A hierarchical extension to the constant bandwidth server framework[*]

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

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

Abstract

The constant bandwidth server (CBS) framework of Abeni and Buttazzo (Integrating multimedia applications in hard real-time systems. In *Proceedings of the Real-Time Systems Symposium* 1998, pp: 3–13) guarantees timely execution to individual threads in certain kinds of real-time environments. An extension to the CBS framework is proposed here, which permits the partitioning of the set of threads comprising the system into subsets representing individual applications, and extends timeliness guarantees to these applications as well.

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

1. Introduction

The Constant Bandwidth Server (CBS) scheduling framework [1] has been proposed as a means of achieving the twin goals of per-thread *performance guarantees* and inter-thread *isolation* in certain kinds of multi-threaded real-time computer systems. In this framework, each thread $T_j = (U_j, P_j)$ is characterized by two parameters — a *(worst case) utilization* U_j, and a *period* P_j. The utilization U_j denotes the amount of processor capacity that is to be devoted to the thread T_j (loosely speaking, it should seem to thread T_j as though it were executing on a dedicated "virtual" processor of computing capacity U_j). The period P_j is an indication of the "granularity" of time from thread T_j's perspective — while this will be elaborated upon later, it suffices for the moment to assume that the smaller the value of P_j, the more fine-grained the notion of real time for T_j. It is assumed that each thread T_j generates a sequence of *jobs* $J_j^1, J_j^2, J_j^3, \ldots$, with job J_j^k becoming ready for execution ("arriving") at time a_j^k ($a_j^k \le a_j^{k+1}$ for all j, k), and having an execution requirement equal to e_j^k time units (no a priori knowledge of either the arrival times or the execution requirements is assumed). Within each thread, we require that the jobs must be executed in FCFS order — i.e, J_j^k must complete before J_j^{k+1} can begin execution. Given a set τ of such threads to be executed on a single shared preemptable processor of computing capacity $\sum_{T_j \in \tau} U_j$, the CBS scheduling framework makes the following performance guarantee: Let F_j^k denote the time instant at which job J_j^k would complete execution, if all jobs of thread T_j were executed on a dedicated processor of capacity U_j. Let f_j^k denote the time instant at which J_j^k completes execution under CBS. It is guaranteed that

$$f_j^k < F_j^k + P_j \ ; \qquad (1)$$

i.e., *each job of each thread T_j is guaranteed to complete under CBS no more than P_j time units later than the time it would complete if executing on a dedicated processor.* (This is what we mean when we refer to the period P_j of a thread T_j as a measure of the "granularity" of time from the perspective of thread T_j — jobs of T_j complete under CBS within a margin of P_j of the time they would complete on a dedicated processor.)

Hierarchical schedulers. One consequence of the guaranteed-performance property of CBS is that a thread T_j that is continually backlogged (i.e., always has jobs awaiting execution) is asymptotically guaranteed a fraction $U_j/(\sum_{T_\ell \in \tau} U_\ell)$ of the processor capacity. There has recently been considerable interest in being able to extend such guarantees to a *hierarchical composition* of threads, while maintaining the timing guarantees to individual threads (i.e., guaranteeing each thread that it continues to get to execute

[*]Supported in part by the National Science Foundation (Grant No. CCR-9972105, CCR-9988327, and CCR-0082866).

in a timely manner). One of the motivations for developing scheduling frameworks that are thus hierarchical in nature [3, 2, 10, 9] is the recognition of the fact that individual applications may be comprised of several sub-applications (and eventually, several threads); in a hierarchical scheduling framework, each application would request a certain amount of the shared resource (which, in the context of this paper, is a single preemptable CPU), and then further distribute this resource among the various sub-applications that comprise it. The goals of guaranteed performance and isolation, which CBS provides to each thread, should now also be met with respect to each *application* in addition to each thread – each application should be guaranteed a certain degree of performance, and applications should be shielded from the effects of mis-behaving applications.

As a concrete example, consider three threads T_1, T_2, and T_3 each with $U_j = 1/3$, executing on a processor of unit computing capacity. Suppose that threads T_1 and T_2 belong to one application which expects to be assigned two-thirds of the total processor capacity, and thread T_3 is the sole thread of another application which desires the remaining one-third of the processor capacity. If each thread were continually backlogged, then CBS —as defined in [1]— would guarantee each thread one third of the processor capacity – this is what the application semantics expect. However, if thread T_2 were idle while the other two were continually backlogged, then CBS would end up assigning each thread one half the processor capacity, rather than assigning two-thirds of the capacity to T_1, and one-third to T_3. Thus while each thread is indeed guaranteed a certain level of performance (i.e., at least one-third of the processor capacity) regardless of the behavior of the other threads, inter-*application* isolation is not achieved – the application $\{T_1, T_2\}$ fails to receive its desired two-thirds of the processor capacity because application $\{T_3\}$ requests more than its reserved share.

