Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2011, Article ID 391215, 18 pages
doi:10.1155/2011/391215

Research Article

Reclaiming Spare Capacity and Improving Aperiodic Response Times in Real-Time Environments

Sathish Gopalakrishnan[1] and Xue Liu[2]

[1] *Department of Electrical and Computer Engineering, the University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada V6T 1Z4*
[2] *Department of Computer Science and Engineering, University of Nebraska-Lincoln, 104 Schorr Center, Lincoln, NE 68588, USA*

Correspondence should be addressed to Sathish Gopalakrishnan, sathish@ece.ubc.ca

Received 30 August 2010; Accepted 28 January 2011

Academic Editor: S. Ramesh

Copyright © 2011 S. Gopalakrishnan and X. Liu. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Scheduling recurring task sets that allow some instances of the tasks to be skipped produces holes in the schedule which are nonuniformly distributed. Similarly, when the recurring tasks are not strictly periodic but are sporadic, there is extra processor bandwidth arising because of irregular job arrivals. The additional computation capacity that results from skips or sporadic tasks can be reclaimed to service aperiodic task requests efficiently and quickly. We present techniques for improving the response times of aperiodic tasks by identifying nonuniformly distributed spare capacity—because of skips or sporadic tasks—in the schedule and adding such extra capacity to the *capacity queue* of a BASH server. These gaps can account for a significant portion of aperiodic capacity, and their reclamation results in considerable improvement to aperiodic response times. We present two schemes: NCLB-CBS, which performs well in periodic real-time environments with firm tasks, and NCLB-CUS, which can be deployed when the basic task set to schedule is sporadic. Evaluation via simulations and implementation suggests that performance improvements for aperiodic tasks can be obtained with limited additional overhead.

1. Introduction

Real-time systems execute periodic and aperiodic tasks, and typically each of these tasks has a *deadline*. Periodic tasks are recurring, and each instance of such a task is called a *job*. A periodic task τ_i is typically characterized by computation time c_i and period p_i; the relative deadline of an instance of a periodic task is equal to the period of the task. Aperiodic tasks are executed only occasionally but often require short response times. The terms aperiodic task and aperiodic job are used interchangeably in this discussion.

Real-time tasks can be classified based on the consequences of a missed deadline as follows.

Hard. If a hard real-time task misses its deadline, it is assumed that consequences for the system are catastrophic. It is therefore imperative that *a priori* guarantees of not missing the deadlines be provided for all hard real-time tasks.

Soft. A soft real-time task may miss deadlines, and missed deadlines lead to degraded performance or lower quality of service.

Firm. Firm real-time tasks are allowed to miss deadlines occasionally. An instance of a firm task can be aborted to create capacity for executing other jobs.

Real-time tasks may be classified along a different dimension: their periodicity. Strictly *periodic tasks* have job instances that release at precisely periodic intervals. On the other hand, consecutive instances of a *sporadic task* have a minimum separation distance, p_i. The distinction between periodic and sporadic tasks are that consecutive instances of a periodic task are released at time instants strictly separated by p_i time units whereas consecutive instances of sporadic tasks are separated by *at least* p_i time units. Aperiodic tasks do not have any constraints on the interarrival times; in general, aperiodic workload does not constitute a majority share of processor utilization.

Liu and Layland [1] were the first to address the problem of scheduling periodic hard real-time tasks; they developed simple schedulability tests for periodic task sets scheduled by the rate monotonic algorithm or the earliest deadline first algorithm. The RM algorithm assigns higher priorities to tasks with higher rates (lower periods), and the EDF algorithm assigns higher priorities to tasks with earlier absolute deadlines. Systems that assign the same priority to every job of a task are called *fixed priority* real-time systems; systems that might assign different priorities to different instances of the same task are called *dynamic priority* systems. The RM assignment is fixed priority, and EDF is a dynamic priority assignment.

Later research on real-time systems extended Liu and Layland's analysis to derive schedulability conditions for hard real-time task sets under more general settings. Feasibility analysis with resource sharing among periodic tasks [2–4] and in the presence of aperiodic tasks [5–10] are important generalizations that have been studied.

Real-time systems also need to execute some tasks that are aperiodic. These tasks may perform operations such as maintenance work or performance logging. These operations may not impact system functionality or criticality, but better QoS can be provided by servicing these requests quickly. For instance, a response to a resource utilization snapshot request is more useful to a human operator if provided earlier rather than later. From a scheduling perspective, the challenge is to schedule aperiodic tasks as soon as possible without causing periodic/sporadic (critical) tasks to miss deadlines. There are applications that require aperiodic tasks with deadlines; *in this work, however, we focus on aperiodic tasks that do not have deadlines associated with them*.

While it is true that there are some safety-critical systems that cannot tolerate a single deadline miss, many systems (e.g., multimedia systems) are capable of tolerating some missed deadlines. Moreover, even in safety-critical systems, not all tasks are hard real-time tasks; for optimal resource allocation, soft and firm real-time tasks need to be handled differently. Skipping a few instances of a firm real-time task allows a scheduler to utilize resources better and schedule task sets that would otherwise overload the system. Hamdaoui and Ramanathan [11] proposed the (m, k)-model for representing firm real-time tasks; in this model, in any window of k instances of a task, at least m instances need to be executed successfully. Koren and Shasha [12] studied the special case of the $(1, k)$-firm real-time task model. One of the major benefits of allowing skips is the ability to improve the response times of aperiodic tasks (more related work is discussed in Section 2).

The main contribution of our work is the development of mechanisms that enable schedulers to reclaim spare bandwidth—which is nontrivial in systems that allow skips or involve sporadic tasks—and advance the execution of aperiodic tasks within the framework of dynamic-priority servers that can guarantee periodic (or sporadic) tasks a fraction of the processor. To establish the correctness of our methods, we first discuss some basic theorems and techniques (Section 3). Because we are concerned with tasks that permit skips, we consider the *red-tasks-only* and the *blue-when-possible* models [12]. We first discuss the use of our NCLB approach with the red-tasks-only model (Section 4) and then discuss the extensions to the blue-when-possible model (Sections 4.2 and 5).

Experimental results (Sections 6 and 7) indicate that, by reclaiming spare processing cycles with our approach, we can improve the response times of aperiodic tasks significantly, and that the overhead imposed by the suggested schemes is low.

2. Related Work

For firm real-time systems, while Hamdaoui and Ramanathan [11] proposed the (m, k)-model for representing firm real-time tasks and described a heuristic priority assignment scheme for such tasks, they did not develop an exact schedulability analysis. Bernat and Burns [13] described a technique for utilizing the (m, k)-model in the presence of aperiodic tasks along with an offline guarantee test using a worst-case formulation for fixed priority scheduling. Koren and Shasha [12] made important contributions when they proved that making optimal use of skips is NP-hard and described two (efficient, but nonoptimal) skip-over algorithms for exploiting skips and increasing the feasible periodic load and schedule task sets that are slightly overloaded. One skip-over algorithm is fixed priority and extends RM scheduling; the other algorithm is dynamic priority and is based on the EDF algorithm. Koren and Shasha modeled a firm real-time task, τ_i, using a skip factor, s_i, which indicates that one instance of τ_i can be skipped every s_i instances. Koren and Shasha also differentiated task instances using the colors *red* and *blue*. Red instances had to be completed to satisfy the 1-*in*-k requirement; blue instances could be executed if possible and when no other task would miss its deadline by such execution. This classification led to algorithms which executed red tasks only (RTO) or scheduled blue instances when possible (BWP).

There has been extensive work in dealing with aperiodic tasks in hard real-time systems. The primary goal has been the scheduling of aperiodic tasks in the presence of hard real-time tasks and improving the performance over background execution of aperiodic tasks.

The simplest approach to aperiodic task scheduling is *background scheduling*: aperiodic tasks are executed only when no periodic (or sporadic) jobs are ready for execution. Clearly, such an approach preserves the deadlines of periodic tasks and is simple to implement. On the other hand, the execution of aperiodic jobs may be delayed needlessly, and, as a result, their response times may be very high.

The predominant approaches to aperiodic real-time scheduling can be classified as *bandwidth-preserving server* schemes [14]. Bandwidth-preserving servers are aperiodic servers which are defined by some *consumption* and *replenishment* rules. These rules guarantee that each task consumes only a certain amount of the bandwidth thereby providing a guarantee that every hard real-time task will meet its deadlines.

Many server schemes have been proposed for both static and dynamic priority scheduling algorithms. Predominant server schemes have been the deferrable server [5, 15], sporadic server [6], and the constant bandwidth server [10]. More recently, there have been efforts to handle execution time overruns in a graceful manner while allowing periodic and aperiodic tasks to coexist. CASH [16], BASH [17], GRUB [18], and IRIS [19] are some of the proposals in this direction. There have also been other efforts to integrate aperiodic tasks with periodic hard real-time tasks [20, 21]. No prior work, however, has explicitly and efficiently addressed the problem of completely reclaiming spare time for aperiodic tasks in a system that tolerates skips or in systems that deal, predominantly, with sporadic tasks.

There have been three main efforts directed at improving aperiodic response time in a firm real-time environment; Marchand and Silly-Chetto [22], Buttazzo and Caccamo [23], and Thomas et al. [24] have addressed the problem to varying extents.

Marchand and Silly-Chetto developed the EDL-RTO and the EDL-BWP algorithms [22] for enhancing the responsiveness of aperiodic tasks in a firm real-time environment. Their work is based on the EDL scheduler [25], and this leads to high overhead (of the order of $O(N^2)$) where N is the number of tasks to be scheduled).

Buttazzo and Caccamo [23] proposed a technique for minimizing aperiodic response times in a firm real-time environment using the model proposed by Koren and Shasha; the underlying scheduler was the EDF scheduler. Buttazzo and Caccamo reclaimed a portion of the spare time created by skipping jobs to improve the response time for aperiodic tasks. They were unable to reclaim all the spare time, however, and observed that the spare time created by skipping jobs has a "granular" distribution across the schedule. They called these unevenly distributed capacities *holes*. Later, Thomas et al. [24] developed the spare CASH approach for locating and reusing these holes via the CASH mechanism [16] in a firm real-time environment with the RTO scheduling paradigm [12]. The work that we present in this article improves upon spare CASH in two ways. The mechanisms we present use the BASH server, which is an improvement over the CASH server. Additionally, Spare CASH cannot be utilized in conjunction with sporadic task sets whereas we also suggest techniques that improve the response time of aperiodic tasks in the presence of firm/hard sporadic real-time tasks.

