Efficient Reclaiming in Reservation-Based Real-Time Systems with Variable Execution Times

Marco Caccamo, *Member*, *IEEE*, Giorgio C. Buttazzo, *Member*, *IEEE*, and Deepu C. Thomas, *Member*, *IEEE*

Abstract—In this paper, we present a general CPU scheduling methodology for managing overruns in a real-time environment, where tasks may have different criticality, flexible timing constraints, shared resources, and variable execution times. The proposed method enhances the Constant Bandwidth Server (CBS) by providing two important extensions. First, it includes an efficient bandwidth sharing mechanism that reclaims the unused bandwidth to enhance task responsiveness. It is proven that the reclaiming mechanism does not violate the isolation property of the CBS and can be safely adopted to achieve temporal protection even when resource reservations are not precisely assigned. Second, the proposed method allows the CBS to work in the presence of shared resources. The enhancements achieved by the proposed approach turned out to be very effective with respect to classical CPU reservation schemes. The algorithm complexity is $\mathcal{O}(lnN)$, where N is the number of real-time tasks in the system, and its performance has been experimentally evaluated by extensive simulations.

Index Terms—Overrun management, overload control, resource reclaiming, variable execution times.

1 Introduction

In most real-time systems, predictability is achieved by enforcing timing constraints on application tasks whose feasibility is guaranteed offline by means of proper schedulability tests based on worst-case execution time (WCET) estimations. Theoretically, such an approach works fine if all the tasks have a regular behavior and all WCETs are precisely estimated. In practical cases, however, a precise estimation of WCETs is very difficult to achieve because several low-level mechanisms present in modern computer architectures (such as interrupts, DMA, pipelining, caching, and prefetching) introduce a form of nondeterministic behavior in tasks' execution whose duration cannot be predicted in advance.

Even though a precise WCET estimation could be derived for each task, a worst-case feasibility analysis would be very inefficient when task execution times have a high variance. In this case, a classical offline hard guarantee would waste the system's computational resources for preserving the task set feasibility under sporadic peak load situations, even though the average workload is much lower. Such a waste of resources (which increases the overall system's cost) can be justified for very critical applications (e.g., military defense systems or safety critical space missions) in which a single deadline miss may cause catastrophic consequences. However, it does not represent a good solution for those applications (the majority) in which several deadline misses can be tolerated by the system as long as the average task rates are guaranteed offline.

There are many soft real-time applications in which the worst-case duration of some tasks is rare but much longer than the average case. In multimedia systems, for instance, the time for decoding a video frame in MPEG players can vary significantly as a function of the data contained in the previous frames. As another example, consider a visual tracking system where, in order to increase responsiveness, the moving target is searched in a small window centered in a predicted position, rather than in the entire visual field. If the target is not found in the predicted area, the search has to be performed in a larger region until, eventually, the entire visual field is scanned in the worst-case. If the system is well-designed, the target is found very quickly in the predicted area most of the times. Thus, the worst-case situation is very rare, but very expensive in terms of computational resources (computation time increases quadratically as a function of the number of trials). In this case, an offline guarantee based on WCETs would drastically reduce the frequency of the tracking task, causing a severe performance degradation with respect to a soft guarantee based on the average execution time. On the other hand, uncontrolled overruns[1] are very dangerous if not properly handled since they may heavily interfere with the execution of other tasks which could be more critical. Consider, for example, the task set given in Table 1, where two tasks, τ_1 and τ_2, have a constant execution time, whereas τ_3 has an average computation time ($C_3^{avg} = 3$) much lower than its

- M. Caccamo is with the Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin, Urbana, IL 61801. E-mail: mcaccamo@uiuc.edu.
- G.C. Buttazzo is with the Department of Computer Science, University of Pavia, Via Ferrata, 1, 27100 Pavia, Italy. E-mail: giorgio@sssup.it.
- D.C. Thomas is with Microsoft Corp., One Microsoft Way, Redmond, WA 98052. E-mail: dethoma@microsoft.com.

Manuscript received 8 Apr. 2004; revised 2 Sept. 2004; accepted 21 Sept. 2004; published online 15 Dec. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0117-0404.

1. A task is said to overrun when it executes for more than its guaranteed execution time (see Section 2 for its formal definition).

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

TABLE 1
Task Set Parameters

Task	C_i^{avg}	$WCET_i$	T_i
τ_1	1	1	6
τ_2	5	5	10
τ_3	3	10	12

worst-case value ($WCET_3 = 10$). Here, if the schedulability analysis is performed using the average computation time C_3^{avg}, the total processor utlization becomes 0.92, meaning that the system is not overloaded; however, under the Earliest Deadline First (EDF) algorithm [13], the tasks can experience long delays during overruns. In particular, as illustrated in Fig. 1, an overrun of task τ_3 affects the behavior of the other tasks, causing interference in their execution and possibly missing their deadline: The example resulted in the deadline miss of τ_2, whose execution time was correctly predicted. By adding temporal protection, such an interference is avoided so that only the misbehaving task (if any) is delayed. Similar examples can be easily found also under fixed priority assignments (e.g., under the Rate Monotonic algorithm [13]) when overruns occur in the high priority tasks.

To prevent overrun from introducing unbounded delays on tasks' execution, the system could either decide to abort the current instance of the task experiencing the overrun or let it continue with a lower priority. The first solution is not safe because the instance could be in a critical section when aborted, thus leaving a shared resource with inconsistent data (very dangerous). The second solution is much more flexible since the degree of interference caused by the overrun on the other tasks can be tuned acting on the priority assigned to the "faulty" task for executing the remaining computation.

A general technique for limiting the effects of overruns is based on a resource reservation approach [14], [21], [1], according to which each task is assigned (offline) a fraction of the available resources and is handled by a dedicated server, which prevents the served task from demanding more than the reserved amount. An efficient method for achieving resource reservation and temporal protection under EDF is the Constant Bandwidth Server (CBS) [1], [2], whose behavior and main properties are briefly recalled in Appendix A.1. Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent on a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources. To overcome this problem, we propose a general scheduling methodology for managing overruns in a controlled fashion. In particular, the proposed technique extends the CBS, providing the following enhancements:

- It performs efficient reclaiming of the unused computation times through a global bandwidth sharing mechanism that allows exploiting early completions to relax the bandwidth constraints enforced by isolation. The reclaiming mechanism preserves the isolation property of the CBS and can be safely adopted to achieve temporal protection even when resource reservations are not precisely assigned.
- It handles tasks with different criticality and flexible timing constraints to enhance the performance of those real-time applications which allow a certain degree of flexibility.
- It provides resource sharing among tasks with different criticality without compromising the real-time guarantee of hard tasks.

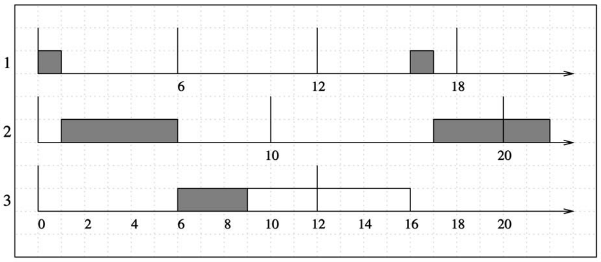


Fig. 1. Negative effects of uncontrolled overruns.

Although the idea of resource reclaiming and bandwidth sharing is not new in the literature, as discussed in Section 6 on related work, the peculiarity of our method is to increase resource utilization while preserving isolation so that not only soft tasks, but also hard real-time tasks can benefit from our approach. A preliminary version of this work has been published in [6], [7]; however, while [6] addresses the resource reclaiming problem for capacity-based aperiodic servers without considering resource sharing and [7] focuses on the resource sharing problem among hard and soft tasks without considering any reclaiming mechanism, this work integrates resource reclaiming and resource sharing in a complete framework. Moreover, a new and improved reclaiming technique is introduced which outperforms the one in [6]; finally, extensive experiments carried out by a real-time scheduling simulator validated all the results predicted by the theory. It is worth noting that, comparing our technique with other resource reclaiming mechanisms which preserve isolation, our method handles resource constraints among soft and hard tasks, preserving the real-time guarantee of hard periodic tasks without the additional cost of reserving extra budget to tasks. Moreover, unlike other similar approaches, our method was not developed for enhancing aperiodic responsiveness of soft tasks, but to efficiently handle overruns in real-time (hard and soft) tasks, where some form of relaxed guarantee is required offline.

The rest of the paper is organized as follows: Section 2 introduces some terminology and assumptions used throughout the paper; Section 3 illustrates the bandwidth sharing algorithm; Section 4 extends the bandwidth sharing algorithm to work in the presence of resource constraints; Section 5 illustrates some experimental results; Section 6 presents the related work; and Section 7 contains our conclusions and future work.

2 Terminology and Assumptions

Throughout the paper, each hard periodic task τ_i is considered as a stream of jobs (or task instances) $\tau_{i,j}$ ($j = 1, 2, \dots$), each characterized by a request time $r_{i,j}$ and a deadline $d_{i,j}$. In the following, P_i denotes the desired activation period of the task, $WCET_i$ its maximum computation time, and C_i^{avg} its average computation time. For the sake of completeness, before formally introducing the notion of *overrun*, the concepts of *computational demand* and *bandwidth utilization* are briefly recalled. The *computational demand* $g_i(t_1, t_2)$ of task τ_i is defined as the total computation time requested by those jobs $\tau_{i,j}$ whose arrival times and deadlines are within $[t_1, t_2]$ (that is, $t_1 \leq r_{i,j} \leq d_{i,j} \leq t_2$). A task τ_i is said to have a *bandwidth utilization* U_i if, in any interval of time $[t_1, t_2]$, its computational demand $g_i(t_1, t_2)$ never exceeds $(t_2 - t_1)U_i$ and there exists an interval $[t_a, t_b]$ such that $g_i(t_a, t_b) = (t_b - t_a)U_i$. A task τ_i is said to *overrun* when there exists an interval of time $[t_1, t_2]$ in which its computational demand g_i exceeds its expected bandwidth U_i multiplied by the length of the interval. This condition may occur either because jobs arrive more frequently than expected (*activation overrun*) or computation times exceed their expected value (*execution overrun*).

In the proposed approach, each task is handled by a dedicated *Constant Bandwidth Server* (CBS) [1], which provides isolation among tasks. In addition, a bandwidth sharing mechanism allows tasks to reclaim the unused computations due to early completions. Notice that 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. Due to the isolation mechanism introduced by the multiple server approach, there are no particular restrictions on the task model that can be handled by the proposed method. Hence, tasks can be hard, soft, periodic, or aperiodic. Notice that, while each job of a hard task is characterized by its deadline $d_{i,j}$, soft tasks do not have a deadline to honor, but their late completion gracefully degrades the performance of the system without causing any damage. Since each task (hard, soft, periodic, or aperiodic) is scheduled by a dedicated server, the task has properly assigned a dynamic *server deadline* used by EDF to correctly schedule the task set. To ensure the real-time behavior of the system, the parameters of a server associated to a hard task are assigned in such a way that the server deadline is coincident with the task deadline: Hence, by meeting server deadlines, it is ensured no hard task misses its own deadline. On the other hand, a soft task simply inherits its server deadline to enhance its responsiveness. Resource constraints can also be taken into account, as explained in Section 4, using a concurrency control protocol for mutually exclusive resources. Although the method is built upon the CBS, it can easily be generalized to be used with any capacity-based server. CBS and its main properties are briefly recalled in Appendix A.1.