This research. In this research, we generalize the CBS framework to permit a 2-level hierarchical organization of the threads comprising a real-time system. That is, we partition the set of threads τ into N (disjoint) sets $S_1, S_2, \ldots, S_N$ satisfying $S_1 \cup S_2 \cup \cdots \cup S_N = \tau$ (the original CBS framework of [1] corresponds to the special case $N = 1$). Each partition S_i is assumed to contain the threads corresponding to a distinct application. We continue to provide each individual thread in τ a guaranteed level of performance, as represented by Equation 1, regardless of the behavior of the other threads in the system. In addition, however, we will ensure that processor capacity not used by a individual thread is preferentially made available to other threads that lie in the *same* partition as that thread, and that this excess capacity is also made available in a timely manner, regardless of the behaviors of threads in other partitions. Hence, the isolation and performance guarantee properties, provided by CBS to each individual thread, are now valid at both the thread level and the partition level.

Organization. The remainder of this paper is organized as follows. In Section 2, we formally define –and justify– the system model that forms the basis of this research, and explain the goals of the scheduling framework we will be proposing. In Section 3, we provide a detailed description of the **H-CBS scheduling algorithm**, our proposed extension to the CBS algorithm of Abeni and Buttazzo [1] which achieves our goals of providing guaranteed performance and isolation at both the thread and the application level. In Section 4, we illustrate the operation of the H-CBS algorithm by detailing the scheduling decisions that would be made by the algorithm on a particular example system. In Section 5, we provide formal justification of our claims that the H-CBS algorithm does indeed meet its design goals.

2. System model

In this paper, we will model a system as being comprised of a set τ of threads, with each thread $T_j = (U_j, P_j)$ being characterized by the order pair of parameters: its *utilization* U_j, and its *period* P_j. Each thread T_j generates a sequence of *jobs* $J_j^1, J_j^2, J_j^3, \ldots$, with job J_j^k becoming ready for execution ("arriving") at time a_j^k ($a_j^k \leq a_j^{k+1}$ for all j, k), and having an execution requirement equal to e_j^k time units. The following assumptions are made regarding the individual threads:

- The arrival times of the individual jobs is not known beforehand; i.e., it is not known prior to time-instant a_j^k when job J_j^k will be arriving.

- The execution requirements of the individual jobs are not known beforehand, nor are they revealed upon job-arrival; instead, the value of e_j^k may only be determined by actually executing J_j^k to completion.

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

The set τ of threads is partitioned into the subsets $S_1, S_2, \ldots, S_N$, such that $S_1 \cup S_2 \cup \cdots \cup S_N = \tau$, and $S_i \cap S_k = \{\}$ for all $i \neq k$. For each such subset S_i, we let $U(S_i)$ denote the sum of the utilizations of all the threads in S_i: $U(S_i) = \sum_{T_j \in S_i} U_j$. All these threads are executed on a single shared preemptable processor; without loss of generality, we will assume that this processor's computing capacity is unity, and that $\sum_{i=1}^{N} U(S_i) = 1$.

Given such system specifications, our goal is to schedule the threads in such a manner that (i) each *thread* T_j is allocated the shared processor in a timely manner, such that the thread's execution closely mimics the behavior it would experience if T_j were executing on dedicated processor of computing capacity U_j, and (ii) each *subset of threads* S_i is allocated the shared processor in a timely manner such that the threads in the subset would execute in a manner that closely mimics the manner in which they would execute if S_i were executing on dedicated processor of computing capacity $U(S_i)$.

Job preemptions. Observe that both these goals would be trivially satisfied in a processor-sharing schedule – a schedule in which the time-line is partitioned into arbitrarily small intervals, and each thread T_j is assigned a fraction U_j of the processor capacity during each such interval. However, such a processor-sharing schedule would necessarily involve a very large number of processor preemptions. To understand why this is an unacceptable solution, let us look at how preemptions are handled in practice.

While we assume in this research that our processor model satisfies the *preempt-resume* property – i.e., a job executing on the processor can be interrupted at any instant in time, and its execution resumed later – we do not assume that preemptions are "free." Rather, these preemption costs are incorporated into the model in the standard manner, by "charging" the cost of each preemption to one of the jobs involved – typically, the one that gets scheduled after the preemption.

It has been shown [8] 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. In EDF-based schedules, therefore, preemption costs can all be accounted for 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.) We will see later in this paper that our H-CBS algorithm schedules a job with a large execution requirement by successively *postponing* the deadline according to which it is scheduled — each such deadline-postponement is essentially equivalent to generating a new job. We will see that job J_j^k's deadline may be changed as many as $\left\lceil \frac{(e_j^k/U_j)}{P_j} \right\rceil$ times – this jobs could therefore be charged for as many as $2 \times \left\lceil \frac{(e_j^k/U_j)}{P_j} \right\rceil$ context switches. For small P_j, this becomes unacceptably large, and much of the processor capacity allocated to T_j could end up being spent thrashing in context switches. As a design rule, it is probably best for the application designer to choose a value for P_j such that $(e_j^k/U_j) \leq P_j$ for most jobs J_j^k generated by T_j — i.e., to choose P_j to be such that the median (or an even higher percentile) of the execution requirements of these jobs is no larger than $U_j \cdot P_j$.