Lin and Brandt identified some principles for slack reclamation in periodic real-time systems and used these principles to develop the BACKSLASH scheme [26]. BACKSLASH is, however, better suited to dealing with dynamic slack that is created by early job completions and allocating it to tasks that may have overruns. In the context of speeding up aperiodic tasks in firm real-time or sporadic environments, our approach is more aggressive and attuned to the specifics of these settings. In particular, BACKSLASH makes no distinction between periodic and aperiodic tasks and provides fair distribution of slack. This may not be the best choice for all systems. In the methods that we will present, slack present in a schedule can be transferred to tasks that need the slack the most. We have chosen to restrict our attention to aperiodic tasks, but the same ideas can be used to allocate slack to other tasks executed in the system.

In this paper, we provide a *simple, integrated framework* for dealing with aperiodic tasks while *allowing reservations* through the use of servers and *efficient resource reclamations* in the case of *early task completions*, *skipped jobs* or *sporadic tasks*. We propose the NCLB (no capacity left back) class of algorithms because we ensure that no background execution occurs in the schedule. We consider two variants, NCLB-CBS and NCLB-CUS, depending on whether a constant bandwidth server or a constant utilization server [14] is chosen as the underlying server. The choice of the server mechanism depends on whether the basic task set to schedule is periodic or sporadic; with sporadic tasks, we advocate a CU server, and, in other circumstances, it may be a matter of implementation preference.

In the next section, we will discuss, briefly, the main results concerning firm real-time scheduling. Our work does address sporadic task systems as well, but, because of the similarity between the treatment of sporadic and strictly periodic task systems, we do not dwell on prior work on sporadic task sets.

3. Preliminaries

3.1. Terminology and Assumptions. Each firm periodic task, τ_i, is characterized by its worst-case computation time, c_i, its period, p_i, a relative deadline that is equal to the period, and a skip parameter, s_i, $2 \le s_i \le \infty$. The skip parameter specifies the minimum distance between two consecutive skips. For example, if $s_i = 6$, 1 in every 6 instances of task τ_i can be skipped. When $s_i = \infty$, no skips are allowed and the task is a hard periodic task. The skip parameter can be viewed as a *quality of service* measure; the higher the s, the better the QoS. $\tau_{i,j}$ is used to denote the jth instance of task τ_i.

Using the terminology introduced by Koren and Shasha [12], every instance of a firm periodic task can be labeled *red* or *blue*. A red instance must complete before its deadline; a blue instance can be aborted at any time. When a blue instance is aborted, we say that it is *skipped*. If a blue instance is skipped, then the next $s - 1$ instances must be red. On the other hand, if a blue instance completes successfully, the next task instance is also blue.

3.2. Basic Results on Firm Periodic Task Scheduling. In the hard periodic model, where all task instances are red (no skips are permitted), the schedulability of a periodic task set can be tested using a simple, necessary, and sufficient condition based upon cumulative processor utilization. Liu and Layland [1] showed that a periodic task set is schedulable by EDF if and only if its cumulative processor utilization, U_p, is no greater than 1. That is,

$$U_p = \sum_{i=1}^{n} \frac{c_i}{p_i} \le 1. \quad (1)$$

Analyzing the feasibility of firm periodic tasks is not equally easy. Koren and Shasha [12] proved that determining whether a set of skippable periodic tasks is schedulable is NP-hard. They also found, given a set $\Gamma = \{\tau_i(p_i, c_i, s_i)\}$ of firm periodic tasks that allow skips, that

$$\sum_{i=1}^{n} \frac{c_i(s_i - 1)}{p_i s_i} \leq 1 \qquad (2)$$

is a necessary condition for the feasibility of Γ, since it represents the utilization based on the computation that must take place.

The concepts mentioned above can be clarified with an example. Consider the task set shown in Table 1 and the corresponding feasible schedule, obtained by EDF, illustrated in Figure 1. Notice that the cumulative processor utilization, U_p, is greater than 1 ($U_p = 1.25$), but condition (2) is satisfied.

Using the *processor demand* criterion, Baruah et al. [27] showed that a set of hard periodic tasks is schedulable by EDF if and only if, for any interval $L \geq 0$,

$$L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i. \qquad (3)$$

Based on this result, Koren and Shasha [12] proved the following theorem, which provides a sufficient condition for guaranteeing a set of skippable periodic tasks under EDF.

Theorem 1. *A set of firm (i.e., skippable) periodic tasks is schedulable if*

$$\forall L \geq 0, \quad L \geq \sum_{i=1}^{n} D(i, [0, L])$$
$$\text{where } D(i, [0, L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i. \qquad (4)$$

Koren and Shasha [12] proposed two online scheduling algorithms, *red tasks only* and *blue when possible*, to handle tasks with skips under EDF. In their theorem, $D(i, [0, L])$ represents the effective time demanded by the periodic task T_i over the interval $[0, L]$.

> *Red tasks only* (RTO) always skips blue instances whereas red ones are scheduled according to EDF.
>
> *Blue when possible* (BWP) is more flexible than RTO and schedules blue instances whenever there are no ready red jobs to execute. Red instances are scheduled according to EDF.

It is easy to find examples to demonstrate that BWP improves upon RTO in the sense that it can schedule task sets that RTO cannot schedule. In the general case, the above algorithms are not optimal, but they are optimal under a special task model, called the *deeply-red* model.

Definition 1. A system is deeply-red if all tasks are synchronously activated and the first $s_i - 1$ instances of every task τ_i are red.

Table 1: A schedulable set of firm periodic tasks.

Task	Task 1	Task 2	Task 3
Computation	1	2	5
Period	3	4	12
Skip parameter	4	3	∞
U_p		1.25	

In the same paper, Koren and Shasha showed that the worst case for a periodic skippable task set occurs when tasks are deeply-red. This means that, if a task set is schedulable under the deeply-red model, it is also schedulable without this assumption. For this reason, all results in this paper will be proved under the assumption above.

Buttazzo and Caccamo [23] defined the equivalent processor utilization, U_p^*, for a set of firm periodic tasks to be

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_i D(i, [0, L])}{L} \right\}. \qquad (5)$$

They then used the remaining (uniformly distributed) capacity, $1 - U_p^*$, to schedule aperiodic tasks. However, the equivalent processor utilization overestimates the system utilization, and there is some processor capacity that is not reclaimed because it has a "granular" distribution [23]. The spare capacity in the system can be calculated, and it is given by

$$U_{\text{spare}} = 1 - U_p + \sum_{i=1}^{n} \frac{c_i}{p_i s_i}, \qquad (6)$$

where U_p is the cumulative processor utilization if there were no skips.

This spare capacity can be categorized into two portions U_{sa} and U_{sh}. A portion of this capacity $U_{\text{sa}} = 1 - U_p^*$ is uniformly distributed and is assigned to the aperiodic server. The remaining portion of the spare capacity is nonuniformly distributed among many *holes* [23], and can be calculated as $U_{\text{sh}} = U_{\text{spare}} - U_{\text{sa}}$.

Table 2 shows a set of skippable tasks that can be feasibly scheduled under the RTO model with $U_p^* = 0.80$. Notice that the capacity distributed among the holes in the schedule accounts for 27 percent of the processor utilization. Being able to reclaim more than a quarter of the processor capacity can result in marked reductions in response times for aperiodic tasks.

In our work, we identify the spare capacity that is irregularly spaced and provide a mechanism for using this capacity to improve response times of aperiodic tasks.

3.3. The BASH Mechanism.

Using the basic results on firm periodic task scheduling, we address the feasibility analysis of hybrid task sets, consisting of firm periodic tasks and soft aperiodic requests.

The bandwidth sharing mechanism (BASH) works in conjunction with the constant bandwidth server (CBS) [28]. CBS provides isolation between tasks in a system; each task is allocated a bandwidth and a server to ensure that

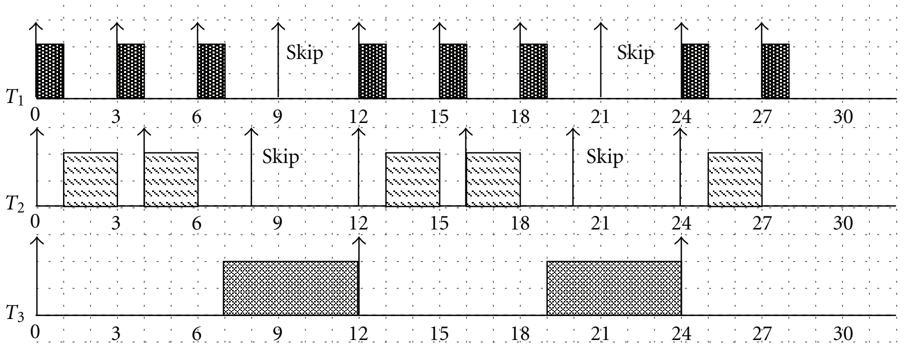


Figure 1: A feasible schedule of the task set shown in Table 1.

Table 2: Task set illustrating the existence of holes.

Task	Task 1	Task 2
Computation	2	2
Period	3	5
Skip parameter	2	2
U_p	1.07	
U_p^*	0.8	
$U_{sa} = 1 - U_p^*$	0.2	
U_{spare}	0.47	
U_{sh}	0.27	

it does not use more than the allotted bandwidth. BASH [17] was proposed as an approach for handling overruns in systems executing periodic tasks while preserving isolation. The primary motivation for bandwidth sharing was the observation that only a few instances of a task execute for the worst-case duration and that reserving resources using the worst-case consumption is expensive. BASH advocates a resource budget based on the bandwidth allocated to each task; when a task exceeds the allocated budget, residual capacities from jobs that finished before their budgets expired can be utilized to handle the overrun. BASH was proposed for periodic task sets with hard deadlines; if U_p is the utilization of the periodic task set, the remaining bandwidth, $1 - U_p$, can be assigned to an aperiodic task server. A global capacity queue, or a BASH queue, is used to keep track of the available excess capacity.

The BASH algorithm is specified by the following rules.

