HAL open science

Multicore CPU Reclaiming: Parallel or Sequential?

Luca Abeni, Giuseppe Lipari, Andrea Parri, Youcheng Sun

To cite this version:

Luca Abeni, Giuseppe Lipari, Andrea Parri, Youcheng Sun. Multicore CPU Reclaiming: Parallel or Sequential?. 31st ACM Symposium on Applied Computing, Apr 2016, Pise, Italy. 10.1145/2851613.2851743 . hal-01286130

HAL Id: hal-01286130

https://hal.science/hal-01286130

Submitted on 18 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Multicore CPU Reclaiming: Parallel or Sequential?

Luca Abeni
University of Trento, Italy
luca.abeni@unitn.it

Giuseppe Lipari
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France
giuseppe.lipari@univ-lille1.fr

Andrea Parri and Youcheng Sun
Scuola Superiore Sant'Anna, Pisa, Italy
{a.parri, y.sun}@sssup.it

March 18, 2016

Abstract

When integrating hard, soft and non-real-time tasks in general purpose operating systems, it is necessary to provide temporal isolation so that the timing properties of one task do not depend on the behaviour of the others. However, strict budget enforcement can lead to inefficient use of the computational resources in the presence of tasks with variable workload. Many resource reclaiming algorithms have been proposed in the literature for single processor scheduling, but not enough work exists for global scheduling in multiprocessor systems. In this paper we propose two reclaiming algorithms for multiprocessor global scheduling and we prove their correctness. We also present their implementation in the Linux kernel and we compare their performance on synthetic experiments.

1 Introduction

The Resource Reservation Framework [20, 1] is an effective technique to integrate the scheduling of real-time tasks in general-purpose systems, as demonstrated by the fact that it has been recently implemented in the Linux kernel [14]. One of the most important properties provided by resource reservations is *temporal isolation*: the worst-case performance of a task does not depend on the temporal behaviour of the other tasks running in the system. This property can be enforced by limiting the amount of time for which each task can execute in a given period.

In some situations, a strict enforcement of the executed runtime (as done by the hard reservation mechanism that is currently implemented in the Linux kernel) can be problematic for tasks characterized by highly-variable, or difficult to predict, execution times: allocating the budget based on the task' Worst Case Execution Time (WCET) can result in a waste of computational resources; on the other hand, allocating it based on a value smaller than the WCET can cause a certain number of deadline misses. These issues can be addressed by using a proper *CPU reclaiming mechanism*, which allows tasks to execute for more than their reserved time **if spare CPU time is available** and if this over-execution does not break the guarantees of other real-time tasks.

While many algorithms (e.g., [18, 15, 10, 17]) have been developed for reclaiming CPU time in single-processor systems, the problem of reclaiming CPU time in multiprocessor (or multicore) systems has been investigated less. Most of the existing reclaiming algorithms (see [15] for a summary of some commonly used techniques) are based on keeping track of the amount of execution time reserved to some tasks, but not used by them, and by distributing it between the various active tasks. In a multiprocessor system, this idea can be extended in two different ways:

1. by considering a *global* variable that keeps track of the execution time *not used by all the tasks in the system* (without considering the CPUs/cores on which the tasks execute), and by distributing such an unused execution time to the tasks. This approach will be referred to as *parallel reclaiming* in this paper, because the execution time not used by one single task can be distributed to multiple tasks that execute in parallel on different CPUs/cores;

2. by considering multiple per-CPU/core (per-runqueue, in the Linux kernel *slang*) variables each representing the unused bandwidth that can be distributed to the tasks executing on the corresponding CPU/core. This approach will be referred to as *sequential reclaiming* in this paper, because the execution time not used by one single task is associated to a CPU/core, and cannot be distributed to multiple tasks executing simultaneously.

This paper compares the two mentioned approaches by extending the GRUB (Greedy Reclamation of Unused Bandwidth) [17] reclaiming algorithm to support multiple processors according to sequential reclaiming and parallel reclaiming. The comparison is performed both from the theoretical point of view (by formally analysing the schedulability of the obtained algorithm) and by running experiments on a real implementation of this extension, named M-GRUB. Such implementation of M-GRUB reclaiming (that can do either parallel or sequential reclaiming) is based on the Linux kernel and extends the `SCHED_DEADLINE` scheduling policy.

The paper is organised as follows: in Section 2 we recall the related work. In Section 3 we present our system model and introduce the definitions and concepts used in the paper. The algorithms and admission tests used as a starting point for this work are then presented in Section 4. The two proposed reclaiming

rules are described in Section 5. In Section 6 we discuss the implementation details and in Section 7 we present the results of our experiments. Finally, in Section 8 we present our conclusions.

2 Related work

The problem of reclaiming unused capacity for resource reservation algorithms has been mainly addressed in the context of single processor scheduling.

The CASH (CApacity SHaring) algorithm [10] uses a queue of unused budgets (also called *capacities*) that is exploited by the executing tasks. However, the CASH algorithm is only useful for periodic tasks. Lin and Brandt proposed BACKSLASH [15], a mechanism based on capacities that integrates four different principles for slack reclaiming. Similar techniques still based on capacities are used by Nogueira and Pinho [18].