It is noteworthy that if all the P_j's are chosen arbitrarily close to zero, then the H-CBS algorithm reduces to the Hierarchical Generalized Processor Sharing (H-GPS) algorithm of Bennett and Zhang [2].

Our objectives. If our goal in this extended system model were to provide a performance guarantee identical to that provided by CBS (i.e., to ensure that each job complete in the shared processor within a margin equal to the period of the thread that generated it, of the time it would complete if this thread were executing on a slower dedicated processor), then it can be shown that we could achieve this goal by simply ignoring the partitioning of the threads into $S_1, S_2, \ldots, S_N$, and scheduling all the threads using the CBS algorithm of Abeni and Buttazzo [1]. However, our performance requirements are somewhat stronger — in addition to wanting to emulate on the shared processor the behavior that would be experienced by each thread on a dedicated virtual processor, we also wish to emulate the behavior that would be experienced by each *subset* if all the threads comprising it were to execute on a dedicated virtual processor. More formally,

- Let F_j^k denote the time instant at which job J_j^k would complete execution if all jobs of thread T_j were executed on a dedicated processor of capacity U_j.
- Let Φ_j^k denote the time instant at which J_j^k would complete execution, if the threads in the partition S_i such that $T_j \in S_i$ were executing the CBS algorithm of [1] on a dedicated virtual processor of computing capacity $U(S_i)$.

- Let f_j^k denote the time instant at which J_j^k completes execution under H-CBS.

The analogue of the CBS performance guarantee would bound the difference between f_j^k and F_j^k:

$$f_j^k < F_j^k + P_j \ .$$

As stated above, this can be achieved by simply scheduling all threads using the CBS algorithm of Abeni and Buttazzo [1]. However, we have an additional goal: we also wish to bound the difference between f_j^k and Φ_j^k. Achieving this turns out to be non-trivial, in the sense that there seems to be a contradiction between achieving performance guarantees at the two different levels – the thread level, and the partition level – of our 2-level scheduling hierarchy. Our approach in this paper towards resolving this apparent contradiction is to design a strategy for identifying processor capacity that is unused by a thread *at the earliest possible instant*, and then transferring this excess capacity to some other thread that is served by the same CBS server. We incorporate and extend the techniques that were introduced in [5], to help us optimally identify such unused processor capacity.

3. The H-CBS algorithm

In this section, we provide a detailed description of the H-CBS scheduling algorithm. As stated previously, we will model our system as being comprised of a set τ of threads partitioned into the subsets $S_1, S_2, \ldots, S_N$, that are to execute on a single shared processor. We let $U(S_i)$ denote the sum of the utilizations of all the threads in S_i: $U(S_i) = \sum_{T_j \in S_i} U_j$. Without loss of generality, we will assume that the threads execute on a shared processor of capacity equal to $U(S_1) + U(S_2) + \cdots + U(S_N)$, and that the capacities have been normalized such that this sum equals 1.

Algorithm Variables. For each thread T_j in the system, the H-CBS algorithm maintains two variables: a *deadline* D_j and a *virtual time* V_j.

- Intuitively, the value of D_j at each instant is a measure of the *priority* that the H-CBS server accords thread T_j at that instant — H-CBS will essentially be performing earliest deadline first (EDF) scheduling based upon these D_j values.

- The value of V_j at any time is a measure of how much of thread T_j's "reserved" service has been consumed by that time. As a first approximation, we can assume that the value of V_j will be updated in such a manner that, *at each instant in time, thread T_j has received the same amount of service that it would have received by time V_j if executing on a dedicated processor of capacity U_j.*

Thread States. At any instant in time during run-time, each thread T_j is in one of three states: inactive, activeContending, or activeNonContending. The initial state of each thread is inactive. Intuitively at time t_o a thread 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, the H-CBS server chooses for execution some thread that is in its activeContending state (if there is no such thread, then the processor is idled). From among all the threads that are in their activeContending state, the next job needing execution of the thread T_j, whose deadline parameter D_j is the smallest, is chosen for execution.

When the first job of T_j arrives, V_j is set equal to this arrival time. While (a job of) T_j is executing, its virtual time V_j is *in*creased at a rate that will be specified later. If at any time this virtual time becomes equal to the deadline ($V_j == D_j$), then the deadline parameter is incremented by P_j ($D_j \leftarrow D_j + P_j$). Notice that this may cause T_j to no longer be the earliest-deadline active thread, in which case it may surrender control of the processor to an earlier-deadline thread.

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

1. If thread T_j is in the inactive state and a job J_j^k arrives (at time-instant a_j^k), then the following code is executed

$$\begin{array}{lcl} V_j & \leftarrow & a_j^k \\ D_j & \leftarrow & V_j + P_j \end{array}$$

   and thread T_j enters the activeContending state.

2. When a job J_j^k of T_j completes (at time-instant f_j^k) — notice that T_j must then be in its activeContending state — the action taken depends upon whether the next job J_j^{k+1} of T_j has already arrived.

