Spare CASH: Reclaiming Holes to Minimize Aperiodic Response Times in a Firm Real-Time Environment*

Deepu C. Thomas† Sathish Gopalakrishnan‡ Marco Caccamo‡
Chang-Gun Lee§

Abstract

Scheduling periodic tasks that allow some instances to be skipped produces spare capacity in the schedule. Only a fraction of this spare capacity is uniformly distributed and can easily be reclaimed for servicing aperiodic requests. The remaining fraction of the spare capacity is non-uniformly distributed, and no existing technique has been able to reclaim it. We present a method for improving the response times of aperiodic tasks by identifying the non-uniform holes in the schedule and adding these holes as extra capacity to the capacity queue of the CASH mechanism. The non-uniform holes can account for a significant portion of spare capacity, and reclaiming this capacity results in considerable improvements to aperiodic response times.

1 Introduction

Real-time systems execute periodic and aperiodic tasks, and 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 assumed to be 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.

*This work is supported in part by the NSF grant CCR-0237884, and in part by NSF grant CCR-0325716.
†Microsoft Corporation, Redmond, WA 98052, USA; formerly with the University of Illinois at Urbana-Champaign
‡Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
§Department of Electrical and Computer Engineering, The Ohio State University, Columbus OH 43210, USA

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 or even skip some instances occasionally. All other instances should complete before their deadlines. When a job of a firm task is skipped, the processor gains some capacity for executing some other job.

Liu and Layland [14] 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 [18, 4, 3] and in the presence of aperiodic tasks [13, 19, 12, 20, 9, 21] are important generalizations that have been studied.

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 tasks; some are soft and others are firm 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 [17] proposed the (m, k)-model for representing a firm real-time task where at least m out of k consecutive jobs must meet their deadlines. They described a heuristic priority assignment scheme for such tasks but did not develop an exact schedulability analysis. Bernat and Burns [5] 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 [11] made important contributions when they proved that making optimal use of skips is NP-hard and described two (efficient, but non-optimal) *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. Liu et al. [15] introduced a novel QoS model for networked feedback control systems: the authors showed that their QoS constraint can directly be related to the control system's performance.

Buttazzo and Caccamo [6] 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, however, were unable to reclaim all the spare time and observed that a significant fraction of the spare time created by skipping jobs has a "granular" distribution across the schedule. They called these non-uniformly distributed capacities *holes*. Reclaiming those holes has been an open issue. Marchand and Silly-Chetto [16] developed two new algorithms, named EDL-RTO and EDL-BWP, which are able to exploit the skip model to enhance the response time of soft aperiodic requests. Since these algorithms are based on an optimal server like EDL, their runtime overhead increases on the order of $O(N^2)$ where N is the number of tasks in the system.

In this paper, we propose a novel technique for reclaiming all the spare time, including both uniformly distributed and non-uniformly distributed fractions, created when jobs are skipped in a firm real-time environment; jobs are prioritized using EDF. The non-uniformly distributed holes are identified offline, and they are utilized for servicing aperiodic jobs online by the Spare CASH mechanism that provides an aggressive resource reclamation technique, building upon the CASH mechanism [8]. Experimental results indicate that the reclamation of the non-uniformly distributed holes leads to significant improvements in the response time of aperiodic tasks.

2 Preliminaries

2.1 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, $s_i, 2 \leq s_i \leq \infty$. 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; 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 [11], every instance of a firm periodic task can either be *red* or *blue*. A red instance must be completed 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.

2.2 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 [14] showed that a periodic task set is schedulable by EDF if and only if its cumulative processor utilization is no greater than 1. That is,

$$U_p = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1. \tag{1}$$

Analyzing the feasibility of firm periodic tasks is not equally easy. Koren and Shasha [11] proved that determining whether a set of skippable periodic tasks is schedulable is NP-hard. They also found that, given a set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ of firm periodic tasks that allow skips, then

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

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.

Task	**Task1**	**Task2**	**Task3**
Computation	1	2	5
Period	3	4	12
Skip Parameter	4	3	∞
U_p	1.25		

Table 1: A schedulable set of firm periodic tasks.

Using the *processor demand* criterion, Jeffay and Stone [10] showed that a set of hard periodic tasks is schedulable by EDF if and only if, for any interval $L \geq 0$,

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

Based on this result, Koren and Shasha [11] proved the following theorem, which provides a sufficient condition for the schedulability of a set of skippable periodic tasks under EDF.

