Giuseppe Lipari and Sanjoy Baruah. Greedy reclaimation of unused bandwidth in constant-bandwidth servers. Proceedings of the EuroMicro Conference on Real-Time Systems, pp 193-200, Stockholm, Sweden. June 2000. IEEE Computer Society Press.

Greedy reclamation of unused bandwidth in constant-bandwidth servers*

Giuseppe Lipari
Scuola Superiore S. Anna, Pisa, Italy
lipari@sssup.it

Sanjoy Baruah
The University of North Carolina
baruah@cs.unc.edu

Abstract

A framework for scheduling a number of different applications on a single shared preemptable processor is proposed, such that each application seems to be executing on a slower dedicated processor. A tradeoff is identified and evaluated between how precise a notion of real time (as measured by the granularity of its clock) an application needs supported on the one hand, and the added context-switch costs imposed by our scheduling framework on the other.

Keywords. *Preemptive scheduling; Earliest deadline first; Inter-application isolation; Constant-bandwidth server; Bandwidth reclamation.*

1. Introduction

Multiprogrammed computer systems are expected to execute several threads concurrently. When some (or all) of these threads correspond to *real-time* applications, it is important that the underlying scheduling policy possess the following features:

1. Each individual thread should be *guaranteed* a certain level of service, and
2. There should be effective *isolation* among threads – an errant thread should not be able to cause an unacceptable degradation in performance in other – well-behaved – threads.

A popular conceptual framework for modelling the behaviour of such application systems is to associate a *server* with each thread, with each server characterized by certain parameters which specify exactly its performance expectations. The goal of the system-wide scheduler is then to schedule run-time resources in such a manner that each server is guaranteed a certain (quantifiable) level of service, with the exact guarantee depending upon the server parameters. That is, a server's parameters represent its *contract* with the system, and the system's global scheduler is obliged to fulfil its part of the contract by providing the level of service contracted for. However, it is incumbent upon each server, and *not* the global scheduler, to ensure that the individual jobs that comprise the thread being modelled by this server perform as expected.

In this paper, we present **Algorithm** GRUB (Greedy Reclamation of Unused Bandwidth), a global scheduling algorithm that achieves these goals in preemptive uniprocessor systems. Our methodology, which is based upon the notion of reserving a fraction of the processor bandwidth for each server, builds upon the *Constant Bandwidth Server* (CBS) of Abeni and Buttazzo [1]; the CBS, in turn, derives inspiration from the service mechanisms proposed in the *Dynamic Sporadic Server* (DSS) [13, 5] and the *Total Bandwidth Server* (TBS) [13, 12].

System Model. In our model, each server S_i is characterized by two parameters — a *processor share* U_i, and a *period* P_i. The processor share U_i denotes the fraction of total processor capacity that is to be devoted to the thread being modelled by S_i (loosely speaking, it should seem to server S_i as though its jobs are executing on a dedicated "virtual" processor, which is of speed U_i times the speed of the actual processor). The period P_i is an indication of the "granularity" of time from server S_i's perspective — while this will be elaborated upon later, it suffices for the moment to assume that the smaller the value of P_i, the more fine-grained the notion of real time for S_i.

Each server S_i generates a sequence of *jobs* $J_i^1, J_i^2, J_i^3, \ldots$, with job J_i^j becoming ready for execution ("arriving") at time a_i^j ($a_i^j \leq a_i^{j+1}$ for all i, j), and having an execution requirement equal to e_i^j time units.

*Supported in part by the National Science Foundation (Grant No. CCR-9704206).

Within each server, we assume that these jobs must be executed in FCFS order — i.e, J_i^j must complete before J_i^{j+1} can begin execution.

