G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. Proceedings of the IEEE Real-time Systems Symposium, pp 217-226, Orlando, FL. November 2000. IEEE Computer Society Press.

A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments *

Giuseppe Lipari John Carpenter Sanjoy Baruah

Abstract

A framework for scheduling a number of different real-time applications on a single shared preemptable processor is proposed. This framework enforces complete isolation *among the different applications, such that the behavior of each application is very similar to its behavior if it had been executing on a slower dedicated processor. A scheduling algorithm that implements this framework is presented and proved correct.*

Keywords. *Hard-real-time systems; Preemptive scheduling; Earliest deadline first; Inter-application isolation.*

1. Introduction

When several real-time applications are multiprogrammed on a single computer system, the underlying scheduling policy must provide each application with an appropriate quality of service. In most computing environments, this necessitates the enforcement of isolation between applications. The scheduler must ensure that an errant application should not be able to cause an unacceptable degradation in performance of other – well-behaved – applications.

An effective conceptual framework for modeling such systems is to associate a *server* with each application, and have a *global scheduler* resolve contention of shared resources among the servers. Each server is characterized by parameters which specify its performance expectations. A *feasibility/ admission control* test determines whether a set of servers can be scheduled on the available resources by the global scheduler such that each server receives its expected level of service (the level it would receive on a slower, dedicated system). If so, the global scheduler allocates resources at run-time in such a manner that each server's performance expectations are met. Each server is still responsible for scheduling competing jobs generated by its application, the global scheduler only makes the sharing of resources among different servers transparent to any particular server.

In this paper we present **Algorithm** PShED (Processor Sharing with Earliest Deadlines First), a global scheduling algorithm that provides guaranteed service and inter-application isolation in preemptive uniprocessor systems. Much previous research dedicated to achieving these goals (see, e.g., [13, 14, 16, 6, 17, 8, 4, 1, 10]) has assumed that jobs to be handled by a particular server are processed in a first-come first-served (FCFS) manner amongst themselves. Because few real-time applications (in particular, *hard* real-time applications) satisfy such a restriction, the guarantees that can be made by these global schedulers are severely limited. The scheduling framework we propose does not correlate the arrival time of a job with its deadline. Service guarantees can then be made to an application regardless of how its individual server schedules its jobs.

System model. In this paper, we consider a system of N applications $1, 2, \ldots, N$, each with corresponding server $S_1, S_2, \ldots, S_N$. Each server S_i is characterized by a single parameter: a *processor share* U_i, denoting the fraction of the total processor capacity devoted to application i. We restrict our attention to systems in which all servers execute on a single shared processor. Without loss of generality, this processor is assumed to have unit capacity, requiring that the sum of the processor shares of all the servers be no more than one (i.e., $\sum_{i=1}^{N} U_i \leq 1$). Algorithm PShED gives application i the appearance that its jobs are executing on a dedicated "virtual" processor of capacity U_i. If this application has hard deadlines which would all be met when scheduled on such a dedicated slower processor, then Algorithm PShED will also guarantee to meet all deadlines of this application, regardless of the behav-

*Supported in part by the National Science Foundation (Grant Nos. CCR-9704206, CCR-9972105, and CCR-9988327).

iors of other applications being scheduled with it.

The goal of having each application i behave as though executing on a dedicated processor of capacity U_i can be achieved trivially in a *processor sharing* schedule which is obtained by partitioning the timeline into infinitesimally small intervals and assigning a share U_i of the processor during each such interval to server S_i. This strategy is not useful in practice because job preemptions require execution time. While preemptions are allowed in our model, their costs are accounted for in such a way that Algorithm PShED produces a valid schedule that is practical at run-time (see Section 5).

Significance of this research. The framework we present is designed to permit several different real-time (and non real-time) applications to coexist on a single processor, while isolating each application from the others. Hence each application can be designed and implemented in isolation, with no assumptions made about the run-time behaviors of other applications that may execute concurrently with it. Our long-term research goal is to extend this framework to permit the sharing of resources other than the processor, and to incorporate resource sharing strategies that permit critical sections and nonpreemptable resources to be shared in a transparent and guaranteed manner, without each application having to make too many assumptions about the behaviors of the other applications.

We expect such a framework to be useful in general-purpose computing environments such as desktop machines, in which real-time (including *soft* and *firm* real-time) and non real-time applications coexist and contend for processors and other resources. More importantly, we are interested in providing a general framework for the development of real-time applications that permits each application to be developed in isolation, with its resource requirements completely characterized by a few parameters (in the research reported here, which does not consider the sharing of resources other than the processor, this would be a single parameter – the processor share expected by the application). We envision dynamic, distributed environments in which such applications may migrate between different processing nodes at run-time for reasons of fault-tolerance, load-balancing, and efficiency, with simple admission tests (in the current paper's framework, "is there sufficient processor capacity available to accommodate the new application?") that determine whether an application is permitted to execute on a particular processing node. Note that we are *not* expecting each application to be periodic or sporadic, or indeed to even execute "forever." It is quite possible for instance, that an individual application represents a onetime transaction that has hard real-time constraints, and that can be modeled as a finite number of real-time jobs. Provided such a transaction can be shown to successfully meet all deadlines on a dedicated processor of a particular capacity U_i, we may model its resource requirements in our framework by associating it with a server of processor-share U_i. Whenever this transaction is to be executed at run-time, this approach would require us to find a processor with sufficient spare capacity to be able to accommodate this server. Upon finding such a processor, a server of processor-share U_i is added to the set of servers being served on this processor for the duration of the transaction, and Algorithm PShED's performance guarantee ensures that the transaction will indeed meet its deadline.