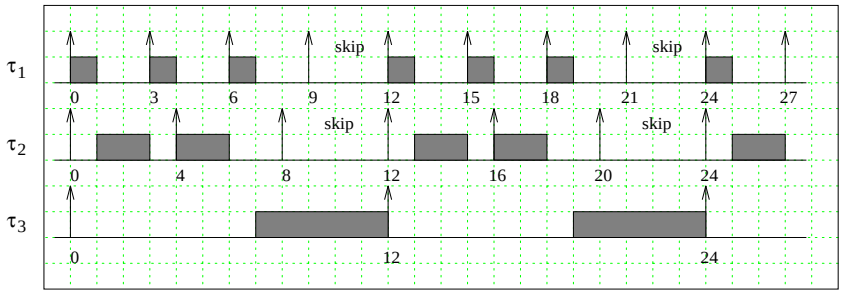


Figure 1: Feasible schedule for tasks in Table 1.

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

$$\forall L \geq 0 \qquad L \geq \sum_{i=1}^{n} D(i,[0,L]) \tag{4}$$

$$\textit{where } D(i,[0,L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i. \tag{5}$$

In their theorem, $D(i,[0,L])$ represents the effective time demanded by the periodic task set over the interval $[0,L]$. Koren and Shasha [11] also proposed two online scheduling

algorithms, *Red Tasks Only* and *Blue When Possible*, to handle tasks with skips under EDF.

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 is able to 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.*

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 using the deeply-red assumption.

Buttazzo and Caccamo [6] defined the equivalent processor utilization, U_p^*, for a set of firm periodic tasks to be

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_i D(i, [0, L])}{L} \right\}. \tag{6}$$

They then used the remaining (uniformly distributed) capacity, $1 - U_p^*$, to schedule aperiodic tasks. However, the equivalent processor utilization over-estimates the system utilization, and there is some processor capacity that is not reclaimed because it has a "granular" distribution [6].

The total spare capacity in the system can be calculated and it is given by

$$U_{spare} = 1 - U_{firm} = 1 - U_p + \sum_{i=1}^{n} \frac{c_i}{p_i s_i}. \tag{7}$$

This spare capacity can be categorized into two portions U_{sa} and U_{sh}. A portion of this capacity $U_{sa} = 1 - U_p^*$ is uniformly distributed and is assigned to the aperiodic server.

The remaining portion of the spare capacity is non-uniformly distributed among many *holes* [6], and can be calculated as $U_{sh} = U_{spare} - U_{sa}$.

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 this paper, we provide a mechanism for identifying the spare capacity that is irregularly spaced and for using this capacity to improve response times of aperiodic tasks. We concentrate on the RTO scheduling approach and defer work on BWP scheduling to a future paper.

Task	**Task1**	**Task2**
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	

Table 2: Illustrating the existence of holes.

2.3 The CASH 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. In order to minimize aperiodic response times, aperiodic tasks are handled by the Spare CASH mechanism. Spare CASH builds upon the CASH algorithm [8]; non-uniformly distributed spare capacities (holes) for a given firm periodic task set are calculated offline and placed in the global capacity queue of the CASH server. Before proceeding further, we provide an outline of the CASH mechanism.

The capacity sharing mechanism (CASH) works in conjunction with the Constant Bandwidth Server (CBS) [2]. CBS provides isolation between tasks in a system; each task is allocated a bandwidth and a server to ensure that it does not use more than the

allotted bandwidth. CASH was proposed as an approach to handling overruns in systems executing periodic tasks while preserving isolation. The primary motivation for capacity sharing was the observation that only a few instances of a task execute for the worst-case duration and reserving resources using the worst-case consumption is expensive. CASH 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. CASH was proposed for periodic task sets with hard deadlines; if U_p is the processor utilization, the unused bandwidth, $1 - U_p$, can be assigned to an aperiodic task server. A global capacity queue, or a CASH queue, is used to keep track of the available excess capacity.

The CASH algorithm is specified by the following rules:

1. Each CBS server S_i is characterized by the current remaining 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. The ratio $U_i = Q_i/T_i$ is the server bandwidth. At each instant, a fixed deadline $d_{i,k}$ is associated with the server. At the beginning $d_{i,0} = 0, \forall i$.

2. Each task instance, $\tau_{i,j}$ with release time $r_{i,j}$, handled by server S_i is assigned a dynamic deadline equal to the current server deadline $d_{i,k}$.

3. 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.

4. 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.

5. When a task instance, $\tau_{i,j}$, arrives and the server is active the request is queued with other pending jobs according to a given (arbitrary) discipline.

6. Whenever instance $\tau_{i,j}$ is scheduled for execution, the server S_i uses the capacity c_q in the CASH queue (if there is one) with the earliest deadline d_q, such that $d_q \leq d_{i,k}$, otherwise its own capacity c_i is used.

7. Whenever job $\tau_{i,j}$ executes for δ time units, the used budget c_q or c_i is decreased by δ. When c_q becomes zero, it is deleted from the CASH queue and the next capacity in the queue with deadline less than or equal to $d_{i,k}$ can be used.

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

9. 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 capacity $c_i > 0$ (if any) is inserted in the CASH queue with deadline equal to the server deadline, and c_i is set equal to zero.