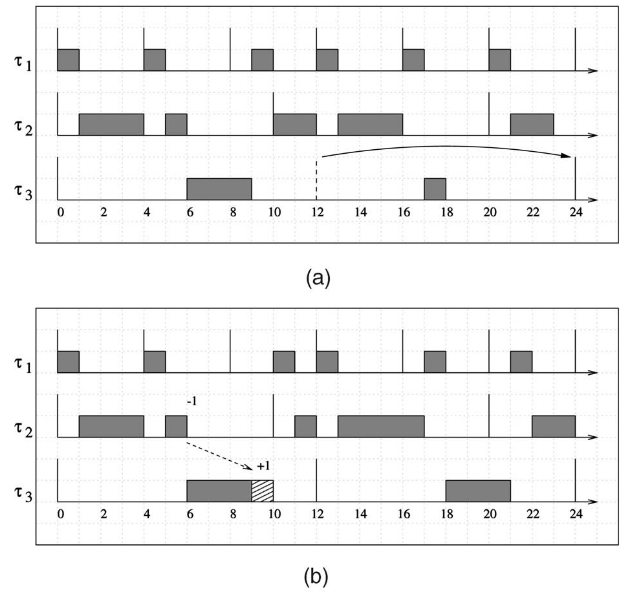


Fig. 2. (a) Overruns handled by a plain CBS versus (b) overruns handled by a CBS with the BASH reclaiming mechanism.

3 The Bandwidth Sharing Approach

The bandwidth sharing (BASH) mechanism proposed in this paper works in conjunction with the CBS. To illustrate the idea behind our approach, we present an example to show the potential improvements that can be achieved by a proper exploitation of the unused computation times coming from early completions. Ideally, we would like to reserve a given bandwidth to each task to achieve isolation, but we would also like to reclaim the unused time left by the other tasks as much as possible, thus a task a chance to handle its overruns without introducing long delays.

Consider the example shown in Fig. 2, where three tasks are handled by three servers with budgets $Q_1 = 1$, $Q_2 = 5$, $Q_3 = 3$, and periods $T_1 = 4$, $T_2 = 10$, $T_3 = 12$, respectively. At time $t = 6$, job $\tau_{2,1}$ completes earlier with respect to the allocated budget, whereas job $\tau_{3,1}$ requires one extra unit of time. Fig. 2a illustrates the case in which no reclaiming is used and tasks are served by the plain CBS algorithm. Notice that, in spite of the budget saved by $\tau_{2,1}$, the third server is forced to postpone its current deadline when its budget is exhausted (it happens at time $t = 9$). As shown in Fig. 2b, however, we observe that the spare capacity saved by $\tau_{2,1}$ can be used by $\tau_{3,1}$ to advance its execution and prevent the server from postponing its deadline.

The reclaiming mechanism, working with the CBS, uses a global queue, the BASH queue, of spare capacities ordered by deadline. Whenever a task completes its execution and its server budget is greater than zero, the residual capacity can be used by any active task to advance its execution. When using a spare capacity, the task can be scheduled using the current deadline of the server to which the spare capacity belongs. In this way, each task can use its own capacity along with the residual capacities derived from other servers.

Whenever a new task instance is scheduled for execution, the server tries to use the residual capacities with deadlines less than or equal to the one assigned to the served instance; if these capacities are exhausted and the instance is not completed, the server starts using its own capacity. Every time a task ends its execution and the server becomes idle, the residual capacity (if any) is inserted with its deadline in the global queue of available capacities. Spare capacities are ordered by deadline and are consumed according to an EDF policy. This method, although developed for overrun control, can also be very effective in different contexts; for example, for improving the average response times of the served tasks, enhancing the performance of control applications, or increasing dependability in fault-tolerant real-time systems using recovery strategies under time redundancy. In such systems, in fact, an efficient reclaiming mechanism is important to exploit the unused computation time of backup copies whose primaries ended successfully.

3.1 The BASH Algorithm

In this section, we formally describe the BASH algorithm, assuming that each task τ_i is handled by a dedicated CBS server S_i running on a uniprocessor system. Bandwidth reclaiming is performed through the use of a global queue, called the BASH queue, containing all the residual capacities ordered by deadlines. The BASH algorithm can be defined as follows:

Algorithm 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 $\forall i,\ d_{i,0} = 0$. Finally, a global variable T^{idle} always maintains the finishing time of the last idle interval and it is initially set equal to zero.
2. Each BASH capacity is represented by an ordered tuple $Cap_q(r_q, d_q, c_q, U_q, T_q)$, where r_q is its release time (in the BASH queue), d_q is its absolute deadline, c_q is its budget, 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 $t < d_q \leq d_{i,k}$; otherwise, its own capacity c_i is used. Supposing a BASH capacity Cap_q is used and $r_q < T^{idle}$, the budget c_q of Cap_q is updated as $c_q = min[T_q U_q, (d_q - T^{idle})U_q]$ before Cap_q is used by server S_i; otherwise, the budget c_q is used as it is. Notice that each BASH capacity with deadline less than or equal to the current time t has already expired and has to be removed from 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 Δ ends.

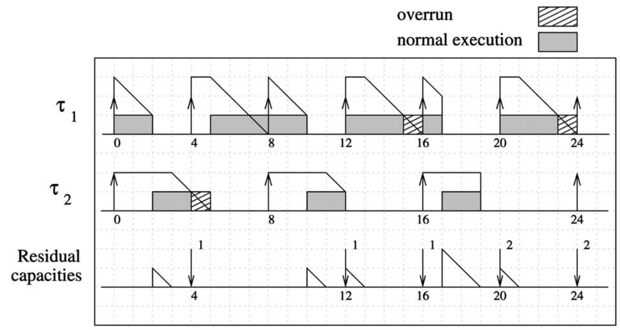


Fig. 3. Example of global resource reclaiming.

3.2 An Example

To better understand the proposed approach, we will describe a simple example which shows how our reclaiming algorithm works. Consider a task set consisting of two periodic tasks, τ_1 and τ_2, with periods $P_1 = 4$ and $P_2 = 8$, maximum execution times $WCET_1 = 4$ and $WCET_2 = 3$, and average execution times $C_1^{avg} = 3$ and $C_2^{avg} = 2$. Each task is scheduled by a dedicated CBS having a period equal to the task period and a budget equal to the average execution time. Hence, a task completing before its average execution time saves some budget, whereas it experiences an overrun if it completes after. A possible execution of the task set is reported in Fig. 3, which also shows the budget of each server and the residual capacities generated by each task. At time $t = 2$, task τ_1 has an early completion and a residual capacity equal to one with deadline equal to four becomes available. After that, τ_2 consumes the above residual capacity before starting to use its own capacity; hence, at time $t = 4$, a τ_2 overrun is handled without

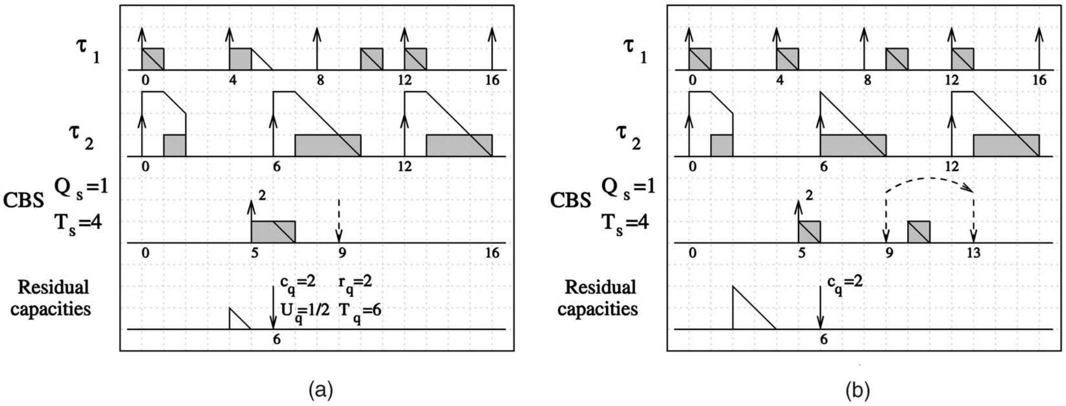


Fig. 4. (a) Example of resource reclaiming with BASH versus (b) example of resource reclaiming with CASH.

postponing its deadline. Notice that each task tries to use residual capacities before using its own capacity and that, if an idle interval occurs (see interval $[19, 20]$), T^{idle} is set equal to its finishing time ($T^{idle} = 20$) and the budget of next available BASH capacity is set as

$$\begin{aligned} c_q &= min[T_q U_q, (d_q - T^{idle}) U_q] \\ &= min[8 * 2/8, (24 - 20)2/8] = 1. \end{aligned}$$

Hence, the budget of BASH capacity with earliest deadline has to be recomputed according to rule 7 to handle the residual capacities correctly. The example above shows that overruns can be handled efficiently without postponing any deadline. A classical CBS, instead, would postpone some deadlines in order to guarantee tasks isolation. Clearly, if all the tasks consume their allocated budget, no reclaiming can be done and our approach performs the same as a plain CBS. However, this situation is very rare, hence our approach helps in improving the average system's performance.

The proposed technique performs efficient reclaiming of unused computation times like the CASH algorithm [6]; however, BASH improves system performance substantially, as shown in Section 5. This performance enhancement is achieved by storing three additional parameters when characterizing each spare capacity of the BASH queue. Notice that, by storing the capacity bandwidth U_q whose value represents the utilization of its generating server, the spare capacities are better preserved against idle times. In fact, it avoids having an idle interval of length $\Delta(t_1, t_2)$ consume the BASH capacities by the amount $t_2 - t_1$: Fig. 4 shows an example where BASH is compared to the CASH algorithm to better understand the key idea and performance gain of BASH. According to the example, the task set consists of two hard periodic tasks, τ_1 and τ_2, with periods $P_1 = 4$ and $P_2 = 6$, maximum execution times $WCET_1 = 1$ and $WCET_2 = 3$. In addition, soft aperiodic activities are handled by a CBS server with maximum budget $Q_s = 1$ and period $T_s = 4$. Each hard periodic task is scheduled by a dedicated CBS having a period equal to the task period and a budget equal to the maximum execution time: Hence, a hard task completing before its maximum execution time saves some budget to advance the execution of soft aperiodic tasks. At time $t = 2$, task τ_2 has an early completion and a residual capacity with budget equal to two and deadline equal to six becomes available. An idle interval occurs within $[2, 4]$, whose duration completely discharges the residual capacity of τ_2 (see Fig. 4b) if CASH is used (spare capacities are discharged during idle intervals). By using BASH instead (see Fig. 4a), job $\tau_{1,2}$ can use τ_2 residual capacity at time $t = 4$, whose budget is set as $c_q = min[T_q U_q, (d_q - T^{idle}) U_q] = min[6 * 1/2, (6 - 4)1/2] = 1$. As a consequence, an aperiodic job (released at time five and requesting two units of computation time) can exploit one unit of extra capacity completing at time $t = 7$, while the same job would complete at time $t = 10$ by using CASH.