(1) Each server S_i is characterized by a budget c_i and by an ordered pair (Q_i, T_i), where Q_i is the maximum budget and T_i is the period of the server. The ratio $U_i = Q_i/T_i$ is denoted as the server bandwidth. At each instant, a fixed deadline $d_{i,k}$ is associated with the server. At the beginning $d_{i,0} = 0$, for all i. A global variable t^{idle} always maintains the finishing time of the last idle interval, and it is initially set to zero.

(2) Each BASH capacity is represented by an ordered tuple $\text{Cap}_q(r_q, d_q, c_q, U_q, T_q)$ where r_q is its release time (in the BASH queue), d_q is its absolute deadline, c_q is its budget, and U_q and T_q are the utilization and period of its generating server, respectively.

(3) Each task instance $\tau_{i,j}$ handled by server S_i is assigned a dynamic deadline equal to the current server deadline $d_{i,k}$.

(4) A server S_i is said to be active at time t if there are pending instances. A server is said to be idle at time t if it is not active.

(5) When a task instance $\tau_{i,j}$ arrives and the server is idle, the server generates a new deadline $d_{i,k} = \max\{r_{i,j}, d_{i,k-1}\} + T_i$, and c_i is recharged to the maximum value Q_i.

(6) When a task instance $\tau_{i,j}$ arrives and the server is active, the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline.

(7) Assuming instance $\tau_{i,j}$ is scheduled for execution at time t, the server S_i uses the capacity Cap_q in the BASH queue (if there is one) with the earliest deadline d_q such that $t < d_q \le d_{i,k}$; otherwise, its own capacity c_i is used. Suppose a BASH capacity Cap_q is used and $r_q < t^{\text{idle}}$, the budget c_q of Cap_q is updated as $c_q = \min\{T_q U_q, (d_q - t^{\text{idle}})U_q\}$ before Cap_q is used by server S_i; otherwise, the budget c_q is used as is. Notice that each BASH capacity with deadline less than or equal to the current time t will expire and be removed from the BASH queue.

(8) Whenever job $\tau_{i,j}$ executes, the used budget c_q or c_i is decreased by the same amount. When c_q becomes equal to zero, Cap_q is extracted from the BASH queue and the next capacity in the queue with deadline less than or equal to $d_{i,k}$ can be used.

(9) When the server is active and c_i becomes equal to zero, the server budget is recharged at the maximum value Q_i and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.

(10) When a task instance finishes, the next pending instance, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle, the residual budget $c_i > 0$ (if any) is inserted in the BASH queue as a capacity with release time equal to the current time, deadline,

bandwidth, and period equal to the server deadline, server bandwidth, and server period, respectively. Finally, c_i is set equal to zero.

(11) Each time the processor becomes idle for an interval of time $\Delta(t_1, t_2)$, the global variable t^{idle} is set equal to t_2 as soon as the idle interval ends.

BASH was developed for hard real-time task sets; our work pushes the envelope further by dealing with firm real-time tasks and sporadic tasks. The holes that occur in a schedule are identified and added (at the appropriate time) to the BASH queue and can be utilized by all tasks, especially aperiodic tasks. BASH performs better than CASH [16] because of improved idle-time handling.

3.4. Sequencing Aperiodic Tasks. When multiple aperiodic tasks are active (ready to be executed) at the same time, we have multiple policy choices to sequence the aperiodic requests. The simplest approach is to execute aperiodic tasks on a first-come-first-served basis. This is the approach we apply in our work, and this is the policy chosen during our simulations and experiments. Other policies for sequencing aperiodic tasks such as shortest remaining processing time (SRPT) can be applied. For instance, SRPT is likely to yield improvements in average response times. We have, however, not applied alternative schemes although they are compatible with the techniques we propose. One may also chose to apply some static priority rules to aperiodic tasks depending on the application context, and this too is easily integrated into the framework that we have developed.

Our emphasis is on showing that reclaiming spare capacity improves the response time of aperiodic tasks. Further optimizations, such as sequencing between aperiodic tasks, can be performed, and these may yield greater benefits.

4. The NCLB-CBS Technique

In this section, we formally describe the NCLB-CBS technique assuming that each task, τ_i, is handled by a dedicated CBS server, S_i, running on a uniprocessor system. Bandwidth is reclaimed through the BASH mechanism. We first present a mechanism for the simplest case—a *strictly periodic firm task system* where tasks are scheduled by EDF *according to the* RTO *model*. Prior to the discussion, we introduce a definition.

Definition 2. Given a set $\Gamma = \{\tau_i(p_i, c_i, s_i)\}$ of n periodic tasks that allow skips, the *metahyperperiod*, $H = \text{lcm}(p_1 \times s_1, p_2 \times s_2, \ldots, p_n \times s_n)$, is defined as the period after which the task schedule repeats itself.

As an example, the metahyperperiod of the task set in Table 2 is 30.

Spare capacities for a given task set are identified offline and added to the global capacity queue online. Holes are identified over the metahyperperiod for the given task set.

4.0.1. An Algorithm to Locate Holes

Definition 3. The *total activity duration* in an interval $[t_1, t_2]$ is defined as

$$A[t_1, t_2] = \int_{t_1}^{t_2} f(t)dt,$$
$$\text{where } f(t) = \begin{cases} 1 & \text{if the processor is busy at } t, \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

Remark 1. It is trivial to observe that $A[t_1, t_2] \leq t_2 - t_1$. Moreover, since $D(i, [0, L])$ is the effective time demand for a firm periodic task τ_i, when a task set is schedulable, we must have the activity duration over any time interval greater than or equal to the effective time demand over that interval. In effect, we can restate Theorem 1 as follows: a set of firm periodic tasks is schedulable if

$$\forall L \geq 0, \quad L \geq A[0, L] \geq \sum_{i=1}^{n} D(i, [0, L]). \tag{8}$$

Definition 4. A time instant t is called a *skip deadline* if it is the deadline for a task instance that is skipped.

The algorithm to locate holes in the schedule inflates the utilization of the task set by the factor $1/U_p^*$. Following this inflation, any spare capacity that exists is associated with a hole in the schedule. Note that a fraction, $U_{\text{sa}} = 1 - U_p^*$, of the processor capacity can be reclaimed simply by using an aperiodic task server of bandwidth U_{sa}. By inflating the execution times, we account for the known spare capacity and obtain only the capacity that is irregularly distributed. The algorithm has a complexity of $O(Hn)$ where H is the metahyperperiod and n represents the number of tasks. Spare capacities (the holes) are calculated at every skip deadline in Algorithm 1 and are characterized by the three-tuple (E_k, r_k, d_k) with E_k being the capacity, r_k the release time, and d_k the hole deadline. Capacities can also be calculated and placed at every task deadline. We will first describe how the holes can be reused when the RTO model is used. Following the description, we will formally prove that the capacities identified by Algorithm 1 are indeed holes and that the schedulability of the periodic task set is preserved.

4.1. RTO-EDF Scheduling with NCLB-CBS. In the RTO model, all the blue instances are rejected. Since all blue instances are skipped uniformly, the task schedule repeats every metahyperperiod. The extra capacities are calculated *offline* according to Algorithm 1.

Figure 2 shows the hole capacities for the task set in Table 2. Holes are identified at every skip deadline. Extra capacity at $t = 6$ is calculated according to (7) and Algorithm 1 as

$$E_0 = (L - A[0, L]) \times U_p^* = (6 - 5) \times 0.8 = 0.8, \tag{9}$$

since $A[0, 6] = 5$ and $U_p^* = 0.8$. Similarly, extra capacity at $t = 10$ is $E_1 = (10 - 7.5) \times 0.8 - 0.8 = 1.2$. Extra capacity E_0

```
Require: A set $\Gamma = \{\tau_i(p_i, c_i, s_i)\}$ of $n$ firm periodic tasks with an equivalent processor utilization $U_p^*$.
  $k \Leftarrow 0$
  $d_{-1} \Leftarrow 0$
  $H \Leftarrow \mathrm{lcm}(p_1 \times s_1, p_2 \times s_2, \dots, p_n \times s_n)$ {metahyperperiod}
  for all tasks $\tau_i$ do
    $c_i^* = c_i/U_p^*$
  end for
  Schedule the task set $\Gamma^* = \{\tau_i(p_i, c_i^*, s_i)\}$ using the EDF-RTO scheduler
  for all time $t$ where ($t$ is a task skip-deadline) and ($t \le H$) to
    $E_k \Leftarrow (t - A[0,t]) \times U_p^* - \sum_{j=0}^{k-1} E_j$
    $r_k \Leftarrow d_{k-1}$
    $d_k \Leftarrow t$
    Add hole $(E_k, r_k, d_k)$ to the hole capacity list
    $k \Leftarrow k + 1$
  end for
```

Algorithm 1: Locate holes and determine capacities.

is assigned a deadline $d_0 = 6$, and released at time 0 while E_1 is assigned a deadline $d_1 = 10$ and released at time 6. In the example, no hole capacities are added at certain skip deadlines (e.g., $t = 12$, $t = 20$) because there is no new hole in the schedule at those time instants.

Extra capacities for the entire metahyperperiod are calculated offline. They are released online according to Algorithm 2.

It is important to observe that holes correspond to idle intervals in the task schedule with inflated execution times; however, identifying holes makes it extremely efficient to exploit spare capacity in the system—this approach is much better than background execution.

The BASH capacity queue can now consist of two types of capacities—the BASH capacity that is a result of early job completions and hole capacities. These capacities need to be distinguished when being consumed. We denote capacities due to holes as $\mathrm{Hole}_q(E_q, d_q)$ (A hole is characterized by the three tuple $\{E_k, r_k, d_k\}$ but when the hole capacity is added to the BASH queue it is sufficient to retain only the capacity and deadline.) and when a hole is released E_q is set to the hole capacity. To BASH (Rule 7) as follows.

Assuming instance $\tau_{i,j}$ is scheduled for execution at time t, the server S_i uses Hole_q or Cap_q in the BASH queue (if there is one) with the earliest deadline d_q such that $t < d_q \le d_{i,k}$, otherwise its own capacity c_i is used.

(a) If the capacity is a hole capacity and $(d_q - E_q/U_p^*) < t^{\mathrm{idle}} < d_i$, the budget c_q is updated as $c_q = (d_q - t^{\mathrm{idle}})U_p^*$ before Hole_q is used by server S_i, otherwise the budget c_q is used as it is.