10. Whenever the processor becomes idle for an interval of time Δ, the capacity c_q (if it exists) with the earliest deadline in the CASH queue is decreased by Δ until the CASH queue becomes empty.

CASH was originally developed for hard real-time task sets; our new work pushes the envelope further by dealing with firm real-time tasks. The holes that occur in a schedule are identified and added (at the appropriate time) to the CASH queue and can be utilized by all tasks, especially aperiodic tasks.

3 Spare CASH

In this section, we formally describe the Spare CASH technique assuming that each task, τ_i, is handled by a dedicated CBS server, S_i, running on a uniprocessor system. Spare capacities for a given task set are identified offline and added to the global capacity queue online. Holes are identified over the meta-hyperperiod for the given task set.

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

As an example, the meta hyper-period of the task set in Table 2 is 30.

3.1 An Algorithm to Locate Holes

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

$$A[t_1, t_2] = \int_{t_1}^{t_2} f(t)dt \text{ where}$$

$$f(t) = \begin{cases} 1 & \text{processor is busy at } t \\ 0 & \text{otherwise} \end{cases}.$$

Algorithm 1 LOCATE HOLES AND DETERMINE CAPACITIES

Require: A set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ of n firm periodic tasks with an *equivalent processor utilization* U_p^*.
  $k \Leftarrow 0$
  $d_{-1} \Leftarrow 0$
  $H \Leftarrow lcm(c_1 \times s_2, c_2 \times s_2, \ldots, c_n \times s_n)$ {meta hyper-period}
  for all tasks T_i **do**
    $c_i^* = c_i / U_p^*$; inflate c_i to c_i^* to include uniformly distributed portion of spare capacity
  end for
  Schedule the task set $\Gamma^* = \{T_i(p_i, c_i^*, s_i)\}$ using the EDF-RTO scheduler
  for all time t where (t is a task skip-deadline) and ($t \leq H$) **do**
    $E_k \Leftarrow (t - A[0,t]) \times U_p^* - \sum_{j=0}^{k-1} E_j$; amount of a hole
    $r_k \Leftarrow d_{k-1}$; hole release time
    $d_k \Leftarrow t$; hole deadline
    Add hole (E_k, r_k, d_k) to the hole capacity list
    $k \Leftarrow k + 1$
  end for

Remark. 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 T_i, when a task set is schedulable, we must have the activity duration over any time interval greater than the effective time demand over that interval. In effect, we can restate Theorem 1 as: 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 1 to locate holes in the schedule first inflates the utilization of the task set by the factor $1/U_p^*$. Note that a fraction, $U_{sa} = 1 - U_p^*$, of the processor capacity is uniformly distributed and can be reclaimed simply by using an aperiodic task server of bandwidth U_{sa}. Thus, inflating the execution times accounts for the known uniformly distributed spare capacity $U_{sa} = 1 - U_p^*$. The task schedule after this inflation gives us only the non-uniformly distributed spare capacities, i.e., holes, which are identified in the second for loop. 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. The algorithm has a complexity of $O(Hn)$ where H is the meta hyper-period and n represents the number of tasks.

In the next section, we will first describe how the identified holes can be reused in the RTO model. 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.

3.2 Scheduling with the RTO model

In the RTO model, all the blue instances are rejected. Since all blue instances are skipped uniformly the task schedule repeats every meta hyper-period. The extra capacities are calculated *offline* according to Algorithm 1.

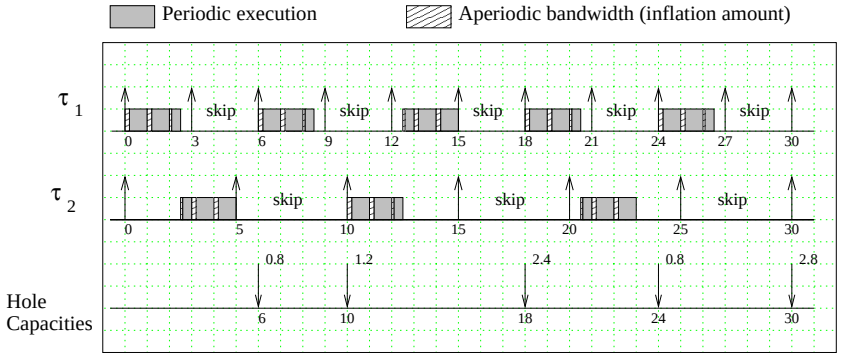


Figure 2: Task set (Table 2) scheduled using the RTO model with inflated computation times.