Organization of this report. The remainder of this report is organized as follows. Section 2 defines Algorithm PShED and formally proves that the desired properties of guaranteed service and isolation among applications are achieved. For simplicity of argument, Section 2 assumes Algorithm PShED can correctly compute a rigorously defined notion of a server's budget. Section 3 then describes how Algorithm PShED explicitly computes these budgets, while Section 4 offers a formal proof that these computations give the budgets as defined in Section 2. Section 5 discusses implementation issues concerning context switch costs, and optimizations when servers idle. Section 6 briefly describes other research on providing guaranteed service to several applications that share a processing platform, while Section 7 concludes with a brief summary of the major points we have attempted to make here.

2. Algorithm PShED: Overview

Algorithm PShED must enforce *inter-application isolation* – it must not permit any application i to consume more than a fraction U_i of the shared processor if this would impact the performance of other applications. In order to do so, Algorithm PShED computes *budgets* for each server: these budgets keep track of the execution history of each server S_i. To arbitrate between the various servers to determine which server should have access to the processor at any instant, Algorithm PShED associates a *server deadline* D_i with each server S_i. We discuss both the server budgets and the server deadline in more detail below.

2.1 Server deadline

Algorithm PShED associates a *server deadline* D_i with S_i. Informally speaking, the value that server S_i assigns D_i at any instant is a measure of the urgency with which S_i desires the processor – the smaller the value assigned to D_i, the greater the urgency (if S_i has no active jobs awaiting execution and hence does not desire the processor, it should set D_i equal to ∞). From the perspective of Algorithm PShED, the current value of D_i is a measure of the *priority* that Algorithm PShED accords server S_i at that instant. Algorithm PShED will be performing earliest deadline first (EDF) scheduling among all eligible servers based on their D_i values.

Algorithm PShED holds each server S_i responsible for updating the value of D_i as necessary. For instance if S_i schedules its associated application's jobs in EDF order, then the value of D_i at each instant t_o should be set equal to the deadline parameter of the earliest-deadline job of server S_i that has not completed execution by time t_o. If there are no such jobs, then server S_i sets D_i equal to ∞. Since S_i is scheduling its jobs according to the EDF discipline, this would imply that D_i be updated in one of three circumstances. (i) The job with deadline D_i completes execution. (ii) A job with an earlier deadline arrives at server S_i. (iii) The job with deadline D_i has not completed, but the budget computations of Algorithm PShED indicate that S_i has used up its processor share up to instant D_i (we explain below in Section 3 how this is done). Case (iii) only occurs when there is an error within server S_i's application. What happens in this case depends upon the semantics of this application – options include *postponing* the deadline, or *aborting* this job. In either case, D_i should be set equal to the deadline of the new earliest-deadline active job of S_i.

Although Algorithm PShED uses EDF, an individual server S_i may schedule its application's jobs with any algorithm that produces reasonably consistent schedules. For the purposes of this paper we require that each server use a fully preemptive local scheduling algorithm that totally orders all jobs by priority (i.e., for any two jobs the scheduling algorithm assigns distinct priorities, based upon the jobs' parameters – arrival times, execution requirements, and deadlines), and never executes a job of lower priority while a higher priority job is active. Examples of appropriate algorithms are EDF and any preemptive fixed priority scheme (for the recurring task model). Examples of algorithms we do not consider in this paper include Least Laxity First [15] and non-preemptive schemes. If all deadlines of S_i are met while using such an algorithm on a dedicated processor of capacity U_i, then Algorithm PShED will guarantee to meet all deadlines of S_i.

No matter which scheduling discipline server S_i is using, S_i should always set its D_i value equal to the earliest deadline of all ready jobs which have not completed execution. If S_i's internal scheduler is not EDF, then at some instant t_0, S_i may choose to schedule some job other than the one whose deadline is the current D_i value.

Algorithm PShED must ensure that an errant server S_i which fails to update its D_i parameter accurately does not impact the performance of other servers in the system. The performance of the errant server S_i itself might degrade however. As we will see below, this is achieved by the server budget computations performed by Algorithm PShED.

2.2 Server budget

At each instant t_o and for all values $d \geq t_o$ that D_i has taken thus far, Algorithm PShED computes a *budget* $\mathsf{bdgt}_i(d, t_o)$ which specifies exactly how much more execution server S_i is to be permitted with D_i set to values $\leq d$. This value is determined by the "tightest" constraint of the following kind

- Let t_s denote a time instant $\leq t_o$ such that the value of D_i just prior to t_s is $> d$, and D_i is assigned a value $\leq d$ at time instant t_s.
- Let $\alpha_i(t_s, d, t_o)$ denote the amount of execution that Algorithm PShED has permitted server S_i with $D_i \leq d$, during the interval $[t_s, t_o)$.
- Clearly, $(U_i \cdot (d - t_s) - \alpha_i(t_s, d, t_o))$ is an upper bound on the remaining amount of execution that S_i is permitted with D_i set to values $\leq d$. If S_i were permitted to execute for *more* than this amount with $D_i \leq d$, then S_i would be executing for more than its permitted fraction of the processor over the interval $[t_s, d)$.

Some further definitions.

- Let $\Psi_i(d, t_o)$ denote the set of all time instants $\leq t_o$ such that the value of D_i just prior to t_s is $> d$, and D_i is assigned a value $\leq d$ at time instant t_s.
- Let $\mathsf{slack}_i(d, t_o)$ be defined as follows:

$$\mathsf{slack}_i(d, t_o) \stackrel{\text{def}}{=} \min_{t_s \in \Psi_i(d, t_o)} \{U_i \cdot (d - t_s) - \alpha_i(t_s, d, t_o)\} \quad (1)$$

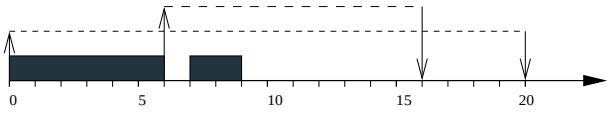

Figure 1. The scenario described in Example 1.

