Implementing constant-bandwidth servers upon multiprocessor platforms

Sanjoy Baruah* Joël Goossens Giuseppe Lipari

Abstract

In **constant-bandwidth server** (CBS) systems, several different applications are executed upon a shared computing platform in such a manner that each application seems to be executing on a slower dedicated processor. CBS systems have thus far only been implemented upon uniprocessors; here, a multiprocessor extension, which can be implemented upon computing platforms comprised of several identical preemptable processors, is proposed and proven correct.

1. Introduction and Motivation

Conventional real-time scheduling theory has tended to focus upon the worst-case analysis of very simple systems that are restricted to execute in strictly-controlled environments. Such systems are typically modelled as finite collections of simple, highly repetitive tasks, each of which generates jobs in a very predictable manner. These jobs have upper bounds upon their worst-case execution requirements, and associated deadlines. Real-time scheduling theory has traditionally focused upon the development of algorithms for *feasibility analysis* (determining whether all jobs can complete execution by their deadlines) and *run-time scheduling* (generating schedules at run-time for systems that are deemed to be feasible) of such systems.

This traditional perspective of real-time scheduling theory has served the safety-critical embedded systems community well. Since the consequences of failure in safety-critical systems are typically unacceptably high, it makes sense that such systems be carefully crafted, and exhaustively analyzed. However over the last decade or so, the field of real-time systems has been turning its attention to applications that are not all safety-critical, but that nevertheless have significant real-time constraints. Such applications include teleconferencing; displaying continuous media (CM) such as audio and video on general-purpose workstations; collaborative tools that permit multiple, distributed users to collaborate in a synchronized, concurrent manner; etc. Such applications have considerably broadened the focus of the discipline of real-time computing: the restricted perspective of real-time systems as small, safety-critical systems that must be subjected to worst-case analysis is proving increasingly inadequate. The new perspective is towards being able to provide significant real-time support within the context of general-purpose multi-tasking operating systems, with the understanding that not all applications need the same "degree" of real-time support — not all applications are equally important, or have deadlines that are equally "hard," or of the same time-scale. This paradigm has been somewhat formalized in the concept of open real-time environments [2].

In an **open real-time environment**, developers of each application that has real-time constraints can develop the application independently, and validate the schedulability of the application independently of other applications that may run together with it. Each such application is characterized by a few timing (and other) parameters, which characterize its real-time and resource requirements. Any real-time application may request to begin execution at run-time. The open system has a simple but accurate acceptance test that does not rely on a global schedulability analysis; rather, it treats each multi-threaded real-time application as a "black box" that is completely characterized by its specified parameters. Upon receiving a request to start a new real-time application, the open system subjects the application to the admission test and accepts the application only if the application passes the test. Once the system accepts a real-time application, it schedules the application according to the algorithm chosen by its developers, and guarantees the schedulability of the application regardless of the behavior of other applications in the system.

Most open real-time environments that have

*Contact author (email: baruah@cs.unc.edu). Supported in part by the National Science Foundation (Grant Nos. CCR-9972105, CCR-9988327, and ITR-0082866).

been implemented are based upon two-level scheduling schemes, commonly known as **bandwidth servers** [12, 1, 2]. In bandwidth server systems, there is a scheduling entity called a *server* associated with each application, and a single scheduling entity called the *resource scheduler* that arbitrates access to the shared CPU. Each server is characterized by certain parameters which specify exactly its performance expectations. The goal of the resource scheduler is 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 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 application being modelled by this server perform as expected.

The resource scheduler determines at each instant in time which application should be permitted to execute: for the application that is selected for execution, the associated *server scheduler* determines which job of that application is to execute. Such bandwidth servers typically base the resource scheduler upon the earliest deadline first scheduling algorithm (EDF) [8, 3], but permit each application to choose the kind of server scheduler that is most appropriate to its needs. Each application is characterized by a parameter known as its **required utilization**, which denotes the fraction of the shared CPU's capacity that is needed by this application. (In some servers, notably CBS [1] each application is characterized by an additional parameter — a **timeliness parameter** — indicating the granularity of time from the application's perspective: the smaller the value of this parameter, the finer the notion of time for the application and the less tolerant the application is to delays. E.g., an audio playback application would choose its timeliness parameter to be smaller than an MPEG player, since the human ear is more sensitive to temporal jitter than the human eye. It is incumbent upon the CBS scheduling algorithm to ensure that these timeliness constraints are met.) Admission control is then a simple utilization-based test: a new application is accepted by an open real-time environment if its desired utilization, when summed with the cumulative utilization of all previously-admitted applications, does not exceed the total capacity of the CPU. Of course, such an admission control test crucially depends upon the fact that the global scheduler is EDF-based, and the optimality of EDF in uniprocessor systems – since EDF is not optimal in multiprocessor environments, *such an approach does not extend directly to multiprocessor open real-time environments.*

Multiprocessor scheduling theory. Over the past few decades, real-time scheduling theory has made dramatic advances. However, many of the new results are applicable only to *uniprocessor* real-time systems. As real-time application software requirements in open environments have grown increasingly more complex, it is rapidly becoming unreasonable to expect to implement them as uniprocessor systems. Even when such uniprocessor implementations are technically feasible (i.e., devices of sufficient computing capacity do exist), they are often not cost-effective: in general, it typically costs far less to purchase k processors of a specified capacity than it does to purchase one processor that is k times as fast. Recently, therefore, there have been significant efforts to extend real-time scheduling theory to make it applicable to multiprocessor systems as well.