The GRUB algorithm [17] modifies the rates at which servers' budgets are decreased so to take advantage of free bandwidth. The algorithm can be also used by aperiodic tasks. We present the GRUB algorithm in Section 4 as it is used as a basis for our multiprocessor reclaiming schemes. For fixed priority scheduling, Bernat et al. proposed to reconsider past execution so to take advantage of the executing slack [6].

Reclaiming CPU time in multiprocessor systems is more difficult (especially if global scheduling is used), as shown by some previous work [9] that ends up imposing strict constraints on the distribution of spare budgets to avoid compromising timing isolation: spare CPU time can only be donated by hard real-time tasks to soft real-time tasks – which are scheduled in background respect to hard tasks – and reservations must be properly dimensioned.

To the authors' best knowledge, the only previous algorithm that explicitly supports CPU reclaiming on all the real-time tasks running on multiple processors without imposing additional constraints (and has been formally proved to be correct) is M-CASH [19]. It is an extension of the CASH algorithm to the multiprocessor case, which additionally includes a rule for reclaiming unused bandwidth by aperiodic tasks. The algorithm uses the utilisation based test by Goossens, Funk and Baruah [12] as a base schedulability test for the servers. It distinguishes two kinds of servers: servers for periodic tasks (whose utilisation is reclaimed using capacity-based mechanism) and servers for aperiodic tasks, whose bandwidth is reclaimed with a technique similar to the parallel reclaiming that we propose in Section 5.1. However, M-CASH has never been implemented in a real OS kernel. On the other hand, the GRUB algorithm [17] has been implemented in the Linux kernel [2], after extending the algorithm to support multiple CPUs, but the multiprocessor extensions used in this implementation have not been formally analysed nor validated from a theoretical point of view.

3 System model and definitions

We consider a set of n real-time tasks τ_i scheduled by a set of n servers S_i $(i = 1, \ldots, n)$.

A real-time *task* τ_i is a (possibly infinite) sequence of jobs $J_{i,k}$: each job has an arrival time $a_{i,k}$, a computation time $c_{i,k}$ and a deadline $d_{i,k}$. *Periodic* real-time tasks are characterised by a period T_i, and their arrival time can be computed as $a_{i,k+1} = a_{i,k} + T_i$. *Sporadic* real-time tasks wait for external events with a minimum inter-arrival time, also called T_i, so $a_{i,k+1} \geq a_{i,k} + T_i$. Periodic and sporadic tasks are usually associated a relative deadline D_i such that $d_{i,k} = a_{i,k} + D_i$.

A *server* is an abstract entity used by the scheduler to reserve a fraction of CPU-time to a task. Each server S_i is characterised by the following parameters: P_i is the *server period* and it represents the granularity of the reservation; U_i is the fraction of reserved CPU-time, also called *utilisation* factor or *bandwidth*. In each period, a server is reserved at least a *maximum budget*, or *runtime*, $Q_i = U_i P_i$.

The execution platform consists of m identical processors (Symmetric Multiprocessor Platform, or SMP). In this paper we use the Global Earliest Deadline First (G-EDF) scheduling algorithm: all the tasks are ordered by increasing deadlines of the servers, and the m active tasks with the earliest deadlines are scheduled for execution on the m CPUs.

The logical priority queue of G-EDF is implemented in Linux by a set of *runqueues*, one per each CPU/core, and some accessory data structures for making sure that the m highest-priority jobs are executed at each instant (see [13] for a description of the implementation).

4 Background

In this section we first recall the Constant Bandwidth Server (CBS) algorithm [1, 4] for both single and multiprocessor systems. We then recall the GRUB algorithm [17], an extension of the CBS. Finally, we present two schedulability tests for Global EDF.

4.1 CBS and GRUB

As anticipated in Section 3, each server is characterised by a period P_i, a bandwidth U_i and a maximum budget $Q_i = U_i P_i$. In addition, each server maintains the following dynamic variables: the *server deadline* d_i, denoting at every instant the server priority, and the *server budget* q_i, indicating the remaining computation time allowed in the current server period.

At time t, a server can be in one of the following states: ActiveContending, if there is some job of the served task that has not yet completed; ActiveNonContending, if all jobs of the served task have completed, but the server has already consumed all the available bandwidth (see the transition rules below

for a characterisation of this state); Inactive, if all jobs of the server's task have completed and the server bandwidth can be reclaimed (see the transition rules below), and Recharging, if the server has jobs to execute, but the budget is currently exhausted and needs to be recharged (this state is generally known as "throttled" in the Linux kernel, or "depleted" in the real-time literature).

The EDF algorithm chooses for execution the m tasks with the earliest server deadlines among the ActiveContending servers. Initially, all servers are in the Inactive state and their state change according to the following rules:

1. When a job of a task arrives at time t, if the corresponding server is Inactive, it goes in ActiveContending and its budget and deadlines are modified as: $q_i \leftarrow U_i P_i$ and $d_i \leftarrow t + P_i$

2a. When a job of S_i completes and there is another job ready to be executed, the server remains in ActiveContending with all its variables unchanged;

2b. When a job of S_i completes, and there is no other job ready to be executed, the server goes in ActiveNonContending.

2c. If at some time t the budget q_i is exhausted, the server moves to state Recharging, and it is removed from the ready queue. The corresponding task is suspended and a new server is executed.