We make the following requirements of our scheduling discipline:

- The arrival times of the jobs (the a_i^j's) are not *a priori* known, but are only revealed on line during system execution.
- The exact execution requirements e_i^j are also not known beforehand: they can only be determined by actually executing J_i^j to completion[1].
- We are interested in integrating our scheduling methodology with traditional real-time scheduling — specifically, we wish to design a scheduler that is at worst a minor variant of the classical *Earliest Deadline First* scheduling algorithm (EDF) [3, 8]. We therefore require that our scheduling strategy be as similar to EDF as possible. *In particular, this rules out the use of scheduling strategies based upon "fair" processor-sharing, such as GPS [11] and its variants.*

In this paper, we will consider a system comprised of n servers $S_1, S_2, \ldots, S_n$, with each server S_i characterized by the parameters U_i and P_i as described above. Furthermore, we restrict our attention to systems where all of these servers execute on a single shared processor (without loss of generality, this processor is assumed to have unit processing capacity) — we therefore require that the sum of the processor shares of all the servers sum to no more than one; i.e., $(\sum_{i=1}^{n} U_i) \leq 1$.

Performance Guarantee. Recall that our goal with respect to designing the global scheduler is to be able to provide complete isolation among the servers, and to guarantee a certain degree of service to each individual server. As stated above, the processor share U_i of server S_i is a measure of the fraction of the total processor that should be devoted to executing (jobs of) server S_i. The performance guarantee that is made by our global scheduler is as follows (Theorem 1):

[1]Clearly, the kinds of performance guarantees that can be made by our scheduling algorithm are very much influenced by this requirement — in particular, it is not possible for the global scheduler to guarantee that individual jobs meet "hard" deadlines. However, we believe that making such guarantees is the responsibility of the individual server serving the hard-real-time job, and not of the global scheduler: it is incumbent upon the server, based upon the performance guarantees made to it by the global scheduler, to ensure that all hard-real-time guarantees will be met by it based upon the amount of service it is guaranteed to receive from the global scheduler. In other words, hard-real-time feasibility analysis in our model is the responsibility of the individual servers, and *not* of the global scheduler.

Suppose that job J_i^j would begin execution at time-instant A_i^j, if all jobs of server S_i were executed on a dedicated processor of capacity U_i. In such a dedicated processor, J_i^j would complete at time-instant $F_i^j \stackrel{def}{=} A_i^j + (e_i^j/U_i)$, where e_i^j denotes the execution requirement of J_i^j. If J_i^j completes execution by time-instant f_i^j when our global scheduler is used, then it is guaranteed that

$$f_i^j \leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i \ .$$

From the above inequality, it directly follows that $f_i^j < F_i^j + P_i$. This is what we mean when we refer to the period P_i of a server S_i as a measure of the "granularity" of time from the perspective of server S_i — jobs of S_i complete under Algorithm GRUB within a margin of P_i of the time they complete on a dedicated processor.

Reclaiming unused capacity. Several other server-based global schedulers (e.g., CBS [1]), can offer performance guarantees somewhat similar to the one made by Algorithm GRUB. However, Algorithm GRUB has an added feature that is not to be found in many of the other schedulers — an ability to *reclaim* unused processor capacity ("bandwidth") that is not used because some of the servers may have no outstanding jobs awaiting execution. While such reclamation does not directly effect the performance guarantee that can be made by Algorithm GRUB (since in the worst case there may be no idle threads and hence no excess capacity to reclaim), we will show that reclamation tends to result in improved system performance, in particular, with regard to the total number of preemptions in the schedule. Furthermore, *the unused capacity reclamation is achieved without any additional cost or complexity* – the computational complexity of Algorithm GRUB is the same as that of previously-proposed schedulers, and reclaiming bandwidth at a particular instant in time does not compromise the ability of the system to live up to its performance guarantees in the future.

Comparison to other work. Much research has been performed on achieving guaranteed service and inter-thread isolation in uniprocessor multi-threaded environments (see, e.g., [12, 5, 13, 1, 6, 14, 9, 7, 2]). Algorithm GRUB is most closely related to the CBS approach of Abeni and Buttazzo [1], hence, we will compare our approach to the CBS server. For an excellent comparison of the features of the other servers, see [1].

Our algorithm differs from the CBS approach in two primary ways. First, we are able to more accurately characterize the behaviour of our servers, by comparing the performance of server S_i under Algorithm GRUB to its behaviour if executed on a dedicated server of capacity U_i. In contrast, many of the performance guarantees made by CBS are somewhat circular, in that assertions can be proved about meeting or missing deadlines which are themselves assigned by the CBS algorithms.