This research. Thus far, research in the two fields of designing open real-time environments and multiprocessor scheduling theory has proceeded independently of each other — open real-time environments that have been designed are *uni*processor systems, while most multiprocessor real-time research has focused upon developing feasibility-analysis and run-time scheduling algorithms for very simple system models (e.g., of systems comprised entirely of *periodic tasks* [8]) that execute in strictly-controlled environments. As stated above, however, the kinds of applications that have motivated the design of open real-time environments are typically very *computation-intensive*: in implementing open real-time environments upon uniprocessor platforms one very soon runs up against a **utilization bottleneck** in that few such applications can simultaneously coexist upon a single processor. The obvious solution to this bottleneck would be to implement such open real-time environments upon *multi*processor platforms; however, the current theory underlying the design of open environments does not seem to generalize to multiprocessor platforms. *The goal of the research described here is to develop the scheduling theory necessary to be able to implement open real-time environments upon multiprocessor platforms.* To this end, we apply recent results that have been obtained concerning the scheduling of periodic task systems upon multiprocessor platforms (see, e.g., [4, 13]) to design (and prove correct) **Algorithm** M-CBS, a scheduling algorithm for open multiprocessor platforms which provides service guarantees to individual servers as well as ensure inter-server isolation.

Organization. The remainder of this paper is organized as follows. In Section 2, we present the abstract model we use to represent a multi-server real-time system, and state some of our assumptions. In Section 3, we provide a detailed description of Algorithm M-CBS;

in Section 4, we prove that Algorithm M-CBS meets its design goals of providing per-application guaranteed percormance and inter-application isolation.

2. System Model

In this paper, we consider a real-time system comprised of n servers $S_1, S_2, \ldots, S_n$, that is to be implemented upon a platform comprised of m identical multiprocessors — without loss of generality, each processor is assumed to have unit processing capacity. We assume that the scheduling model allows for processor **preemption** and interprocessor **migration** – a job executing upon a processor may be interrupted at any instant, and its execution resumed later upon the same or a different processor, with no additional cost or penalty.

In our model, each server $S_i = (U_i, P_i)$ is characterized by two parameters — a *processor share* U_i, and a *period* P_i. The processor share U_i is required to be ≤ 1, and denotes the fraction of the capacity of one processor that is to be devoted to the application 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 processors). 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. 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.

Below, we will describe the properties we desire of our global scheduler. But first, a definition.

Definition 1 (Fixed-priority algorithms.) *A scheduling algorithm is said to be* **fixed-priority** *if and only if it satisfies the condition that* for every pair of jobs J_i and J_j, if J_i has higher priority than J_j at some instant in time, then J_i *always* has higher priority than J_j. ■

(Note that fixed-priority algorithms are distinct from *static*-priority algorithms, which are defined for periodic task systems and require that all the jobs generated by a particular periodic task have the same priority. All static-priority algorithms defined for periodic task systems are fixed-priority algorithms, but not all fixed-priority algorithms defined for periodic tasks are static-priority – the earliest deadline first algorithm is an example of a fixed-priority algorithm that is not static-priority. Liu [9] refers to fixed-priority algorithms as *job-level fixed priority algorithm*.)

From an implementation perspective, there are significant advantages to using fixed-priority algorithms in real-time systems. While it is beyond the scope of this document to describe in detail all these advantages, some of the more important ones are: (i) very efficient implementations of fixed-priority scheduling algorithms have been designed (see, e.g., [10]); (ii) it can be shown that when a set of jobs is scheduled using a fixed-priority algorithm then the total number of preemptions is bounded from above by the number of jobs in the set (and consequently, the total number of context switches is bounded at twice the number of jobs); similarly, (iii) it can be shown that the total number of interprocessor migrations is bounded from above by the number of jobs.

As with constant-bandwidth servers [1], we make the following requirements of the global scheduling discipline:

- We desire that our global scheduler satisfy the property of being **fixed-priority**[1]. *Note that this requirement rules out the use of scheduling strategies based upon "fair" processor-sharing, such as GPS [11] and its variants.*
- 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. Hence, our scheduling strategy cannot require knowledge of future arrival times.
- 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. Nor do we require an *a priori* upper bound (a "worst-case execution time") on the value of e_i^j.