(b) If Cap_q is used and $r_q < t^{\mathrm{idle}}$, the budget c_q of Cap_q is updated as $c_q = \min[T_q U_q, (d_q - t^{\mathrm{idle}})U_q]$ before Cap_q is used by server S_i, otherwise the budget c_q is used as it is.

The need for the modified rule can be understood easily: if a hole was released *before* an idle interval ended, some of the idle time leads to a reduction in hole capacity. Hole capacities are included in the effective processor utilization, U_p^*, therefore idling leads to a reduction in hole capacities at the rate U_p^*. (idling will also lead to a reduction in directly reserved aperiodic capacity at the rate of $1 - U_p^*$.)

The keystone for this work on exploiting holes is to transform background time into reserved bandwidth by reclaiming resources. Each CBS server is able to reclaim bandwidth by consuming spare capacity while preserving its own budget. A formal discussion of this intuition follows.

4.1.1. Theorems and Proofs

Theorem 2. *Given a set $\Gamma = \{\tau_i(p_i, c_i, s_i)\}$ of n firm periodic tasks with an equivalent processor utilization factor $U_p^* \le 1$, the set $\Gamma^* = \{\tau_i(p_i, c_i^*, s_i)\}$ where $c_i^* = c_i/U_p^*$ is schedulable.*

Proof. We need to prove that

$$\forall L \ge 0, \quad L \ge \sum_{i=1}^{n} D(i,[0,L])$$
$$\text{where } D(i,[0,L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times c_i^*. \tag{10}$$

Since $c_i^* = c_i/U_p^*$, alternatively, we need to prove that

$$\forall L \ge 0, \quad L \ge \left(\sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) * \frac{c_i}{U_p^*} \right). \tag{11}$$

By the definition of U_p^* (5), we have

$$U_p^* \ge \left(\sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times \frac{c_i}{L} \right)$$
$$\Longrightarrow L \ge \left(\sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times \frac{c_i}{U_p^*} \right). \tag{12}$$

□

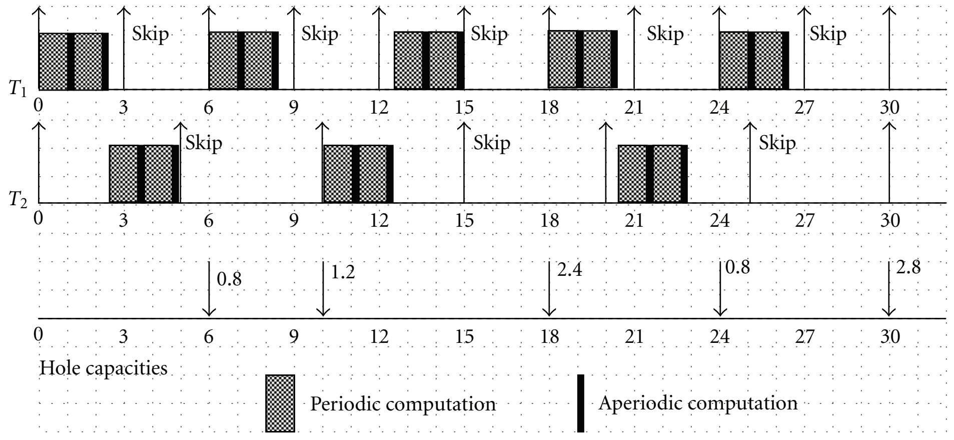


Figure 2: Task set (Table 2) scheduled using the RTO model with inflated computation times.

Theorem 3. *Algorithm 1 preserves the aperiodic bandwidth over any time interval $[t_1, t_2]$.*

Proof. We will consider three cases.

Case 1 (Processor is fully occupied during interval $[t_1, t_2]$). Algorithm 1 assumes that the time between $[t_1, t_2]$ is divided into discrete time units such that each unit resembles Figure 3.

Hence, for every $\Delta = t_2 - t_1$, $U_{\text{sa}} \times \Delta$ is available as aperiodic bandwidth. This, however, increases the computation for task τ_i to c_i/U_p^*; the schedulability for which is proved in Theorem 2.

Case 2 (Processor is idle during interval $[t_1, t_2]$). In Algorithm 1, when spare capacity is calculated at every skip deadline, only a fraction, U_p^*, of it is identified as hole capacity. The remaining spare capacity is the aperiodic bandwidth $U_{\text{sa}} = 1 - U_p^*$. Therefore, for any idle interval, $[t_1, t_2]$, the aperiodic bandwidth is conserved.

Case 3 (Processor is partially busy during interval $[t_1, t_2]$). This case is a combination of Cases 1 and 2. Since the theorem holds for Cases 1 and 2, it holds for this case. □

Theorem 4. *Addition of extra capacities does not affect the schedulability of the original task set $\Gamma = \{\tau_i(p_i, c_i, s_i)\}$.*

Proof. We need to prove

$$\forall L, \quad L \geq \sum_{i=1}^{n} D(i, [0, L]) + \left(1 - U_p^*\right)L + \sum_{k, d_k \leq L} E_k, \tag{13}$$

where $D(i, [0, L])$ is the effective time demanded by $\tau_i(p_i, c_i, s_i)$, $(1 - U_p^*)L$ is the total aperiodic bandwidth in $[0, L]$, and E_k is the hole capacity with deadline d_k.

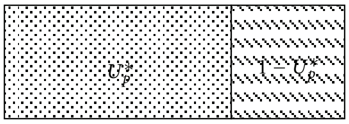


Figure 3: A single unit of time.

By taking the $(1 - U_p^*)L$ term to the left-hand side and in (13) and then dividing throughout by U_p^*, we need to show

$$L \geq \frac{\sum_{i=1}^{n} D(i, [0, L])}{U_p^*} + \frac{\sum_{k, d_k \leq L} E_k}{U_p^*}. \tag{14}$$

Algorithm 1 uses the task set with inflated computation times, Γ^*. However, since $U_p^* \leq 1$, Γ^* is schedulable. From (8), we have

$$A[0, L] \geq \left\{ \sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times \frac{c_i}{U_p^*} \right\} = \frac{\sum_{i=1}^{n} D(i, [0, L])}{U_p^*}. \tag{15}$$

Using the above inequality in (14), to prove the theorem, we need to show that

$$(L - A[0, L]) \times U_p^* \geq \sum_{k, d_k \leq L} E_k. \tag{16}$$

This, however, follows directly from Algorithm 1 when L is a skip deadline because

$$E_{k, d_k = L} = (L - A[0, L]) \times U_p^* - \sum_{j, d_j < L} E_j. \tag{17}$$

If L is not a skip deadline, let L' be the greatest skip deadline such that $L' < L$ (L' can be 0.) Then, we have

$$(L' - A[0, L']) \times U_p^* = \sum_{k, d_k \leq L'} E_k. \tag{18}$$

Rewriting $L - A[0, L]$ as $(L - L') - A[L', L] + L' - A[0, L']$, we need to show that

$$((L - L') - A[L', L] + L' - A[0, L']) \times U_p^* \geq \sum_{k, d_k < L} E_k. \quad (19)$$

Using (18), we simply need to prove that $(L - L') - A[L', L] \geq 0$. This is trivial because the activity over a time interval cannot exceed the length of the interval. □

4.2. BWP-EDF Scheduling with NCLB-CBS. Having discussed scheduling firm periodic tasks and aperiodic tasks under the RTO model, we turn our attention to the BWP model. The focus is still on firm real-time task systems.

In Section 4.0.1, we presented an algorithm to determine the holes in a schedule consisting of firm real-time tasks. Locating the holes provides an elegant integration with BASH that can reduce the response times of aperiodic tasks. We implicitly assumed that task releases were strictly periodic, and that the RTO model alone was employed for scheduling. These assumptions simplified the problem of locating holes, because it is possible to construct the schedule offline.

It is possible to improve the scheduling behavior by using the BWP model: whenever a job belonging to a firm periodic task can be executed, it is. Jobs are skipped only when their execution can cause some other job to miss its deadline. The BWP model performs at least as well as the RTO model. However, BWP executions create irregular schedules, especially when periodic and aperiodic tasks are both present. As a result, it is not possible to precompute the parameters for the holes (release times, deadlines, and hole sizes).

An idea that easily extends the approach used in the RTO case employs *task patterns.*

Definition 5. A *task pattern* is defined as a fixed series of skipped and red instances such that the minimum distance between two skipped instances is equal to the skip parameter *s*.

It is easy to see that the total number of unique task patterns for a task τ_i is equal to s_i—any one of the first s_i jobs may be skipped, and depending on which job is dropped a pattern starts. For a task set with n tasks, the total number of pattern combinations is $\Psi = s_1 \times s_2 \times \cdots \times s_n$. The total number of unique task patterns for a hard task is equal to 1.

For the task set in Table 2, the total number of task pattern combinations is $\Psi = s_1 \times s_2 = 2 \times 2 = 4$. One example of these task pattern combinations is shown in Figure 2. Another example is shown in Figure 4.

The BWP model schedules blue instances when there are no ready red instances of periodic tasks or aperiodic jobs to schedule. This causes the blue instances to always execute in background. When a blue instance completes successfully, the next task instance is made blue; this leads to a change in the task pattern impacting the way the hole capacities are distributed across the schedule.

The spare capacity is calculated by Algorithm 1 under the assumption that all blue instances are rejected. We could recalculate the extra capacities each time a blue instance completes successfully, but the operation has to be performed online unlike in Section 4.1 leading to an overhead $O(Hn)$. We propose a scheme that has lower computational overhead but requires extra storage.

Given a set $\Gamma = \{\tau_i(p_i, c_i, s_i)\}$ of n periodic tasks that allow skips, the distribution of the extra capacity is calculated for all $\Psi = s_1 \times s_2 \times \cdots \times s_n$ combinations. Each pattern results in a unique *hole capacity distribution* which is stored in a hash table indexed by the corresponding pattern combination.

When a blue instance completes successfully at time t^{blue}, the task pattern change is detected and

(i) the current hole capacity (from the old pattern) present in the global capacity queue is deleted,

(ii) hole capacities computed offline for the new pattern are released starting with the hole with the nearest skip deadline in the new pattern.

The online cost is minimal since the cost of pattern lookup is $O(1)$ (hash table). Rules for entering holes into the BASH queue are identical to those specified in Algorithm 2. The pattern remains unchanged until a blue instance is completed.