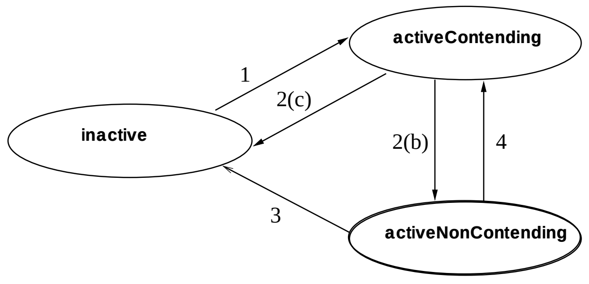


Figure 1. State transition diagram. The labels on the nodes and edges denote the name by which the respective states and transitions are referred to in this paper.

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

$$D_j \longleftarrow V_j + P_j \;;$$

the thread remains in the activeContending state.

(b) If there is no job of T_j awaiting execution and $V_j > f_j^k$ (i.e., the current value of V_j is *greater* than the current time) then thread T_j changes state, and enters the activeNonContending state.

(c) If there is no job of T_j awaiting execution and $V_j \leq f_j^k$ (i.e., the current value of V_j is *no larger* than the current time) then, too, thread T_j changes state and enters the inactive state.

3. For thread T_j to be in the activeNonContending state at any instant t, it is required that $V_j > t$. When this ceases to be true, because time has elapsed since T_j entered the activeNonContending state but V_j does not change for threads in this state, then the thread enters the inactive state.

4. If a new job J_j^k arrives while thread T_j is in the activeNonContending state, then the deadline parameter D_j is updated as follows:

$$D_j \longleftarrow V_j + P_j \;,$$

and thread T_j returns to the activeContending state.

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

3.1 Incrementing virtual time

It now remains to specify how the virtual time V_j of a thread T_j changes when a job of T_j is executing. If we were implementing the CBS algorithm of Abeni and Buttazzo [1] rather than H-CBS, V_j would be updated in such a manner that, at each instant in time, thread T_j has received the same amount of service that it would have received by time V_j if executing on a dedicated processor of capacity U_j. And, this can be achieved by incrementing V_j at a rate $1/U_j$:

$$\frac{d}{d\,t}V_j \overset{\text{def}}{=} \begin{cases} \frac{1}{U_j}, & \text{if } T_j \text{ is executing} \\ 0, & \text{otherwise} \end{cases} \qquad (2)$$

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

If we were to increment virtual time according to Equation 2 above, we would be emulating on a shared processor the behavior that would be experienced by each thread if it were executing on a single dedicated processor. However, recall that our extended system model allows for a two-level hierarchy of threads – individual threads T_j are grouped into subsets, and our design goal is to be able to emulate the behavior that would be experienced by each *subset* if all the threads comprising it were to execute on a dedicated virtual processor, in addition to emulating on the shared processor the behavior that would be experienced by each thread on a dedicated virtual processor. In order to achieve our goal, we must be able to identify computing capacity that is not going to be used by each individual thread T_j as soon as possible, and be able to assign such excess capacity to other threads that happen to be in the same subset S_i as the thread T_j. Below, we identify two sources of this excess capacity, and describe how this excess capacity is reclaimed by H-CBS; briefly, these sources are (i) threads in the inactive state, which by virtue of being in this state are not making use of their reserved capacities at the current instant, and (ii) threads that make the transition labelled "2(c)" in Figure 1 – in order to make this transition a thread must have its virtual time strictly <u>less</u> that the current time and is therefore not making use of the capacity that it had reserved during the interval between its virtual time and the current instant.

(§i). One source of excess capacity is the processor capacity that remains unused because some threads are in the inactive state. In reclaiming such excess processor capacity, though, we must be very careful to not end up using any of the *future* capacity of currently inactive threads, since we have no idea at any instant when the currently inactive threads will become active. That is, we should devise strategies for updating the virtual times — the V_j's — in order to maximize reclamation of the processor capacity of currently inactive threads, without compromising the future performance guarantees of these threads.

Definition 1 *We define a thread T_j 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 thread 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.

For each subset S_i, H-CBS maintains an additional variable the *current excess capacity* ρ_i, which at each instant in time is equal to the sum of the capacities U_j of all threads T_j that are not active at that instant in time. The current excess capacity ρ_i is initially set equal to $U(S_i)$ for all i; whenever a thread $T_j \in S_i$ undergoes the state-transition labelled "1" in Figure 1, ρ_i is decremented by U_j; whenever T_j undergoes the state-transitions labelled "2(c)" or "3", ρ_i is *in*cremented by U_j.