Figure 2 shows the hole capacities for the task set in Table 2. Holes are identified at every skip deadline. The hole capacity at $t = 6$ is calculated as $E_0 = (L - A[0, L]) \times U_p^* = (6 - 5) \times 0.8 = 0.8$ since $A[0, 6] = 5$ and $U_p^* = 0.8$. Similarly, the hole capacity at $t = 10$ is $E_1 = (10 - 7.5) \times 0.8 - 0.8 = 1.2$. The hole capacity E_0 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. The hole capacities for the entire meta hyper-period are calculated offline. They are released online according to Algorithm 2.

It is important to note 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 far better than background execution. The keystone for this work on exploiting holes is to transform background time into reserved bandwidth by reclaiming resources. In fact, each CBS server is able to reclaim bandwidth by consuming spare capacity while preserving its own budget. A formal discussion of this

Algorithm 2 HOLE CAPACITY RELEASE

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

intuition follows.

3.3 Theorems and Proofs

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

Proof. We need to prove that

$$\begin{aligned} \forall L \geq 0 : L &\geq \sum_{i=1}^{n} D(i, [0, L]) \\ where \quad D(i, [0, L]) &= \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times c_i^*. \end{aligned}$$

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

$$\forall L \geq 0 : L \geq \left(\sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) * \frac{c_i}{U_p^*} \right).$$

By the definition of U_p^* (Equation (6)), we have

$$\begin{aligned} U_p^* &\geq \left(\sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times \frac{c_i}{L} \right) \Rightarrow \\ L &\geq \left(\sum_{i=1}^{n} \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) \times \frac{c_i}{U_p^*} \right). \quad \square \end{aligned}$$

Theorem 3 *Algorithm 1 preserves the aperiodic bandwidth, i.e.,* $U_{sa} = 1 - U_p^*$ *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.

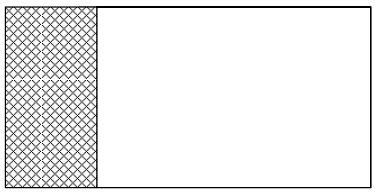


Figure 3: A single unit of time.

Hence for every $\Delta = t_2 - t_1$, $U_{sa} \times \Delta$ is available as aperiodic bandwidth. This however increases the computation for task T_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 hole capacity is calculated at every skip deadline, only a fraction, U_p^*, of it is identified. The remaining spare capacity is the aperiodic bandwidth $U_{sa} = 1 - U_p^*$. 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 Case 1 and Case 2. Since the theorem holds for Case 1 and Case 2, it holds for this case. □

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

Proof. We need to prove

$$\forall L :\geq \sum_{i=1}^{n} D(i, [0, L]) + (1 - U_p^*)L + \sum_{k, d_k \leq L} E_k \tag{9}$$

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

By taking the $(1-U_p^*)L$ term to the left-hand side and in (9) and then dividing throughout by U_p^*, we need to show

$$L \geq \frac{\sum_{i=1}^{n} D(i, [0, L])}{U_p^*} + \frac{\sum_{k, d_k \leq L} E_k}{U_p^*}. \tag{10}$$

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

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

Using the above inequality in (10), to prove the theorem, we need to show that $(L - A[0,L]) \times U_p^* \geq \sum_{k,d_k \leq L} E_k$. This, however, follows directly from Algorithm 1 when L is a skip deadline because $E_{k,d_k=L} = (L - A[0,L]) \times U_p^* - \sum_{j,d_j<L} E_j$. 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

$$(L' - A[0,L']) \times U_p^* = \sum_{k,d_k \leq L'} E_k. \tag{11}$$

Rewriting $L - A[0,L]$ as $(L - L') - A[L',L] + L' - A[0,L']$, we need to show that $((L - L') - A[L',L] + L' - A[0,L']) \times U_p^* \geq \sum_{k,d_k<L} E_k$. Using (11), 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. □

3.4 Scheduling with the BWP model

Having discussed scheduling of firm periodic tasks and aperiodic tasks under the RTO model, we turn our attention to the BWP model. Before we can do this, we need to introduce the notion of *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 is created. For 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.

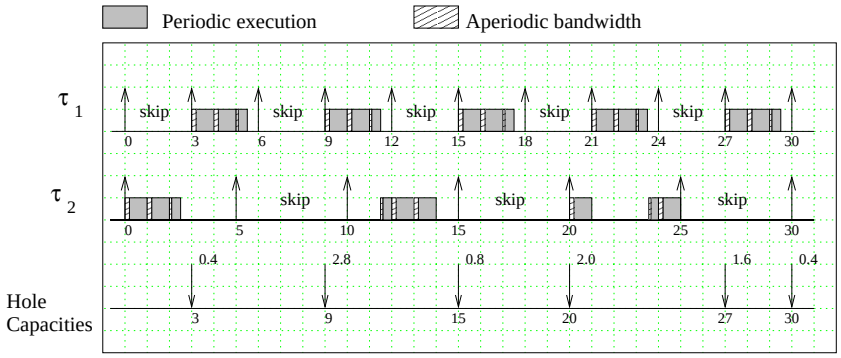


Figure 4: A combination of task patterns for the task set shown in Table 2.