Second and more important, Algorithm GRUB is able to efficiently *reclaim* excess processor capacity. (CBS, too, reclaims excess processor capacity in the sense that the processor is not allowed to idle while there are jobs awaiting execution; however, we will show that Algorithm GRUB is able to use reclaimed excess capacity in a more "intelligent" manner than other schedulers — i.e., to obtain a schedule with better characteristics (fewer preemptions, etc.) than that obtained by CBS and similar servers.)

2. Algorithm GRUB

In this section, we provide a detailed description of Algorithm GRUB, our global scheduler.

Algorithm Variables. For each server S_i in the system, Algorithm GRUB maintains two variables: a *deadline* D_i and a *virtual time* V_i.

- Intuitively, the value of D_i at each instant is a measure of the *priority* that Algorithm GRUB accords server S_i at that instant — Algorithm GRUB will essentially be performing earliest deadline first (EDF) scheduling based upon these D_i values.
- The value of V_i at any time is a measure of how much of server S_i's "reserved" service has been consumed by that time. Algorithm GRUB will attempt to update the value of V_i in such a manner that, *at each instant in time, server S_i has received the same amount of service that it would have received by time V_i if executing on a dedicated processor of capacity U_i.*

Algorithm GRUB is responsible for updating the values of these variables, and will make use of these variables in order to determine which job to execute at each instant in time.

Server States. At any instant in time during runtime, each server S_i is in one of three states: inactive, activeContending, or activeNonContending. The initial state of each server is inactive. Intuitively at time t_o a server is in the activeContending state if it has some jobs awaiting execution at that time; in the activeNonContending state if it has completed all jobs that arrived prior to t_o, but in doing so has "used up" its share of the processor until beyond t_o (i.e., its virtual time is greater than t_o); and in the inactive state if it has no jobs awaiting execution at time t_o, and it has *not* used up its processor share beyond t_o.

At each instant in time, Algorithm GRUB chooses for execution some server that is in its activeContending state (if there are no such servers, then the processor is idled). From among all the servers that are in their activeContending state, Algorithm GRUB chooses for execution (the next job needing execution of) the server S_i, whose deadline parameter D_i is the smallest.

While (a job of) S_i is executing, its virtual time V_i *in*creases (the exact rate of this increase will be specified later); while S_i is not executing V_i does not change. If at any time this virtual time becomes equal to the deadline ($V_i == D_i$), then the deadline parameter is incremented by P_i ($D_i \leftarrow D_i + P_i$). Notice that this may cause S_i to no longer be the earliest-deadline active server, in which case it may surrender control of the processor to an earlier-deadline server.

The following lemma establishes a relationship that will be useful in the discussion that follows.

Lemma 1 *At all times and for all servers S_i during run-time, the values of the variables V_i and D_i maintained by Algorithm* GRUB *satisfy the following inequalities*

$$V_i \leq D_i \leq V_i + P_i \ . \tag{1}$$

Proof: Follows immediately from the preceding discussion.

State Transitions. Certain (external and internal) events cause a server to change its state (see Figure 1):

1. If server S_i is in the inactive state and a job J_i^j arrives (at time-instant a_i^j), then the following code is executed

$$\begin{array}{lcl} V_i & \leftarrow & a_i^j \\ D_i & \leftarrow & V_i + P_i \end{array}$$

   and server S_i enters the activeContending state.

2. When a job J_i^{j-1} of S_i completes (notice that S_i must then be in its activeContending state), the action taken depends upon whether the next job J_i^j of S_i has already arrived.

   (a) If so, then the deadline parameter D_i is updated as follows:

$$D_i \longleftarrow V_i + P_i \ ;$$

the server remains in the activeContending state.

(b) If there is no job of S_i awaiting execution, then server S_i changes state, and enters the activeNonContending state.

3. For server S_i to be in the activeNonContending state at any instant t, it is required that $V_i > t$. If this is not so, (either immediately upon transiting into this state, or because time has elapsed but V_i does not change for servers in the activeNonContending state), then the server enters the inactive state.

4. If a new job J_i^j arrives while server S_i is in the activeNonContending state, then the deadline parameter D_i is updated as follows:

$$D_i \longleftarrow V_i + P_i \ ,$$

and server S_i returns to the activeContending state.