As a concluding remark, it is worth noting that the BASH algorithm allows a capacity to become more resilient against idle intervals and to be independent of an idle interval length. In fact, while a long idle interval would likely discharge all spare capacities according to CASH rules, the new approach allows the leftover budget of each BASH capacity to become a function of its absolute deadline and processor utilization, no matter how long an idle interval lasts.

3.3 Theoretical Validation for Independent Tasks

In this section, we analyze the schedulability condition for a hybrid task set consisting of hard periodic and soft aperiodic tasks. Each task is scheduled using a dedicated CBS. If each hard periodic task is scheduled by a server[2] with maximum budget equal to the task WCET and with period equal to the task period, it behaves like a standard hard task scheduled by EDF. The difference is that each task can gain and use extra capacities and yields its residual capacity to other tasks. Notice that BASH is able to improve the average responsiveness of soft tasks by performing resource reclaiming, but cannot provide a hard guarantee unless hard tasks have been assigned a server budget greater than or equal to their WCETs. When assigning the server parameters for a soft task though, smaller server utilization diminishes soft task responsiveness while permitting us to guarantee more hard tasks. Moreover, since BASH improves the average performance of soft tasks by handling execution overruns, a good trade off consists of assigning server budget and period according to average execution time and average interarrival time of the handled soft task. The runtime exchange performed by BASH,

2. This assumption holds throughout the entire paper.

however, does not affect schedulability; thus, the task set can be guaranteed using the classical Liu and Layland condition:

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1,$$

where Q_i is the maximum server budget and T_i is the server period. Before proving the schedulability test, a lemma ensures that all the generated capacities are used before their own deadlines.

Lemma 1. *Given a set Γ of capacity-based servers along with the BASH algorithm, each BASH capacity used during the scheduling is exhausted before its deadline if and only if:*

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1, \quad (1)$$

where Q_i is the maximum server budget and T_i is the server period.

Proof. If. Assume (1) holds and a capacity Cap^* is not exhausted at time t^* when the corresponding deadline is reached. Let $t_a \geq 0$ be the last instant before t^* during which the CPU is idle (if there is no such time, set $t_a = 0$); let $t_b \geq 0$ be the last time before t^* at which a capacity with deadline after t^* is discharging (if there is no such time, set $t_b = 0$). If we take $t = max(t_a, t_b)$, one of the following two properties holds:

1. Only capacities released after t with deadline less than or equal to t^* are used during $[t, t^*]$.
2. One or more capacities, whose deadline is less than or equal to t^*, were released before the last idle interval $\Delta(t_1, t_2)$, their budget c_q is computed as $c_q = min[T_q U_q, (d_q - T^{idle}) U_q]$ (see rule 7 of Section 3.1), and execute within $[t, t^*]$.

Let us consider the following two cases:

CASE A. Assume that property one holds. Let $Q_T(t_x, t_y)$ be the sum of capacities created after t_x and with deadline less than or equal to t_y; since a capacity misses its deadline at time t^*, the following inequality holds:

$$Q_T(t, t^*) > (t^* - t).$$

In the interval $[t, t^*]$, we can write that:

$$(t^* - t) < Q_T(t, t^*) \leq \sum_{i=1}^{n} \left\lfloor \frac{t^* - t}{T_i} \right\rfloor Q_i \leq (t^* - t) \sum_{i=1}^{n} \frac{Q_i}{T_i},$$

which is a contradiction.

CASE B. Assume that property two holds. Hence, there exists a BASH capacity $Cap_q(r_q, d_q, c_q, U_q)$ with release time $r_q \leq t_1$ and deadline $d_q \leq t^*$, which is used within the interval of continuous utilization $[t, t^*]$. Since Cap_q is used within $[t, t^*]$ and $\Delta(t_1, t_2)$ is the last idle interval before the deadline miss, it follows that $T^{idle} = t_2 = t$. In fact, none of the capacities $\overline{Cap}$ with deadline greater than t^* can execute within interval $[t_2, t^*]$ and only capacities with deadline $d < d_q$ can preempt Cap_q and execute before it; in addition, Cap_q cannot execute within the interval of continuous utilization $[t, t^*]$ if a low priority capacity $\overline{Cap}$ (whose absolute deadline is $\overline{d} > t^*$) executes within $[t_2, t^*]$. It follows that only Cap_q and capacities created after t_2 with deadline less than or equal to t^* are used during $[t_2, t^*]$. Let $Q_T(t_2, t^*)$ be the sum of capacities created after t_2 and with deadline less than or equal to t^*. Since a capacity misses its deadline at time t^*, the following inequality holds:

$$Q_T(t_2, t^*) + c_q > (t^* - t_2),$$

where c_q is the new budget of Cap_q after the idle interval, that is, $c_q = min[T_q U_q, (d_q - T^{idle}) U_q]$. Assuming that server S_z generated Cap_q and $U_z = U_q$, it follows that:

$$\begin{aligned}
(t^* - t_2) &< Q_T(t_2, t^*) + c_q \\
&\leq \sum_{\substack{i=1 \\ i \neq z}}^{n} \left\lfloor \frac{t^* - t_2}{T_i} \right\rfloor Q_i + \left\lfloor \frac{t^* - d_q}{T_z} \right\rfloor Q_z + (d_q - t_2) U_q \\
&\leq (t^* - t_2) \sum_{\substack{i=1 \\ i \neq z}}^{n} U_i + (t^* - d_q) U_z + (d_q - t_2) U_q \\
&= (t^* - t_2) \sum_{i=1}^{n} U_i.
\end{aligned}$$

Finally, it follows that $1 < \sum_{i=1}^{n} U_i$, which is a contradiction.

Only if. Suppose that $\sum_i \frac{Q_i}{T_i} > 1$. Then, we show there exists an interval $[t_1, t_2]$ in which $Q_T(t_1, t_2) > (t_2 - t_1)$. Assume that all the servers are activated at time 0; then, for $L = lcm(T_1, \ldots, T_n)$, we can write that:

$$Q_T(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor Q_i = \sum_{i=1}^{n} \frac{L}{T_i} Q_i = L \sum_{i=1}^{n} \frac{Q_i}{T_i} > L,$$

hence, the "*only if* condition" follows. □

According to the above lemma, each *server deadline* is never missed when scheduling the BASH capacities if and only if the sum of servers' utilization does not exceed one. Notice that no statement is made on setting the server parameters (server maximum budget and period) to meet hard tasks' deadlines; the following theorem instead provides a hard tasks' schedulability condition under the assumption that the server parameters of each hard task are correctly set.

Theorem 2. *Let $\mathcal{T}_h$ be a set of periodic hard tasks, where each task τ_i is scheduled by a dedicated server with $Q_i = WCET_i$ and $T_i = P_i$ and let $\mathcal{T}_s$ be a set of soft tasks scheduled by a group of servers with total utilization U^{soft}. Then, $\mathcal{T}_h$ is feasible if and only if*

$$\sum_{\tau_i \in \mathcal{T}_h} \frac{Q_i}{T_i} + U^{soft} \leq 1. \quad (2)$$

Proof. The theorem follows immediately from Lemma 1; in fact, we can notice that each hard task instance has available at least its own budget equal to the task's WCET. Lemma 1 states that each capacity is always discharged before its deadline, hence it follows that each hard task instance has to finish by its deadline. □

It is worth noting that Theorem 2 also holds under a generic capacity-based server having a periodic behavior and a limited bandwidth.

TABLE 2
Parameters of the Task Set

$task$	$type$	Q_s	T_s	R_a	R_b
J_1	soft aperiodic	4	10	-	3
τ_2	hard periodic	2	12	2	-
τ_3	hard periodic	6	24	4	2

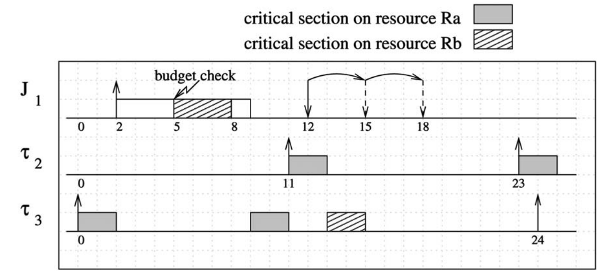


Fig. 5. BASH+SRP with static preemption levels.

4 Handling Resource Constraints

The BASH technique presented in the previous sections enhances the Constant Bandwidth Server (CBS) with the ability of managing overruns under the assumption that soft real-time tasks and hard real-time tasks are independent. Unfortunately, in a multiprogrammed system, tasks are rarely independent, but must often cooperate to provide the expected service. In a shared memory programming paradigm, such a cooperation is achieved through shared resources, which must be used in mutual exclusion to preserve data consistency during concurrent accesses. In this section, BASH will be extended to handle resource sharing among tasks with different criticality without compromising the real-time guarantee of hard tasks. The proposed solution is based on extending the BASH algorithm to maintain the key properties of Baker's Stack Resource Policy (SRP) [3] for resource sharing (the SRP policy and its main properties are briefly recalled in Appendix A.2).

Enabling resource sharing among hard periodic and soft aperiodic tasks is not straightforward. In particular, there are two main challenges in integrating BASH with the SRP:

1. Preemption levels under the SRP were developed under the assumption that relative deadlines are fixed, making the resulting preemption levels static values. Unfortunately, under BASH, server relative deadlines vary, thus preemption levels become dynamic.
2. If the CBS exhausts its budget while the served task is inside a critical section, high priority tasks would experience long blocking delays due to the budget replenishment rule.

Both problems will be analyzed in the next sections and suitable solutions will be provided to properly extend the BASH algorithm to efficiently support resource sharing.