Let $[t, t + \Delta t)$ denote a time interval during which no scheduling events occur – i.e., no thread undergoes a state-transition, and the same job is scheduled throughout this interval. If a thread of S_i is executing during this interval, then let T_j denote this executing thread; else, let T_j denote the highest-priority (earliest-deadline) thread T_j in S_i – note that T_j may in this case be either active contending or active non-contending. We refer to this thread T_j as the **beneficiary** thread of server S_i during the interval $[t, t + \Delta t)$, and *we will assign all the excess capacity of S_i during this interval — a quantity of $\Delta t \cdot \rho_i$ — to this beneficiary thread for "free".* Consequently, the beneficiary T_j gets an amount of excess capacity equal to $(\Delta t \cdot \rho_i)$ during this interval; equivalently, its virtual time V_j should decrease by an amount equal to $\frac{\Delta t \cdot \rho_i}{U_j}$. However, if T_j were currently being executed – i.e., H-CBS has chosen the thread T_j for execution during $[t, t + \Delta t)$ – then T_j has used an amount equal to

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

of its *own* processor capacity during this interval; equivalently, its virtual time V_j should <u>in</u>crease by an amount equal to $\frac{\Delta t \cdot (1-\rho_i)}{U_j}$. That is, *for each S_i the virtual time V_j of the beneficiary thread $T_j \in S_i$ defined as above is updated as follows by H-CBS:*

$$\frac{d}{dt} V_j = \begin{cases} \frac{1-\rho_i}{U_j}, & \text{if } T_j \text{ is executing} \\ -\frac{\rho_i}{U_j} & \text{otherwise} \end{cases} \tag{3}$$

(§ii). There is one other source of excess processor capacity that H-CBS reclaims. Recall that a thread T_j which makes the transition labelled "2(c)" in Figure 1 at some time-instant t_o satisfies the property that $V_j < t_o$ at that instant. That is, T_j has only used its reserved processor capacity until some time-instant V_j that is strictly *less than* the current instant — its remaining capacity, which equals $((t_o - V_j) \cdot U_j)$, is available for H-CBS to reclaim. Once again, H-CBS will allocate all this reclaimed capacity in a greedy manner, to the active thread with the highest priority (i.e., smallest deadline parameter) that is in the same subset. Suppose that this thread is T_k – i.e., after T_j undergoes the transition labelled "2(c)" in Figure 1 at instant t_o, thread T_k is the active thread in the same subset as T_j with the earliest value of its deadline variable. Then all the reclaimed capacity is assigned to T_k, and this is done by *de*crementing T_k's virtual time by the appropriate amount, as follows:

$$V_k \leftarrow V_k - \frac{(t_o - V_j) \cdot U_j}{U_k} \tag{4}$$

H-CBS thus observes the following rules for updating virtual time:

- If $T_j \in S_i$ is the beneficiary thread in $\tau(S_i)$

$$\frac{d}{dt} V_j \stackrel{\text{def}}{=} \begin{cases} \frac{1-\rho_i}{U_j}, & \text{if } T_j \text{ is executing} \\ -\frac{\rho_i}{U_j} & \text{otherwise} \end{cases}$$

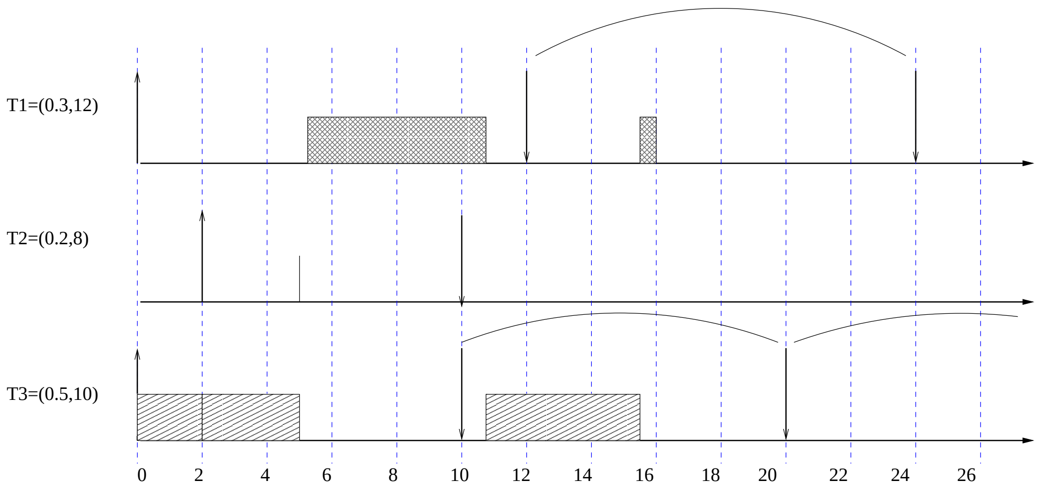
- If $T_j \in S_i$ undergoes transition "2(c)" at time t_o,and T_ℓ is the active thread with the next-earliest deadline in S_i:

$$V_\ell \leftarrow V_\ell - \frac{(t_o - V_j) \cdot U_j}{U_\ell}$$

4. An Example

We will now illustrate the operation of the H-CBS algorithm, by tracing the scheduling of a simple system.