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 a processor that should be devoted to executing (jobs of) server S_i. The performance guarantee that is made by Algorithm M-CBS is as follows (this will be formally proved in Section 4):

[1] In the uniprocessor case, CBS [1] uses **EDF** as its global scheduling algorithm. For our purposes, it is overly restrictive to require the use of EDF, and it can be shown that most of the benefits of using EDF in CBS are a direct consequence of EDF being fixed-priority; hence, we do not lose out on these benefits by using some fixed-priority algorithm other than EDF.

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 Algorithm M-CBS *is used, then it is guaranteed that*

$$f_i^j \leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i \qquad (1)$$

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 M-CBS within a margin of P_i of the time they complete on a dedicated processor.

3. Algorithm M-CBS

Let $\tau = \{S_1, S_2, \ldots, S_n\}$ denote a collection of servers with $S_i = (U_i, P_i)$, $1 \leq i \leq n$, that is to be scheduled on m unit-capacity processors, and let $U(\tau) \stackrel{def}{=} \sum_{S_i \in \tau} U_i$ denote the utilization of the collection of servers τ. Without loss of generality, we assume that *servers are indexed according to non-increasing utilization; i.e., $U_i \geq U_{i+1}$ for all i, $1 \leq i < n$.* (Recall that we require that $U_i \leq 1$ for all i, $1 \leq i \leq n$; thus, it follows that $U_1 \leq 1$.) We introduce the notation $\tau^{(i)}$ to refer to the collection of servers comprised of the $(n-i+1)$ minimum-utilization servers in τ:

$$\tau^{(i)} \stackrel{def}{=} \{S_i, S_{i+1}, \quad , S_n\}$$

(According to this notation, $\tau \equiv \tau^{(1)}$.)

3.1 Acceptance test

The collection of servers τ is accepted by Algorithm M-CBS if and only if it satisfies the following test:

$$m \geq \min_{k=1}^{n} \left\{ (k-1) + \frac{U(\tau^{(k+1)})}{1-U_k} \right\} \qquad (2)$$

Suppose that Inequality 2 is satisfied, and let $\kappa(\tau)$ denote the smallest value of k that causes Inequality 2 to be satisfied, i.e.,

$$\left(m \geq (\kappa(\tau)-1) + \left\lceil \frac{U(\tau^{(\kappa(\tau)+1)})}{1-U_{\kappa(\tau)}} \right\rceil \right) \qquad (3)$$

$$\bigwedge$$

$$\left(\forall k : 1 \leq k < \kappa(\tau) : m < (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1-U_k} \right\rceil \right)$$

Servers $S_1, S_2, \ldots, S_{\kappa(\tau)-1}$ are denoted **high-priority** servers by Algorithm M-CBS, and servers $S_{\kappa(\tau)}, \ldots, S_n$ are denoted **deadline-based** servers. During run-time, Algorithm M-CBS assigns highest priority to jobs of the high-priority servers, and schedules jobs of the deadline-based servers according to deadlines that are assigned to these jobs.

This admission-control algorithm is closely based upon an algorithm presented in [13] for scheduling systems of *periodic tasks* upon identical multiprocessor platforms. It was shown in [13] that the algorithm presented there successfully schedules any periodic task system with utilization no more than $m^2/(2m-1)$ upon m processors; furthermore, no fixed-priority scheduling algorithm can successfully schedule all periodic task systems with utilization greater than $(m+1)/2$. Very similar technqiques can be used to prove that *Condition 2 above is satisfied by any system of servers τ satisfying* $\left(U(\tau) \leq \frac{m}{2m-1}\right)$, and that no fixed-priority algorithm can guarantee both reserved capacity and inter-application isolation for systems of servers τ with $\left(U(\tau) > \frac{m+1}{2}\right)$; we omit the proofs due to space considerations. As $m \to \infty$, note that these upper and lower bounds coincide; hence from the perspective of utilization bounds, Algorithm M-CBS is an asymptotically optimal fixed-priority algorithm.

Run-time complexity. It is not difficult to see that Condition 2 can be verified for a given τ in $\mathcal{O}(n)$ time (where n denotes the number of servers in τ), if the servers are given to us in non-increasing order of utilizations; else, the servers must be sorted in $\mathcal{O}(n \log n)$ time to be in non-increasing order of utilization and Condition 2 then verified in $\mathcal{O}(n)$ time (for a total run-time complexity of $\mathcal{O}(n \log n)$). In our opinion, this is a reasonable run-time complexity for admission control. For *on-line* admission control — i.e., for deciding whether to admit a new server during run-time, while some servers are already in the system — simple sufficient (but not necessary) tests that run in $\mathcal{O}(1)$ time can be designed; we omit the details here.

3.2 Run-time behaviour

In this section, we provide a detailed description of the working of Algorithm M-CBS once a decision has been reached that real-time system τ be accepted.