4.1 Preventing Budget Exhaustion Inside Critical Sections

When shared resources are accessed in mutual exclusion by tasks handled by a capacity-based server, problems arise if the server exhausts its budget when a task is inside a critical section. In order to prevent long blocking delays due to the budget replenishment rule, a job which exhausted its budget could be allowed to continue executing with the same deadline, using extra budget until it leaves the critical section. At this time, the budget can be replenished at its full value and the deadline postponed. The maximum interference created by the budget overrun mechanism occurs when the server exhausts its budget immediately after the job entered its longest critical section. Thus, if ξ is the duration of the longest critical section of task τ handled by server S, the bandwidth demanded by the server becomes $\frac{Q_s+\xi}{T_s}$. This approach inflates the server utilization.

Alternatively, a job can perform a budget check before entering a critical section at time t. If the current budget c_s is not sufficient to complete the job's critical section, the budget is replenished and the server deadline postponed. The remaining part of the job follows the same procedure until the job completes. This approach dynamically partitions a job into *chunks*. Each chunk has execution time such that the consumed bandwidth is always less than or equal to the available server bandwidth U_s. By construction, a chunk has the property that it will never suspend inside a critical section. Notice that both techniques rely on the knowledge of the worst-case execution time of each critical section; however, while an overestimation of ξ causes a waste of bandwidth if the server utilization is inflated (first approach), the second technique (by using budget check and early replenishment) does not have this problem and simply postpones the server deadline and recharges its budget. The following example illustrates two different solutions using the BASH algorithm with the SRP protocol still maintaining static preemption levels:

Example. The task set consists of an aperiodic job, J_1, and two periodic tasks, τ_2 and τ_3, each one handled by a dedicated CBS. The task set shares two resources, R_a and R_b. In particular, J_1 and τ_3 share resource R_b, whereas τ_2 and τ_3 share resource R_a. The task set parameters are shown in Table 2, where R_a and R_b represent the worst-case execution time of critical sections accessing resources R_a and R_b, respectively.

A simple-minded solution could maintain a fixed relative deadline whenever the budget must be replenished and the deadline postponed. The advantage of having a fixed relative deadline is to keep the SRP policy unchanged for handling resource sharing between soft and hard tasks. In this way, the budget is recharged by a variable amount according to the formula: $c_s = c_s + (d_s^{new} - d_s^{old})U_s$, where d_s^{new} is the postponed server deadline and d_s^{old} is the previous server deadline.

A possible solution produced by BASH+SRP is shown in Fig. 5. Notice that the ceiling of resource R_a is $ceil(R_a) = 1/12$ and the ceiling of R_b is $ceil(R_b) = 1/10$. When job J_1 arrives at time $t = 2$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + T_1 = 12$ according to the BASH algorithm. At that time, τ_3 is already inside a critical section on resource R_a, however, $H_{1,1}$ of job J_1 is able to preempt, having its preemption level $\pi_1 = 1/10 > \Pi_s$. At time $t = 5$, J_1 tries to access a critical

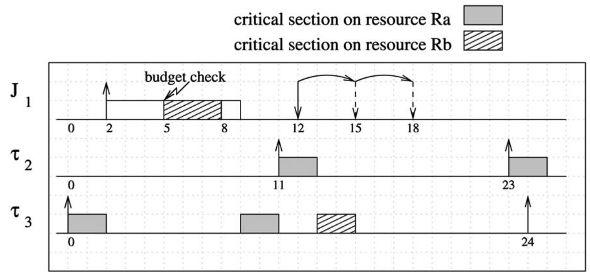


Fig. 6. BASH+SRP with static preemption levels and job suspension.

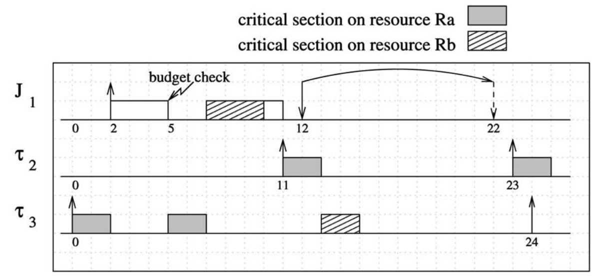


Fig. 7. BASH+SRP with dynamic preemption levels.

section, however, its residual budget is equal to 1 and is not sufficient to complete the whole critical section. As a consequence, a new chunk $H_{1,2}$ is generated with an arrival time $a_{1,2} = 5$ and a deadline $d_{1,2} = a_{1,2} + T_1 = 15$ (the relative deadline is fixed). The budget is replenished according to the available server bandwidth; hence, it follows that $c_1 = c_1 + (d_1^{new} - d_1^{old})U_1 = 1 + 1.2$. Unfortunately, the current budget is not sufficient to complete the critical section and an extra budget equal to 0.8 is needed. Hence, we have to inflate the budget, wasting bandwidth. The remaining part of the job follows the same procedure until the job completes. This approach has two main drawbacks: An extra budget still needs to be reserved and jobs are cut in too many chunks, so increasing the algorithm overhead. Another simple-minded solution could suspend a job whenever its budget is exhausted until the current server deadline. Only at that time would the job again become eligible and a new chunk would be ready to execute with the budget recharged at its maximum value ($c_s = Q_s$) and the deadline postponed by a server period.

The schedule produced using this approach on the previous example is shown in Fig. 6. When job J_1 arrives at time $t = 2$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + T_1 = 12$ according to the BASH algorithm. As previously shown, at time $t = 5$, J_1 tries to access a critical section; however, its residual budget is equal to one and is not sufficient to complete the whole critical section. As a consequence, J_1 is temporarily suspended and a new chunk is released at time $t = 12$, with deadline $d_{1,2} = 22$ and the budget replenished ($c_1 = Q_1 = 4$). This approach also has a drawback: It increases the response time of aperiodic tasks.

4.2 Dynamic Preemption Levels

The two methods described in the previous section show that, although the introduction of budget check can prevent budget exhaustion inside a critical section without inflating the server size, fixed relative deadline and static preemption levels do not permit providing an easy and efficient solution to the addressed problem.

We now show that using dynamic preemption levels for aperiodic tasks allows achieving a simpler and more elegant solution to the problem of sharing resources under BASH+SRP. According to the new method, whenever there is a replenishment, the server budget is always recharged by Q_s and the server deadline is postponed by T_s. It follows that the server is always eligible, but each aperiodic task gets a dynamic relative deadline.

Since, to maintain the main properties of the SRP, preemption levels must be inversely proportional to relative deadlines, we define the preemption level $\pi_{i,j}$ of a job chunk $H_{i,j}$ as $\pi_{i,j} = 1/(d_{i,j} - a_{i,j})$. Notice that $\pi_{i,j}$ is assigned to each chunk at runtime and cannot be computed offline. As a consequence, a job J_i is characterized by a *dynamic preemption level* π_i^d equal to the preemption level of the current chunk. To perform an offline guarantee of the task set, it is necessary to know the *maximum preemption level* that can be assigned to each job J_i by its server. From the deadline assignment rule of the BASH algorithm, it follows that each chunk has a minimum relative deadline D_i^{min} equal to its server period.

By setting $D_i^{min} = T_i$, we can assign each aperiodic task τ_i a *maximum preemption level* π_i^{max} inversely proportional to the server period ($\pi_i^{max} = 1/D_i^{min} = 1/T_i$). In order to use a uniform notation for all the tasks in the system, we define the maximum preemption level of a periodic hard task as the classical preemption level typically used in the original SRP protocol ($\pi_i^{max} = \pi_i = \frac{1}{D_i}$). The maximum preemption levels will be used to compute the ceiling of each resource offline. We note that $\pi_i^d \leq \pi_i^{max}$, in fact, by definition,

$$\forall i, j \;\; \pi_{i,j} = \frac{1}{d_{i,j} - a_{i,j}} = \pi_i^d \leq \frac{1}{D_i^{min}} = \frac{1}{T_i} = \pi_i^{max}. \qquad (3)$$

The schedule produced by BASH+SRP under dynamic preemption levels is shown in Fig. 7. When job J_1 arrives at time $t = 2$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + T_1 = 12$ according to the BASH algorithm. At that time, τ_3 is already inside a critical section on resource R_a; however, $H_{1,1}$ of job J_1 is able to preempt, having a preemption level $\pi_{1,1} = 1/10 > \Pi_s$. At time $t = 5$, J_1 tries to access a critical section; however, its residual budget is equal to one and is not sufficient to complete the whole critical section. As a consequence, the deadline is postponed and the budget replenished ($c_1 = c_1 + Q_1 = 1 + 4$). Hence, the next chunk $H_{1,2}$ of J_1 starts at time $a_{1,2} = 5$ with deadline $d_{1,2} = d_{1,1} + T_1 = 22$ and budget $c_1 = 5$. However, chunk $H_{1,2}$ cannot start because its preemption level $\pi_{1,2} = 1/17 < \Pi_s$. It follows that τ_3 executes until the end of its critical section. When the system ceiling becomes zero, J_1 is able to preempt τ_3. We note that the bandwidth consumed by any chunk is no greater than U_1 since, whenever the budget is refilled by Q_1, the absolute deadline is postponed by T_1. The main advantage of the proposed approach is that it does not require reserving extra budget for synchronization purposes and does not jeopardize the response time of aperiodic tasks. However, we need to determine the effects that dynamic preemption levels have on the properties of the SRP protocol.

We first note that, since each chunk is scheduled by a fixed deadline assigned by the CBS, each chunk inherits the SRP properties. In particular, each chunk can be blocked for at most the duration of one critical section by the preemption test and, once started, it will never be blocked for resource contention. However, since a soft aperiodic job may consist of many chunks, it can be blocked more than once. The behavior of hard tasks remains unchanged, permitting resource sharing between hard and soft tasks without jeopardizing the hard tasks' guarantee. The details of the proposed technique are described in the next section.

4.3 BASH with Resource Constraints

In this section, we first define the rules governing the BASH algorithm with resource constraints, BASH-R, that have been informally introduced in the previous section. We then prove its properties. Under the BASH-R, each job J_i starts executing with the server current budget c_i and the server current deadline $d_{i,k}$. Whenever a chunk $H_{i,j}$ exhausts its budget at time $\bar{t}$, that chunk is terminated and a new chunk $H_{i,j+1}$ is released at time $a_{i,j+1} = \bar{t}$ with an absolute deadline $d_{i,j+1} = d_{i,j} + T_i$ (where T_i is the period of the server). When the job chunk $H_{i,j}$ attempts to lock a semaphore, the BASH-R checks whether there is sufficient budget to complete the critical section. If not, a replenishment occurs and the execution performed by the job is labeled as chunk $H_{i,j+1}$, which is assigned a new deadline $d_{i,j+1} = d_{i,j} + T_i$. This procedure continues until the last chunk completes the job.