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 also blue; which 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 performed online unlike in Section 3.2 leading to an overhead $O(Hn)$. We propose a scheme that has lower computational overhead but requires extra storage.

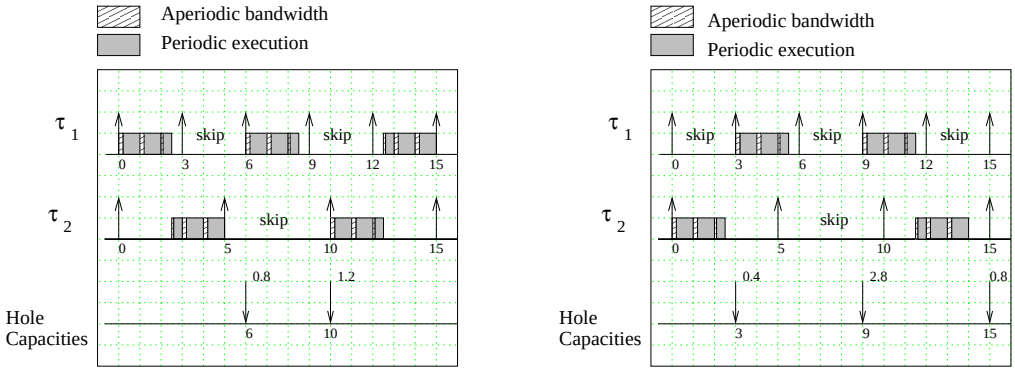
Given a set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ of n periodic tasks that allow skips, the distribution of the hole capacity is calculated for all $\Psi = s_1 \times s_2 \times \cdots \times s_n$ combinations. Each pattern combination results in a unique *hole capacity distribution* which is stored in a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

When a blue instance completes successfully, the task pattern change is detected and

- The current hole capacity (from the old pattern combination) present in the global capacity queue is deleted.
- Hole capacities computed offline for the new pattern combination are released starting from the nearest skip deadline of the new pattern combination.

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

An example



(a) Pattern combination 1 (b) Pattern combination 2

Figure 5: Task pattern combinations for task set in Table 2

Figure 5 shows the hole capacities for two pattern combinations for the task set in Table 2. Notice that the total time duration of each pattern combination is equal to the hyper-period of the task set in Table 2, namely $H = 15$.

Let us now consider a schedule in which a blue instance of a task is able to complete successfully. 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 combination shown in Figure 5(a) to the new pattern combination illustrated in Figure 5(b) at the nearest skip deadline of Pattern combination 2, time $t = 18$.

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 CASH apply. This results in the hole capacity placed at time $t = 18$ decreasing from 2.4 to $2.4 - c_2 = 0.4$, and being deleted when the pattern switches. The schedule continues to release hole capacity from the nearest skip deadline, $t = 18$, of Pattern combination 2.

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 hole capacities across all Ψ task patterns by Theorem 4.

Let T^c be the time at which a blue instance completes and T^s be the time at which the

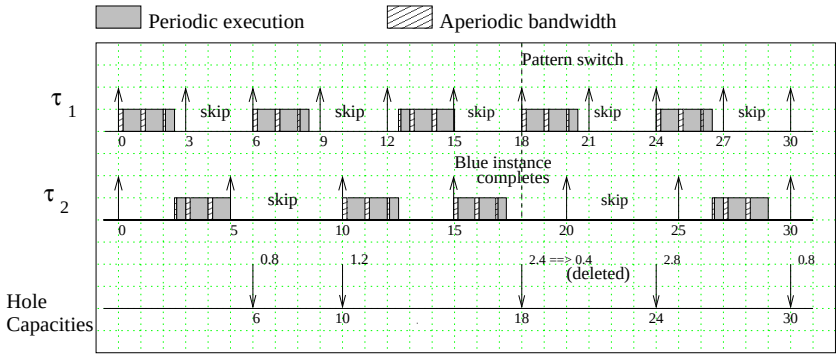


Figure 6: Schedule produced by BWP for the task set shown in Table 2.

task pattern switch occurs.

We consider two cases:

- Case A: $\forall t, \quad T^c \leq t \leq T^s$: 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 $t \leq T^c$ do not impact the schedule beyond T^c. Since the active hole capacity is deleted at time instant T^c, there exists no hole capacity for the time interval $T^c \leq t \leq T^s$. Since the BWP algorithm can find a schedule when test condition (2) is satisfied [11], the task set remains schedulable in the range $[T^c, T^s]$.

- Case B: $\forall t, \quad \geq T^s$: This time interval belongs exclusively to the new task pattern. This case 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 sub-optimal results.