Thus $\mathsf{slack}_i(d, t_o)$ is an upper bound on how much more S_i can safely execute at or after time t_o with deadline $\leq d$ without using more than its share of the processor. As we move toward a formal definition of $\mathsf{bdgt}_i(d, t_o)$, we see that it must have the property that

$$\mathsf{bdgt}_i(d, t_o) \;\leq\; \mathsf{slack}_i(d, t_o). \tag{2}$$

Since at any time t_o, Algorithm PShED will permit server S_i to contend for the processor with D_i set equal to d only if $\mathsf{bdgt}_i(d, t_o) \geq 0$, the following lemma clearly holds for any definition of $\mathsf{bdgt}_i(d, t_o)$ satisfying Condition 2.

Lemma 1 *In systems scheduled using Algorithm* PShED*,*

$$(\forall i)(\forall d)(\forall t_o)\ \mathsf{slack}_i(d, t_o) \geq 0. \tag{3}$$

□

By requiring that $\mathsf{bdgt}_i(d, t_o) \leq \mathsf{slack}_i(d, t_o)$, Algorithm PShED prohibits S_i from executing for more than its fraction U_i of the shared processor, and thus *isolates* the remaining applications from i. In addition to isolating other applications from i, we would also like to *guarantee* a certain level of performance for application i. That is, we would like to permit S_i the maximum amount of execution possible without interfering with the executions of other applications. Equivalently, we would like to set $\mathsf{bdgt}_i(d, t_o)$ to be as large as possible, while still ensuring isolation. We may be tempted to try setting $\mathsf{bdgt}_i(d, t_o)$ equal to $\mathsf{slack}_i(d, t_o)$ – i.e., the "$\leq$" of Condition 2 could be replaced by an equality. The following example however illustrates that this would be incorrect.

Example 1 Consider the server S_i with $U_i = 0.5$. Suppose that D_i is initially ∞ and that S_i sets D_i to value 20 at time instant zero. Algorithm PShED schedules S_i over the interval $[0, 6)$. At $t = 6$, S_i sets D_i to 16, and is scheduled over the interval $[7, 9)$. From the definition, we see that $\Psi_i(16, 10) = \{6\}$ while $\Psi_i(20, 10) = \{0\}$. Furthermore, $\alpha_i(6, 16, 10) = 2$ while $\alpha_i(0, 20, 10) = 8$. Therefore, $\mathsf{slack}_i(16, 10) = (\frac{1}{2}(16 - 6) - 2) = 3$, and $\mathsf{slack}_i(20, 10) = (\frac{1}{2}(20 - 0) - 8) = 2$. At time instant $t_o = 10$, Algorithm PShED would compute $\mathsf{bdgt}_i(16, 10) = 2$, even though $\mathsf{slack}_i(16, 10) = \frac{1}{2} \cdot (16 - 6) - 2 = 3$. Notice that this is what one would intuitively expect. If S_i were executing on a dedicated processor half as fast as the shared one, it could have executed for ten time units over the interval $[0, 20)$. Since Algorithm PShED has permitted it to execute for eight time units with deadlines ≤ 20, it can execute for just two more time units with deadline ≤ 20, and consequently with deadline ≤ 16.

□

As Example 1 illustrates, the budget $\mathsf{bdgt}_i(d, t_o)$ depends not just upon the amount of processor consumed by S_i with deadline set $\leq d$, but is also bounded from above by the amount of processor consumed by S_i with deadline set $> d$. Thus the budget values computed by Algorithm PShED are as follows.

$$\mathsf{bdgt}_i(d, t_o) \stackrel{\text{def}}{=} \min_{d' \geq d} \{\mathsf{slack}_i(d', t_o)\} \tag{4}$$

Let $\hat{d}$ be the smallest value $> d$ which D_i has been assigned prior to t_o. Equation 5 below immediately follows.

$$\mathsf{bdgt}_i(d, t_o) = \min\left\{\mathsf{slack}_i(d, t_o), \mathsf{bdgt}_i(\hat{d}, t_o)\right\} \tag{5}$$

Consider d_1, d_2 with $d_1 < d_2$. While we cannot draw conclusions regarding the relative values of $\mathsf{slack}_i(d_1, t_o)$ and $\mathsf{slack}_i(d_2, t_o)$, the following property immediately follows from repeated applications of Condition 5.

Lemma 2 *For all i and all t_o*

$$d_1 < d_2 \Rightarrow \mathsf{bdgt}_i(d_1, t_o) \leq \mathsf{bdgt}_i(d_2, t_o).$$

□

Algorithm PShED can now be described. At each instant t_o, Algorithm PShED assigns the processor to a server S_i satisfying the following two conditions

1. $\mathsf{bdgt}_i(D_i, t_o) > 0$
2. $D_i = \min\{D_j \mid \mathsf{bdgt}_j(D_j, t_o) > 0\}$

That is, *the processor is assigned to a server that has the earliest server deadline of all the servers which have a positive budget for their current server deadlines.*

(We note that Algorithm PShED does not explicitly compute $\mathsf{bdgt}_i(D_i, t_o)$ for all i from the definition in Equation 4. Rather than computing the slack, α, and Ψ quantities at each instant, Algorithm PShED maintains data structures that allow the efficient computation of these quantities as needed. These data structures are defined and proven appropriate in Sections 3 and 4.)

2.3 Correctness of Algorithm PShED

The hard real-time guarantee property of Algorithm PShED is stated in the following theorem.

Theorem 1 . *Consider a system of N servers $S_1, S_2, \ldots, S_N$ with processor shares $U_1, U_2, \ldots, U_N$ respectively, such that $(U_1 + U_2 + \cdots + U_N) \leq 1$. If all jobs of S_i make their deadlines when scheduled on a dedicated slower processor of computing capacity U_i, then all jobs of S_i will make their deadlines when scheduled using Algorithm* PShED*.*

Proof: We argue the contrapositive, i.e., if (a job of) server S_i misses a deadline at time instant d using Algorithm PShED, then (some job of) S_i would have missed a deadline at or prior to time instant d if S_i were scheduled on a dedicated slower processor of computing capacity U_i. Assume that server S_i misses a deadline, and let d denote the first instant at which this happens. This can happen in one of two ways. (i) The $\mathsf{bdgt}(d, t_o)$ becomes equal to zero at some instant t_0 precisely at or prior to d, preventing S_i from contending for the processor by setting its deadline equal to d. (ii)The $\mathsf{bdgt}(d, d) > 0$, in which case the job missed its deadline at d despite S_i being able to contend for the processor with D_i set equal to d.