3. When $t = d_i$, the server variables are updated as $d_i \leftarrow d_i + P_i$ and $q_i \leftarrow U_i P_i$. The server is inserted in the ready queue and the scheduler is called to select the earliest deadline server, hence a context switch may happen.

4. If a new job arrives while the server is in ActiveNonContending, the server moves to ActiveContending without changing its variables.

5. A server remains in ActiveNonContending only if $q_i < (d_i - t)U_i$. When $q_i \geq (d_i - t)U_i$ the server moves to Inactive.

Only servers that are ActiveContending can be selected for execution by the EDF scheduler. If S_i does not execute, its budget is not changed. When S_i is executing, its budget is updated as $\mathrm{d}q_i = -\,\mathrm{d}t$.

When serving a task, a server generates a set of *server jobs*, each one with an arrival time, an execution time and a deadline as assigned by the algorithm's rules. For example, when the server at time t moves from Inactive to ActiveContending a new server job is generated with arrival time equal to t, deadline equal to $d_i = t + P_i$, and worst-case computation time equal to Q_i. A similar thing happens when the server moves from Recharging to ActiveContending, and so on.

We say that a server is *schedulable* if every server job can execute the budget Q_i before the corresponding server job deadline. It can be proved that the *demand bound function* dbf (see [5] for a definition) generated by the server jobs of S_i is bounded from above as follows: $dbf(t) \leq U_i t$ for each t. Hence, for

single processor systems it is possible to use the utilisation test of EDF as an admission control test, i.e.,

$$\sum_{i=1}^{n} U_i \leq 1. \tag{1}$$

It has been proved that if Equation (1) holds, then all servers are schedulable (i.e. all servers jobs will complete before their scheduling deadlines). Based on this result, it is possible to guarantee the respect of the deadlines of a task by setting $P_i \leq T_i$ and $Q_i \geq C_i$ (see the original paper [1] for a more complete description).

The CBS algorithm has been extended to multiprocessor global scheduling in [4]. The authors prove the temporal isolation and the hard schedulability properties of the algorithm when using the schedulability test of Goossens, Funk and Baruah [12], which we will recall next.

The GRUB algorithm [17] extends the CBS algorithm by enabling the reclaiming of unused bandwidth, while preserving the schedulability of the served tasks. The main difference between the CBS and GRUB algorithms is the rule for updating the budget. In the original CBS algorithm, the server budget is updated as $\mathrm{d}q_i = -\,\mathrm{d}t$, independently of the status of the other servers. To reclaim the excess bandwidth, GRUB maintains one additional global variable U_{act}, the total utilisation of all active servers

$$U_{\mathsf{act}} := \sum_{S_i \notin \mathsf{Inactive}} U_i$$

and uses it to update the budget q_i as

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = -(1 - (U_{\mathsf{sys}} - U_{\mathsf{act}})) \tag{2}$$

where U_{sys} is the utilisation that the system reserves to the set of all servers. As in the original CBS algorithm, the budget is not updated when the server is not executing. The executing server gets all the free bandwidth in a *greedy* manner, hence the name of the algorithm. The GRUB algorithm preserves the Temporal Isolation and Hard Schedulability properties of the CBS [16].

4.2 Admission control tests

When using the CBS or the GRUB algorithm it is important to run an admission test to check if all the servers' deadlines are respected. In single processor systems, the utilisation based test of Equation (1) is used for both the CBS and the GRUB algorithm. We now present two different schedulability tests for the multiprocessor case: an utilisation-based schedulability test for G-EDF by Goossens, Funk and Baruah [12] (referred to as GFB in this paper), and an interference-based schedulability test for G-EDF proposed by Bertogna, Cirinei and Lipari [8] (referred to as BCL in this paper).

GFB and BCL are not the most advanced tests in the literature: in particular, as discussed in [7], more effective tests – i.e. tests that can admit a larger number of task sets – are now available. The reason we chose these two in particular is their low complexity (so they can be used as on-line admission tests), and the fact that currently we are able to prove the correctness of the reclaiming rules with respect to these two tests in particular. In fact, we need to guarantee that the temporal isolation property continues to hold even when some budget is donated by one server to the other ones according to some reclaiming rule.

At the time of preparation of this paper, we have formally proved the correctness of the two reclaiming rules proposed in Section 5 with respect to the GFB and the BCL test – the proofs are not reported here, due to space constraints, and can be found in a separate Technical Report [3]. Using some other, more effective, admission control test may be unsafe, hence, for the moment we restrict our attention to GFB and BCL.

The GFB test is based on the notion of uniform multiprocessor platform, and it allows to check the schedulability of a task set based on its utilisation.

In practice, according to GFB, a set of periodic or sporadic tasks is schedulable by G-EDF if:

$$U \leq m - (m-1)U_{\mathsf{max}} \tag{3}$$

where $U_{\mathsf{max}} = \max_i\{U_i\}$.

The maximum utilisation we can achieve depends on the maximum utilisation of all tasks in the system: the largest is U_{max}, the lower is the total achievable utilisation. This test is only sufficient: if Equation (3) is not verified, the task set can still be schedulable by G-EDF.

The authors of [12] also proposed to give higher priority to largest utilisation tasks. In this paper we will not consider these further enhancements.

