application interface parameters. Section 5 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 [23, 5] by a 3-tuple: $(\textit{WCET, relative deadline, period})$. 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 [22, 15, 14] with the exception of the $H_k(R_\ell)$ parameter.

Resource holding times $H_k(R_\ell)$. For open environments which choose to execute all global resources non-preemptively (such as the designs proposed in [11, 6]), $H_k(R_\ell)$ is simply the worst-case execution time upon the VP of the longest critical section holding global resource R_ℓ. We have recently [17, 8] 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) [27, 25] are instead used to arbitrate access to these global resources; in [17, 8], 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 5.

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.

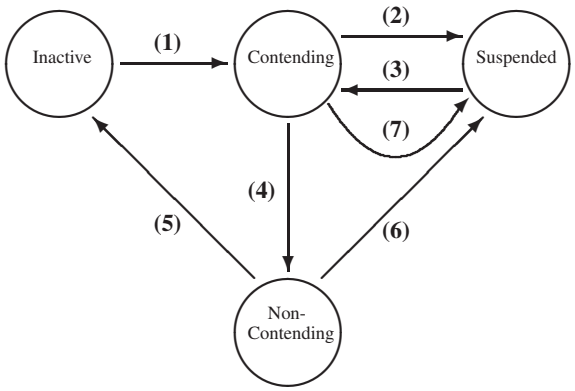

Figure 1. State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

We first describe the global scheduling algorithm used by our open environment, in Section 3.1. A description of our admission control algorithm appears in Section 3.2. The entire framework is proved correct in Section 3.3. The use of other local scheduling algorithms within our framework is briefly discussed in Section 3.4.

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[2].

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 a shared global resource R_ℓ must perform a *budget check* (i.e. is there enough execution budget to complete execution of the resource prior to D_k?). If $\alpha_k(D_k - V_k) < H_k(R_\ell)$ 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 immediately prior to entering an outermost critical section locking a global resource[3] 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 also been used in [18, 19]. 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 [9] and [11], 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

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

[2]Note there is an implicit *executing* state with transitions to and from the *contending* state. The *executing* state is omitted from our description for space reasons and because it is clear when the server will transition to and from the *executing* state.

[3]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).

(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

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

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. [9]) or implicitly, by allowing lock-holding applications to continue executing non-preemptively even when their current execution capacities are exhausted (e.g., [6, 11]).

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[4].

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. 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. Effectively choosing the server parameters is an interesting open question beyond the scope of the current paper.
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?

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 [20, 24, 4] 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.

3.2.1 Admission Control Algorithms

If the system has full knowledge of each application's usage of global resources, then the maximum blocking expe-

[4]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.

rienced by any applicatin A_k is:

$$B_k = \max_{P_j > P_k} \{H_j(R_\ell) | \exists H_x(R_\ell) \neq 0 \wedge P_x \leq P_k\} \quad (3)$$

In other words, the maximum amount of time for which A_k 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. A proof of the theorem is contained in [16].

Theorem 1 *Applications $A_1, \ldots, A_q$ may be composed upon a unit-capacity processor together without any server missing a deadline, if*

$$\forall k \in \{1, \ldots, q\} \; : \; \sum_{P_i \leq P_k} \alpha_i + \frac{B_k}{P_k} \leq 1 \quad (4)$$

where the blocking term B_k is defined in Equation 3.

However, this 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 this 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 is to effectively *ignore* the exact resource-usage of the applications in the online setting, instead considering only the maximum amount of time for which an application may choose to hold any resource; also, we did not consider the identity of this resource. That is, we required a simpler interface than the one discussed in Section 2, 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:

$$H_k \stackrel{\text{def}}{=} \max_{\ell=1}^{m} H_k(R_\ell) \quad (5)$$

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 1, and is as follows:

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

▷ Check if A_k is schedulable:
1 **if** $\max_{P_i > P_k} H_i > P_k(1 - \sum_{P_j \leq P_k} \alpha_j)$ **return** "reject"
▷ Check if already admitted applications remain schedulable:
2 **for** each $(P_i < P_k)$
3 **do if** $H_k > P_i(1 - \sum_{P_j \leq P_i} \alpha_j)$ **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.

3.2.2 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, 18]), 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* [12] 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 Proofs

In this section, we show (Theorem 2) that an admitted application that behaves as specified in its interface is guaranteed to meet all deadlines. A complete proof of the correctness of BROE is contained in the companion technical report [16]; the proof sketches contained in this section outline and present all the major ideas to give the reader a glimpse of the issues involved.

The following lemma demonstrates that our scheduling algorithm guarantees a certain minimum amount of service to each application that needs to execute, in a timely manner. This result is very similar to results that have been obtained concerning the bounded-delay resource partition model [22, 15, 14]; the major difference is that the bounded-delay resource partition model does not, to our knowledge, consider shared resources.

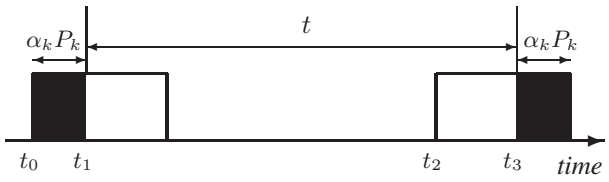

Figure 2. Worst case scenario discussed in the proof of Lemma 1. The application receives execution during the shaded intervals.