§1. Let us consider the first possibility: $\mathsf{bdgt}(d, t_o) = 0$. By Equation 4, this implies that $\exists d' \geq d$ such that $\mathsf{slack}_i(d', t_o) = 0$. Recall (Equation 1) that

$$\mathsf{slack}_i(d', t_o) \stackrel{\text{def}}{=} \min_{t_s \in \Psi(d', t_o)} \{U_i \cdot (d' - t_s) - \alpha_i(t_s, d', t_o)\} \ ,$$

and let $\mathbf{t}_s$ denote the value of $t_s \in \Psi(d', t_o)$ corresponding to the minimum of the RHS of this expression. Over the interval $[\mathbf{t}_s, d')$, server S_i has already executed as much as it would have on a slower dedicated processor of computing capacity U_i times the computing capacity of the shared processor while jobs with deadlines $\leq d$ were ready. Application i therefore must miss a deadline during this interval on the slower processor as well.

§2. Consider now the case where $\mathsf{bdgt}(d, d) > 0$. Recall that d is the earliest instant at which S_i misses a deadline. Therefore at instant d, D_i is set equal to d. Recall that Algorithm PShED schedules according to the deadline parameters D_k of the servers S_k, $k = 1, 2, \ldots, i, \ldots, n$. Let $\hat{t} < d$ be the last time instant prior to d when the processor was either idle, or was allocated to a server S_k with $D_k > d$. At that instant, all the D_k's must have been $> d$ (or blocked from execution because their budget was exhausted). Over the interval $(\hat{t}, d]$, the processor has been assigned exclusively to servers with deadline $\leq d$.

For each $k = 1, 2, \ldots, N$, let b_k denote the earliest instant $\geq \hat{t}$ at which D_k becomes $\leq d$. This implies $b_k \in \Psi_k(d, d)$, since b_k is a time instant prior to d at which D_k's value was changed from being $> d$ to being $\leq d$.

By Lemma 1, $\mathsf{slack}_k(d, d) \geq 0$ for each server S_k. From the definition of slack (Equation 1), it follows that

$$U_k \cdot (d - b_k) - \alpha_k(b_k, d, d) \geq 0.$$

Equivalently $\alpha_k(b_k, d, d) \leq U_k \cdot (d - b_k)$, which implies (since $b_k \geq \hat{t}$)

$$\alpha_k(b_k, d, d) \leq U_k \cdot (d - \hat{t}) \ . \tag{6}$$

By our choice of $\hat{t}$, the processor is never idled over the interval $(\hat{t}, d]$. Therefore, $\alpha_i(b_i, d, d)$ – the amount of execution that Algorithm PShED has permitted server S_i during the interval $[b_i, d)$ with $D_i \leq d$, – is given by

$$\begin{aligned}
\alpha_i(b_i, d, d) &= (d - \hat{t}) - \sum_{\substack{k=1 \\ k \neq i}}^{N} \alpha_k(b_k, d, d) \\
&\geq (d - \hat{t}) - \sum_{\substack{k=1 \\ k \neq i}}^{N} \left(U_k \cdot (d - \hat{t})\right) \\
&= (d - \hat{t}) - \left((d - \hat{t}) \cdot \sum_{\substack{k=1 \\ k \neq i}}^{N} U_k \right) \\
&= (d - \hat{t}) \left(1 - \sum_{\substack{k=1 \\ k \neq i}}^{N} U_k \right) \\
&\geq (d - \hat{t}) \cdot U_i. \qquad (7)
\end{aligned}$$

Since $b_i \in \Psi_i(d, d)$, it follows from the definition of slack (Equation 1) that

$$\begin{aligned}
&\mathsf{slack}_i(d, d) \leq U_i(d - b_i) - \alpha_i(b_i, d, d) \\
&\quad \Rightarrow \quad \mathsf{slack}_i(d, d) \leq U_i(d - \hat{t}) - \alpha_i(b_i, d, d) \\
&\quad \Rightarrow \quad \mathsf{slack}_i(d, d) \leq 0 \ .
\end{aligned}$$

By the definition of bdgt (Equation 4), $\mathsf{bdgt}_i(d, d) \leq \mathsf{slack}_i(d, d)$ — thus, the above inequality contradicts the assumption that $\mathsf{bdgt}_i(d, d) > 0$. □

3. Algorithm PShED: computing the bdgts

As we have seen in Section 2.2, Algorithm PShED makes scheduling decisions based on the budgets corresponding to the current server deadlines. The crucial

factor determining the run-time complexity of Algorithm PShED is the efficiency with which these budgets are computed.

Algorithm PShED maintains a **residual list** R_i associated with each server S_i from which budgets $\mathsf{bdgt}_i(d, t_o)$ are easily and efficiently computed. A residual list is a set of 3-tuples (d, β, γ) where d and β are nonnegative real numbers and γ is one of $\{\mathsf{val}, \mathsf{bnd}\}$. At any instant t_o, R_i contains a 3-tuple (d, β, γ) for each value $d \geq t_o$ that D_i has been assigned thus far, which is interpreted as follows.

- if $\gamma = \mathsf{val}$ then $\mathsf{bdgt}_i(d, t_o) = \beta$,
- if $\gamma = \mathsf{bnd}$ then $\mathsf{bdgt}_i(d, t_o) = \min\{(d - t_o) \cdot U_i, \beta\}$

That is, β is either the value of $\mathsf{bdgt}_i(d, t_o)$, or an upper bound on this value.