To comply with the SRP rules, a chunk $H_{i,j}$ starts its execution only if its priority is the highest among the active tasks and its *preemption level* $\pi_{i,j} = 1/(d_{i,j} - a_{i,j})$ is greater than the system ceiling. For the SRP protocol to be correct, every resource R_i is assigned a static[3] ceiling $ceil(R_i)$ (we assume binary semaphores) equal to the highest maximum preemption level of the tasks that could be blocked on R_i when the resource is busy. Hence, $ceil(R_i)$ can be computed as follows:

$$ceil(R_i) = \max_k \{\pi_k^{max} \mid \tau_k \;\; needs \;\; R_i\}. \quad (4)$$

It is easy to see that the ceiling of a resource computed by (4) is greater than or equal to the one computed using the dynamic preemption level of each task. In fact, as shown by (3), the maximum preemption level of each aperiodic task represents an upper bound on its dynamic value.

Finally, in computing the blocking time for a periodic/aperiodic task, we need to take into account the duration of the critical section of an aperiodic task without considering its relative deadline. In fact, the actual relative deadline of a chunk belonging to an aperiodic task is assigned online and it is not known in advance. The blocking times can be computed as a function of the minimum relative deadline of each aperiodic task, as follows:

$$B_i = \max\{s_{j,h} \mid (T_i < T_j) \wedge \pi_i^{max} \le ceil(\rho_{j,h})\}, \quad (5)$$

where $s_{j,h}$ is the worst-case execution time of the *h*th critical section of task τ_j, $\rho_{j,h}$ is the resource accessed by the critical section $s_{j,h}$, and T_i is the period of the dedicated server. The B_i parameter computed by (5) is the blocking time experienced by a hard or soft task. In fact, $T_i = D_i^{min}$ for a soft aperiodic task and $T_i = D_i$ for a hard periodic task.

3. In the case of multiunits resources, the ceiling of each resource is dynamic as it depends on the number of units actually free.

TABLE 3
Parameters of the Task Set

task	*type*	Q_s	T_s	R_a	R_b
J_1	soft aperiodic	4	8	-	3
τ_2	hard periodic	3	10	1	1
τ_3	hard periodic	4	24	2	-

The correctness of our approach will be formally proven in Section 4.4. We will show that the modifications introduced in the BASH and SRP algorithms do not change any property of SRP and permit keeping a static ceiling for the resources even though the relative deadline of each chunk is dynamically assigned at runtime by the CBS server. As shown in the examples illustrated above, an additional constraint has to be introduced to handle resource constraints. In particular, the correctness of the proposed technique relies on the following statement: A task must never exhaust its budget when it is inside a critical section. As a consequence, with respect to the original definition given in Section 3.1, we have to add the following rule:

- Whenever a served job J_i tries to access a critical section, if $c_i < \xi_i$ (where ξ_i is the duration of the longest critical section of job J_i), a budget replenishment occurs, that is, $c_i = c_i + Q_i$, and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.

The above rule has been added to prevent a task from exhausting its budget when it is using a shared resource. This is done by performing a budget check before entering a critical section. If the current budget is not sufficient to complete a critical section, the budget is replenished and the deadline postponed. This minor change allows the BASH algorithm to become compliant with the proposed approach without modifying its global behavior.

4.3.1 An Example

The following example illustrates the usage of the BASH-R algorithm in the presence of resource constraints. The task set consists of an aperiodic job J_1, handled by a server with maximum budget $Q_1 = 4$ and server period $T_1 = 8$ and two periodic tasks τ_2, τ_3, which share two resources R_a and R_b; in particular, J_1 and τ_2 share resource R_b, while τ_2 and τ_3 share resource R_a. The task set parameters are shown in Table 3, where R_a and R_b represent the worst-case execution time of critical sections accessing resources R_a and R_b, respectively.

The schedule produced by BASH-R+SRP is shown in Fig. 8. When job J_1 arrives at time $t = 3$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + T_1 = 11$ according to the BASH-R algorithm. At that time, τ_3 is already inside a critical section on resource R_a; however, $H_{1,1}$ of job J_1 is able to preempt, having a preemption level $\pi_{1,1} = 1/8 > \Pi_s$. At time $t = 6$, J_1 tries to access a critical section; however, its residual budget is equal to one and is not sufficient to complete the whole critical section. As a consequence, the

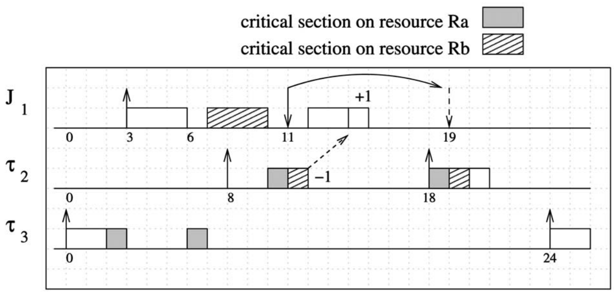


Fig. 8. Schedule produced by BASH-R+SRP.

deadline is postponed and the budget replenished. Hence, the next chunk $H_{1,2}$ of J_1 starts at time $a_{1,2} = 6$ with deadline $d_{1,2} = 19$. The chunk $H_{1,2}$ of J_1 cannot start because its preemption level $\pi_{1,2} = 1/13 < \Pi_s$. It follows that τ_3 executes until the end of its critical section. When the system ceiling becomes zero, J_1 is able to preempt τ_3. When J_1 frees resource R_b, τ_2 starts executing. Task τ_2 has an early completion at time $t = 12$, saving one unit of spare capacity. Such a capacity is used by chunk $H_{1,2}$ of J_1. It follows that J_1 can finish its execution, avoiding an additional deadline postponement. It is worth noting that each chunk can be blocked for at most the duration of one critical section by the preemption test and, once it is started, it will never be blocked for resource contention. In the next section, the SRP properties are formally proven and the validity of the guarantee test is analyzed.

4.4 Theoretical Validation for Resource Constrained Tasks

In this section, we prove that all SRP properties are preserved for hard periodic tasks and for each chunk of soft aperiodic tasks. Finally, we provide a sufficient guarantee test for verifying the schedulability of hybrid task sets consisting of hard and soft tasks. Since a preemption level is always inversely proportional to the relative deadline of each chunk, the following properties can be derived in a straightforward fashion:

Property 1. *A chunk $H_{i,h}$ is not allowed to preempt a chunk $H_{j,k}$, unless $\pi_{i,h} > \pi_{j,k}$.*

Property 2. *If the preemption level of a chunk $H_{i,j}$ is greater than the current system ceiling, then there are sufficient resources available to meet the requirement of $H_{i,j}$ and the requirement of every chunk that can preempt $H_{i,j}$.*

Property 3. *If no chunk $H_{i,j}$ is permitted to start until $\pi_{i,j} > \Pi_s$, then no chunk can be blocked after it starts.*

Property 4. *Under the BASH-R+SRP policy, a chunk $H_{i,j}$ can be blocked for at most the duration of one critical section.*

Property 5. *The BASH-R+SRP prevents deadlocks.*

The proofs of properties listed above are similar to those of original Baker's paper [3]. The following lemma shows how hard periodic tasks maintain their behavior unchanged:

Lemma 3. *Under BASH-R+SRP, each job of a hard periodic task can be blocked at most once.*

Proof. The schedule of hard periodic tasks produced by EDF is the same as the one produced by handling each hard periodic task by a dedicated server with a maximum budget equal to the task WCET and server period equal to the task period; it follows that each hard task can never be cut into multiple chunks. Hence, using Property 4, it follows that each instance of a hard periodic task can be blocked for at most the duration of one critical section. □

The following theorem provides a simple sufficient condition to guarantee the feasibility of hard tasks when they share resources with soft tasks under the BASH-R+SRP algorithm.

Theorem 4. *Let Γ be a task set composed of n hard periodic tasks and m soft aperiodic tasks, each one (soft and hard) scheduled by a dedicated server. Suppose tasks are ordered by decreasing maximum preemption level (so that $\pi_i^{max} \geq \pi_j^{max}$ only if $i < j$), then the hard tasks are schedulable by BASH-R+SRP if*

$$\forall i, \quad 1 \leq i \leq n + m \qquad \sum_{j=1}^{i} \frac{Q_j}{T_j} + \frac{B_i}{T_i} \leq 1, \tag{6}$$

where Q_j is the maximum budget of the dedicated server, T_j is the server period, and B_i is the maximum blocking time of task τ_i.

Proof. Suppose (6) is satisfied for each τ_i. Notice that aperiodic tasks get dynamic relative deadlines due to the deadline assignment rule of the BASH-R algorithm; hence, it follows that each task chunk has a relative deadline greater than or equal to its server period. Therefore, we have to analyze two cases:

Case A. Task τ_i has a relative deadline $D_i = T_i$. Using Baker's guarantee test (see (12) in Appendix A.2), it follows that the task set Γ is schedulable if

$$\forall i, \quad 1 \leq i \leq n + m \qquad \sum_{j=1}^{i-1} \frac{Q_j}{D_j} + \frac{Q_i}{T_i} + \frac{B_i^{new}}{T_i} \leq 1, \tag{7}$$

where D_j ($D_j \geq T_j$) is the relative deadline of task τ_j and B_i^{new} is the blocking time τ_i might experience when each τ_j has a relative deadline equal to D_j. Notice that a task τ_j can block as well as preempt τ_i, varying its relative deadline D_j; however, τ_j cannot block and preempt τ_i simultaneously. In fact, if the current instance of τ_j preempts τ_i, its absolute deadline must be before τ_i's deadline; hence, the same instance of τ_j cannot also block τ_i; otherwise, it should have its deadline after τ_i's deadline. From the considerations above, the worst-case scenario happens when τ_i experiences the maximum number of preemptions: It occurs by shortening, as much as possible, the relative deadline of each task τ_j, that is, setting $D_j = T_j$. In addition, even though B_i might be less than B_i^{new}, τ_j's interference due to preemption is always greater than or equal to its blocking effect, that is, $\sum_{j=1}^{i-1}(Q_j/T_j - Q_j/D_j) \geq (B_i^{new} - B_i)/T_i$. Hence, it follows that:

$$\forall i, \quad 1 \leq i \leq n + m$$
$$\sum_{j=1}^{i-1} \frac{Q_j}{D_j} + \frac{Q_i}{T_i} + \frac{B_i^{new}}{T_i} \leq \sum_{j=1}^{i-1} \frac{Q_j}{T_j} + \frac{Q_i}{T_i} + \frac{B_i}{T_i}.$$

Finally,

$$\sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\frac{Q_i}{T_i}+\frac{B_i}{T_i}\;\leq\;1.$$

Notice that the last inequality holds for the theorem hypothesis; hence, (7) is satisfied and the task set is schedulable.