An Example. Figure 5 shows the hole capacities for two patterns for the task set in Table 2. Let us now consider a schedule in which a blue instance of a task is able to complete successfully at $t^{\text{blue}} = 17$. Figure 6 shows such a schedule: the blue instance of task τ_2 released at time $t = 15$ completes execution. This triggers a pattern switch from the current pattern shown in Figure 5(a) to the new pattern illustrated in Figure 5(b) at the nearest skip deadline of Pattern 2, time $t = 17$.

A blue instance executes with background priority, since both periodic and aperiodic tasks can preempt it. The execution time of a blue instance is analogous to idle time, and idle time rules of BASH apply. This results in the hole capacity placed at time $t = 18$ being switched from 2.4 to 0.4; the hole capacity corresponding to the old pattern is deleted, and the hole capacity corresponding to the new pattern is inserted. (This pattern change can be seen by comparing Figures 5(a) and 5(b).)

Theorem 5. *A task pattern switch, which leads to a new hole capacity distribution, does not cause a deadline miss for periodic tasks.*

Proof. The task set Γ is schedulable with the addition of extra capacities across all Ψ task patterns by Theorem 4.

Let t^{blue} be the time at which a blue instance completes triggering a pattern switch.

The execution time of a blue instance is analogous to idle time, and we can apply the idle interval lemma to conclude that events occurring at time $t \leq t^{\text{blue}}$ do not impact the schedule beyond t^{blue}. The new hole that is introduced will not violate any other task execution because of BASH's idle time rules.

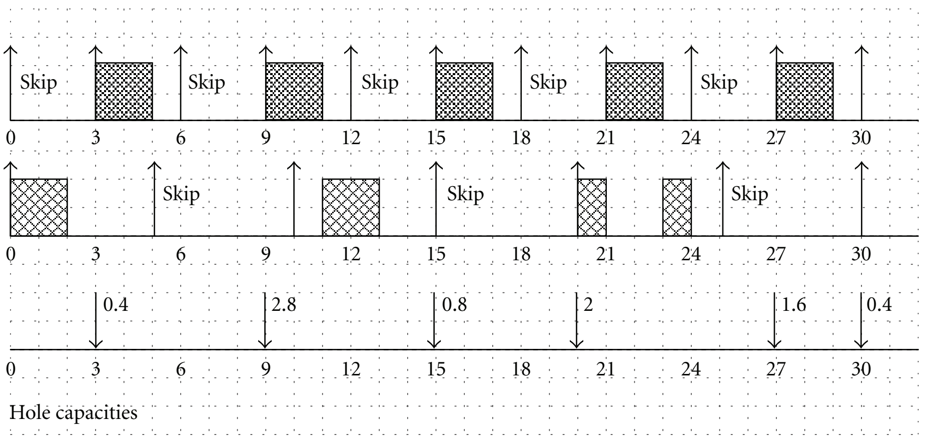


Figure 4: A combination of task patterns for the task set shown in Table 2.

```
$k \Leftarrow 0$
**loop**
    $t = current\_time()$
    **if** $r_k = t \bmod H$ **then**
        Insert $(E_k, (\lfloor t/H \rfloor \times H) + r_k, (\lfloor t/H \rfloor \times H) + d_k)$ into the global capacity queue.
        $k \Leftarrow (k+1) \bmod listSize$
    **end if**
**end ioop**
```

Algorithm 2: Hole capacity release.

For all $t \geq t^{\text{blue}}$: this time interval belongs exclusively to the new task pattern. The schedulability guarantee follows directly from Theorem 4. □

It is also possible to store only a subset of pattern combinations. Then, successful completion of a blue instance may not lead to the next instance being blue. In such situations, the overhead is reduced because there are fewer pattern switches, but this will produce suboptimal results.

In this section, we outlined a possible approach to support the BWP scheduling model for firm real-time tasks. This approach, using task patterns, extends the approach outlined for the RTO model using a pattern library to alter hole information when a blue instance completes.

Next, we will explore the case when the main task set is sporadic as opposed to being periodic. Sporadic tasks require a different method because it is not possible to *a priori* compute the idle times (holes) in a sporadic setting.

5. NCLB-CUS: Easy Reclamation with Sporadic Task Sets

The BWP model alone does not introduce difficulties in determining the locations of the holes. Sporadic tasks present similar problems. Sporadic tasks are not strictly periodic; successive jobs belonging to the same task arrive (are released) with a minimum separation distance. The unpredictability in arrival times (subject to the minimum separation) makes it difficult to identify holes ahead of time. In this section, we will discuss a technique, NCLB-CUS, that can accommodate sporadic tasks. This technique can be used for both hard and firm deadlines.

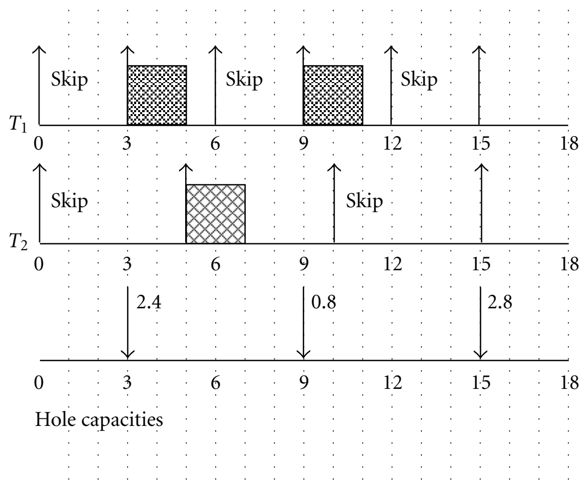
A CBS-based server can provide task isolation and ensure that no task misses deadlines as long as it does not exceed its resource reservation. Background execution occurs when the task set does not utilize the processor fully and idle times (specifically, intervals when the processor would have idled if there were only periodic tasks without skips) occur. In a traditional CBS mechanism, tasks utilizing these intervals are penalized with lost capacity. We can also state this observation about CBS as follows: "By automatically postponing a server's deadline when the server exhausts its budget, CBS never forces idle time."

When tasks are strictly periodic, indeed we can locate holes using an offline scheme (Algorithm 1), but, when tasks are sporadic, idle times need to be identified online. At worst, sporadic tasks are periodic, and, therefore, reservations must be made in cognizance of this fact.

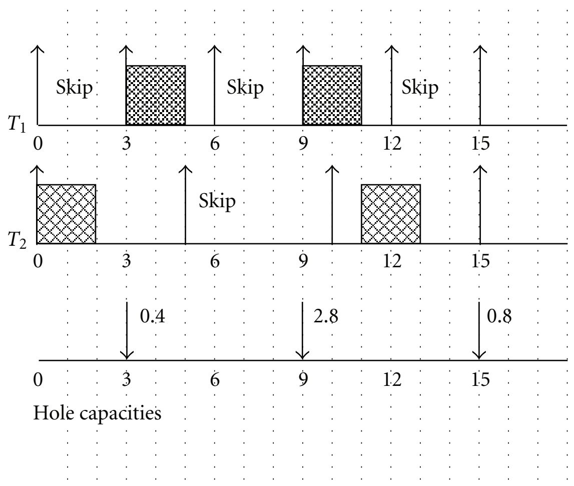
To ensure that idle time (with respect to a strictly periodic schedule) can be identified and reclaimed with the knowledge that any execution that occurs during such intervals need not be associated with budget decrements, one can employ hard reservations using a constant utilization server [29]. CBS and CUS differ only in the manner in which they update server budgets.

BASH can be integrated with constant utilization (instead of constant bandwidth) servers. This requires no changes to BASH.

The CUS replenishment rules (What we present here is a slightly modified version of the CUS [14]. Originally,



(a) Pattern 1



(b) Pattern 2

Figure 5: Task patterns for task set in Table 2.

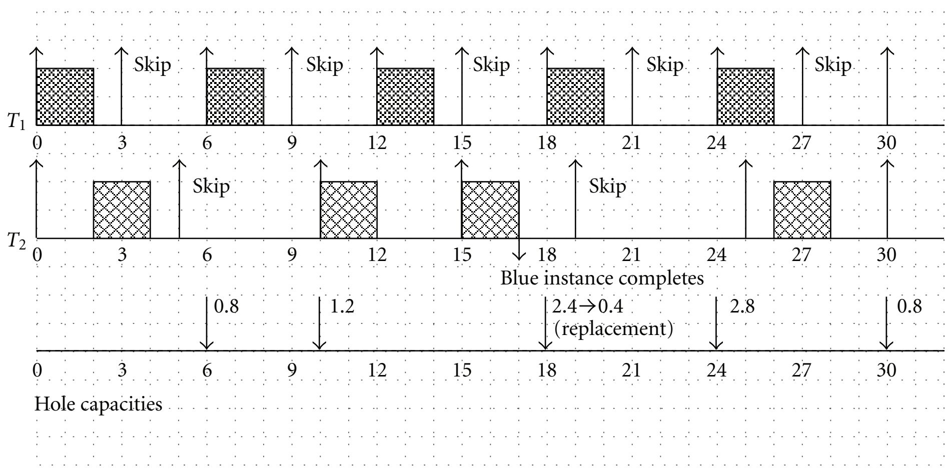


Figure 6: Schedule produced by BWP for the task set shown in Table 2.

CUS sets a deadline on the basis of the execution time of the aperiodic job. We assume a server period and assign the budget based on the server bandwidth.) for a server $S_i(Q_i, T_i)$ with budget c_i and deadline d_i are as follows.

(1) Initially, $c_i = 0, d_i = 0$.

(2) When an aperiodic job arrives at time t to an empty aperiodic job queue,

  (a) if $t < d_i$, do nothing,

  (b) if $t \geq d_i$, $d_i = t + T_i$, and $c_i = Q_i$.

(3) At the deadline d_i of the server,

  (a) if the server is backlogged, set the server deadline to $d_i + T_i$ and $c_i = Q_i$;

  (b) if the server is idle, do nothing.

By default, the CUS approach does not schedule an aperiodic task if it arrives at time $t < d_i$ (where d_i is the server's current deadline) even if there is leftover budget from an earlier execution. In conjunction with BASH, this situation is easy to handle because all unused budgets are stored in the BASH queue as capacities, and these capacities can be used when a new aperiodic instance arrives.