Algorithm PShED maintains its list of residuals R_i in *sorted* order, sorted by the first coordinate of each 3-tuple. For the remainder of this section, we will let

$$\langle (d_1, \beta_1, \gamma_1), (d_2, \beta_2, \gamma_2), \ldots, (d_\ell, \beta_\ell, \gamma_\ell), \ldots \rangle$$

denote the sorted residual list R_i at the current time instant, with $d_1 < d_2 < \cdots$. The list of residuals R_i is updated when jobs of S_i are *executed*, or server S_i *changes the value of its server deadline* D_i.

3.1 When D_i's value is changed

Suppose that server S_i changes the value of its server deadline at instant t_o. Let D_i^{old} denote the value of D_i prior to the change, and D_i^{new} the value afterwards. We assume that there is a 3-tuple $(t_o, 0, \mathsf{val})$ in the first position of R_i, and a 3-tuple $(\infty, \infty, \mathsf{val})$ in the last position of R_i. This leaves two cases.

D_i^{new} already in R_i. Suppose that there is a 3-tuple $(d_j = D_i^{\mathsf{new}}, \beta_j, \gamma_j)$ already in R_i, at the j'th location.

If $(\gamma_j \equiv \mathsf{val})$, then no change is necessary.

else $(\gamma_j \equiv \mathsf{bnd})$, in which case the assignment $\gamma_j \leftarrow \mathsf{val}$ is performed and

$$\beta_j \leftarrow \min\{\beta_j, (d_j - t_o) \cdot U_i\}. \tag{8}$$

D_i^{new} not in R_i. Suppose that the 3-tuple $(d_j = D_i^{\mathsf{new}}, \beta_j, \gamma_j)$ would occupy the j'th position in the sorted list R_i. Then $\gamma_j \leftarrow \mathsf{val}$, and β_j is computed as follows.

If $(\gamma_{j-1} \equiv \mathsf{val})$, then

$$\beta_j \leftarrow \min\{\beta_{j-1} + (d_j - d_{j-1}) \cdot U_i,\ \beta_{j+1}\} \tag{9}$$

else $(\gamma_{j-1} \equiv \mathsf{bnd})$, and

$$\beta_j \leftarrow \min\{\beta_{j-1} + (d_j - d_{j-1}) \cdot U_i,\ (d_j - t_o) \cdot U_i,\ \beta_{j+1}\}\ . \tag{10}$$

Stack of deadlines. Algorithm PShED maintains a *stack of deadlines* SoD_i with each server S_i. At each instant, the top of SoD_i contains the current value of D_i. SoD_i also contains some past values of D_i in increasing order. When D_i changes value, SoD_i is modified as follows.

if $D_i^{\mathsf{new}} < D_i^{\mathsf{old}}$, then D_i^{new} is pushed onto SoD_i.

else (i.e., $D_i^{\mathsf{new}} > D_i^{\mathsf{old}}$), values from the top of SoD_i are repeatedly popped, until the value on top of SoD_i is $\geq D_i^{\mathsf{new}}$. The γ field of the 3-tuple corresponding to each popped deadline is set equal to bnd (i.e., for each deadline $\hat{d}$ popped, the 3-tuple $(d_\ell = \hat{d}, \beta_\ell, \gamma_\ell)$ is identified and $\gamma_\ell \leftarrow \mathsf{bnd}$). If the value now on top of SoD_i is not equal to D_i^{new}, the D_i^{new} is pushed onto SoD_i.

The following lemma follows from the manner in which the stack of deadlines SoD_i is used by Algorithm PShED.

Lemma 3 *At any instant t_o, if there is a 3-tuple (d_j, β_j, γ_j) in R_i such that $d_j < D_i$ at t_o, then $\gamma_j \equiv \mathsf{bnd}$.*

That is, all β_j's stored in R_i at time instant t_o, for deadlines $<$ the current server deadline, are bounds rather than exact values. (Note that this lemma is *not* asserting the converse – i.e., it is quite possible that $\gamma_j \equiv \mathsf{bnd}$ even if $d_j > D_i$).

Proof of Lemma 3: If (d_j, β_j, γ_j) exists in R_i at instant t_0, and $d_j < D_i$, then at some moment prior to t_0, S_i must have set its D_i value to be d_j (otherwise the residual (d_j, β_j, γ_j) would not be in R_i). Call the most recent such moment (where D_i was set to d_j) t_{-1}. At t_{-1}, (d_j, β_j, γ_j) must have been on the stack because it would have been added to the stack when D_i was assigned the value d_j. Since at time $t_0 > t_{-1}$ we know $D_i > d_j$, then at some moment over $(t_{-1}, t_0]$, the value of D_i was increased from being equal to d_j, to being greater than d_j. At that moment, (d_j, β_j, γ_j) would have been popped off of the stack and γ_j would be set to bnd. Since t_{-1} was the last instant at which D_i was set to d_j, the value of γ_j in the residual has been uneffected and thus at t_0, $\gamma_j = \mathsf{bnd}$. □

3.2 Upon execution

Recall that Algorithm PShED performs earliest-deadline-first scheduling among the eligible servers, with the server's deadline parameter D_i playing the role of its deadline. Algorithm PShED considers a server eligible only if the server's budget associated with its current deadline D_i is not exhausted. Algorithm PShED monitors the budget of each server S_i via the residual list R_i associated with server S_i. More formally, at any instant t_0, each server S_i will have a residual of the form $(D_i, \beta_i, \mathsf{val})$ in the residual list R_i maintained by Algorithm PShED. Algorithm PShED then will assign the processor to server S_i only if

1. $\beta_i > 0$ in the residual $(D_i, \beta_i, \mathsf{val})$, and
2. for all other servers S_j either $D_j \geq D_i$ or $\beta_j = 0$ in the residual $(D_j, \beta_j, \mathsf{val})$.