4 Experimental Results

Spare CASH has been simulated using RTSIM [1] to evaluate the performance of the proposed technique. In this section, we present the results of our experiments. To evaluate the performance improvement, the Spare CASH algorithm is compared against the CASH algorithm that utilizes only uniformly distributed portion of space capacity. Periodic tasks are handled using the RTO scheduling policy.

The six experiments described in this section can be grouped into two sets. The first set shows the performance of the algorithms as a function of the aperiodic load, for three different values of hole capacity (U_{sh}). The second set of experiments tests the sensitivity of the algorithms to the average computation time of aperiodic requests.

The performance of the algorithms were measured by computing the average aperiodic response time as a function of the mean aperiodic load. Each aperiodic response time has been normalized with respect to the average aperiodic computation 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.

The results have been averaged over 20 runs, each of duration 1,000,000 time units. The 98% confidence interval is tight (but not plotted) and demonstrates the accuracy of the simulations. Execution times of aperiodic requests were chosen from a uniform distribution over a predefined interval, whereas their inter-arrival 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}. The objective of the experiments is to measure the improvement in the response time of aperiodic tasks when using Spare CASH. We use the CASH mechanism to queue holes that result from skips and our intention is not to model early completions that motivated the initial development of CASH [8].

4.1 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 experiment. The equivalent processor utilization, U_p^*, is kept constant at 0.90 for all three experiments and thus the aperiodic server has a fixed bandwidth $U_{sa} = 1 - U_p^* = 0.10$.

Figure 7 shows the results of the first experiment, with $U_{sh} = 0.12$, in which very few periodic instances are skipped and includes a periodic hard task. U_{spare} represents the total spare capacity, which is $U_{sa} + U_{sh}$. As the reader can see, the Spare CASH algorithm

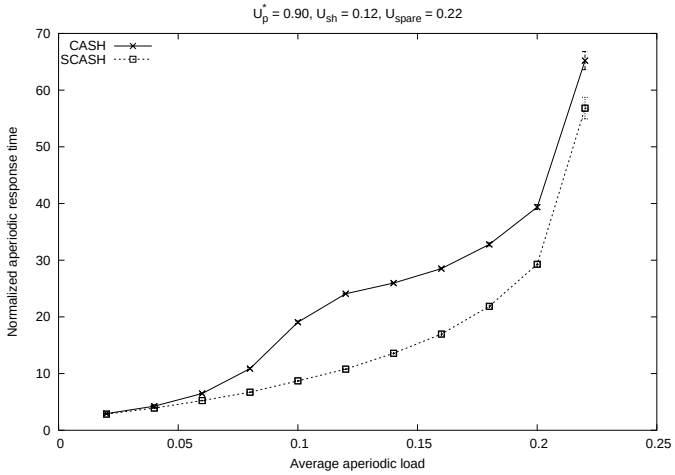


Figure 7: Performance results of simulation 1.

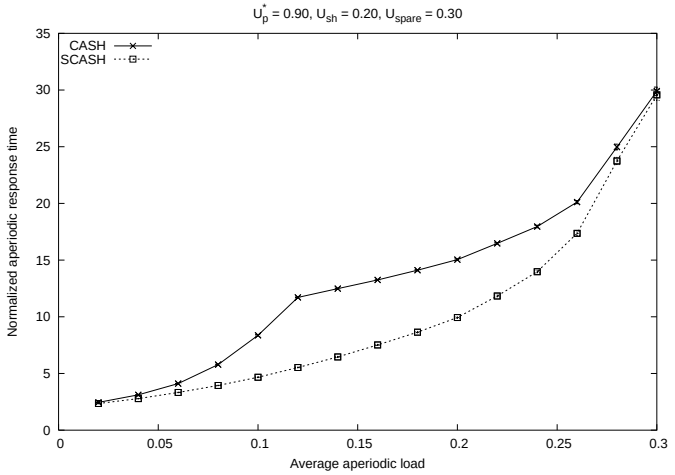


Figure 8: Performance results of simulation 2.

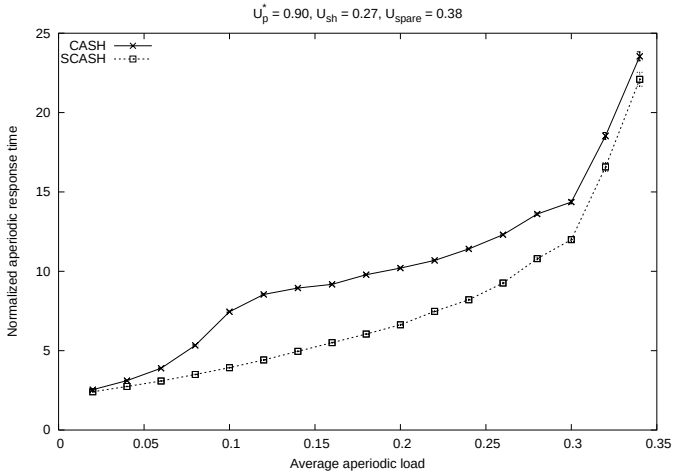


Figure 9: Performance results of simulation 3.