The key point to note is that with CUS, a server may be backlogged and may have run out of budget; the budget can be recharged only at the server deadline. We call this a server's *dormant* state. When a server is backlogged and has budget to expend, it is in the *active* state. When a server has no pending jobs, it is in the *inactive* state. Now, under a strictly periodic task set, there is idle time when all servers are either dormant or inactive.

To improve aperiodic response times, our NCLB-CUS scheme prioritizes aperiodic jobs when all servers are dormant or inactive. Because this is time that should have been idle time, giving preference to aperiodic tasks does not affect any of the sporadic (or periodic) tasks. This has the same effect as allowing the aperiodic server to use a hole under the NCLB-CBS scheme.

The proof of correctness follows trivially from the original proofs for CUS [14] and BASH [17].

NCLB-CUS can be implemented by distinguishing between the deadline and the replenishment time. By enforcing a replenishment time (via CUS), we ensure that there are no unnecessary deadline postponements. The approach can be trivially incorporated in a deadline-based scheduler by setting the deadlines of all servers in the dormant state to infinity (some suitably large value) and introducing a capacity into the BASH queue with infinite budget and a deadline of infinity (as can be done with dormant tasks). This allows the same control mechanism for BASH capacities. (We cannot insert a BASH capacity at each hyperperiod with budget $1 - U$ where U is the peak utilization of the sporadic task set because that is not enough to reclaim all idle times. For this reason, we introduce a capacity with infinite budget and infinite suitably large deadline. The aperiodic server, when in the dormant state, can access this capacity because it too will have a sufficiently large deadline.) When a server's budget is recharged, its deadline can be updated. Ties among dormant servers can be broken in favor of the aperiodic task server.

The technique we have described in this section requires no knowledge of holes. Holes are background cycles that are difficult to identify with traditional BASH/CBS rules, but using CUS rules provides sufficient information to identify the holes and use them without deducting from server budgets. This simplifies the implementation of the resource reclamation mechanism.

6. Empirical Results

We tested the NCLB algorithms using the OMNe T++ simulator [30] to quantify the performance gains. In this section, we present the results of our experiments. We compare the performance of the techniques proposed in this paper with simple resource reclamation as enforced by BASH.

The metric for comparison in all our experiments is the normalized aperiodic response time. Thus, a value of 5 on the y-axis actually means an average response time five-times longer than the task computation time; a value of 1 corresponds to the minimum achievable response time. For all experiments, the results have been averaged over 25 runs, each of a duration of 1,000,000 time units. We do not explicitly plot a 98% confidence interval, but we note that the width of the confidence intervals was not greater than 7% of the mean in any of the evaluations.

In this section, we will use the term NCLB when we refer to both the NCLB-CBS and NCLB-CUS algorithms. When we wish to distinguish between the two algorithms, we make use of the specific term.

6.1. Performance Evaluation for Firm Real-Time Environments. The first five experiments were to assess the impact of NCLB in a firm real-time environment. These experiments can be divided into two groups. The first group shows the performance of the algorithms as a function of the aperiodic load, for three different values of U_{sh}. The second group of experiments tests the sensitivity of the algorithms to the average computation time of aperiodic requests.

Execution times of aperiodic requests were chosen from a uniform distribution over a predefined interval whereas their interarrival times were generated according to an exponential distribution, with the mean computed to impose a specific aperiodic load ρ_a. The periodic task set consists of five periodic tasks with $U_p^* = 0.90$ and different hole capacities, U_{sh}.

We do not employ BWP in any of our experiments. Our main objectives are to determine the improvements in aperiodic response using NCLB when compared to a simple BASH implementation, and to investigate the difference between NCLB-CBS and NCLB-CUS with RTO scheduling.

We emphasize that the basic implementation that uses BASH alone is identical to EDF-RTO when no task uses less than its budget. That is the model we use for our empirical analysis, and, therefore, the BASH evaluation represents the performance that can be achieved by the basic approach proposed by Koren and Shasha [12]. We demonstrate that our approach improves the behaviour of aperiodic tasks significantly in comparison to the simplest possible approach.

6.1.1. Performance with Varying Aperiodic Load. The first set of experiments includes three simulations which show the performance of the algorithms as a function of the aperiodic load for low, medium, and high values of U_{sh}. Execution times of aperiodic requests were chosen to be uniformly distributed in the interval $[2, 10]$. Periods, computation times, and skip parameters of the tasks for every simulation are shown in Table 3. Notice that the value of U_{sh} is increased from the first to the third simulation, which means that more instances are skipped in the second and third experiments. The equivalent processor utilization, U_p^*, is kept constant at 0.90 for all three experiments. The aperiodic server has a fixed bandwidth $U_{\text{sa}} = 1 - U_p^* = 0.10$.

Figure 7 shows the results of the first experiment, with $U_{\text{sh}} = 0.12$, in which very few periodic instances are skipped and includes a periodic hard task. As the reader can see, the NCLB algorithms outperform vanilla BASH, and, in fact, the lowest average response times are achieved when NCLB-CUS policy is used because it can handle BWP as well. NCLB algorithms outperform BASH for values of ρ_a in the range (0.08–0.24). This range is approximately equal to U_{sh}. For values of ρ_a outside this range, the aperiodic response behavior for both the algorithms is similar. The aperiodic response time under the BASH algorithm grows at a moderate pace after an initial spurt, since aperiodic requests continue to be serviced during the holes with deadlines periodically postponed according to CBS rules.

NCLB, in general, performs better than BASH because of aggressive reclamation. This holds for all our experiments. Among the NCLB algorithms, we find that using NCLB-CBS (determining holes offline and inserting them into the BASH queue) yields better response times for aperiodic tasks. This is to be expected because CBS is set up to reduce response times of tasks. CBS advances a server's deadline and recharges

Table 3: Simulations parameters for the first set of experiments.

Simulation	Task	Task 1	Task 2	Task 3	Task 4	Task 5
I	Computation	8	5	35	15	20
	Period	90	100	150	60	60
	Skip	5	3	∞	5	5
II	Computation	12	5	50	20	25
	Period	90	100	150	60	60
	Skip	2	3	3	2	2
III	Computation	10	2	46	18	26
	Period	90	100	150	55	55
	Skip	2	2	2	2	2

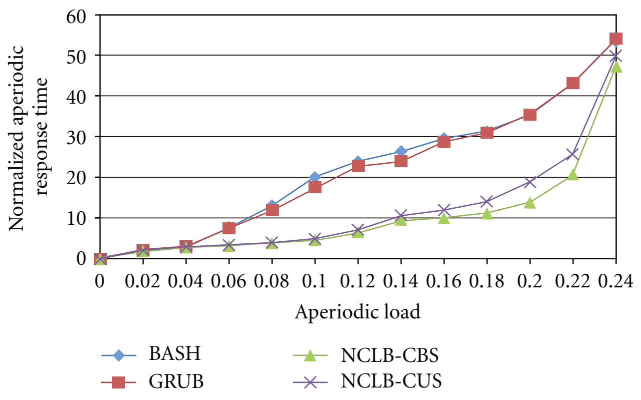


Figure 7: Varying aperiodic load; $U_p^* = 0.9$, $U_{\text{sh}} = 0.12$, $U_{s_{\max}} = 0.22$.

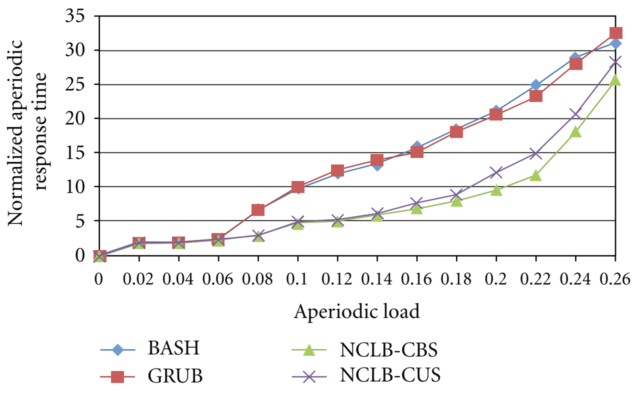


Figure 8: Varying aperiodic load; $U_p^* = 0.9$, $U_{\text{sh}} = 0.20$, $U_{s_{\max}} = 0.30$.

the budget as soon as the budget expires. This gives CBS an opportunity to schedule the task immediately; CUS, on the other hand, replenishes the budget only at specific times and may cause tasks to wait.

Figure 8 refers to the second experiment, in which $U_{\text{sh}} = 0.20$. More periodic instances are skipped which results in a lower aperiodic response when compared to the first experiment, as aperiodic load remains identical. NCLB improves aperiodic response time for values of ρ_a in the range (0.08–0.26). This range is higher than the first experiment since $U_{\text{sh}} = 0.20 > 0.12$. Again, the performance of both algorithms is seen to be similar for values outside this range.

The results of the third experiment is shown in Figure 9. In this case, $U_{\text{sh}} = 0.27$ is the highest value in all the experiments. The improvement in aperiodic response time occurs over a larger range (0.08–0.32); thus, NCLB can yield better response times even when a large fraction of processor cycles are unused.

The NCLB algorithms work by identifying holes (idle times that would lead to background execution of aperiodic tasks) and explicitly avoiding unnecessary deadline postponements for the aperiodic server; this enables better aperiodic response times.

According to the first set of experiments, three distinct zones can be identified in terms of achieved performance.

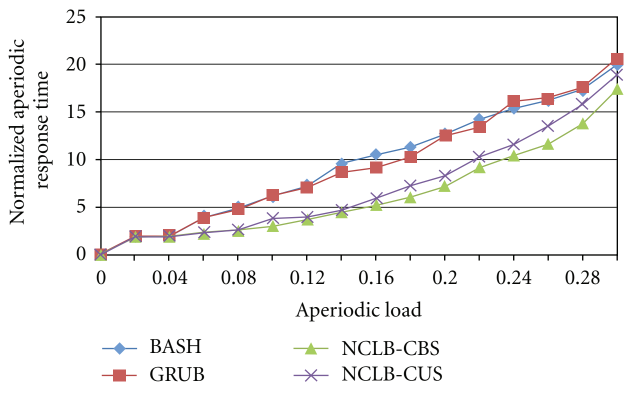


Figure 9: Varying aperiodic load; $U_p^* = 0.9$, $U_{\text{sh}} = 0.27$, $U_{s_{\max}} = 0.38$.