5. There is one additional possible state change — if the processor is ever idle, then *all* servers in the system return to their inactive state.

Incrementing virtual time. It now remains to specify how the virtual time V_i of a server S_i changes when a job of S_i is executing. Let us first consider incrementing V_i at a rate $1/U_i$:

$$\frac{d}{dt}V_i \stackrel{\text{def}}{=} \begin{cases} \frac{1}{U_i}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases} \qquad (2)$$

—intuitively, executing S_i for one time unit is equivalent to executing it for $1/U_i$ time units on a dedicated processor of capacity U_i, and we are updating V_i accordingly.

When V_i is incremented as above (i.e., at a rate $1/U_i$ while S_i is executing, and not at all the rest of the time), Algorithm GRUB is very similar to the CBS algorithm of Abeni and Buttazzo [1], and performance guarantees similar to the ones made by CBS can be proven for Algorithm GRUB as well. However, recall that one of the motivations driving our design of Algorithm GRUB is that we be able to *reclaim* processor capacity that may remain unused because some servers are in the inactive state, and that we make *efficient* use of this reclaimed bandwidth. In using excess processor capacity, though, we must be very careful to not end up using any of the *future* capacity of currently inactive servers, since we have no idea at any instant when the currently inactive servers will become active.

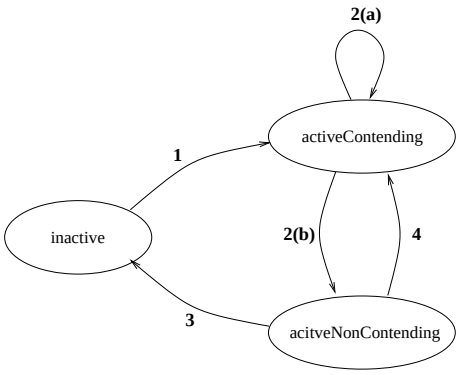

Figure 1. State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

1. Algorithm GRUB always executes the job of the server S_i which is in the activeContending state, and whose deadline parameter D_i is the smallest.
2. While a job of S_i is being executed, V_i is incremented at a rate $\frac{U}{U_i}$.
3. If V_i becomes equal to D_i while a job of S_i is executing, then D_i is incremented by an amount P_i ($D_i \leftarrow D_i + P_i$).
4. When job J_i^j arrives
   (a) If S_i is in the inactive state:
      i. $V_i \leftarrow a_i^j$
      ii. $D_i \leftarrow V_i + P_i$
      iii. $U \leftarrow U + U_i$
      iv. enter the activeContending state
   (b) If S_i is in the activeContending state:
      i. make a note of the time — a_i^j — at which this arrival occurs
   (c) If S_i is in the activeNonContending state:
      i. $D_i \leftarrow V_i + P_i$
      ii. enter the activeContending state
5. When job J_i^j completes execution (S_i must be in the activeContending state at this instant)
   <u>if</u> J_i^{j+1} has already arrived
   (a) <u>then</u> $D_i \leftarrow V_i + P_i$
   (b) <u>else</u> enter the activeNonContending state
6. If S_i is in the activeNonContending state (in which case V_i must be greater than the current time) <u>and</u> the current time becomes equal to V_i
   (a) $U \leftarrow U - U_i$
   (b) enter the inactive state.

Figure 2. PseudoCode for Algorithm GRUB.

Definition 1 *We define a server S_i to be* active *at a particular instant in time if it is in either the* activeContending *or the* activeNonContending *state at that time, and* inactive *if it is in the* inactive *state.* □

Intuitively, a server is active at time t if it is either waiting to execute jobs at instant t, or if it has already consumed its reserved processor capacity for time t.

Algorithm GRUB maintains an additional variable the *system utilization* U, which at each instant in time is equal to the sum of the capacities U_i of all servers S_i that are active at that instant in time. U is initially set equal to zero; whenever a server S_i undergoes the state-transition labelled "1" in Figure 1, U is incremented by U_i; whenever S_i undergoes the state-transition labelled "3", U is *de*cremented by U_i.