The BCL test was developed for sporadic tasks, and here we adapt the notations to server context. We focus on the schedulability of a target server S_k; particularly, we choose one arbitrary server job of S_k. Execution of the target server job may be interfered by jobs from other servers with earlier absolute deadlines. The *interference* over the target server job by an interfering server S_i, within a time interval, is the cumulative length of sub-intervals such that the target server is in ActiveContending but cannot execute, while S_i is running.

A *problem window* is the time interval that starts with the target server job's arrival time and ends with the target server job's deadline. As a result, the interference from an interfering server S_i is upper bounded by its *workload*, which is the cumulative length of execution that S_i conducts within the problem window. Let us denote the worst-case workload of a server S_i in the problem window as $\hat{W}_{i,k}$.

The formulation of the workload used in this paper is the same as the one proposed in [8]. In order not to compromise the schedulability when reclaiming CPU time (see [3]), we need to add one additional term to this upper bound to take into account the interference caused by the reclaimed bandwidth by servers that may be activated aperiodically.

Thus, in this paper the workload upper bound is defined as follows:

$$\hat{W}_{i,k} = \left\lfloor \frac{P_k}{P_i} \right\rfloor Q_i + \min\{Q_i, \Delta\} + \max\{\Delta - Q_i, 0\} U_i \tag{4}$$

where $\Delta = (P_k \bmod P_i)$.

On the other hand, when S_i and S_k execute in parallel on different processors at the same time, S_i does not impose interference on S_k. Thus, in case S_k is schedulable, the interference upon the target job by S_i cannot exceed $(P_k - Q_k)$.

In the end, according to the formulation of BCL used in this paper a task set is schedulable if one of the following two conditions is true:

$$\begin{aligned} a) \quad & \sum_{i \neq k} \min\{\hat{W}_{i,k}, P_k - Q_k\} < m(P_k - Q_k) \\ b) \quad & \sum_{i \neq k} \min\{\hat{W}_{i,k}, P_k - Q_k\} = m(P_k - Q_k) \\ & \qquad \wedge \neg \exists h \neq k, \hat{W}_{h,k} \leq P_k - Q_k \end{aligned} \tag{5}$$

Between the two tests presented so far (GFB and BCL) no one dominates the other: there are task sets that are schedulable by GFB but not by BCL, and vice versa. In general terms, BCL is more useful when a task has a large utilisation, whereas GFB is more useful for a task set with many small tasks.

5 Reclaiming rules

In this section we propose two new reclaiming rules for G-EDF. The first one, that we call *parallel reclaiming* equally divides the reclaimed bandwidth among all executing servers. The second one, that we call *sequential reclaiming* assigns the bandwidth reclaimed from one server to one specific processor.

Note that, when bandwidth reclaiming is allowed, served jobs within a server may run for more than the server's budget, as the bandwidth from other servers may be exploited. Due to space constraints, the proofs of correctness are not reported here. They can be found in [3].

5.1 Parallel reclaiming

In parallel reclaiming, we define one global variable U_{inact}, initialized to 0, that contains the total amount of bandwidth in the system that can be reclaimed. The rules corresponding to transitions 1 and 5 are modified as follows.

5. A server remains in ActiveNonContending only if $q_i < (d_i - t)U_i$. When $q_i \geq (d_i - t)U_i$ the server moves to Inactive. Correspondingly, variable U_{inact} is incremented by U_i.

1. When a job of a task arrives, if the corresponding server is Inactive, it goes to ActiveContending and its budget and deadline are modified as in the original rule. Correspondingly, U_{inact} is decremented by U_i.

While a server S_i executes on processor p, its budget is updated as follows:

$$\mathrm{d}q_i = -\max\left\{U_i, 1 - \frac{U_{\mathsf{inact}}}{m}\right\}\mathrm{d}t. \tag{6}$$

This rule is only valid for the GFB test. That is, if a set of servers are schedulable by GFB without bandwidth reclaiming, it is still schedulable when parallel reclaiming is allowed.

Initialization of U_{inact} While it is safe to initialise U_{inact} to be 0, we would like to take advantage of the initial free bandwidth in the system. Therefore, we initialise U_{inact} to the maximum initial value that can be reclaimed without jeopardizing the existing servers. From Equation (3), we have:

$$U + U_{\mathsf{inact}} \leq m - (m-1)U_{\mathsf{max}}.$$

That is,

$$U_{\mathsf{inact}} = m - (m-1)U_{\mathsf{max}} - U. \tag{7}$$

This is equivalent to having one or more servers, whose cumulative bandwidth is U_{inact} that are always inactive.

5.2 Sequential reclaiming

In sequential reclaiming, we define an array of variables $U_{\mathsf{inact}}[]$, one per each processor. The variable corresponding to processor p is denoted by $U_{\mathsf{inact}}[p]$. More specifically, $U_{\mathsf{inact}}[p]$ is the reclaimable bandwidth from inactive servers that complete their executions in processor p, and $U_{\mathsf{inact}}[p]$ can only be used by a server running on p. For any p, $U_{\mathsf{inact}}[p]$ can be safely initialised to be 0. Then, we modify the rules corresponding to transitions 1 and 5 as follows.