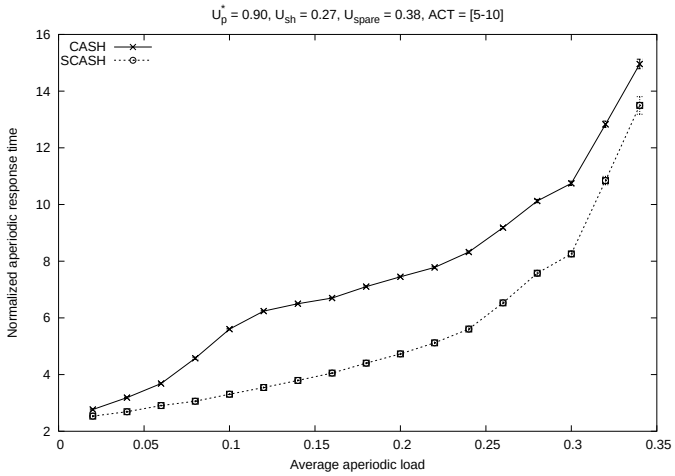


Figure 10: Performance results of simulation 4.

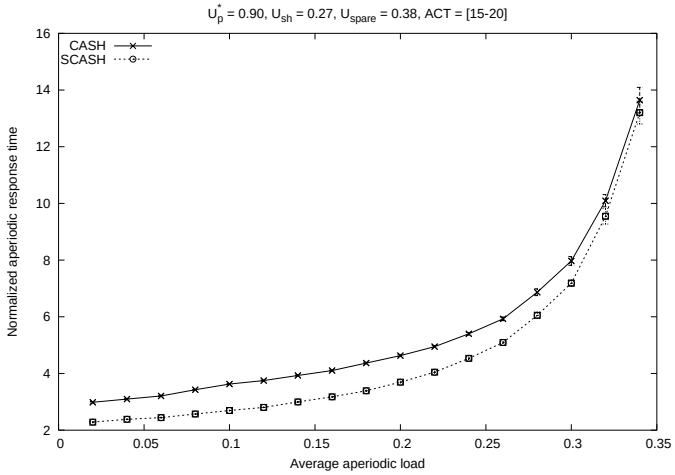


Figure 11: Performance results of simulation 5.

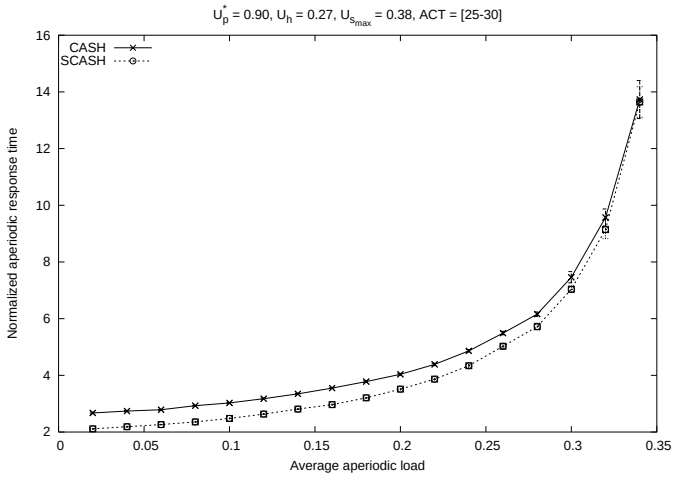


Figure 12: Performance results of simulation 6.

Simulation #	Task	Task1	Task2	Task3	Task4	Task5
I	Computation	8	5	35	15	20
	Period	90	100	150	60	60
	Skip Parameter	5	3	∞	5	5
II	Computation	12	5	50	20	25
	Period	90	100	150	60	60
	Skip Parameter	2	3	3	2	2
III	Computation	10	2	46	18	26
	Period	90	100	150	55	55
	Skip Parameter	2	2	2	2	2

Table 3: Parameters: first set of simulations.

outperforms CASH for values of ρ_a in the range [0.08,0.22]. This range is approximately equal to U_{sh}. For values of ρ_a outside this range the aperiodic response behavior for both the algorithms are similar. The aperiodic response time under the CASH 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.

Figure 8 refers to the second experiment, in which $U_{sh} = 0.20$. More periodic instances are skipped which results in a lower aperiodic response when compared to the first experiment, as aperiodic load remains identical. Spare CASH improves aperiodic response time for values of ρ_a in the range [0.08,0.28]. This range is higher than the first experiment since $U_{sh} = 0.20 > 0.12$. 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, $U_{sh} = 0.27$, the highest value in all the experiments. The improvement in aperiodic response time occurs over a larger range [0.08-0.33]; thus the Spare CASH algorithm performs best for higher values of hole capacities.