Let $[t, t + \Delta t)$ denote a time interval during which U does not change, and during which (a job of) S_i is executing. *We will assign all the excess processor capacity during this interval — a quantity of* $\Delta t \cdot (1-U)$ *— to server S_i for "free".* Consequently, S_i has used an amount equal to

$$(\Delta t - \Delta t \cdot (1 - U)) \quad = \quad \Delta t \cdot U$$

of its *own* processor capacity during this interval; equivalently, its virtual time V_i should increase by an amount equal to $\frac{\Delta t \cdot U}{U_i}$. **Algorithm GRUB's rule for updating virtual time is therefore as follows:**

$$\frac{d}{dt} V_i \stackrel{\text{def}}{=} \begin{cases} \frac{U}{U_i}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

The complete pseudocode for Algorithm GRUB is given in Figure 2.

3. Formal Analysis of Algorithm GRUB

In this section, we will formally prove that Algorithm GRUB (i) closely emulates the performance that the servers would experience if they were each executing on dedicated processors of lower capacity (Theorem 1), and (ii) distributes excess bandwidth in a somewhat fair manner (Theorem 2). But first, some definitions.

Let A_i^j and F_i^j denote the instants that job J_i^j would begin and complete execution respectively, if server S_i were executing on a dedicated processor of capacity U_i. The following expressions for A_i^j and F_i^j are easily seen to hold:

$$\begin{aligned} A_i^1 &= a_i^1 \\ F_i^1 &= A_i^1 + \frac{e_i^1}{U_i} \\ A_i^j &= \max\left(F_i^{j-1}, a_i^j\right), \text{ for } j > 1 \\ F_i^j &= A_i^j + \frac{e_i^j}{U_i}, \text{ for } j > 1 \end{aligned} \tag{4}$$

The following theorem formally states the performance guarantee that can be made by Algorithm GRUB *vis a vis* the behaviour of each server when executing on a dedicated processor:

Theorem 1 *Let f_i^j denote the time at which Algorithm* GRUB *completes execution of job J_i^j. Then the following inequality holds:*

$$f_i^j \leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i \ . \tag{5}$$

Proof: In the appendix.

Corollary 1 *The completion time of a job of server S_i when scheduled by Algorithm* GRUB *is less than P_i time units after the completion-time of the same job when S_i has its own dedicated processor.*

Proof: Observe that

$$\begin{aligned} f_i^j &\leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i \\ &< A_i^j + \left(\frac{e_i^j}{U_i \cdot P_i} + 1 \right) \cdot P_i \\ &= A_i^j + \frac{e_i^j}{U_i} + P_i \\ &= \left(A_i^j + \frac{e_i^j}{U_i} \right) + P_i \\ &= F_i^j + P_i \qquad \text{(By Equation 4)} \end{aligned}$$

Thus, f_i^j — the completion time of the j'th job of server S_i when scheduled by Algorithm GRUB — is strictly less than P_i plus F_i^j — the completion-time of the same job when S_i has its own dedicated processor.

Choosing job deadlines. A natural question to ask at this point may be: why would each server S_i not choose P_i to be arbitrarily small, and hence obtain exact emulation of its behaviour on a dedicated server? To answer this question, we need to look at the issue of *job preemptions.*

It has been shown [10] 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 *in*creasing 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 next, when it completes execution and returns control of the processor to the job with the next highest deadline. (It is easily seen that all context switches in the system are accounted for in this manner.)

As we saw in Section 2, Algorithm GRUB schedules a job with a large execution requirement by successively *postponing* the deadline according to which it is scheduled. In particular, job J_i^j's deadline may be changed as many as $\left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil$ times. For small P_i, this becomes unacceptably large, and much of server S_i's capacity could end up being spent thrashing in context switches. As a rule of thumb, it is probably best to choose a value for P_i such that $(e_i^j/U_i) \leq P_i$ for most jobs J_i^j generated by S_i.

It is noteworthy that if all the P_i's are chosen arbitrarily close to zero, then Algorithm GRUB reduces to the Generalized Processor Sharing (GPS) algorithm of Parekh and Gallagher [11].

This next theorem concerns the manner in which Algorithm GRUB distributes excess bandwidth among needy servers.