(1) $\rho_a \leq U_{\text{sa}}$: in this zone, aperiodic response of BASH and NCLB are identical. If aperiodic load is less than U_{sa}, BASH can be as competitive as NCLB in scheduling aperiodic tasks. The hole capacity, U_{sh}, is not utilized much in this traffic zone.

(2) $U_{\text{sa}} < \rho_a \leq U_{\text{spare}}$: here NCLB outperforms BASH. The workload is consistently greater than U_{sa}, and, therefore, the holes are necessary, and NCLB is able to serve aperiodic tasks better.

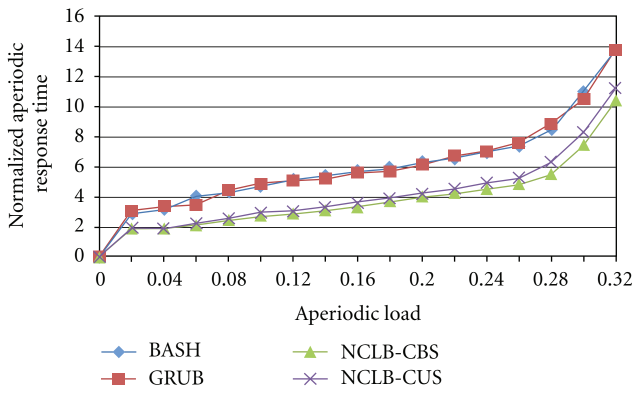


Figure 10: Medium length aperiodic tasks; $U_p^* = 0.9$, $U_{\text{sh}} = 0.27$, $U_{s_{\max}} = 0.38$.

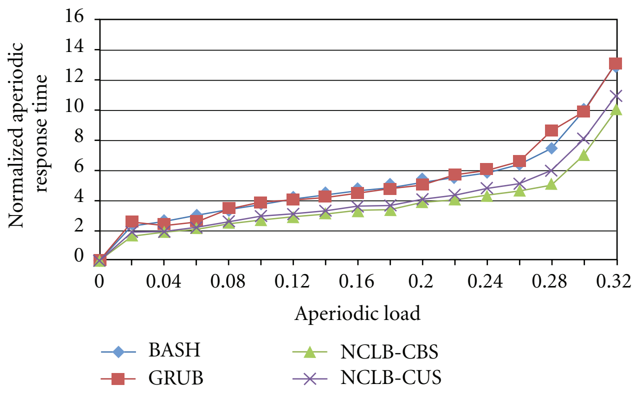


Figure 11: Long aperiodic tasks; $U_p^* = 0.9$, $U_{\text{sh}} = 0.27$, $U_{s_{\max}} = 0.38$.

(3) $\rho_a > U_{\text{spare}}$: aperiodic response is identical again. When the aperiodic workload exceeds $U_{\text{sa}} + U_{\text{sh}}$, the response times increase rapidly for BASH and NCLB. The aperiodic tasks saturate all capacity, and this leads to the convergence in performance.

Moreover, we observe that when the number of skips increases (but aperiodic load is held constant), the gap between BASH and NCLB decreases. This might seem to be counterintuitive, but the reason is straightforward: when more jobs are skipped, more idle time is created. Even if CBS postpones deadlines, aperiodic tasks still complete early. Thus, the gap between BASH and NCLB reduces when we increase the skips. In our experiment, when all tasks have a skip factor of 2, the effect is almost the same as doubling the period of the tasks. A lot of spare capacity is uniformly distributed and can be reclaimed quite easily by an aperiodic task server.

6.1.2. Sensitivity to Aperiodic Computation Time. To test the sensitivity of the algorithms with respect to the length of aperiodic tasks, three simulations were carried out using task sets with short, medium, and long aperiodic computation times (ACT). In particular, execution times of aperiodic requests were chosen from the uniform distribution over the interval [15,20] (medium length) and [25,30] (long). To limit the total number of graphs, the periodic tasks used were only those used in the earlier group of experiments. Additionally, in the earlier set of experiments we used aperiodic tasks with execution time uniformly distributed over the [2,10] interval, and they represent short aperiodic tasks. The results of the experiments to measure the impact of aperiodic computation time are shown in Figures 10 and 11, respectively.

The improvement in performance achieved by NCLB over BASH is more significant when aperiodic requests have short computation times (compare Figures 9 and 10, e.g.). As the ACTs become longer, the performance of both NCLB algorithms tend to be similar to the one achieved by BASH. This is because, for long aperiodic tasks, advancing the position of small slack intervals in the schedule does not make a great impact on the response times.

6.2. Aperiodic Tasks Along with a Set of Sporadic Hard Real-Time Tasks

In this section, we elaborate on the performance of NCLB-CUS when scheduling aperiodic tasks along with sporadic tasks. For these experiments, we generated random task sets with varying levels of utilization. Aperiodic tasks were generated in the same manner as in the previous experiments.

For sporadic task sets, the utilization factor is, more precisely, the peak utilization factor (when all tasks are periodic). For each task generated, the minimum interarrival time, p_i, between instances was chosen at random, and during the experiments, the actual interarrival time between jobs belonging to the same task were chosen as follows.

Moderate Sporadicity. For each task τ_i, the interarrival time was chosen from the uniform distribution over the range $[p_i, 1.4p_i]$.

High Sporadicity. For each task τ_i, the interarrival time was chosen from the uniform distribution over the range $[p_i, 2p_i]$.

We first kept the peak utilization of the sporadic task set constant at 0.75 and varied the aperiodic load (For these experiments, we generated 20 sporadic task sets at random with peak utilizations of 0.75.). We considered the two cases: moderately sporadic tasks (Figure 12) and highly sporadic tasks (Figure 13).

NCLB-CUS performs much better than BASH because BASH is ideally suited for strictly periodic task sets and is unable to account for extra processor bandwidth that results from sporadicity. NCLB-CUS can recognize this additional capacity and does not waste the budget of the aperiodic server when this capacity can be exploited. With moderately sporadic task sets, NCLB-CUS provides greater performance by reclaiming even the small capacity excess that is a result of moderate sporadicity. As sporadicity increases and aperiodic load is held constant, there is a uniform increase in available bandwidth, and the gap between BASH and NCLB-CUS

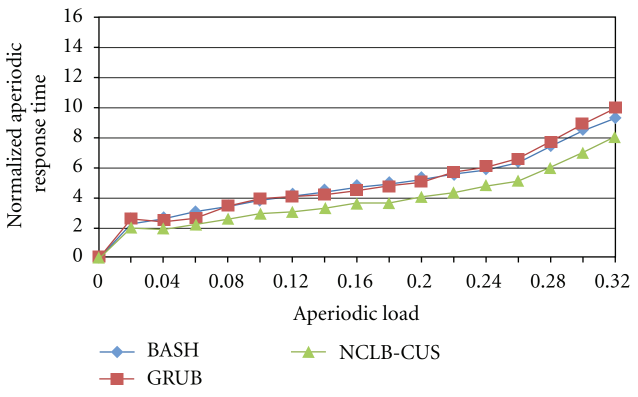


Figure 12: Task set with moderate sporadicity.

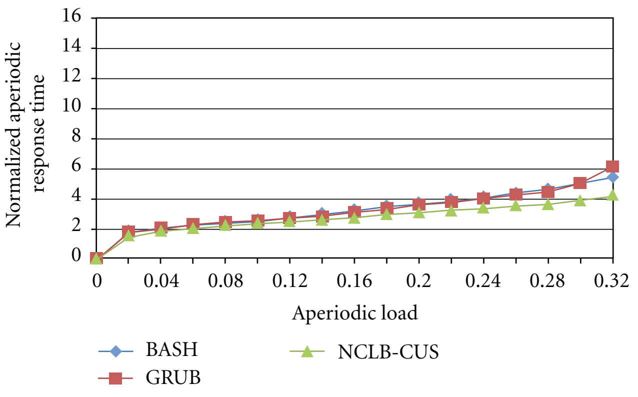


Figure 13: Task set with high sporadicity.

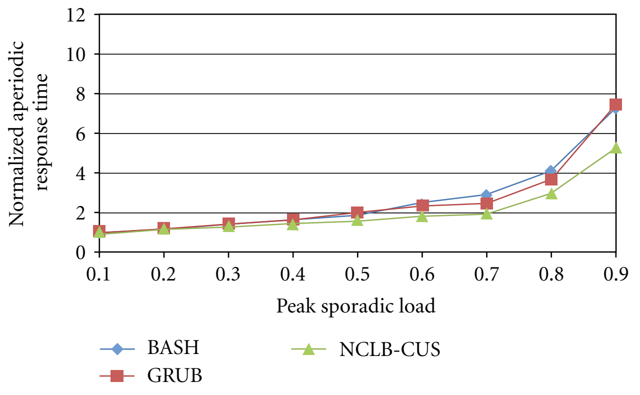


Figure 14: Impact of varying peak task set utilization; moderate sporadicity.

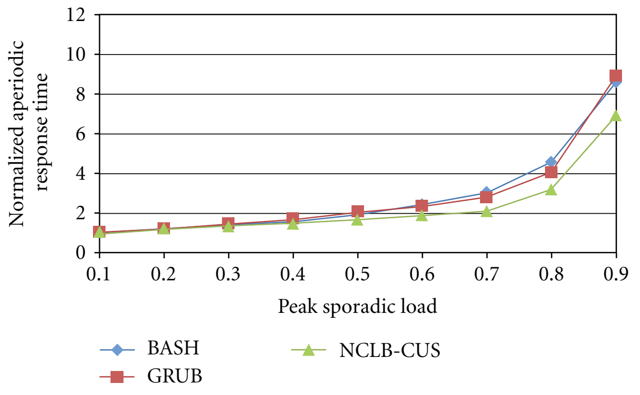


Figure 15: Impact of varying peak task set utilization; high sporadicity.

decreases. Both of them perform better when sporadicity is high when compared to the situation when sporadicity is moderate.

The last set of experiments measure the impact of the peak sporadic workload (Figures 14 and 15). For these experiments, the aperiodic load was kept constant at 0.25. When peak utilization of the sporadic task set is low, most of the processor bandwidth is directly available to the aperiodic server, and there is not much difference between BASH and NCLB-CUS. As the peak utilization increases, a smaller fraction of the processor can be reserved *a priori* for the aperiodic server, and the aperiodic response times increase. NCLB-CUS continues to perform better, and the gap between NCLB-CUS and BASH is greatest when the utilization of the sporadic task set is high and sporadicity is moderate.