The Spare CASH algorithm works by locating hole capacities and placing them in the global capacity queue. Thus the aperiodic server can prevent unnecessary deadline postponements while executing in the hole region, enabling a better aperiodic response time.

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

1. $\rho_a \leq U_{sa}$: In this zone, aperiodic response of CASH and Spare CASH are identical.

If aperiodic load is less than U_{sa}, CASH can be as competitive as Spare CASH is scheduling aperiodic tasks. The hole capacity, U_{sh}, is not utilized much in this traffic zone.

2. $U_{sa} < \rho_a \leq U_{spare}$: Here Spare CASH outperforms CASH. The workload is consistently greater than U_{sa} and therefore the holes are necessary and Spare CASH is able to serve aperiodic tasks better.

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

Moreover, we observe that when the number of skips increases, the gap between CASH and Spare CASH decreases. This might seem to be counter-intuitive but the reason is straightforward: when more jobs are skipped, U_{sa} also increases and reclamation of holes is overshadowed by the increase in U_{sa}. Thus the gap between CASH and Spare CASH 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 a simple CASH server.

4.2 Analysis of Sensitivity to Aperiodic Computation Times

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 $[5, 10]$ for Simulation 4, $[15, 20]$ for Simulation 5, and $[25, 30]$ for Simulation 6. To limit the total number of graphs, the periodic tasks used were only those used in Simulation 3. The results of these experiments are shown in Figures 10, 11, and 12 respectively.

The improvement in performance achieved by Spare CASH over CASH is more significant when aperiodic requests have short computation times. As the ACTs become longer, the performance of Spare CASH tends to be similar to the one achieved by CASH. This is because, for long aperiodic tasks, advancing the position of small slack intervals in the

schedule does not create a great impact on the response times.

5 Conclusion

In this paper, we presented an algorithm for reclaiming holes that are created when scheduling tasks that allow skips. The holes are identified offline, and are introduced online as capacities in the CASH [8] queue. These holes are then utilized for minimizing the response times of aperiodic tasks.

To the best of our knowledge, this is the first work that describes a technique for reclaiming holes in a firm periodic real time environment. 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 CASH technique by utilizing it in a firm real time environment.

We have considered the RTO (Red Tasks Only) and the BWP (Blue When Possible) strategy for scheduling periodic tasks. Extensive experimentation with the RTO strategy reveals that our approach can significantly improve response times for aperiodic tasks.

In our future investigations, we will seek for algorithms that are more efficient, and continue experiments with the BWP strategy. We would also like to improve the idle time handling of the CASH algorithm to obtain even better response times. So far, we have dealt with only the *deeply-red* task model and we need to generalize our approach to systems with arbitrary offsets because our approach requires us to know how holes are distributed in a schedule and this distribution will change when tasks have different offsets. Finally, we note that we can improve response times by advancing hole deadlines (using techniques similar to TB* [7]) and plan to study this extension and present our results in a subsequent publication.

References

[1] Real-time system simulator (RTSIM). http://www.rtsim.sssup.it.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1998.

[3] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 5(8):284–292, September 1993.

[4] T.P. Baker. Stack-based scheduling of real-time processes. *The Journal of Real-Time Systems*, 1(3):67–100, 1991.

[5] G. Bernat and A. Burns. Combining (n, m)-hard deadlines and dual priority scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 46–57, December 1997.

[6] G. Buttazzo and M. Caccamo. Minimizing aperiodic response times in a firm real-time environment. *IEEE Transactions on Software Engineering*, 25(1):22–32, January/February 1999.

[7] G.C. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Transactions on Computers*, 48(10):1035–1052, October 1999.

[8] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2000.

[9] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, pages 21–36, 1995.

[10] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 212–221, December 1993.

[11] G. Koren and D. Sasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 110–117, December 1995.

[12] J.P. Lehoczky and S.R. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 110–123, December 1992.

[13] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, December 1987.

[14] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 1(20):40–61, 1973.

[15] D. Liu, X.S. Hu, M.D. Lemmon, and Q. Ling. Firm real-time system scheduling based on a novel QoS constraint. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2003.

[16] A. Marchand and M. Silly-Chetto. QoS and aperiodic tasks scheduling for real-time linux applications. In *Proceedings of the 6th RTL Workshop*, November 2004.

[17] M.Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 12(44):1443–1451, December 1995.

[18] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 9(39):1175–1185, September 1990.

[19] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic scheduling for hard real-time system. *Journal of Real-Time Systems*, (1):27–60, 1989.

[20] M. Spuri and G.C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time System Symposium*, pages 2–11, December 1994.

[21] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 2(10):179–210, 1996.