5. A server remains in ActiveNonContending only if $q_i < (d_i - t)U_i$. When $q_i \geq (d_i - t)U_i$ the server moves to Inactive. Correspondingly, one of the variables $U_{\mathsf{inact}}[p]$ is incremented by U_i. The server remembers the processor where its utilisation has been stored, so that it can recuperate it later on.

1. When a job of a task arrives, if the corresponding server is Inactive, it goes in ActiveContending and its budget and deadline are modified as in the original rule. Correspondingly, $U_{\mathsf{inact}}[p]$ (where p is the processor where the utilisation was stored before) is decremented by U_i.

While a server S_i executes, its budget is updated as follows:

$$\mathrm{d}q_i = -\max\{U_i, 1 - U_{inact}[p]\}\,\mathrm{d}t. \tag{8}$$

Notice that, for the moment, we do not explore more sophisticated methods for updating $U_{\mathsf{inact}}[p]$ when a server becomes inactive. In fact, there are several possible choices: for example, we could use a Best-Fit algorithm to concentrate

all reclaiming in the smallest number of processors, or Worst-Fit to distribute as much as it is possible the reclaimed bandwidth across all processors. In the current implementation, for simplicity we chose to update the variable $U_{\mathsf{inact}}[p]$ corresponding to the processor where the task has been suspended.

This rule works for both the GFB test of Equation (3) and and for the modified BCL test of Equation (5).

Initialization of $U_{inact}[]$ Similarly to the parallel reclaiming case, we would like to initialise each $U_{inact}[p]$ to be an as large as possible value so to reclaim the unused bandwidth in the system. Let us denote this value as U_x.

In case the GFB test is used, the maximum free bandwidth is computed as in Equation (7). Then, to keep the set of servers still schedulable w.r.t. GFB, we can initialise each variable to a value $U'_x = \frac{m-(m-1)U_{max}-U}{m}$.

When it comes to the BCL test, we can think of adding m servers to the system, each one with infinitesimal period and bandwidth equal to U''_x. To allow each server to use free bandwidth as much as possible while still guaranteeing the schedulability, the following condition should hold.

$$\forall k, \sum_{i \neq k} min(\hat{W}_{i,k}, P_k - Q_k) + mU''_x P_k < m(P_k - Q_k)$$

$$U''_x < \min_k \left\{ \frac{P_k - Q_k}{P_k} - \frac{\sum_{i \neq k} min(\hat{W}_{i,k}, P_k - Q_k)}{mP_k} \right\}$$

Finally we take the maximum between U'_x and U''_x, since only one of the two test needs to be verified.

$$U_{inact}[p] = \max \{U'_x, U''_x\}. \tag{9}$$

6 Implementation

The parallel and sequential reclaiming techniques described in the previous sections have been implemented in the Linux kernel, extending the `SCHED_DEADLINE` scheduling policy [14]. The modified Linux kernel has been publicly released at `https://github.com/lucabe72/linux-reclaiming`. These kernel modifications are based on a previous implementation of the GRUB algorithm [2], which however did not guarantee schedulability of server jobs.

6.1 Parallel reclaiming implementation

Parallel reclaiming requires to keep track of the total inactive bandwidth in a global (per-root domain) variable U_{inact} which is updated when tasks move from an Active states to the Inactive state or vice-versa.

Then, the budget decrease rate of every executing server depends on this global variable (see Equation (6)). For each executing `SCHED_DEADLINE` task,

the scheduler periodically accounts the executed time, decreasing the current budget (called "runtime" in the Linux kernel) of the task at each tick (or when a context switch happens). When a reclaiming strategy is used, the amount of time decreased from the budget depends on the value of the global variable U_{inact}. This means that, when a server changes its state to Inactive (or Active-Contending) and the value of U_{inact} changes all the CPUs should be signalled to update the budgets of the executing task before U_{inact} is changed. Such an inter-processor signalling can be implemented using Inter-Processor Interrupts (IPIs). However, this may substantially increase the overhead of the scheduler and increase its complexity; furthermore the combination of global variables and inter-processor interrupts may lead to race conditions very difficult to identify.

Therefore, parallel reclaiming has been implemented by introducing a small approximation: we avoid IPIs and the value of U_{inact} is sampled only at each scheduling tick. In this regard it is worth noting that every real scheduler implements an approximation of the theoretical scheduling algorithm: for example, `SCHED_DEADLINE` accounts the execution time at every tick (hence, a task can consume up to 1 tick more than the reserved runtime / budget).

Despite of the approximations introduced when implementing parallel reclaiming, during our experiments with randomly generated tasks we never observed any server deadline miss, probably because the GFB schedulability test is pessimistic and hence a certain amount of slack is available in the great majority of cases. It is important to underline, however, that from a purely theoretical point of view our current implementation of the parallel reclaiming rule cannot guarantee the respect of every server deadline.

6.2 Sequential reclaiming implementation

Implementing sequential reclaiming is easier under certain assumptions. In particular, we need to make sure that the code executing on the p-th runqueue only accesses variables local to the same runqueue.