Consider a system τ comprised of three threads $T_1 = (0.3, 12)$, $T_2 = (0.2, 8)$, and $T_3 = (0.5, 10)$. Suppose that τ is partitioned into $S_1 = \{T_1, T_2\}$ and $S_2 = \{T_3\}$ (therefore, $U(S_1) = 0.5$ and $U(S_2) = 0.5$). Suppose that the first job J_1^1 of T_1 arrives at instant 0 and has an execution requirement equal to 6, and the first job



No	**time**	ρ_1	$\mathbf{V}_1$	$\mathbf{D}_1$	$\mathbf{V}_2$	$\mathbf{D}_2$	ρ_2	$\mathbf{V}_3$	$\mathbf{D}_3$	**comment**
1	0^-	0.5	-	∞	-	∞	0.5	-	∞	prior to start
2	0	0.2	0	12	-	∞	0.0	0	10	J_1^1 and J_3^1 arrive
3	2^-	0.2	$-\frac{4}{3}$	12	-	∞	0.0	4	10	J_3^1 executes over $[0, 2)$; V_1 in incremented according to Equation 4
4	2	0.0	$-\frac{4}{3}$	12	2	10	0.0	4	10	J_2^1 arrives
5	5^-	0.0	$-\frac{4}{3}$	12	2	10	0.0	10	10	T_3 has exhausted its budget
6	5	0.0	$-\frac{4}{3}$	12	2	10	0.0	10	20	T_3 increments its deadline
7	$5+\epsilon$	0.0	$-\frac{4}{3}$	12	$2+\frac{\epsilon}{0.2}$	10	0.0	10	20	J_2^1 completes execution over $[5, 5+\epsilon)$, for $\epsilon \to 0$
8	$5+\epsilon$	0.2	$-\frac{10}{3}$	12	$2+\frac{\epsilon}{0.2}$	∞	0.0	10	20	T_2 transfers its excess budget to T_1, as per Equation 3
9	10.75	0.2	12	24	$2+\frac{\epsilon}{0.2}$	∞	0.0	10	20	J_1^1 executes over $[5, 10.75)$; V_1 is incremented according to Equation 3. V_1 is incremented by P_1
10	15.75^-	0.2	12	24	$2+\frac{\epsilon}{0.2}$	∞	0.0	20	20	J_3^1 executes for five time units, thus exhausting its budget
11	15.75	0.2	12	24	$2+\frac{\epsilon}{0.2}$	∞	0.0	20	30	J_3^1 increments its deadline
12	16	0.5	$12\frac{2}{3}$	24	$2+\frac{\epsilon}{0.2}$	∞	0.0	20	30	J_1^1 executes for 0.25 time units, and completes. V_1 is incremented according to Equation 3.

Figure 2. Scenario described in Section 4.

J_3^1 of T_3 also arrives at instant 0 and has a large enough execution requirement that J_3^1 will utilize the processor whenever it is scheduled. Suppose that the first job J_2^1 of T_2 arrives at instant 2, and has an arbitrarily small execution requirement of ϵ ($\epsilon \to 0$). The resulting schedule is depicted in Figure 2; the steps taken by H-CBS in generating this schedule are explained below.

Initially, the variables ρ_1 and ρ_2 are both equal to 0.5 (indicating that no threads are active, and ρ_i therefore equal $U(S_i)$ for $i = 1, 2$).

At time $\mathbf{t}_o = 0$, jobs J_1^1 and J_3^1 arrive. As a consequence, threads T_1 and T_3 make the transition labelled "1" in Figure 1; thus,

- $\rho_1 \leftarrow \rho_1 - U_1$, and $\rho_2 \leftarrow \rho_2 - U_3$.
- $V_1 \leftarrow t_o$ and $V_3 \leftarrow t_o$.
- $D_1 \leftarrow t_o + P_1$ and $D_3 \leftarrow t_o + P_3$.

Since $D_3 < D_1$, H-CBS begins executing J_3^1.

Over the interval $[0, 2)$, V_3 is incremented at a rate $(1 - \rho_2)/U_3 = 2$ while V_1 is <u>de</u>cremented at a rate $\rho_1/U_1 = \frac{2}{3}$, both according to Equation 3.

At time $\mathbf{t}_o = 2$, J_2^1 arrives, and T_1 makes the transition labelled "1" in Figure 1; thus,

- $\rho_1 \leftarrow \rho_1 - U_2 = 0$.
- $V_2 \leftarrow t_o$.
- $D_2 \leftarrow t_o + P_2$.

D_2 and D_3 are now both equal; let us assume that H-CBS breaks the tie in favor of T_3, and thus continues executing J_3^1.

Over the interval $[2, 5)$, T_3's job is executed, and V_3 thus continues to be incremented according to Equation 3 at a rate $(1 - \rho_2)/U_3 = 2$. Since ρ_1 now equals zero, it follows from Equation 3 that V_1 and V_2 do not change over this interval.

At time $\mathbf{t}_o = 5$, V_3 becomes equal to $D_3 - T_3$ has thus exhausted its budget. Since T_3 has exhausted its budget, it must increment its deadline parameter: $D_3 \leftarrow D_3 + P_3$. D_2 now becomes the earliest deadline, and H-CBS begins executing J_2^1.

Over the interval $[5, 5 + \epsilon)$, J_2^1 is executed.