Lemma 1 *In any time interval of length t during which Application A_k served by a* BROE *server is continually backlogged, it receives at least $(t - \Delta_k)\alpha_k$ units of execution.*

Proof Sketch:

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).

From the optimality of EDF and the fact that the admission control algorithm ensures that the sum of the α_i parameters of all admitted applications does not exceed one, it can be shown that the execution received over this interval is $(t-2P_k(1-\alpha_k))\alpha_k$. By the definition of P_k (Equation 1, this is equal to $(t - \Delta_k)\alpha_k$, and the lemma is proved. ■

We are now ready to prove our major result:

Theorem 2 *If all jobs of application A_k always complete execution at least Δ_k time units prior to their deadlines when executing upon a dedicated VP of computing capacity α_k , then all jobs of A_k meet their deadlines when scheduled by* EDF + SRP *on the* BROE *server.*

Proof Sketch: Let d_o denote any time-instant, and consider a sequence of job arrivals of application A_k such that *(i)* some job has a deadline at time-instant d_o; and *(ii)* this job completes as late as possible when A_k is scheduled in

isolation upon a dedicated virtual processor (VP) of computing capacity α_k. Let t_f denote this completion time — by definition of the interface parameter Δ_k, it must be the case that $d_o - t_f \geq \Delta_k$. We will prove that all jobs with deadline $\leq d_o$ complete by their deadlines when scheduled in the open environment.

Let t_s denote the latest time-instant prior to t_f during which there are no jobs with deadline $\leq d_o$ awaiting execution in the VP schedule ($t_s \leftarrow 0$ if there was no such instant). Hence over $[t_s, t_f)$, the VP is only executing jobs with deadline $\leq d_o$, or jobs that were *blocking* the execution of jobs with deadline $\leq d_o$. Further, the total amount of such execution during $[t_s, t_f)$ upon the VP is equal to $(t_f - t_s)\alpha_k$.

Since the open environment's scheduling algorithm is EDF based, it, too, will execute these jobs in preference to jobs with deadline later than d_o after time-instant t_s, as long as there are such jobs awaiting execution.

By Lemma 1 above, application A_k is guaranteed to receive enough execution to have completed all this work over the interval $[t_s, t')$, where t' is the earliest time instant that satisfies $((t'-t_s)-\Delta_k)\times\alpha_k \geq (t_f-t_s)\alpha_k \equiv t' \geq t_f+\Delta_k$.

That is, all jobs of application A_k with deadline $\leq d_o$ complete by time-instant $t_f + \Delta_k$ upon the open environment. But as noted above it follows from the definition of the interface parameter Δ_k that $t_f + \Delta_k \leq d_o$, and the theorem is proved. ■

3.4 Application-Level Scheduling with Algorithms Other Than EDF

The application may require that a scheduler other than EDF + SRP be used to as an application-level scheduler. As mentioned in the previous subsection, our BROE server provides a similar guarantee as a bounded-delay resource partition (α_k, Δ_k) (the companion technical report [16] shows that our server execution, in fact, has the *same* guarantee as (α_k, Δ_k)). 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 [22, 28, 29, 19] and easily applied to our open environment. We believe 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.

4 Related work

We consider this paper to be a generalization of earlier ("first-generation") open environments (see, e.g., [22, 15, 7, 28, 14, 10]), 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 [11], the work in progress at Malardalen outlined in the work-in-progress paper [6], 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 [11, 6, 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 5 below.

Another difference between our work and the results presented in [11] 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, [11] 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.

The brief presentation in the work-in-progress paper [6] 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 [22, 15, 14], and the other the compositional framework studied by Shin and Lee [28]. 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).

5 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 re-

[5] We should point out that [11] considers static-priority scheduling while we adopt an EDF-based approach.

source sharing (e.g. [11, 6]) 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 [4] that no (work-conserving) scheduling strategy can result in this application being feasible upon a VP of the specified speed.
- 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. A theorem formally stating this is presented below. In some cases, it may be still 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.

Theorem 3 *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 with local preemptions disabled on a* BROE *server with parameter (α_k, Δ_k).*

The theorem above is proved correct in [16]. If the 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 3.2.1. This increases the likelihood of A_k being admitted; the reason is because the amount of time A_k could block applications A_i (with $P_i < P_k$) is decreased.

Reducing Resource-Hold Times $H_k(R_\ell)$. In [17], we present an algorithm for computing resource hold times when applications are scheduled using EDF+SRP; this algorithm essentially identifies the worst-case 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 [17] 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 that allow for the computation of resource hold times on our server will be presented in a journal version of this paper.

6 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[6]. Our framework, in fact, is general enough to handle a variety of different task models and scheduling algorithms. Section 3.4 and the companion technical report [16] briefly discusses how local schedulers other than EDF may be used within our framework; this will be elaborated further in a future journal version of this paper, in preparation.

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. 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[6] Shin and Lee [28] refer to this property as *universality*.

[12] 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.

[13] 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.

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

[15] 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.

[16] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. Technical report, Department of Computer Science, The University of North Carolina at Chapel Hill, 2007. Available at `http://www.cs.unc.edu/~fishern/pubs.html`.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.