In sequential reclaiming, we need to provide one variable U_{inact} for each runqueue. When a server on the p-th runqueue becomes Inactive, we update the corresponding variable; at this point, only the budget of the task executing on this runqueue needs to be updated. When a server becomes Active, we make sure that the corresponding handler is executed on the same processor where the server was previously suspended and became Inactive (the task will be migrated later, if necessary). Therefore, we just need to modify the local U_{inact} and update the budget of the task executing on this CPU. While this may not be the optimal way to distribute the spare bandwidth, we do not need any IPI to implement the exact reclaiming rule (the only approximations are the ones introduced by the `SCHED_DEADLINE` accounting mechanism).

Notice that for both parallel and sequential reclaiming the transition between ActiveNonContending and Inactive is handled by setting up an *inactive timer* that fires when such a transition should happen (for more details, see [2], where such a time is named *0-lag time*).

7 Experimental Evaluation

The effectiveness of the reclaiming algorithms has been evaluated through some experiments with the patched Linux kernel described in the previous section. The kernel version used for all the experiments is based on version 3.19 (in particular, the `global-reclaiming` and `refactored-reclaiming` branches of the `linux-reclaiming` repository have been used). All the tests were executed on a 4-cores Intel Xeon CPU.

7.1 Randomly generated tasks

The first set of experiments has been performed by executing sets of randomly generated real-time tasks with the `rt-app` application[1].

A set of 100 task sets with utilisation $U = 2.5$ has been generated using the `taskgen` [11] script, and the task sets that are schedulable according to the BCL and GFB tests have been identified. Some first `rt-app` runs confirmed that these task sets can actually run on the Linux kernel (using `SCHED_DEADLINE`) without missing any deadline. Then, the reclaiming mechanisms have been tested as follows: for each schedulable task set $\gamma = \{(C_i, T_i)\}$ generated by `taskgen`, a task set $\Gamma = \{\tau'_i\}$ has been generated, where τ'_i has period T_i and execution time uniformly distributed between $\alpha\gamma C_i$ and γC_i (hence, the WCET of task τ_i is γC_i), and is scheduled by a server with parameters $(Q_i = C_i, P_i = T_i)$. Notice that γ represents the ratio between the task's WCET and the maximum budget allocated to the task; hence increasing γ increases the amount of CPU time that the task needs to reclaim to always complete before its deadline; on the other hand, α represents the ratio between the BCET and the WCET of a task (so, $\alpha \leq 1$). Decreasing α increases the amount of CPU time that a task can donate to the other tasks through the reclaiming mechanism. When $\gamma \leq 1$, the WCET of each task is smaller than the maximum budget Q_i used to schedule the task, so all the tasks' deadlines will be respected. The experiments confirmed this expectation. When $\gamma > 1$, instead, the situation is more interesting because some deadlines can be missed and enabling the reclaiming mechanism allowed to reduce the amount of missed deadlines.

Figure 1 reports the percentage of missed deadlines for $\gamma = 1.1$ as a function of α when using no reclaiming, parallel reclaiming and sequential reclaiming. For parallel and sequential reclaiming, the results are reported both when initialising U_{inact} to 0 and when initialising it according to Equations (7) and (9) (reclaiming the initial spare utilisation). From the figure, it can be seen that both reclaiming algorithms allow to reduce the percentage of missed deadlines; however, parallel reclaiming tends to perform better than sequential reclaiming. When α increases, the average utilisation of the tasks increases, and the amount of CPU time that can be reclaimed decreases; hence, the differences between the efficiency of the various algorithms become more evident; however, parallel reclaiming performance do not seem to depend too much on the value used to

[1] `https://github.com/scheduler-tools/rt-app`

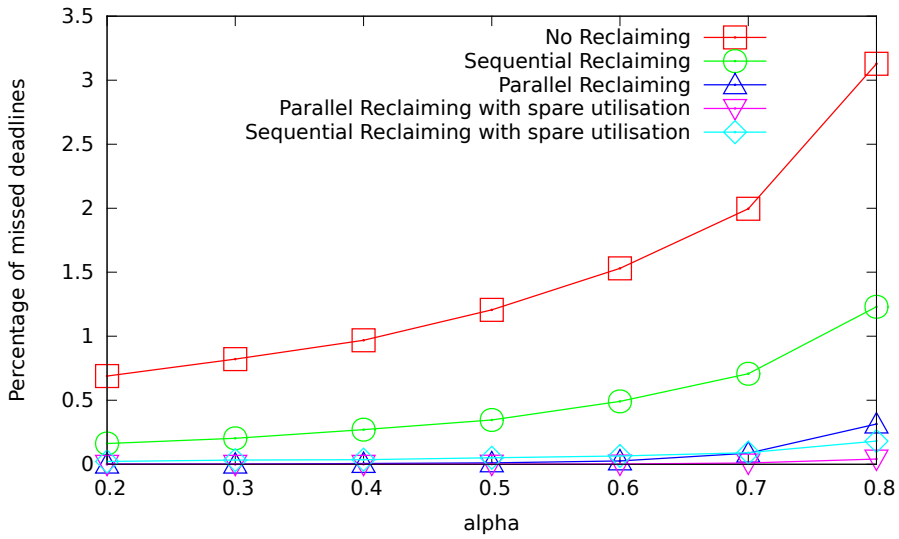

Figure 1: Percentage of missed deadlines when using `rt-app` with different reclaiming strategies. α varies from 0.2 to 0.8, and $\gamma = 1.1$.

initialise U_{inact}. This happens because with a small value of $\gamma = 1.1$, the tasks do not need to reclaim much execution time.