§3.2.1: Server States. Three server states are defined by Algorithm M-CBS: inactive,

activeContend, and activeNonContend. High-priority servers (i.e., servers $S_1, \ldots, S_{\kappa(\tau)-1}$) are always in one of the two states inactive or activeContend; deadline-based servers may be in any of these three states. The initial state of each server is inactive. Intuitively at time t_o a server is in the activeContend state if it has some jobs awaiting execution at that time; in the activeNonContend 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; 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.

§3.2.2: Algorithm Variables. For each server S_i in the system, Algorithm M-CBS maintains a variable called a *deadline* D_i; for each deadline-based server S_i, it maintains an additional variable called the *virtual time* V_i.

- Intuitively, the value of D_i at each instant is a measure of the *priority* that Algorithm M-CBS accords server S_i at that instant — Algorithm M-CBS will essentially be performing earliest deadline first (EDF) scheduling based upon these D_i values. The value of D_i for an active high-priority server is set always equal to $(-\infty)$; for deadline-based servers, D_i is updated in a manner that is explained later in this section.

- 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. We will see that Algorithm M-CBS updates 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 computing capacity equal to U_i.*

Algorithm M-CBS 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.

At each instant in time, Algorithm M-CBS chooses for execution (at most) m servers that are in the activeContend state — from among all the servers that are in their activeContend state, Algorithm M-CBS chooses for execution (the next job needing execution of) the servers with smallest deadline parameters, with ties broken arbitrarily. (If there are fewer than m activeContend servers, then some processors are idled.)

When the first job of a deadline-based server S_i arrives, V_i is set equal to this arrival time. While (a job of) S_i is executing, its virtual time V_i *in*creases; while S_i is not executing V_i does not change

$$\frac{d}{dt}V_i \stackrel{\text{def}}{=} \begin{cases} \frac{1}{U_i}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

—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.

The treatment accorded the deadline parameter D_i is different for the high-priority and the deadline-based servers. D_i for each activeContend high-priority server is always equal to $-\infty$ (thus, such servers are always selected for execution). For the remaining servers (the deadline-based ones), if at any time the virtual time of 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 one of the m earliest-deadline active servers, in which case it may surrender control of its 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 deadline-based servers S_i during run-time, the values of the variables V_i and D_i maintained by Algorithm* M-CBS *satisfy the following inequalities*

$$V_i \leq D_i \leq V_i + P_i \quad (5)$$

Proof: Follows immediately from the preceding discussion. ■

§3.2.3: State Transitions. As stated above, each high-priority server is always in one of two states: if it has a job awaiting execution, then it is activeContend; else, it is inactive. Thus if the server is in the inactive state and a job arrives, the server transits to the activeContend state. If it is in the activeContend state and its last waiting job completes execution, then it transits to the inactive state.

The state-transition relationship for a deadline-based server is somewhat more complicated (see Figure 3.2):

1. If the 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

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

   and server S_i enters the activeContend state.

2. When a job J_i^j of S_i completes (at time-instant f_i^j) — notice that S_i must then be in its activeContend

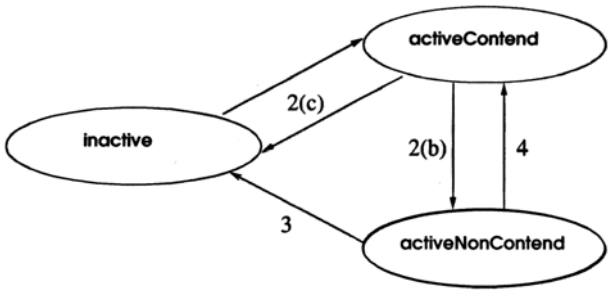

Figure 1. State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

state — the action taken depends upon whether the next job J_i^{j+1} 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 activeContend state.

(b) If there is no job of S_i awaiting execution and $V_i > f_i^j$ (i.e., the current value of V_i is *greater* than the current time) then server S_i changes state, and enters the activeNonContend state.

(c) If there is no job of S_i awaiting execution and $V_i \leq f_i^j$ (i.e., the current value of V_i is *no larger* than the current time) then, too, server S_i changes state and enters the inactive state.

3. For server S_i to be in the activeNonContend state at any instant t, it is required that $V_i > t$. When this ceases to be true, because time has elapsed since S_i entered the activeNonContend state but V_i does not change for servers in this state, then the server enters the inactive state.

4. If a new job J_i^j arrives while server S_i is in the activeNonContend 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 activeContend state.

5. There is one additional possible state change — if all m processors are ever idle at the same instant, then *all* servers in the system return to their inactive state.

For each server $S_i \in \tau$ and each integer $j \geq 1$, 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:

$$\begin{aligned} A_i^1 &= a_i^1 \\ F_i^1 &= A_i^1 + \frac{e_i^1}{U_i} \\ A_i^j &= \max\left(F_i^{j-1}, a_i^j\right), \text{ for } j > 1 \\ F_i^j &= A_i^j + \frac{e_i^j}{U_i}, \text{ for } j > 1 \end{aligned} \tag{6}$$

The following lemma will be used later in this paper.