At time $\mathbf{t}_o = 5 + \epsilon$, J_2^1 completes execution, and hence undergoes the state-transition "2(c)" in Figure 1. As a consequence,

- $\rho_1 \leftarrow \rho_1 + U_2 = 0.2$.
- $D_2 \leftarrow \infty$.
- The excess capacity of T_2 is transferred to T_1 — this is achieved by decrementing V_1 according to Equation 4.

D_1 is now the smallest-deadline thread in the system, and H-CBS consequently chooses T_1 for execution.

Over the interval $[5 + \epsilon, 10.75 + \epsilon)$, T_1's job is executed, and T_3 is thus incremented according to Equation 3 at a rate $(1 - \rho_1)/U_1 = \frac{8}{3}$. Since ρ_2 equals zero, V_3 does not change over this interval.

At time $\mathbf{t}_o = 10.75 + \epsilon$, V_1 becomes equal to $D_1 - T_1$ has thus exhausted its budget. Since T_1 has exhausted its budget, it must increment its deadline parameter: $D_1 \leftarrow D_1 + P_1$. D_3 now becomes the earliest deadline, and H-CBS begins executing J_3^1.

Over the interval $[10.75 + \epsilon, 15.75 + \epsilon)$, J_3^1 is executed, and V_3 incremented according to Equation 3 at a rate $(1 - \rho_2)/U_3 = 2$. Also, V_1 is <u>de</u>cremented at a rate $\rho_1/U_1 = \frac{2}{3}$ over this interval.

At time $\mathbf{t}_o = 15.75 + \epsilon$, V_3 becomes equal to $D_3 - T_3$ has thus exhausted its budget. Since T_3 has exhausted its budget, it must increment its deadline parameter: $D_3 \leftarrow D_3 + P_3$. D_1 now once again becomes the earliest deadline, and H-CBS begins executing J_1^1.

Over the interval $[15.75 + \epsilon, 16 + \epsilon)$, J_1^1 is executed, and V_1 incremented according to Equation 3 at a rate $(1 - \rho_1)/U_1 = \frac{8}{3}$.

At time $\mathbf{t}_o = 16 + \epsilon$, J_1^1 completes execution and undergoes the state-transition "2(c)" in Figure 1. As a consequence,

- $\rho_1 \leftarrow \rho_1 + U_1 = 0.5$.
- $D_1 \leftarrow \infty$.
- Since there are no active threads in S_1, T_1 cannot transfer its excess capacity (as mandated by Equation 4) to any other thread.

5. Formal Analysis of H-CBS

In this section, we provide formal justification of our claims that the H-CBS algorithm does indeed meet its design goals. That is, we show (Theorem 1) that the H-CBS algorithm provides service to each thread in a timely manner (as does the CBS algorithm of Abeni & Buttazzo [1]); furthermore, we show (Theorem 2) that the H-CBS algorithm provides service to each *partition* in a timely manner as well.

Recall that we are modelling our system as a set τ of threads partitioned into the subsets $S_1, S_2, \ldots, S_N$, with $U(S_i)$ denoting the sum $\sum_{T_j \in S_i} U_j$. We assume that all these threads execute on a single shared processor of unit capacity: $\sum_{i=1}^{N} U(S_i) = 1$.

Thread-level guarantee. As before, let F_j^k denote the time instant at which job J_j^k would complete execution, if all jobs of thread T_j were executed on a dedicated processor of capacity U_j. Let f_j^k denote the time instant at which J_j^k completes execution under H-CBS.

Theorem 1 *It is guaranteed that*

$$f_j^k < F_j^k + P_j \ ; \qquad (5)$$

i.e., H-CBS makes the same per thread *performance guarantee as does the CBS framework of [1].*

Proof Sketch: It can be shown that a thread T_j receives identical service under CBS and H-CBS *if the other threads in the system are continually backlogged* — i.e., if there is no excess processor capacity to reclaim and distribute. Since CBS has been shown [1, 5] to satisfy Inequality 5 under all circumstances, it thus follows that H-CBS guarantees that Inequality 5 will be satisfied, in the absence of any excess execution capacity.

Since capacity is reclaimed by H-CBS from a thread only when it is deemed to be not needed by the thread — i.e., only that capacity is reclaimed from a thread which becomes excess due to a thread being inactive at present (Equation 3) or in the past(Equation 4) — the finish time of a job of an active thread must be $\leq$ its finish time in the absence of any excess execution capacity from other threads. Thus in Equation 5 above, f_j^k under H-CBS will be no larger than the value of f_j^k if there had been no excess capacity available. And, we have argued above that in the absence of any excess capacity the correctness of CBS implies that H-CBS satisfies Inequality 5.