Increasing the value of γ to $\gamma = 1.3$ (Figure 2), the tasks need to reclaim more execution time and the effect of U_{inact} initialisation becomes more evident. In particular, with $\alpha = 0.8$ the tasks cannot donate enough execution time, so if U_{inact} is initialised to 0 (only the utilisation of the "existing tasks" can be reclaimed) the two reclaiming algorithms (parallel and sequential) do not seem to be very effective (the percentage of missed deadlines is similar to the "No Reclaiming" case). If, instead, U_{inact} is initialised according to Equations (7) and (9) the reclaiming algorithms are able to reclaim the spare utilisation and are able to reduce the percentage of missed deadlines.

Some partial conclusions can be drawn from this set of experiments. In general, the parallel reclaiming strategy performs better than sequential reclaiming. This is probably due to the fact that parallel reclaiming tends to fairly distribute the spare bandwidth across all processors, whereas in the current implementation of the sequential reclaiming we have no control on which server uses the reclaimed bandwidth. In fact, with sequential reclaiming, in the worst-case all reclaiming could go on one single processor and benefit only the tasks that by chance execute on that processor.

On the other hand, using sequential reclaiming we can admit a larger number of tasks sets, because the mechanism is valid both for the GFB and a modified version of the BCL test. Furthermore, as previously discussed a precise implementation of the parallel reclaiming is more costly in terms of overhead and programming effort. For the moment we conclude that sequential reclaiming

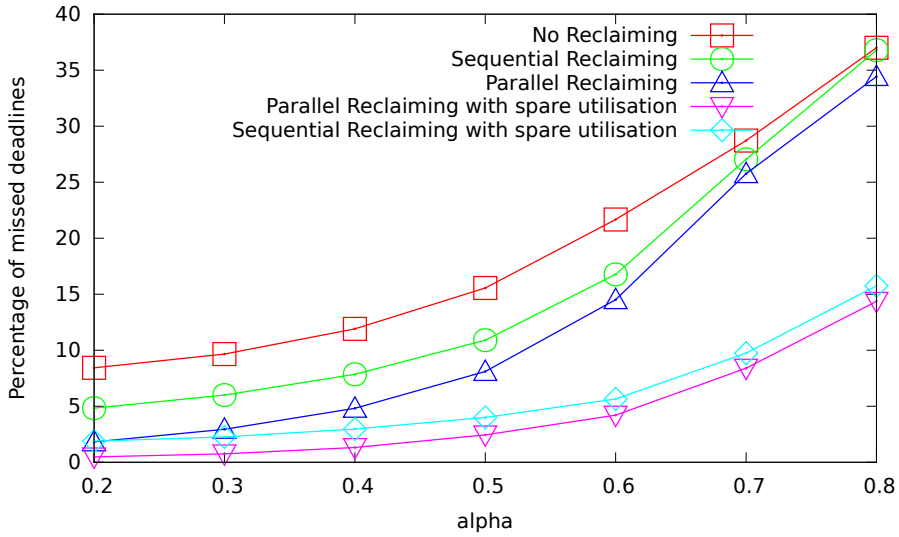

Figure 2: Percentage of missed deadlines when using `rt-app` with different reclaiming strategies. α varies from 0.2 to 0.8, and $\gamma = 1.3$.

seems to be preferable from an implementation point of view. However we acknowledge that further investigation is needed to perform a full assessment.

7.2 Experiments on real applications

In the next set of experiments, the performance of a real application (the `mplayer` video player[2]) has been evaluated. In particular, `mplayer` has been modified to measure the "Audio/Video delay", defined as the difference of the presentation timestamps of two audio and video frames that are reproduced simultaneously. A negative value of the Audio/Video delay means that video frames are played in advance with respect to the corresponding audio frames, while a positive value indicates that the video is late with respect to the audio (probably because `mplayer` has not been able to decode the video frames in time). When this value becomes too large, audio and video are perceived out of synch, and the quality perceived by the user is badly affected.

When `mplayer` is executed as a `SCHED_DEADLINE` task, it is pretty easy to set the reservation period $P = 1/fps$, where fps is the frame rate (in frames per second) of the video; however, correctly dimensioning the maximum budget/runtime Q is much more difficult. If Q is slightly under-dimensioned (larger than the average time needed to decode a frame, but smaller than the maximum time), the Audio/Video delay can become too large affecting the quality, and a reclaiming mechanism can help in improving the perceived quality.