Lemma 2 *Let S_i be any deadline-based server. At the instant that job J_i^j is first considered for scheduling by Algorithm* M-CBS, $V_i = A_i^j$.

Proof: If S_i is in the inactive state when job J_i^j arrives and is considered, then Algorithm M-CBS immediately sets V_i to a_i^j (which is in turn equal to A_i^j), and the result is seen to hold.

If S_i is in either the activeContend or activeNonContend state at the instant when J_i^j is first considered, then by definition of virtual time V_i, server S_i has by that instant 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. But that is exactly the value of A_i^j. ■

4. Formal Analysis of Algorithm M-CBS

In this section, we prove that Algorithm M-CBS does indeed make the formal performance guarantee for all jobs J_i^j:

$$f_i^j < F_i^j + P_i \ , \tag{7}$$

where f_i^j denotes the time job J_i^j completes execution under Algorithm M-CBS, and F_i^j denotes the time it

completes execution if all jobs of server S_i were executing (in a FCFS manner) on a dedicated processor of computing capacity U_i.

During any particular run, the servers $S_1, S_2, \ldots, S_n$ will generate a specific collection of jobs, and our proof obligation is to prove that the completion time of each job J_i^j in this collection satisfies Equation 7. Therefore, we will consider an arbitrary collection of jobs $\mathcal{J} \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \left(\right.$ $\left. \right)_{j \geq 1}$ representing an arbitrary run of the open real-time system τ.

- We will first prove (Section 4.1) that all jobs in $\mathcal{J}$ that were generated by the *high-priority servers* do indeed complete in a timely manner as required by Equation 7.
- We will next (Section 4.2) consider the jobs in $\mathcal{J}$ that were generated by the deadline-driven servers. Notice that Algorithm M-CBS assigns strictly higher priority to jobs of the high-priority servers than to jobs of the deadline-driven servers. There are $(\kappa(\tau)-1)$ high-priority servers, which at each instant of time consume at most $\kappa(\tau)-1$ processors; hence, $m-(\kappa(\tau)-1)$ processors are always available for the scheduling of the deadline-driven servers' jobs. We will therefore study the scheduling of only the jobs generated by the deadline-driven servers, upon $m-(\kappa(\tau)-1)$ processors; for this problem – scheduling the jobs in $\mathcal{J}$ that were generated by servers in $\tau^{(\kappa(\tau))}$ upon $(m-\kappa(\tau)+1)$ processors – we will prove that Algorithm M-CBS again completes all jobs in a timely manner, in accordance with Equation 7.
- Finally (Section 4.3), we will integrate the separate results concerning the jobs in $\mathcal{J}$ generated by high-priority and deadline-based servers, to be able to conclude that *all* jobs in $\mathcal{J}$ complete within the time-bounds required by Equation 7.

4.1 High-priority servers

Theorem 1 *For each high-priority server S_i (i.e., for all i, $1 \leq i < \kappa(\tau)$)*

$$f_i^j < F_i^j + P_i$$

Proof: From the definition of $\kappa(\tau)$ (Equation 3), it follows that $(\kappa(\tau) - 1) \leq m$; i.e., there are at most m high-priority servers (where m denotes the number of processors). Since (i) each high-priority server in the activeContend state always sets its deadline parameter equal to $(-\infty)$, (ii) servers are selected for execution by Algorithm M-CBS in order of smallest deadline parameters, and (iii) there are no more high-priority servers than there are processors, it follows that each activeContend-state high-priority server is always assigned a processor. It is therefore the case that, for all $i < \kappa(\tau)$

$$f_i^1 = a_i^1 + e_i^1 \quad (8)$$

$$f_i^j = \max\left\{a_i^j, f_i^{j-1}\right\} + e_i^j, \text{ for all } j > 1 \quad (9)$$

Thus, we see that each high-priority server S_i executing under Algortithm M-CBS experiences behaviour identical to the behaviour it would experience if it were executing on a dedicated processor of computing capacity one; since $U_i \leq 1$ for each server S_i, it follows that the completion time of each job of each such server S_i is no later than its completion time if all jobs of S_i were executing on a dedicated server of computing capacity equal to U_i. The theorem follows. ■

4.2 Deadline-driven servers

Next, we consider the jobs in $\mathcal{J}$ that were generated by the collection of servers servers $\tau^{(\kappa(\tau))} = \{\tau_{\kappa(\tau)}, \tau_{\kappa(\tau)+1}, \ldots, \tau_n\}$, comprised of all the deadline-based servers in τ.