Suppose that D_i has not changed during the interval $[t_o - \Delta, t_o)$, and that Algorithm PShED has assigned the processor to S_i during this entire interval. Then $\mathsf{bdgt}_i(d, t_o)$ is equal to $(\mathsf{bdgt}_i(d, t_o - \Delta) - \Delta)$ for all $d \geq D_i$, and consequently β_ℓ should be decremented by Δ for all $d_\ell \geq D_i$. Additionally, Algorithm PShED maintains R_i such that Lemma 2 is always satisfied: if decrementing the residual corresponding to D_i causes this residual to become smaller than the residual corresponding to a deadline $< D_i$, then the residual corresponding to the deadline $< D_i$ is also decremented to conform to Lemma 2 (such decrementing of a residual corresponding to a deadline $d_j < D_i$ occurs when the value of $\mathsf{bdgt}_i(d_j, t_o)$ is equal to $\mathsf{slack}_i(d', t_o)$ for some $d' \geq D_i$). Thus if the residual list R_i prior to the execution of S_i for Δ time units is

$$\langle (d_1, \beta_1, \gamma_1), (d_2, \beta_2, \gamma_2), \ldots, (d_\ell, \beta_\ell, \gamma_\ell), \ldots \rangle \ ,$$

then the residual list *after* the execution is

$$\langle (d_1, \beta'_1, \gamma_1), (d_2, \beta'_2, \gamma_2), \ldots, (d_\ell, \beta'_\ell, \gamma_\ell), \ldots \rangle \ ,$$

where

$$\beta'_\ell = \begin{cases} \beta_\ell - \Delta, & \text{if } d_\ell \geq D_i \\ \min\left(\beta_\ell, \beta'_{\ell+1}\right), & \text{if } d_\ell \leq D_i. \end{cases} \qquad (11)$$

4. Proof of correctness

Theorem 1 shows that Algorithm PShED meets its performance guarantees, assuming that it can accurately compute the budgets. To complete the proof of correctness of Algorithm PShED,, we now show that the method described in Section 3 accurately computes budgets.

Theorem 2 ***At any instant*** t_o, *the values of the* (d_j, β_j, γ_j) *tuples stored in* R_i *satisfy the property that*

$$\mathsf{bdgt}_i(d_j, t_o) = \begin{cases} \beta_j, & \text{if } \gamma_j = \mathsf{val} \\ \min\{(d_j - t_o) \cdot U_i, \beta_j\} & \text{if } \gamma_j = \mathsf{bnd} \end{cases}$$

Proof: The events of significance during a run of Algorithm PShED are: (i) Algorithm PShED changes its scheduling decision – i.e., either the processor transits from an idle state to executing some server, or it completes executing a server, and (ii) some server changes the value of its server deadline.

The proof is by induction on the instants at which these events of significance occur, in the order in which these events occur. If two servers change their D_i's simultaneously, they are considered serially in arbitrary order.

Base case: Each R_i is initially empty. The first 3-tuple is inserted into R_i at the instant that D_i is first assigned a value $d < \infty$. Say this occurs at time instant t_o – the 3-tuple $(d, (d - t_o) \cdot U_i, \mathsf{val})$ is inserted into R_i. By Equation 4, clearly $\mathsf{bdgt}_i(d, t_o)$ is exactly $(d - t_o) \cdot U_i$.

Induction step: Our inductive hypothesis is that each 3-tuple in R_i is "correct" at time instant $\tilde{t}$ when an event of significance occurred – i.e., each 3-tuple in R_i has the interpretation stated in the body of Theorem 2. Suppose that the next event of significance occurs at time instant $t_o > \tilde{t}$.

§1: If server S_i executed over the interval $[\tilde{t}, t_o = \tilde{t} + \Delta)$, then the update of R_i at time instant t_o is correct, since

- For each $d_j \geq D_i$, β_j is decremented by the amount Δ, reflecting the fact that the remaining (bound on) budget corresponding to d_j has been decremented by the amount executed.
- For each $d_j < D_i$ β_j is decremented as necessary to maintain monotonicity, in accordance with Lemma 2.

§2: If some other server S_j executed over the interval $[\tilde{t}, t_o = \tilde{t} + \Delta)$, then the optimality of EDF [12, 5] ensures that R_i remains correct at time instant t_o, i.e., S_i will get to execute for β_j time units prior to d_j.

§3: Suppose that the event of significance that occurs at time instant t_o is that some server S_i changes the value of its server deadline from D_i^{old} to D_i^{new}. In that case, the value of the residual β_j corresponding to server deadline $d_j = D_i^{\mathsf{new}}$ is computed according to one of Equations 8, 9, or 10, and γ_j set equal to val (thus indicating that $\mathsf{bdgt}_i(d_j, t_o)$ is exactly equal to the computed value of β_j).

To show that this computation is correct – that $\mathsf{bdgt}_i(d_j, t_o)$ is indeed exactly equal to β_j as computed – we must show that the rhs's of Equations 8, 9, and 10 do indeed compute $\mathsf{bdgt}(d_j, t_o)$. To do so, we assume the inductive hypothesis — i.e., that all 3-tuples stored in R_i are indeed correct prior to this update – and then consider each of the three equations separately, proving that the value of β_j computed in each case equals $\mathsf{bdgt}_i(d_j, t_o)$.

Equation 8: Since $\gamma_j = \mathsf{bnd}$, the value of $\mathsf{bdgt}_i(d_j, t_o)$ is, according to the inductive hypothesis, equal to $\min\{\beta_j, (d_j - t_o) \cdot U_i\}$.