Theorem 2 *Suppose that the system utilization U is bounded from above by a constant $c < 1$ during run-time (i.e., the value of the variable U, as maintained by Algorithm* GRUB*, never exceeds c), and suppose that a server S_i has jobs awaiting execution at all times. Then S_i receives at least U_i/c of the processor capacity – i.e., the total amount of execution allowed jobs of S_i is at least U_i/c times the total capacity of the processor.*

Proof Sketch: Since S_i is active at all times, D_i is initially set to P_i and V_i to 0. V_i is now incremented at a rate U/U_i while S_i executes, until $V_i = D_i$ — i.e., S_i executes for an interval of time equal to $(P_i \cdot U_i)/U \geq (P_i \cdot U_i)/c$. At that point, D_i is incremented by P_i, and the process is repeated: i.e., for every increment of D_i by P_i, server S_i executes for $\geq (P_i \cdot U_i)/c$ time units. □

We thus see that in systems in which the overall system utilization is always bounded from above by a constant, Algorithm GRUB allocates excess capacity to needy servers in direct proportion to their processor shares. In addition, our simulation experiments indicate that Algorithm GRUB tends to exhibit similar behaviour — allocating excess capacity to needy servers in direct proportion to their processor shares — even when the excess bandwidth is not always present, but rather is randomly distributed across the time-line.

4. Evaluation of Algorithm GRUB

In designing Algorithm GRUB, we made several design decisions that may, at first glance, seem somewhat arbitrary. In this section, we will explain our reasoning behind these decisions, and will argue why we believe that these are the right decisions to have made. We have also conducted extensive experiments to evaluate the performance of Algorithm GRUB, and to compare this performance with that of other bandwidth-sharing server algorithms. While space limitations prevent us from describing our simulation experiments in much detail, we will attempt to briefly summarize our findings.

We now explain the rationale for some of the design decisions we have made with respect to Algorithm GRUB.

§1. *Why does Algorithm* GRUB *always attempt to increment the deadline parameter when considering a new job?* That is, why do we choose to execute the statement

$$D_i \leftarrow V_i + P_i$$

when first considering each job J_i^j, regardless of the current value of V_i? For instance, if V_i is smaller than D_i at the instant that J_i^j arrives (because J_i^{j-1} executed for less than expected), the CBS algorithm [1] would not increment D_i as we have done, but continue executing J_i^j with the old deadline (and consequently, with greater priority than in Algorithm GRUB).

In deciding whether to increment D_i as above or not, we had to consider the following tradeoff: If we choose to *not* increment the deadline and e_i^j (whose value is currently unknown as per our model) turns out to be quite small, then J_i^j is likely to complete within the current deadline, and hence to complete before it would in the presence of deadline-incrementing. On the other hand, not incrementing the deadline makes it more likely that J_i^j would *not* be able to complete within the current deadline, requiring deadline postponement and consequently, further preemptions.

What finally drove our design decision to always increment deadlines in Algorithm GRUB was this: by our assumption, the period parameter of a server is an indication of the granularity of the time from the perspective of the server. By not incrementing deadlines, we may obtain a response that is *quicker* than this granularity, but presumably this is not of much significance to the application (recall that our goal in designing

Algorithm GRUB is *not* to obtain a fast-response system, but rather to have a system which behaves as predicted based upon the analysis of individual servers on dedicated processors — we are distinguishing between "fast" and "predictable" systems, and our primary goal is predictability and not speed). On the other hand, the potential drawback of additional preemptions *is* a very real concern — particularly in systems where most of the job execution requirements e_i^j are known to be no larger than server "quota" $U_i \cdot P_i$, the likelihood of being able to complete without deadline postponement if we do implement deadline-incrementing is quite high, and seems worth the tradeoff.