First, some notation: recall that J_i^j is the j'th job generated by server S_i, and that it has an execution requirement e_i^j (the value of e_i^j is not known prior to completing the execution of J_i^j). Let $t \stackrel{\text{def}}{=} \left\lceil e_i^j / P_i \right\rceil$. We will consider J_i^j as t separate **"subjobs"** $J_i^j(1)$, $J_i^j(2), \ldots, J_i^j(t)$; these subjobs have execution requirements $e_i^j(1)$, $e_i^j(2), \ldots, e_i^j(t)$, respectively, where

- $e_i^j(\ell) = U_i \cdot P_i$, for $1 \leq \ell < t$, and
- $e_i^j(t) = e_i^j - \sum_{\ell=1}^{k-1} e_i^j(\ell)$

The deadlines of these jobs are set equal to $d_i^j(1)$, $d_i^j(2), \ldots, d_i^j(t)$, respectively, where

- $d_i^j(\ell) = A_i^j + \ell \cdot P_i$, for $1 \leq \ell \leq t$.

That is, we break J_i^j into equal-sized subjobs, each of execution-requirement $U_i \cdot P_i$ and deadlines P_i units apart (except that the last subjob may have an execution requirement $e_i^j(t) < U_i \cdot P_i$).

Now when J_i^j is first considered for scheduling by Algorithm M-CBS, it follows from Lemma 2 that V_i at this instant is equal to A_i^j. Consequently, the value assigned to D_i by Algorithm M-CBS at this instant (by executing the statement: "$D_i \leftarrow V_i + P_i$") is exactly

$d_i^j(1)$, and the subsequent values assigned D_i during the scheduling of J_i^j ("$D_i \leftarrow D_i + P_i$") are exactly the values $d_i^j(2), d_i^j(3), \ldots$. That is, *Algorithm* M-CBS *considers J_i^j to be a sequence of t jobs with execution requirements $e_i^j(1)$, $e_i^j(2), \ldots, e_i^j(t)$, respectively and deadlines $d_i^j(1)$, $d_i^j(2), \ldots, d_i^j(t)$, respectively, with the first subjob arriving at time-instant A_i^j and each subsequent subjob $J_i^j(\ell)$ arriving when subjob $J_i^j(\ell-1)$ completes execution.* Furthermore, the following relationship will hold:

$$d_i^j(t) < F_i^j + P_i \ , \tag{10}$$

where F_i^j denotes the time-instant at which J_i^j would complete execution of all jobs of server S_i were to execute upon a dedicated processor of computing capacity U_i.

Although our interest is the scheduling of τ upon identical multiprocessor platforms — multiprocessor machines in which all the processors are identical— we find it useful to reason in terms of a more general model of multiprocessor machines — the *uniform multiprocessor platform.*

Definition 2 (Uniform multiprocessors.) *A uniform multiprocessor platform is comprised of several processors. Each processor P is characterized by a single parameter — a* speed *(or* computing capacity*)* $\mathsf{speed}(P)$*, with the interpretation that a job that executes on processor P for t time units completes* $\mathsf{speed}(P) \times t$ *units of execution.*

Let π denote a uniform multiprocessor platform. We introduce the following notation:

$$\begin{aligned} s_1(\pi) &\stackrel{def}{=} \max_{P \in \pi} \{\mathsf{speed}(P)\} \\ S(\pi) &\stackrel{def}{=} \sum_{P \in \pi} \mathsf{speed}(P) \ . \end{aligned}$$

That is, $s_1(\pi)$ denotes the computing capacity of the fastest processor in π, and $S(\pi)$ the total computing capacity of all the processors in π. ■

Theorem 2 *There is a uniform multiprocessor platform π upon which $\tau^{(\kappa(\tau))}$ can be executed (by a clairvoyant scheduler) such that each subjob $J_i^j(\ell)$ completes at or before time-instant $d_i^j(\ell)$, which satisfies the following two properties:*

- *The fastest processor in π has computing capacity equal to the largest utilization of any server in $\tau^{(\kappa(\tau))}$. Recalling that servers are indexed according to non-increasing utilization, this is equivalent to*

$$s_1(\pi) = u_{\kappa(\tau}} \tag{11}$$

- *The cumulative computing capacity of π is equal to the sum of the utilizations of all the servers in $\tau^{(\kappa(\tau))}$; i.e.,*

$$S(\pi) = U(\tau^{(\kappa(\tau))}) \tag{12}$$

Proof Sketch: Observe that in the schedule obtained by executing server S_i on a dedicated server of capacity U_i, each subjob of J_i^j will execute such that subjobs $J_i^j(1), J_i^j(2), \ldots, J_i^j(t-1)$ all complete exactly at their deadlines $d_i^j(1), d_i^j(2), \ldots, d_i^j(t-1)$, and subjob $J_i^j(t)$ completes at or before its deadline $d_i^j(t)$.

Consequently, the uniform multiprocessor platform π obtained by dedicating a server of computing capacity U_i to each server S_i is an example platform that satisfies the conditions of this theorem. ■

The following theorem, from [4], relates feasibility of a collection of jobs upon a particular uniform multiprocessor platform π to the ability of EDF to successfully schedule this collection of jobs upon some identical multiprocessor platform.