Case B. Task τ_i has a relative deadline $D_i > T_i$. As in *Case A*, the task set Γ is schedulable if

$$\forall i,\;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i-1}\frac{Q_j}{D_j}+\frac{Q_i}{D_i}+\frac{B_i^{new}}{D_i}\;\leq\;1. \tag{8}$$

From the considerations above, it follows that the worst-case scenario also occurs when $\forall j,\;\; D_j = T_j$:

$$\forall i,\;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i-1}\frac{Q_j}{D_j}+\frac{Q_i}{D_i}+\frac{B_i^{new}}{D_i} \leq \sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\frac{Q_i}{D_i}+\frac{B_i^{new}}{D_i}.$$

Notice that tasks are sorted in decreasing order of maximum preemption levels and each task τ_j has the relative deadline set as $D_j = T_j$, except task τ_i, whose relative deadline is $D_i > T_i$. Since τ_i has an unknown relative deadline whose value changes dynamically, (8) has to be checked for each D_i, where D_i is greater than T_i. Hence, from (11) (see Appendix A.2) we derive that the blocking time B_i^{new} of task τ_i is a function of the actual relative deadline D_i as follows:

$$\begin{array}{rcl} T_i \leq\; D_i &<& T_{i+1} \;\Rightarrow\; B_i^{new} = B_i,\\ T_{i+1} \leq\; D_i &<& T_{i+2} \;\Rightarrow\; B_i^{new} = B_{i+1},\\ &\vdots& \\ T_{n+m-1} \leq\; D_i &<& T_{n+m} \;\Rightarrow\; B_i^{new} = B_{n+m-1},\\ T_{n+m} \leq\; D_i && \Rightarrow\; B_i^{new} = B_{n+m} = 0. \end{array}$$

It is worth noting that the terms $B_i, B_{i+1}, \ldots, B_{n+m}$ are the blocking times computed by (5) and are experienced by hard or soft tasks if the relative deadline of each task is set equal to the period of its dedicated server. Finally, a $k \geq i$ will exist such that:

$$T_k \leq\; D_i \;<\; T_{k+1} \;\Rightarrow\; B_i^{new} = B_k,$$

so, it follows that:

$$\sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\frac{Q_i}{D_i}+\frac{B_i^{new}}{D_i} = \sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\frac{Q_i}{D_i}+\frac{B_k}{D_i}$$
$$\leq \sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\frac{Q_i}{T_i}+\frac{B_k}{T_k} \;\;\leq\;\; \sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\sum_{h=i}^{k}\frac{Q_h}{T_h}+\frac{B_k}{T_k},$$

the last inequality holds because k must be greater than or equal to i and $D_i \geq T_k \geq T_i$. Finally:

$$\sum_{j=1}^{i-1}\frac{Q_j}{T_j}+\sum_{h=i}^{k}\frac{Q_h}{T_h}+\frac{B_k}{T_k} \;=\; \sum_{j=1}^{k}\frac{Q_j}{T_j}+\frac{B_k}{T_k}\;\leq\;1.$$

The above inequality holds for the theorem hypothesis; hence, (8) is satisfied and the task set is schedulable. □

5 Performance Evaluation

The BASH algorithm has been implemented in the real-time simulator RTSIM [16] to measure the performance gain introduced by the bandwidth sharing mechanism and to verify the results predicted by the theory. In particular, several complex task set scenarios were generated to analyze the BASH behavior. In this section, we present the experimental results of the simulations that have been conducted: In particular, BASH has been compared with the CASH [6] and GRUB [12] algorithms. The seven experiments described in this section can be grouped into three sets. The first set shows the performance of the algorithms as a function of the ratio

$$\alpha = \frac{C_i^{avg}}{WCET_i}, \tag{9}$$

where C_i^{avg} is the average computation time, and $WCET_i$ is the worst-case execution time of periodic task τ_i. The second set of experiments compares the performance against varying aperiodic server utilization U_s, for a constant value of α. Finally, the third set of experiments illustrates the performance of the algorithms in terms of overhead, that is, it shows how BASH's algorithmic overhead compares with GRUB for varying aperiodic load. The performance of the algorithms was measured by computing the average aperiodic response time as a function of α and U_s. In particular, the response time has been normalized with respect to the average 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.

Each point in the plots has been computed over 50 runs, each having a duration of 100,000 units of time. A 98 percent confidence interval is plotted to show the simulations' accuracy. The hard periodic task set consists of 10 tasks; moreover, the execution times of aperiodic requests were chosen to be uniformly distributed in a predefined interval to impose a specific aperiodic load according to U_s. Finally, aperiodic interarrival times were generated according to an exponential distribution.

5.1 First Set of Experiments

The first set of experiments includes three simulations which show the performance of the algorithms as a function of α, for varying values of aperiodic server utilization U_s. Periods of hard tasks were chosen to be uniformly distributed in the interval $[100, 200]$, while their computation times were randomly generated such that their total utilization equaled $1 - U_s$. Computation times, interarrival times of the aperiodic tasks, and aperiodic server parameters for each simulation are shown in Table 4. The value of U_s is increased from the first to the third simulation to study the performance impact of aperiodic server utilization on aperiodic task response times.

Fig. 9 shows the results of the first experiment by setting $U_s = 0.2$. It is worth noting that BASH outperforms CASH by a wide margin, while it exhibits a near equivalent

TABLE 4
Simulation Parameters for the First Set of Experiments

Aperiodic Utilization	Aperiodic Task		Aperiodic Server	
	Computation	AIT[4]	Capacity	Period
$U_s = 0.20$	[4,8]	10	1	5
$U_s = 0.33$	[6,10]	12	2	6
$U_s = 0.50$	[6,9]	10	2	4

[4] *AIT represents the Average Interarrival Time of aperiodic tasks.*

performance to GRUB for values of α in the range $[0.2, 0.65]$. Notice that GRUB has a lower aperiodic response time than BASH for the range $[0.65, 0.90]$; however, the difference is marginal at best and BASH outperforms GRUB for higher values of α. This indicates that BASH is the best algorithm in heavily loaded systems where the average execution time is very close to the worst-case value.

Fig. 10 refers to the second experiment in which the aperiodic server utilization is $U_s = 0.33$. As expected, BASH outperforms CASH for all values of α; however, the margin of difference is lower when compared to the first experiment. BASH and GRUB are nearly equivalent in aperiodic response times for values of α in the range $[0.2, 0.65]$. For the range $[0.65, 0.90]$, the difference is more notable than it was in the first experiment; however, BASH continues to outperform GRUB for higher values of α.

The results of the third experiment are shown in Fig. 11 and assume $U_s = 0.5$. It is important to highlight that, again, BASH continues to outperform CASH for all values of α, but the performance difference has been considerably narrowed down from the first experiment to the third.

According to the first set of experiments, three distinct zones can be identified in terms of achieved performance: 1) $\alpha \leq 0.6$, where the aperiodic response of BASH and GRUB are the same; 2) $0.6 < \alpha \leq 0.9$, where GRUB performs better than BASH; 3) $\alpha > 0.9$, where BASH performs better than GRUB. Such behaviors are due to some intrinsic differences between GRUB and BASH algorithms; in fact, by setting $\alpha \leq 0.6$, the aperiodic response times of both GRUB and BASH are similar since the average periodic load is relatively low and the system is not heavily loaded. In medium load conditions ($0.6 < \alpha \leq 0.9$), GRUB offers better performance because it is able to fully reclaim the bandwidth of any inactive server as soon as the current time exceeds its virtual time. Finally, when the system becomes heavily loaded and the average computation time of periodic tasks is close to WCET ($\alpha > 0.9$), BASH outperforms GRUB because it can immediately exploit a spare capacity as soon as a periodic task completes, while GRUB has to wait until the current time (system time) reaches the server virtual time: This phenomenon becomes dominant when $\alpha > 0.9$ because the virtual time increment is proportional to the task execution time.

5.2 Second Set of Experiments

To test the sensitivity of the algorithms with respect to the aperiodic server utilization, three simulations have been performed for U_s ranging between $[0.10, 0.50]$ with α values 0.85, 0.70, and 0.50. CASH aperiodic response time is used to baseline the aperiodic response time of the other two algorithms. Periodic tasks have periods in the range $[200, 800]$ and computation times are randomly generated subject to the constraint that total periodic utilization remains constant and equal to $1 - U_s$. Aperiodic tasks follow an exponential distribution for the interarrival times with an average value equal to 40. Notice that, to avoid wide variation of aperiodic response times across different values of U_s, we increase the computation demand of the aperiodic tasks as U_s grows. The aperiodic server has capacity $Q_s = U_s * T_s$, where $T_s = 20$; the results of these experiments are shown in Figs. 12, 13, and 14. Notice that aperiodic response time increases with an increase in U_s: This seems counterintuitive at first, but we increase the aperiodic load to match the increase in aperiodic utilization on the x-axis.

As the reader can see from the three graphs, BASH outperforms CASH again by a significant margin. According

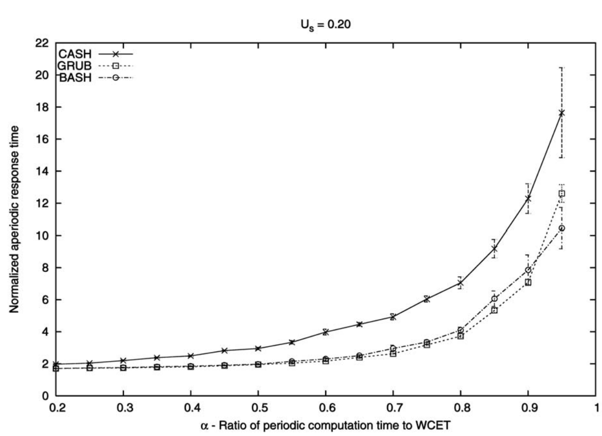


Fig. 9. Performance results of simulation 1.

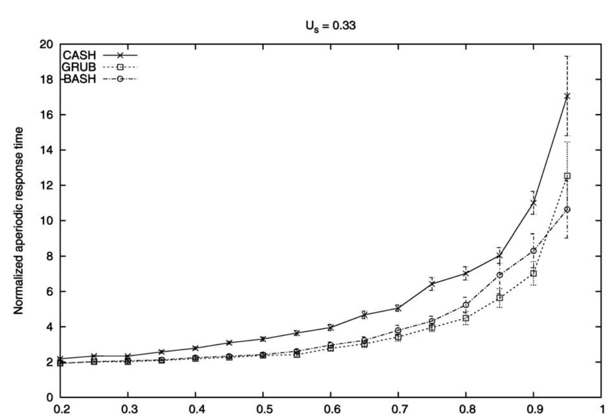


Fig. 10. Performance results of simulation 2.