§2. *Why does Algorithm* GRUB *assign all its excess bandwidth to the currently-executing job, rather than attempting to distribute it evenly to all active servers?* Once again, we carefully considered another alternative — assigning some of the unused bandwidth to *each* active server (perhaps in proportion to their processor share parameters). There are some advantages to such a scheme: in particular, all excess capacity is very evenly distributed, and this scheme is fair in precisely the same sense that the GPS scheme [11] is fair. However, the factors that led us to instead assign all excess bandwidth to the currently executing server included:

computational complexity: If excess capacity is to be assigned to all active servers, then we would need to update the virtual times of all these servers — this could take time linear in the number of active servers.

preemption count: Once again, recall that our primary goal in distributing excess capacity is to reduce the number of deadline postponenents — intuitively, it makes sense to devote as much excess capacity to one (or a few) server[s] in order that its job [their jobs] may complete without deadline postponement, rather than to distribute this capacity among a large number of servers, without providing enough excess capacity to any one to avoid deadline postponement. And if this be the case, then the excess capacity may as well be assigned "greedily" to the currently executing job, since its completion prior to its deadline – the earliest deadline in the system – would avoid a deadline postponement and consequent possible preemption.

In any case, while we can construct artificial workloads under which such greedy assignment of excess capacity is provably unfair to certain servers, we believe the results in Section 3 show that Algorithm GRUB does nevertheless tend to be "fair" under random workloads.

We have conducted extensive experiments to evaluate the performance of Algorithm GRUB, and to compare this performance with that of other bandwidth-sharing server algorithms. We believe that Algorithm GRUB is most similar to the CBS algorithm of Abeni and Buttazzo [1]; hence, we for the most part focused on comparing Algorithm GRUB with CBS. Space limitations prevent us from describing our experimental findings here in much detail – details will be provided in a more complete version of this report, currently under preparation. In brief, our experiments demonstrated two important additional properties of Algorithm GRUB:

1. First, schedules generated by Algorithm GRUB tend to have fewer context-switches than schedules generated by the CBS bandwidth-allocation scheme

2. Second, excess capacity *always* tends to be distributed in a weighted max-min fair manner among servers that need this excess capacity, even when the excess bandwidth is not always present, but rather is randomly distributed across the timeline (i.e, even if the amount of this excess capacity varies over time depending upon the behaviour of the individual servers).

5. Conclusions

We have proposed a global scheduling algorithm for use in preemptive uniprocessor systems in which several different applications are to execute simultaneously, such that each application is assured certain *performance guarantees* — the illusion of executing on a dedicated processor — and *isolation* from any ill-effects of other misbehaving applications. In addition, our proposed algorithm has a well-defined strategy for exploiting excess processor capacity in a manner that

- uses this excess capacity to reduce the number of preemptions and deadline-postponements in the resulting schedule, and

- is, generally speaking, *fair* (although pathological scenarios can be constructed in which some applications get most of the excess capacity while others get nothing beyond their guaranteed capacities).

We have simulated and implemented our algorithm, and have preformed extensive experiments comparing its run-time behaviour with that of other bandwidth-sharing algorithms.

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] 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.

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

[4] Jr. E. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[12] 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.

[13] 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.

[14] 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.

A. Appendix: Proof of Theorem 1

Space limitations prevent us from providing all details of the proof of correctness of Theorem 1; instead, we provide a brief sketch of the technique used. Complete details will be presented in an extended version of this paper, currently under preparation.

In order to prove the correctness of Theorem 1, we make the following observations: For any sequence of job arrivals,

- There is a processor-sharing schedule[2] in which each job J_i^j completes at or before instant $A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i$. This is simply a schedule obtained by scheduling each server S_i's jobs on a dedicated server of capacity U_i – since $\sum U_i \leq 1$, the schedule obtained by superimposing all these individual schedules is the desired processor-sharing schedule.

- From this processor-sharing schedule, we can obtain a preemptive schedule in which at most one job executes at each instant in time, by using the technique of Coffman and Denning [4, Chapter 3][3].

- And finally, we apply the well-known result concerning the optimality of the EDF scheduling algorithm to assert that Algorithm GRUB will therefore also generate a schedule in which each job J_i^j completes at or before instant $A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i$.

[2] I.e., a schedule in which fractions of the processor capacity at each instant in time may be assigned to several different jobs, provided that the sums of these fractions do not exceed one.

[3] For our purposes, this technique reduces to considering maximal contiguous time-intervals during which no (sub)job has an arrival or a deadline and partitioning this interval among the jobs that execute within it, such that each job executes for the same amount of time during the interval in both schedules. Of course, such a schedule may have an unacceptably large number of preemptions, but this needn't concern us here — we are not going to actually construct this schedule, but are merely asserting its existence.