Theorem 3 (from [4], Theorem 3) *Let π denote a uniform multiprocessor platform with cumulative processor-capacity $S(\pi)$, and in which the fastest processor has computing capacity $s_1(\pi)$, $s_\pi < 1$. Let I denote a collection of jobs, each of which is characterized by an arrival time, an execution requirement, and a deadline, that can be scheduled (by a clairvoyant scheduler) on π such that each job completes at or before its deadline. Let $\hat{m}$ denote any positive integer. If the following condition is satisfied:*

$$\hat{m} \geq \frac{S(\pi) - s_1(\pi)}{1 - s_1(\pi)} \tag{13}$$

then all jobs in I will complete at or before their deadlines when scheduled using the EDF algorithm executing on $\hat{m}$ identical unit-capacity processors.

■

We now apply Theorem 3 above to the scheduling of $\tau^{(\kappa(\tau))}$ by Algorithm M-CBS.

Theorem 4 *If $\tau^{(\kappa(\tau))}$ is scheduled using Algorithm* M-CBS *on $(m - \kappa(\tau) + 1)$ processors, then each job J_i^j will complete at or before $F_i^j + P_i$.*

Proof: As stated previously, Algorithm M-CBS considers each job J_i^j of each deadline-driven server S_i to be a sequence of t jobs with execution requirements $e_i^j(1), e_i^j(2), \ldots, e_i^j(t)$, respectively and deadlines $d_i^j(1)$, $d_i^j(2), \ldots, d_i^j(t)$, respectively, with the first subjob arriving at time-instant A_i^j and each subsequent subjob

$J_i^j(\ell)$ arriving when the $J_i^j(\ell-1)$ completes execution. Furthermore, in the absence of any high-priority servers, Algorithm M-CBS reduces to the earliest-deadline first scheduling algorithm (Algorithm EDF), executed upon $m-\kappa(\tau)+1$ processors. By Theorem 2, all these subjobs in $\mathcal{J}$ that are generated by servers in $\tau^{(\kappa(\tau))}$ can be scheduled (by a clairvoyant scheduler) upon a uniform multiprocessor platform π satisfying Conditions 11 and 12 of Theorem 2 above), such that each subjob completes at or before its deadline.

By Theorem 3, this implies that all the subjobs in $\mathcal{J}$ that are generated by servers in $\tau^{(\kappa(\tau))}$ will meet all deadlines if scheduled using Algorithm EDF upon

$$\begin{aligned}
&\left\lceil \frac{S(\pi)-s_1(\pi)}{1-s_1(\pi)} \right\rceil \\
&= \left\lceil \frac{U(\tau^{(\kappa(\tau))})-u_{\kappa(\tau)}}{1-u_{\kappa(\tau)}} \right\rceil \\
&= \left\lceil \frac{U(\tau^{(\kappa(\tau)+1)})}{1-u_{\kappa(\tau)}} \right\rceil \\
&\leq m-\kappa(\tau)+1
\end{aligned}$$

processors. As argued above, Algorithm M-CBS reduces to Algorithm EDF in the absence of any high-priority servers; hence, all the subjobs in $\mathcal{J}$ that are generated by servers in $\tau^{(\kappa(\tau))}$ will meet all deadlines if scheduled using Algorithm EDF upon $m-\kappa(\tau)+1$ processors. The theorem follows. ■

4.3 Putting the pieces together

In Section 4.1 above, we saw that all jobs generated by all high-priority servers satisfy the performance guarantee (Inequality 7) claimed by Algorithm M-CBS:

$$f_i^j < F_i^j + P_i$$

Similarly in Section 4.2, we saw that all jobs generated by all remaining (i.e., deadline-driven) servers also satisfy the performance guarantee of Inequality 7, if scheduled by Algorithm M-CBS upon $(m-\kappa(\tau)+1)$ processors. We now integrate these results and prove that all jobs of all servers in τ satisfy the performance guarantee of Inequality 7. In order to do so, we will use some ideas from Ha and Liu [7]:

Definition 3 (Predictability) *Let A denote a scheduling algorithm, and I any set of jobs. Consider any set I' of jobs obtained from I as follows: for each job $J \in I$, there is a job $J' \in I'$ such that the execution requirement of J' is $\leq$ the execution requirement of J, and all other parameters of J' and J — their arrival times, their deadlines (if defined in the model) — are identical. Scheduling algorithm A is said to be* **predictable** *if and only if for any set of jobs I and for any such I' obtained from I, it is the case that for each $J \in I$ and coresponding $J' \in I'$, the time at which J' completes execution when I' is scheduled by Algorithm A is no later than the time at which J completes execution when I is scheduled by Algorithm A.*

■

Informally, Definition 3 recognizes the fact that the specified execution-requirement parameters of jobs are typically only *upper bounds* on the actual execution-requirements during run-time, rather than the exact values. For a predictable scheduling algorithm, one may determine an upper bound on the completion-times of jobs by analyzing the situation under the assumption that each job executes for an amount equal to the upper bound on its execution requirement; it is guaranteed that the actual completion time of jobs is no later than this determined value.