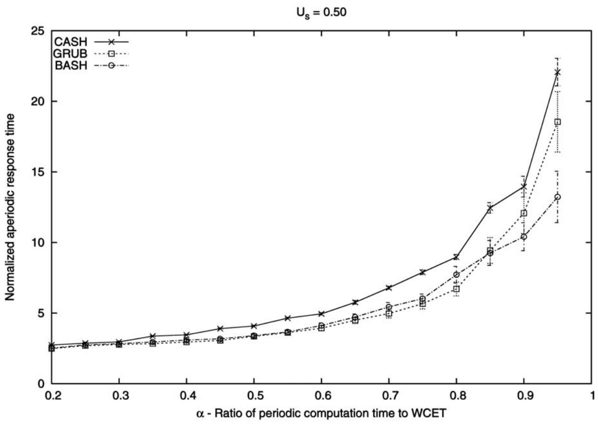


Fig. 11. Performance results of simulation 3.

to these three experiments, it is worth noting that BASH and GRUB have very similar behavior for low U_s values (range $[0.1, 0.2]$). Finally, this second set of experiments reinforces the conclusions of the first set of experiments. In particular, GRUB outperforms BASH if the average execution time of periodic hard tasks becomes small. In fact, if U_s increases on the x-axis of these experiments, the execution time of hard periodic tasks decreases in order to keep the system fully loaded but not overloaded: It follows that GRUB outperforms BASH as U_s increases. As a final remark, notice that BASH has better performance when the execution time of periodic tasks becomes longer since the resource reclaiming capability of GRUB is delayed.

5.3 Third Set of Experiments

The third experiment illustrates the algorithmic overhead of BASH and GRUB in terms of processor cycles. The experiments were performed on a Pentium IV 2.8 MHz computer. The cycles were measured using the Pentium benchmarking instruction RDTSC, which returns the number of clock cycles since the CPU was powered up or reset. The use of this 64-bit on processor register allowed us to minimize experimental errors, as evident in the plot with 98 percent confidence interval.

When considering BASH, we measured the number of processor cycles for all accesses to the BASH queue as its overhead, while GRUB overhead was measured as the total number of cycles executed for each state transition (possible

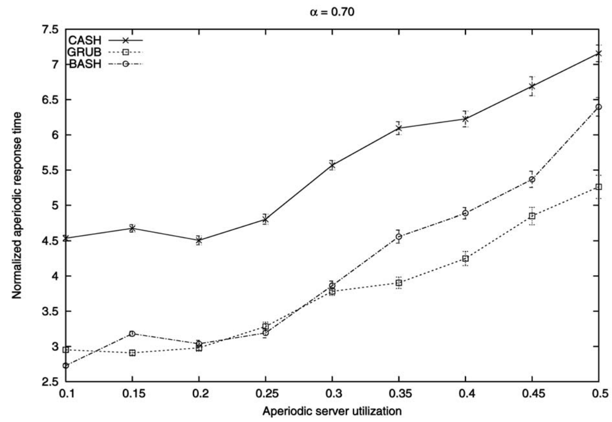


Fig. 13. Performance results of simulation 5.

states are: *inactive, activeContending, activeNonContending*). Notice that the Y-axis shows the sum of processor cycles spent by GRUB and CASH to update their data structures during the entire experiment. Each point has been computed as the average over 50 runs and the graph is plotted as a function of the aperiodic load: The X-axis shows the number of aperiodic task instantiations occurring in a certain window of time (100 units).

Fig. 15 illustrates the results of experiment 7, where $U_s = 0.2$ and $\alpha = 0.95$. The experiment shows that, for all values of aperiodic loads, GRUB has close to two-three times the overhead of BASH. The experiment was repeated for α ranging between 0.50-0.95 and U_s ranging between 0.5-0.90: Since the results were almost identical to the one of Fig. 15, we decided to include only one graph for this set of experiments. As a concluding remark, it is important to highlight that BASH queue can be implemented as a *Heap-based priority queue* [10] whose insert and extract operations take $\mathcal{O}(lnN)$ time, where N is the number of real-time tasks in the system; while looking up the capacity with earliest deadline takes only $\mathcal{O}(1)$.

6 Related Work

Different approaches have been proposed in the literature to deal with overruns and variable execution times. In [20], the authors provide an upper bound of completion times of

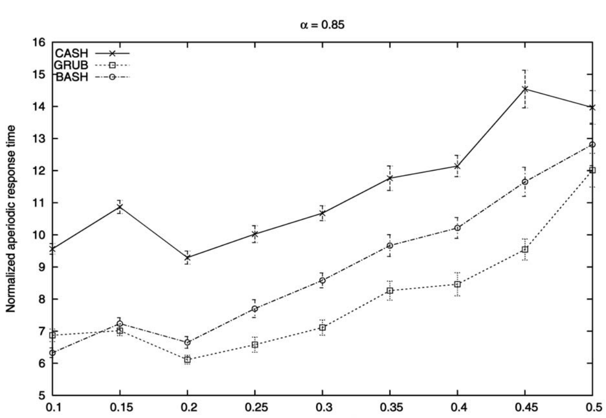


Fig. 12. Performance results of simultion 4.

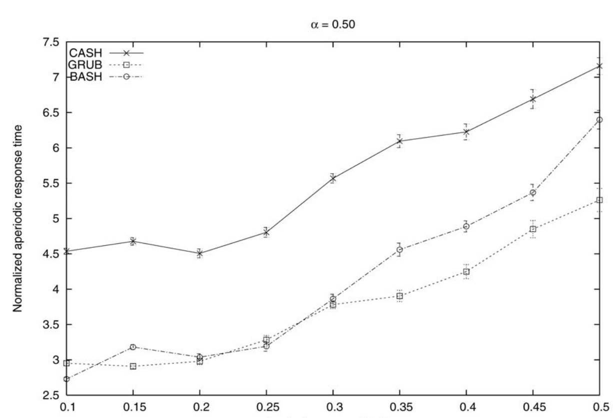


Fig. 14. Performance results of simulation 6.

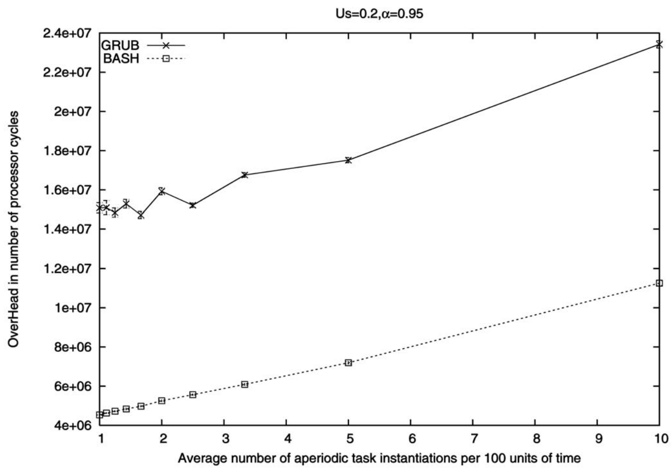


Fig. 15. Performance results of simulation 7.

jobs chains with variable execution times and arbitrary release times. In [15], a guarantee is computed for tasks whose jobs are characterized by variable computation times and interarrival times, occurring with a cyclical pattern. In [14], a capacity reservation technique is used to bound the computational demand of tasks with variable computation times in a fixed priority environment. According to this approach, a fraction of the CPU bandwidth is reserved to each task to achieve temporal isolation. Although such a solution prevents unbounded interference, overruns are not handled efficiently. In fact, whenever a job consumes the reserved budget, its remaining portion is scheduled in the background, thus prolonging its completion for an unpredictable amount of time. In [21], the authors present a Transform-Task Method (TTM) according to which a task is split into two pieces, where the second piece (i.e., the exceeding computation time causing the overrun) is handled as a job served by a Sporadic Server [19]. Using this approach, a probabilistic guarantee is performed on tasks whose execution times have known distribution. In [9], the authors propose two approaches for handling overruns. The first approach, called the Overrun Server Method (OSM), extends the TTM method to combine a general baseline algorithm for scheduling normal periodic tasks with a generic aperiodic server for handling overruns. Although, this method performs better than handling overruns in background, it cannot ensure that the remaining portion of a task instance is always executed before the next one. The second approach, called the Isolation Server Method (ISM), can achieve isolation among tasks, but it cannot provide a priori guarantee.

A more efficient technique, namely, the *Constant Bandwidth Server* (CBS), is proposed in [1] under a dynamic priority environment. As in [14], a fraction of the CPU bandwidth is reserved to each task, but tasks are scheduled by EDF using a suitable deadline, computed as a function of the reserved bandwidth and the actual requests. If a task requires executing more than expected, its deadline is postponed and its budget replenished. This method allows us to achieve isolation among tasks and overruns are handled efficiently based on their actual deadline. As mentioned in the introduction, although isolation mechanisms are essential for increasing system's reliability in the presence of tasks with variable execution times, the correct behavior of the system strongly depends on a correct reservation policy. Recently, this problem has been addressed by a number of authors who proposed new techniques to reduce such a negative aspect of isolation.

In [11], the authors proposed the Bandwidth Sharing Server (BSS) to handle several multithread applications on a single processor by allowing threads belonging to the same application to reclaim the spare time due to early completions. Although the algorithm provides isolation among applications, no isolation is guaranteed among tasks belonging to the same application. A multiapplication environment is also treated in [8], where a two-level scheduling architecture is used to handle each application by a dedicated server. This approach is able to isolate the effect of overloads at the application level, rather than at the task level, but does not provide a global reclaiming mechanism to efficiently exploit the reserved bandwidths.

In [5], the authors proposed a methodology for improving the performance of hard control applications using a resource reservation approach combined with a suitable offline analysis, based on the Seto et al. algorithm [17]. A less pessimistic analysis and a local reclaiming mechanism are used to increase the average task rates, while a proper overrun control mechanism is adopted to guarantee each task a minimum rate. However, since the reclaiming is local to each task (i.e., no capacity sharing is allowed), the improvement achieved over the Seto et al. algorithm is not so significant. In [6], [7], the authors address the problem of resource reclaiming and resource sharing separately and in an independent manner; compared to them, this work integrates resource reclaiming and resource sharing in a complete framework. Moreover, a new and improved reclaiming technique is introduced which outperforms the one in [6]. In [12], the authors proposed an elegant technique for scheduling a set of real-time tasks on a single processor so that each task runs as it is executing on a slower dedicated processor. The method achieves isolation and allows reclaiming most of the spare time unused by tasks. A critical parameter of this approach is the time granularity used in the algorithm; in fact, a small quantum reduces the scheduling error, but increases the overhead due to context switches. In [4], the authors propose a capacity sharing protocol for enhancing soft aperiodic responsiveness in a fixed priority environment, where each soft task is handled by a dedicated server. Although the basic idea of capacity sharing is the same as the one proposed in our paper, the main difference from our BASH algorithm is that, in [4], each server can "steal" capacity from the other servers to advance the execution of the served task, thus losing isolation among the served tasks (a low priority server could receive less bandwidth than requested). In our case, instead, a capacity is given only after a job is completed and a new replenishment is always performed (with a suitable deadline) when a new job arrives. These rules allow the algorithm to preserve the isolation property. Moreover, with respect to the capacity sharing protocol, the BASH algorithm is used to solve a different problem (overrun control) in a different context (dynamic deadline scheduling with resource reservation).