Equation 9: Since $\gamma_{j-1} = \mathsf{val}$ in this case, $\mathsf{bdgt}_i(d_{j-1}, t_o) = \beta_{j-1}$ by the inductive hypothesis. By Equation 5 this implies that $\beta_{j-1} = \min\{\mathsf{slack}_i(d_{j-1}, t_o), \mathsf{bdgt}_i(d_{j+1}, t_o)\}$. From the definition of slack and the fact that D_i has thus far taken on no values between d_{j-1} and d_j, it follows that $\mathsf{slack}_i(d_j, t_o) = \mathsf{slack}_i(d_{j-1}, t_o) + (d_j - d_{j-1}) \cdot U_i$; it therefore follows that $\mathsf{bdgt}_i(d_j, t_o) = \min(\beta_{j-1} + (d_j - d_{j-1}) \cdot U_i, \beta_{j+1})$.

Equation 10: Since $\gamma_{j-1} = \mathsf{bnd}$ in this case, $\mathsf{bdgt}_i(d_{j-1}, t_o) = \min\{\beta_{j-1}, (d_{j-1} - t_o) \cdot U_i\}$ by the inductive hypothesis. By the same argument as above, $\mathsf{slack}_i(d_j, t_o) = \mathsf{slack}_i(d_{j-1}, t_o) + (d_j - d_{j-1}) \cdot U_i$; therefore, $\mathsf{bdgt}_i(d_j, t_o) = \min(\min(\beta_{j-1}, (d_{j-1} - t_o) \cdot U_i) + (d_j - d_{j-1}) \cdot U_i, \beta_{j+1})$. By algebraic simplification, the rhs of this expression reduces to $\min(\beta_{j-1} + (d_j - d_{j-1}) \cdot U_i, (d_{j-1} - t_o) \cdot U_i + (d_j - d_{j-1}) \cdot U_i, \beta_{j+1})$, which equals $\min(\beta_{j-1} + (d_j - d_{j-1}) \cdot U_i, (d_j - t_o) \cdot U_i, \beta_{j+1})$.

Suppose $D_i^{\mathsf{new}} < D_i^{\mathsf{old}}$. It follows from the optimality of EDF scheduling [12, 5] that if the residual (the "β") corresponding to D_i^{old} was exactly equal to the budget (i.e., the corresponding "γ" equals val), then this residual remains exactly equal to the budget even after the server deadline change. On the other hand, if $D_i^{\mathsf{new}} > D_i^{\mathsf{new}}$, then it does not follow from the optimality of EDF that the residuals β_j corresponding to deadlines d_j in R_i that lie between D_i^{old} and D_i^{new} which were exactly equal to $\mathsf{bdgt}_i(d_j, \tilde{t})$ prior to the deadline change, remain exactly equal to budget guarantees $\mathsf{bdgt}_i(d_j, t_o)$ at instant t_o. These residuals become upper bounds on the available budgets. This change in semantics of β_j – from perhaps being an exact value to being an upper bound – is recorded by Algorithm PShED when deadlines popped off the stack of deadlines SoD_i have their corresponding γ-values set to bnd. □

This is illustrated by the following example.

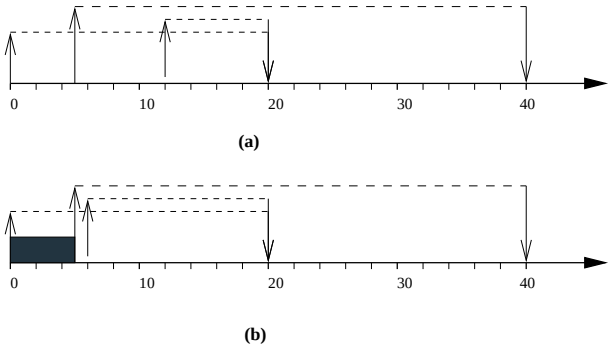

Figure 2. Scenarios described in Example 2

Example 2 Consider first server S_i with its processor share parameter U_i equal to 0.5. Suppose that D_i is initially ∞ and that S_i sets D_i to value 20 at time instant zero. The 3-tuple $(d_j, \beta_j, \gamma_j) = (20, 10, \mathsf{val})$ would be inserted in R_i. Suppose that S_i changes D_i to value 40 at time instant 5, prior to getting scheduled at all by Algorithm PShED. Algorithm PShED no longer guarantees S_i 10 units of execution by deadline 20. For instance, suppose that S_i were to then set D_i back to 20 at time instant 10. Despite the presence of the 3-tuple $(20, 10, \gamma_j)$ in R_i, $\mathsf{bdgt}_i(20, 10)$ is certainly not 10, but is $\frac{1}{2} \cdot (20 - 10) = 5$.

Consider next the same server, and suppose again that D_i is initially ∞ and that S_i sets D_i to value 20 at time instant zero. As is the case above, the 3-tuple $(20, 10, \mathsf{val})$ is inserted in R_i. Suppose that Algorithm PShED executes S_i during $[0, 5)$, at which instant the 3-tuple is $(20, 5, \mathsf{val})$. Suppose that S_i changes D_i to 40 at time instant 5, and then set it back to 20 at time instant 6. In this scenario, $\mathsf{bdgt}_i(20, 10)$ is not $\frac{1}{2} \cdot (20 - 6) = 7$, rather it is bounded from above by the value stored as the second component of the 3-tuple $(20, 5, \gamma_j)$. In this second scenario, the value of β_j represents an upper bound on the budget of S_i for execution by deadline $d_j = 20$.

□

5. Implementation issues

An idled processor. As described in Section 3, the lists of residuals maintained by Algorithm PShED track of the execution history of each server. This information is used to ensure that no one server is able to compromise the performance of others by consuming more than its reserved share of the processor. Under certain well-defined circumstances however, it is possible to permit a server to safely consume more than its reserved share of the processor. Suppose, for instance that no server has any active jobs awaiting execution at some time instant t because all jobs that arrived prior to t have either already completed execution or missed their deadlines. Algorithm PShED will detect this when all server deadlines are set to ∞. It can be shown that the future performance of all the servers is *not* compromised if the entire history stored thus far is discarded. Algorithm PShED can reinitialize the lists of residuals $R_1, R_2, \ldots, R_N$[1]. Similarly, Algorithm PShED can discard any residual (d, β, γ), for any d greater than the current instant. These two optimizations increase runtime efficiency by reducing the residual list size.