7. Implementation

To further validate the proposed approach, we implemented the NCLB-CBS and NCLB-CUS scheduling policies within the RTLinux kernel [31]. We modified an existing implementation of CBS on RTLinux [32] to support our schemes.

To test our implementation, we built a prototype application for building security. This application retrieves image streams, over the local area network, from web cameras. These streams are essentially a sequence of JPEG images retrieved at configurable rates. We chose this design for its simplicity. Displaying a sequence of JPEG images requires an extremely simple viewer, and it avoids the need for MPEG (or similar) decoding that is more sensitive to the frames that can be dropped. With a sequence of still images, we could obtain sufficient video fidelity and skip frames without considering semantic information. We varied the number of camera feeds from 4 to 10 to change the periodic workload. The skip factor for each feed was 12, that is, one frame could be dropped in a window of 12 frames. The frame rate for each camera was chosen to be 24 frames per second. As a source of aperiodic workload, we ran an instance of the Apache web server that would process simple HTTP requests.

This prototype loosely mimics a security system that can process many video streams (periodic, firm, real-time tasks) and handle other aperiodic requests that could include messages to security personnel.

The purpose of this implementation was to obtain a measure of the overhead introduced by adding complexity to the scheduler. We performed our evaluation on an Intel Pentium III processor running at 1 GHz with 512 MB RAM. The overhead was measured in CPU cycles using the RDTSC (read timestamp counter) instruction. Our results for the two scheduling policies are summarized below.

> NCLB-CBS: we observed that for up to 10 image streams (with the rates and skip factors mentioned earlier), and with the web server responding with 28 KB HTML pages at the rate of 1 page every 5 seconds, the scheduler overhead was never more than 2600 CPU cycles. When the number of image streams was only 4, the CPU overhead was never more than 1800 CPU cycles. The memory requirements were also nominal: the scheduler never required more than 1 KB memory.
>
> NCLB-CUS: for a similar workload as the earlier experiment and with 10 image streams, the scheduler overhead was never more than 2000 CPU cycles. When the number of image streams was only 4, the CPU overhead was never more than 1500 CPU cycles. The memory requirements were less than that imposed by the NCLB-CBS scheduler, as can be expected.

To understand the CPU cycles overhead, we measured the overhead of running the same set of tasks with the default EDF scheduler and found the overhead to be not more than 1200 CPU cycles. These results reflect that there is some penalty paid for the extra complexity, but this penalty is no more than a factor of 2.4 over the normal scheduling overhead. With increasing processor speeds, this is acceptable penalty. These results are, in fact, consistent with earlier measurements made by Brandt et al. [33].

We did not extensively evaluate the performance of the aperiodic workload because our goal was predominantly to understand if the scheduling overheads were reasonable. The few experiments performed to measure aperiodic response times were consistent with the results from simulations; we do not include them here for brevity. We have tested our implementation with low to moderate workload, and we find the performance satisfactory. It is possible that the overhead may dominate with many tasks, but we expect that our approach will produce shorter response times for aperiodic tasks when the periodic (or sporadic) task set is not large. (The BWP scheme that we have articulated provides completeness to our work on reclaiming spare capacity, but we found that realizing an actual implementation of such a scheduler may introduce excessive overhead.)

8. Discussion

In this article, we have presented two different approaches for improving the completion times of aperiodic tasks. The first approach, NCLB-CBS, uses CBS as the underlying scheduling mechanism and BASH to reclaim holes. The holes, of course, need to be located beforehand and, during runtime, get added to the BASH queue at appropriate instants. NCLB-CBS can handle both the RTO and the BWP models for firm real-time systems but is not capable of working in a sporadic environment.

To reclaim idle time arising because of sporadic tasks, we presented the NCLB-CUS algorithm. This variant of our resource reclamation methodology does not require any precomputation. It identifies intervals where the processor can idle (in the absence of aperiodic computation requirements) and prioritizes aperiodic requests during these durations. We suggest enforcing this prioritization within the BASH framework.

NCLB-CUS has wider applicability: it can be used for firm real-time task sets and sporadic task sets. In a firm real-time setting, NCLB-CUS can automatically deal with BWP scheduling, but it is not always the best scheme to adopt. Our evaluation demonstrates that NCLB-CBS, which locates holes offline and utilizes them at runtime, leads to better aperiodic response times in a firm real-time system. As a result, when the basis set of tasks to be scheduled is firm and periodic, NCLB-CBS is preferable and has a clean integration with BASH.

A parameter that affects both NCLB resource reclamation algorithms is the ratio between the largest period (or interarrival time) and the smallest period (or interarrival time). If T_1 is the smallest period and T_n is the largest period, the ratio $\alpha = T_n/T_1$ determines the effectiveness of the NCLB algorithms. When α is small (closer to 1), we can reclaim holes quickly and improve the response times of aperiodic jobs. As α increases, the task with period T_n has a distant deadline and will always be eligible to execute pushing the idle time closer to its deadline. As a result, aperiodic jobs may have to wait a long time before exploiting the holes. When α is large, we suggest period transformations (reducing the largest period in the system by dividing the task into portions) to create a new task set with a smaller α. For example, if $T_1 = 10$ and $T_n = 90$ and the execution time of the task with period $T_n = 90$ is 20 time units, an equivalent task set has $T_1 = 10$ and $T_n = 45$ and the task with period $T_n = 45$ has an execution time of 10 units. This changes α from 9 to 4.5. Period transformations do not affect schedulability,but they do create more opportunities for resource reclamation. In the extreme but impractical case—impractical because of timer granularity and context switch overheads—all tasks can be made to have periods of length 1 and any unused cycles can be reclaimed within every unit of activity.

9. Conclusion

In this paper, we presented the NCLB class of algorithms for reclaiming holes that are created when scheduling tasks that allow skips and when scheduling sporadic tasks. Skips and sporadic tasks create extra processor capacity (holes)—capacity that is hard to identify—that can be used to improve the response times of aperiodic tasks.

Our evaluations can be summarized as follows.

(i) Reclaiming spare time leads to significant reductions in aperiodic response time.

(ii) In a firm real-time environment, NCLB-CBS and NCLB-CUS can be used; NCLB-CBS provides shorter response times and is the better scheme for this situation.

(iii) When the basic set of tasks are sporadic, NCLB-CUS can be used; it is not possible to use NCLB-CBS because background time cannot be identified offline.

Identifying and reclaiming holes transforms background capacity into reserved capacity; this transformation results in improved behavior of constant bandwidth servers. In this work, we push the envelope on the applications of the BASH technique by utilizing it in a firm real time environment. By applying BASH and the constant utilization server in conjunction, we also open up a new domain for BASH: sporadic real-time systems.

We relax the need for offline computations of holes and improve the results from prior work [24]. By generalizing the resource reclamation techniques to sporadic and firm real-time tasks, we provide a comprehensive framework for managing aperiodic tasks in real-time systems. The work described in this article can be extended to support energy-efficient scheduling mechanisms that utilize slack to slow the processor down; this is a direction for future investigations.

References

[1] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 40–61, 1973.

[2] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[3] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–100, 1991.

[4] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[5] J. Lehoczky, L. Sha, and J. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261–270, 1987.

[6] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[7] J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 110–123, 1992.

[8] M. Spuri and G. Buttazzo, "Efficient aperiodic service under earliest deadline scheduling," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 2–11, 1994.

[9] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems*, vol. 9, no. 1, pp. 21–36, 1995.

[10] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.

[11] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1443–1451, 1995.

[12] G. Koren and D. Shasha, "Skip-over: algorithms and complexity for overloaded systems that allow skips," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 110–117, December 1995.

[13] G. Bernat and A. Burns, "Combining (n, m)-hard deadlines and dual priority scheduling," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 46–57, December 1997.

[14] J. W.-S. Liu, *Real-Time Systems*, Prentice-Hall, 2000.

[15] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "Deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[16] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS '00)*, pp. 295–304, 2000.

[17] M. Caccamo, G. C. Buttazzo, and D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Transactions on Computers*, vol. 54, no. 2, pp. 198–213, 2005.

[18] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," in *Proceedings of the Euromicro Conference on Real-Time Systems*, 2000.

[19] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "IRIS: a new reclaiming algorithm for server-based real-time systems," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*, pp. 211–218, May 2004.

[20] G. Fohler, T. Lennvall, and G. Buttazzo, "Improved handling of soft aperiodic tasks in offline schedule real-time systems using total bandwidth server," in *Proceedings of the IEEE Symposium on Emerging Technologies and Factory Automation (ETFA '01)*, vol. 1, pp. 151–157, 2001.

[21] D. Isovic and G. Fohler, "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS '00)*, pp. 207–216, 2000.

[22] A. Marchand and M. Silly-Chetto, "QoS and aperiodic task scheduling for real-time Linux applications," in *Proceedings of the 6th RTL Workshop*, 2004.

[23] G. C. Buttazzo and M. Caccamo, "Minimizing aperiodic response times in a firm real-time environment," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 22–32, 1999.

[24] D. C. Thomas, S. Gopalakrishnan, M. Caccamo, and C. G. Lee, "Spare CASH: reclaiming holes to minimize aperiodic response times in a firm real-time environment," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS '05)*, pp. 147–156, July 2005.

[25] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1261–1269, 1989.

[26] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack management," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 3–14, 2005.

[27] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301–324, 1990.

[28] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 4–13, December 1998.

[29] Z. Deng, J. W.-S. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," in *Proceedings of the Euromicro Workshop on Real-Time Systems*, pp. 191–199, 1997.

[30] A. Varga, "OMNeT++ 2.0: Discrete Event Simulation System," Department of Telcommunications (BME-HIT), Technical University of Budapest, Hungary, 2000, http://www.hit.bme.hu/phd/vargaa/omnetpp.htm.

[31] RTLinux. 2008, http://www.rtlinuxfree.com/.

[32] P. Mendoza and P. Balbastre, "Constant bandwidth server in RTLinux," 2003, http://www.ocera.org/download/components/WP5/rtlcbs-0.1-1.html.

[33] S. A. Brandt, S. Banachowski, C. Lin, and T. Bissom, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, pp. 396–407, December 2003.