Application-level guarantee. Without loss of generality, let us assume that $\min_{T_j \in \tau}\{a_j^1\} = 0$; i.e., the first job (of any thread) arrives at time-instant 0. For each application S_i, $1 \leq i \leq N$, consider the schedule if S_i were to execute on a dedicated virtual processor of computing capacity $U(S_i)$ in a work-conserving FCFS manner. Let

$$[\sigma_i^1, \phi_i^1), [\sigma_i^2, \phi_i^2), \ldots, [\sigma_i^k, \phi_i^k), \ldots$$

denote the *busy intervals* in this schedule, with $\sigma_i^k < \phi_i^k$ and $\phi_i^k < \sigma_i^{k+1}$ for all $k \geq 1$. That is,

- the dedicated virtual processor is idle prior to instant σ_i^1
- For all $k \geq 1$; the dedicated virtual processor is executing jobs generated by threads in S_i throughout the interval $[\sigma_i^k, \phi_i^k)$
- For all $k \geq 1$; the dedicated virtual processor is idle throughout the interval $[\phi_i^k, \sigma_i^{k+1})$.

For each server S_i and for all $k \geq 1$, let B_i^k denote the amount of execution that jobs of threads in S_i will have received during its first k busy intervals, if executing on a dedicated processor of computing capacity $U(S_i)$:

$$B_i^k \stackrel{\text{def}}{=} U(S_i) \cdot \left(\sum_{\ell=1}^{k} (\phi_i^\ell - \sigma_i^\ell) \right) .$$

Algorithm H-CBS guarantees that jobs of threads in S_i will receive this amount of service in a timely manner — more specifically,

Theorem 2 *Under the H-CBS scheduling algorithm, jobs of (threads in) S_i are guaranteed to receive B_i^k units of execution by time-instant*

$$\left(\phi_i^k + \max_{T_j \in S_i} \{P_j\} \right) \ ;$$

i.e., each partition of threads S_i is guaranteed to complete under H-CBS all the jobs that it would complete in its first k busy-periods no more than $\max_{T_j \in S_i}\{P_j\}$ time units later than the time it would complete if this partition were executing on a dedicated processor.

(This is, intuitively speaking, the "best" guarantee of timely service that we can reasonably make — since each thread T_j in S_i has a notion of timeliness that is as accurate as its period parameter P_j, we cannot expect the partition to collectively respect a finer-grained notion of time than its most coarse-grained component thread.)

Proof Sketch: It can be shown that all the excess capacity that is reclaimed from a thread T_j according to Equation 3 is reclaimed as soon as it becomes available, and all the excess capacity that is reclaimed according to Equation 4 is reclaimed within P_j time units of it becoming available. And, any reclaimed capacity is assigned at a priority corresponding to the deadline of the next-earliest deadline thread of the same server — thus, this reclaimed capacity is guaranteed to be obtained by some thread in the server within one period of the time it is reclaimed.

6. Conclusions and future work

We have proposed a global scheduling algorithm for use in preemptive uniprocessor systems in which several different time-sensitive applications, each of which can be considered to be comprised of several threads, are to execute on a single preemptable processor. Our scheduling algorithm – the **H-CBS algorithm** – is an extension of the CBS algorithm of Abeni and Buttazzo [1]. As with the CBS algorithm, each thread in a system scheduled using the H-CBS algorithm is assured certain *performance guarantees* — the illusion of executing on a dedicated processor — and *isolation* from any ill-effects of other misbehaving threads. Unlike the CBS algorithm, however, the H-CBS algorithm extends these guarantees to each application (considered to be a subset of the set of threads comprising the system) as well.

In the original CBS algorithm [1], the threads comprising the system being scheduled have a "flat" organization – each thread, characterized by its two parameters, bears the same relationship with every other thread. In this paper, we have essentially extended the results of Abeni and Buttazzo [1] to the case where the threads comprising the system can be considered organized into a two-level hierarchy — individual threads are grouped into partitions (applications), and all of these partitions together comprise the system. In our opinion, the most interesting set of issues left open concerns applying the techniques presented here to a model in which threads can be arranged in a hierarchy that is *more* than two deep. It seems clear that our techniques can indeed be applied to systems that have such a deeper hierarchical structure; however, we do not yet know precisely what kinds of guarantees these techniques will yield in these more complex hierarchical systems.

It can be argued that restrictive thread model assumed in this paper — that jobs generated by each thread must be executed in FCFS order — severely limits the applicability of these results since jobs generated by individual threads in many multi-threaded real-time application systems do not satisfy the FCFS property. Work has been done [6, 4, 7] on designing scheduling algorithms for use in real-time multi-threaded systems under which each thread is assured performance guarantees and isolation from misbehaving threads, even when jobs generated by each thread are not constrained to execute in a FCFS order. In the future, we will study the extension of these algorithms to hierarchically-structured systems as well.

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] Jon C.R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, Oct 1997.

[3] 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.

[4] 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.

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

[6] 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.

[7] Giuseppe Lipari, John Carpenter, and Sanjoy Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Proceedings of the Real-Time Systems Symposium*, Orlando, FL, November 2000. IEEE Computer Society Press.

[8] 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.

[9] John Regehr, Jack Stankovic, and Marty Humphrey. The case for hierarchical schedulers with performance guarantees. Technical Report CS-2000-07, Department of Computer Science, University of Virginia, March 2000.

[10] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *Proceedings of ACM SIGCOMM'97*, August 1997.