The result from the work of Ha and Liu [6, 7, 5] that we will be using can be stated as follows.

Theorem 5 (Ha and Liu) *Any preemptive* fixed-priority[2] *scheduling algorithm is predictable.*

■

Since Algorithm M-CBS meets the conditions of Definition 1, it is a fixed-priority algorithm. Theorem 6 follows.

Theorem 6 *Algorithm* M-CBS *is predictable.*

■

Notice that Algorithm M-CBS always schedules jobs of high-priority servers, if these servers have any jobs awaiting execution. Let us now consider the collection of jobs $\hat{\mathcal{J}}$, obtained by *adding* jobs to $\mathcal{J}$ such that each high-priority server S_i always has a job awaiting execution. This will require us to add, for each high-priority server S_i (i.e., for each i, $1 \leq i \leq \kappa(\tau)-1$), a job which arrives at each time-instant at which S_i would enter the inactive state if executing $\mathcal{J}$, with an execution requirement exactly equal to the length of the interval during which S_i is in the inactive state.

Consider the scheduling of $\hat{\mathcal{J}}$ upon m processors by Algorithm M-CBS. Since the high-priority servers always have a job awaiting execution, these servers each completely consume a processor. There are $\kappa(\tau)-1$ high-priority processors; hence, all the remaining servers (which are exactly the ones in $\tau^{(\kappa(\tau))}$) must be scheduled exclusively upon the remaining $m-\kappa(\tau)+1$ processors. By Theorem 4, Algorithm M-CBS executes all jobs of $\tau^{(\kappa(\tau))}$ such that they complete according

[2] Defined above: Definition 1 in Section 2.

to the performance guarantee of Equation 1. We may therefore conclude that, if $\hat{\mathcal{J}}$ is scheduled using Algorithm M-CBS, then all jobs in $\mathcal{J} \subseteq \hat{\mathcal{J}}$ complete in accordance with the performance guarantee of Equation 1.

Finally, consider the scheduling of $\tilde{\mathcal{J}}$ upon m processors by Algorithm M-CBS, where $\tilde{\mathcal{J}}$ is obtained from $\hat{\mathcal{J}}$ by reducing the execution requirement of each job in $\hat{\mathcal{J}} \setminus \mathcal{J}$ to zero. By the result of Ha and Liu (Theorem 6), the completion time of each job when $\tilde{\mathcal{J}}$ is scheduled using Algorithm M-CBS is no later than the completion time of each job when $\hat{\mathcal{J}}$ is scheduled using Algorithm M-CBS. The correctness of Algorithm M-CBS now follows from the observations that (i) it therefore follows that all all jobs in $\mathcal{J} \subseteq \tilde{\mathcal{J}}$ complete in accordance with the performance guarantee of Equation 1 when $\tilde{\mathcal{J}}$ is scheduled using Algorithm M-CBS, and (ii) the non-degenerate jobs of $\tilde{\mathcal{J}}$ are exactly the jobs in $\mathcal{J}$.

We have thus shown that Algorithm M-CBS does indeed make the formal performance guarantee for *all* jobs J_i^j:

$$f_i^j < F_i^j + P_i \ ,$$

where f_i^j denotes the time job J_i^j completes execution under Algorithm M-CBS, and F_i^j denotes the time it completes execution if all jobs of server S_i were executing (in a FCFS manner) on a dedicated processor of computing capacity U_i.

5. Conclusions

The constant-bandwidth server abstraction has proved very useful in designing, implementing, and reasoning about applications that do not fit the traditional definitions of "safety-critical" and "hard" real-time, but that nevertheless expect significant real-time support within the context of general-purpose multitasking operating systems. In this paper, we have proposed Algorithm M-CBS, a global scheduling algorithm for use in preemptive multiprocessor systems in which several different time-sensitive 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. Algorithm M-CBS requires that each application be characterized by a pair of parameters, indicating the *amount* of execution that is to be reserved for the application and the time *granularity* by which this execution is to be made available to the application; based upon these specifications, Algorithm M-CBS guarantees each application its share of execution within the specified timeliness bounds.

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] Joel Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on uniform multiprocessors. *Real Time Systems*. To appear.

[5] Rhan Ha. *Validating timing constraints in multiprocessor and distributed systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995. Available as Technical Report No. UIUCDCS-R-95-1907.

[6] Rhan Ha and Jane W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. Technical Report UIUCDCS-R-93-1833, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1993.

[7] Rhan Ha and Jane W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, Los Alamitos, June 1994. IEEE Computer Society Press.

[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] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 2000.

[10] A. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proc. 5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, Washington D.C., May 1988.

[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*, pages 228–237, San Juan, Puerto Rico, 1994. IEEE Computer Society Press.

[13] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*. To appear.