7 Conclusions

In this paper, we presented a bandwidth sharing (BASH) mechanism which allows us to achieve temporal protection on tasks' execution while performing efficient reclaimation of the unused computation times. The algorithm is able to handle tasks with soft, hard, as well as flexible, timing constraints.

The BASH algorithm has been implemented in the RTSIM simulator in order to evaluate its performance and validate our theoretical results. The experiments show the effectiveness of the reclaiming mechanism in enhancing the system performance under different workload conditions. Specific experiments on the reclaiming mechanism showed that the overhead introduced by the algorithm is significantly lower than other reclaiming techniques; moreover, BASH is easily implemented, allowing its use in real applications. Finally, the algorithm complexity resulted to be $\mathcal{O}(lnN)$, where N is the number of real-time tasks in the system. As future work, we plan to apply this technique for handling fault-tolerant applications where each task is composed of a primary and a backup copy.

Appendix A

A.1 The CBS Algorithm

A CBS is characterized by an ordered pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth. At each instant, a fixed deadline $d_{s,k}$ and a budget c_s are associated with the server. Every time a new job $\tau_{i,j}$ has to be served, it is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline $d_{s,k}$. The current budget c_s represents the amount of computation time schedulable by the CBS using the current server deadline. Whenever a served job executes, the budget c_s is decreased by the same amount and, every time $c_s = 0$, the server budget is recharged to the maximum value Q_s and a new server deadline is generated as $d_{s,k+1} = d_{s,k} + T_s$.

Fig. 16 illustrates an example in which a task τ_1, with maximum computation time $WCET_1 = 2$ and period $P_1 = 5$, is scheduled by EDF together with another task, τ_2, served by a CBS having a budget $Q_s = 3$ and a period $T_s = 6$. Initially, $c_s = 0$ and $d_{s,0} = 0$. When job $\tau_{2,1}$ (requiring five units of computation) arrives at time $t = 3$, c_s is charged at the value $Q_s = 3$ and the job is assigned a deadline

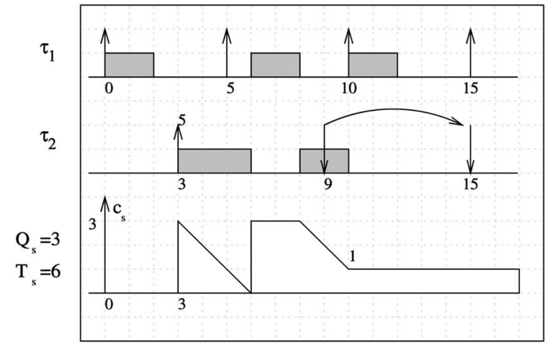


Fig. 16. Example of a CBS server.

$d_{s,1} = t + T_s = 9$. At time $t = 6$, the budget is exhausted, so c_s is replenished and a new deadline $d_{s,2} = d_{s,1} + T_s = 15$ is generated by the server and assigned to job $\tau_{2,1}$.

In [1], it is proven that, in any interval of time of length L, a CBS with bandwidth U_s will never demand more than U_sL, independently from the actual task requests. Such a property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most important consequence of this result is that such tasks can be scheduled together with hard tasks without affecting the a priori guarantee, even in the case in which soft requests exceed the expected load.

A.2 The Stack Resource Policy

The Stack Resource Policy (SRP) is a concurrency control protocol proposed by Baker [3] to bound the priority inversion phenomenon in static as well as dynamic priority systems.

Under the EDF scheduling algorithm, each task τ_i is assigned a dynamic priority p_i inversely proportional to its absolute deadline d_i and a static *preemption level* π_i such that the following property holds:

Property 6. *Task τ_i is not allowed to preempt task τ_j unless $\pi_i > \pi_j$.*

Under EDF, Property 6 is verified if periodic task τ_i is assigned the following preemption level:

$$\pi_i = \frac{1}{D_i},$$

where D_i is its relative deadline. In addition, every resource R_k is assigned a static[4] *ceiling* defined as

$$ceil(R_k) = \max_i \{\pi_i \mid \tau_i \ \ needs \ \ R_k\}. \tag{10}$$

Finally, a dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max[\{ceil(R_k) \mid R_k \text{ is currently busy}\} \cup \{0\}].$$

Then, the SRP scheduling rule states that

> a task is not allowed to start executing until its priority is the highest among the active tasks and its preemption level is greater than the system ceiling.

4. In the case of multiunits resources, the ceiling of each resource is dynamic as it depends on the number of units actually free.

The SRP ensures that, once a task is started, it will never block until completion; it can only be preempted by higher priority tasks.

This protocol has several interesting properties. For example, it applies to both static and dynamic scheduling algorithms, prevents deadlocks, bounds the maximum blocking times of tasks, reduces the number of context switches, can be easily extended to multiunit resources, allows tasks to share stack-based resources, and its implementation is straightforward.

Under the SRP there is no need to implement waiting queues. In fact, a task never blocks its execution: It simply cannot start executing if its preemption level is not high enough. As a consequence, the blocking time B_i considered in the schedulability analysis refers to the time for which task τ_i is kept in the ready queue by the preemption test. Although the task never blocks, B_i is considered a "blocking time" because it is caused by tasks having lower preemption levels.

The maximum blocking time for a task τ_i is bounded by the duration of the longest critical section among those that can block τ_i. Assuming relative deadlines equal to periods, the maximum blocking time for each task τ_i can be computed as the longest critical section among those with a ceiling greater than or equal to the preemption level of τ_i:

$$B_i = \max\{s_{jh} \mid (D_i < D_j) \ \wedge \ \pi_i \le \mathrm{ceil}(\rho_{jh})\}, \qquad (11)$$

where s_{jh} is the worst-case execution time of the *h*th critical section of task τ_j, D_j is its relative deadline, and ρ_{jh} is the resource accessed by the critical section s_{jh}. Given these definitions, the feasibility of a task set with resource constraints (when only periodic and sporadic tasks are considered) can be tested by the following sufficient condition [3]:

$$\forall i, \ \ 1 \le i \le n \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \ \le \ 1, \qquad (12)$$

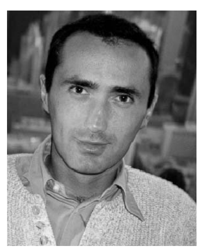
which assumes that all the tasks are sorted by decreasing preemption levels so that $\pi_i \ge \pi_j$ only if $i < j$.

Acknowledgments

This work is supported in part by US National Science Foundation (NSF) grant CCR-0237884, and in part by NSF grant CCR-0325716.

References

[1] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1998.
[2] L. Abeni and G. Buttazzo, "Resource Reservations in Dynamic Real-Time Systems," *Real-Time Systems,* vol. 27, no. 2, pp. 123-165, 2004.
[3] T.P. Baker, "Stack-Based Scheduling of Real-Time Processes," *J. Real-Time Systems,* vol. 3, no. 1, pp. 67-100, 1991.
[4] G. Bernat and A. Burns, "Multiple Servers and Capacity Sharing for Implementing Flexible Scheduling," *Real-Time Systems,* vol. 22, nos. 1-2, pp. 49-75, Jan.-Mar. 2002.
[5] M. Caccamo, G. Buttazzo, and L. Sha, "Elastic Feedback Control," *Proc. IEEE 12th Euromicro Conf. Real-Time Systems,* June 2000.
[6] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity Sharing for Overrun Control," *Proc. IEEE Real-Time Systems Symp.*, Dec. 2000.
[7] M. Caccamo and L. Sha, "Aperiodic Servers with Resource Constraints," *Proc. IEEE Real-Time Systems Symp.*, Dec. 2001.
[8] Z. Deng and J.W.S. Liu, "Scheduling Real-Time Applications in an Open Environment," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1997.
[9] M.K. Gardner and J.W.S. Liu, "Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload," *IEEE Proc. 11th Euromicro Conf. Real-Time Systems*, June 1999.
[10] D.E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1998.
[11] G. Lipari and G. Buttazzo, "Scheduling Real-Time Multi-Task Applications in an Open System," *Proc. IEEE 11th Euromicro Conf. Real-Time Systems*, pp. 234-241, June 1999.
[12] G. Lipari and S. Baruah, "Greedy Reclaimation of Unused Bandwidth in Constant-Bandwidth Servers," *IEEE Proc. 12th Euromicro Conf. Real-Time Systems*, June 2000.
[13] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 40-61, 1973.
[14] C.W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," *Proc. IEEE Intl Conf. Multimedia Computing and Systems,* May 1994.
[15] A.K. Mok and D. Chen, "A Multiframe Model for Real-Time Tasks," *Proc. IEEE Real-Time System Symp.*, Dec. 1996.
[16] Real-Time System SIMulator (RTSIM), http://rtsim.sssup.it, 2004.
[17] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control Systems," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1996.
[18] K.G. Shin, C.M. Krishna, and Y.-H. Lee, "A Unified Method for Evaluating Real-Time Computer Controllers and Its Application," *IEEE Trans. Automatic Control,* pp. 357-365, Apr. 1985.
[19] B. Sprunt, L. Sha, and J.P. Lehoczky, "Aperiodic Scheduling for Hard Real-Time System," *J. Real-Time Systems,* vol. 1, pp. 27-60, 1989.
[20] J. Sun and J.W.S. Liu, "Bounding Completion Times of Jobs with Arbitrary Release Times and Variable Execution Times," *Proc. IEEE Real-Time System Symp.*, Dec. 1996.
[21] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.W.-S. Liu, "Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times," *Proc. IEEE Real-Time Technology and Applications Symp.*, Jan. 1995.



Marco Caccamo graduated in computer engineering from the University of Pisa in 1997 and received the PhD degree in computer engineering from the Scuola Superiore S. Anna in 2002. He is an assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include real-time operating systems, real-time scheduling and resource management, wireless sensor networks, and quality of service control in next generation digital infrastructures. He is recipient of a US National Science Foundation CAREER Award (2003) and a member of the IEEE.

Giorgio C. Buttazzo graduated in electronic engineering from the University of Pisa in 1985, received the master's degree in computer science from the University of Pennsylvania in 1987, and the PhD degree in computer engineering from the Scuola Superiore S. Anna of Pisa in 1991. He is an associate professor of computer engineering at the University of Pavia, Italy. His main research interests include real-time operating systems, dynamic scheduling algorithms, quality of service control, multimedia systems, advanced robotics applications, and neural networks. He is a member of the IEEE.

Deepu C. Thomas received the MS degree in computer science from the University of Illinois at Urbana-Champaign in 2004. He is a design engineer with the Core Operating System Divison of Microsoft Corporation. He previously worked with SANYO Semiconductors R&D in the area of real-time embedded systems and his main research interests include real-time computing and operating systems. He is a member of the IEEE.