Accounting for preemption costs. The correctness of Algorithm PShED implies that when a server sets its deadline to a certain value, it is guaranteed to receive its share of the processor by that deadline. Once the budget associated with the current server deadline equals zero, it must *in*crease the value of its server deadline to receive more processing time. A natural question to ask is: why would each server not always initially set its server deadline to be arbitrarily close to the current instant, and always increase it by an arbitrarily small amount when the associated budget becomes equal to zero? This strategy allows it to obtain exactly its share of the processor over arbitrarily small intervals, and hence experience arbitrarily accurate emulation of its behavior on a dedicated server[2]. To answer this question, we need to look at the issue of *job preemptions.*

It has been shown [15] that if a set of jobs is scheduled using EDF, then the total number of context switches due to preemptions is bounded from above at twice the number of jobs. The standard way in which these preemption costs are incorporated into the schedule is by increasing the execution requirement of each job by two context switch times, and making each such job responsible for switching context twice – first, when it preempts another job to seize control of the processor for the first time, and again when it completes execution and returns control of the processor to the job with the next highest deadline. This accounts for all context switches in the system. In the framework of Algorithm PShED, this effect is achieved by "charging" each server two context switch times whenever the server deadline parameter is changed. Hence, a strategy of emulating processor sharing results in excessive charges for context switches, causing the server to waste too much of its assigned execution time with context switch charges.

Implementing residual lists. A naive implementation of the residual list, which maintains each list R_i as a linked list of 3-tuples, would result in a computational complexity of $\Theta(n)$ for each deadline-change and execution update, where n is the number of ordered pairs in R_i at the time of the operation. With a bit more effort, however, these operations can generally be done in $\mathcal{O}(\log n)$ time per operation, by storing the residual list in the form of a balanced binary tree: in particular, as a variant of the *AVL tree* [2] data structure. Space constraints prevent us from describing this any further in this paper; however, we are currently working on a more complete writeup containing these implementation details.

6. Comparison to other work

A great deal of research has been conducted on achieving guaranteed service and inter-application isolation in uniprocessor multi-application environments (see, e.g., [16, 6, 17, 1, 7, 18, 13, 8, 4, 10]). The PShED approach differs from most of these approaches in one very significant manner — in all of the above research, it has been assumed that *the jobs to be serviced by each server are processed in a first-come first-served (FCFS) manner.* (In our notation and terminology, this would require that $d_i^j \leq d_i^{j+1}$, where d_i^j denotes the deadline of the j'th job that arrives at server S_i.) In most of this earlier work the jobs do not have hard deadline parameters *a priori* assigned, but are to be scheduled to complete as soon as possible within the context of its own server. Placing such a FCFS requirement on the jobs generated by each application is a serious limitation — there are indeed very few real-time applications (in particular, *hard* real-time applications) that will satisfy such a restriction. *The* PShED *framework*

[1] In the terminology of feasibility analysis (see, e.g., [3]), the equivalent of a new "busy period" can be assumed to start with the first arrival of a job after instant t, and this busy period can be analyzed independently of what occurred in earlier busy periods.

[2] It is noteworthy that if such a strategy is employed, then Algorithm PShED reduces to the standard processor sharing algorithm – each server S_i gets the processor for $U_i \cdot \delta t$ for time units over each interval $[t, t + \delta t)$.

places no such restriction on the jobs generated by each application; the arrival time of a job is not correlated with its deadline.

The PShED approach builds directly upon the *Bandwidth Sharing Server* (BSS) [11] of Lipari and Buttazzo (see also [9]). Residual lists as a means of capturing the history of servers were introduced in [11], as *ordered pairs* (rather than 3-tuples, as in the current paper) of data. The major difference between the results described here and the ones in [11, 9] lies in the context — while BSS was designed to facilitate isolation between cooperative servers in a processor sharing environment, Algorithm PShED makes fewer requirements that the servers cooperate with each other (e.g., by being honest in the manner in which deadline parameters are updated, or in reporting arrival times of jobs). Algorithm PShED extends the work in [11, 9] by (i) placing fewer restrictions on the scheduling framework, (ii) providing a precise formulation and formal proof of the kind of hard real-time guarantee that can be made by the scheduling framework, and (iii) adding several optimizations to the framework design which permit a more efficient implementation, and provide a "cleaner" demarcation of responsibilities between the applications and the global scheduling algorithm.

7. Conclusions

We have proposed a global scheduling algorithm for use in preemptive uniprocessor systems in which several different real-time applications can execute simultaneously such that each is assured *performance guarantees*. Each application has the illusion of executing on a dedicated processor and is *isolated* from any effects of other misbehaving applications. Unlike all previous approaches to achieving such behavior, which require that jobs of each individual application be processed in first-come first-served order, our algorithm permits each server to schedule its application's jobs however it chooses. We have formally proven that an application which is feasible on a slower processor in isolation remains feasible when scheduled together with other applications using our algorithm, regardless of whether the other applications are "well-behaved" or not.

References

[1] Luca Abeni and Giorgio 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] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Math Doklady*, 3:1259–1263, 1962.

[3] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park Norwell, MA 02061, USA, 1997.

[4] 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.

[5] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

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

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

[8] H. Kaneko, J. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 206–217, Washington, DC, December 1996.

[9] Giuseppe Lipari and Sanjoy Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 166–175, Washington, DC, May–June 2000. IEEE Computer Society Press.

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

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

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

[13] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, 1993.

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

[15] 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.

[16] Marco Spuri and Giorgio Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the Real-Time Systems Symposium*, San Juan, Puerto Rico, 1994. IEEE Computer Society Press.

[17] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems: The International Journal of Time-Critical Computing*, 10(2), 1996.

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