[2] `http://www.mplayerhq.hu`

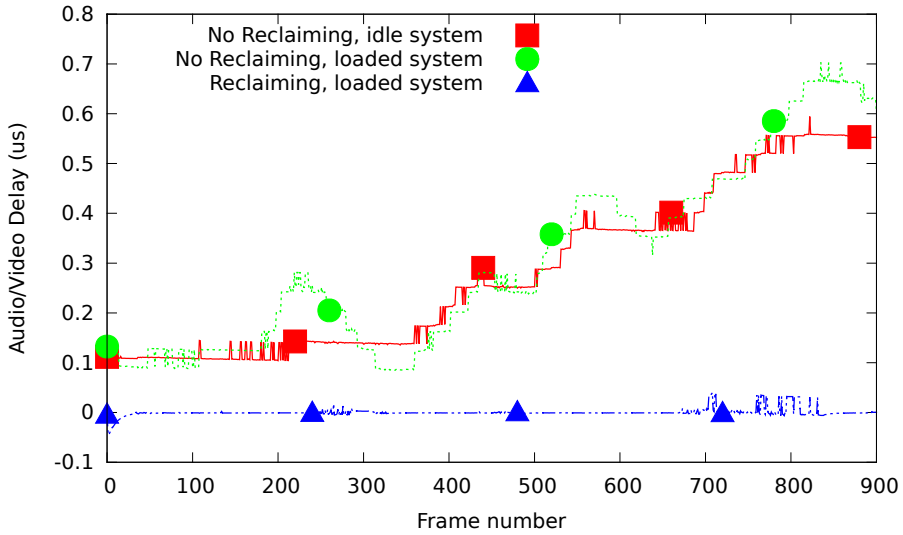

Figure 3: Audio/Video delay experienced by `mplayer` reproducing a video when scheduled with `SCHED_DEADLINE`. The three plots indicate `mplayer` executing alone or together with other real-time tasks without reclaiming, or with the M-GRUB reclaiming mechanism.

For example, Figure 3 shows the evolution of the Audio/Video delay experienced by `mplayer` when reproducing a full-D1 mpeg4 video (with vorbis audio) with $Q = 4.5ms$ and $P = 40ms$ (the video is 25 frames per second, so $P = 1s/25 = 40ms$). The experiment has been repeated executing `mplayer` alone on an idle 4-cores system ("No Reclaiming, idle system" line) or together with other real-time tasks (implemented by `rt-app`, in the "No Reclaiming, loaded system" line). In the "loaded system" case, the total utilisation was about 2.2 and the task set resulted to be schedulable according to both BCL and GFB. As it can be noticed, in both cases the Audio/Video delay continues to increase, and becomes noticeable for the user. When the M-GRUB reclaiming mechanism is activated, `mplayer` can use some spare time left unused from the other tasks, and the Audio/Video delay is unnoticeable (see "Reclaiming, loaded system").

The experiment has been repeated with parallel and sequential reclaiming, obtaining identical results. Hence, in this specific case using one policy instead of the other does not bring any particular advantage.

Notice that the scheduling parameters (reservation period and maximum budged) of the `rt-app` real-time tasks have been dimensioned so that no deadline is missed. During the experiments, it has been verified that the number of missed deadlines for such tasks is actually 0, even when the reclaiming mechanism is enabled.

8 Conclusions

In this paper, we proposed two different reclaiming mechanisms for real-time tasks scheduled by G-EDF on multiprocessor platforms, named *parallel* and *sequential reclaiming*. After proving their correctness, we described their implementation on the Linux OS, and compared their performance on synthetic experiments. Parallel reclaiming requires more approximations in its implementation, however, in average it performs better than the sequential reclaiming. On the other hand, sequential reclaiming can guarantee the real-time schedulability of a large number of tasks, as it allows to use a different admission test, and is characterised by a simpler implementation. However, it performs slightly worse in average. In the future we plan to conduct further investigations comparing the two strategies, and to use more advanced admission tests.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13. IEEE, 1998.

[2] L. Abeni, J. Lelli, C. Scordino, and L. Palopoli. Greedy CPU reclaiming for SCHED_DEADLINE. In *Proceedings of the Real-Time Linux Workshop (RTLWS)*, Dusseldorf, Germany, October 2014.

[3] L. Abeni, G. Lipari, A. Parri, and Y. Sun. Parallel and sequential reclaiming in multicore real-time global scheduling. Technical report, arXiv, December 2015.

[4] S. K. Baruah, J. Goossens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessor platform. In *IEEE Real Time Technology and Applications Symposium*, pages 154–163, 2002.

[5] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time systems*, 2(4):301–324, 1990.

[6] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 328–335. IEEE, 2004.

[7] M. Bertogna and S. Baruah. Tests for global EDF schedulability analysis. *Journal of Systems Architecture*, 57(5):487–497, 2011. Special Issue on Multiprocessor Real-time Scheduling.

[8] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proc. 17th Euromicro Conf. Real-Time Systems (ECRTS 2005)*, pages 209–218. Published by the IEEE Computer Society, 2005.

[9] B. B. Brandenburg and J. H. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 61–70, Pisa, July 2007.

[10] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 295–304. IEEE, 2000.

[11] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the Synthesis of Multiprocessor Tasksets. In *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, Brussels, Belgium, July 2010.

[12] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25(2-3):187–205, 2003.

[13] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the international workshop on operating systems platforms for embedded real-time applications (OSPERT)*, 2011.

[14] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, pages n/a–n/a, 2015.

[15] C. Lin and S. Brandt. Improving soft real-time performance through better slack reclaiming. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 12–pp. IEEE, 2005.

[16] G. Lipari. *Resource reservation in real-time systems*. PhD thesis, PhD thesis, Scuola Superiore S. Anna, Pisa, Italy, 2000.

[17] G. Lipari and S. K. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proc. 12th Euromicro Conf. Real-Time Systems Euromicro RTS 2000*, pages 193–200. IEEE, IEEE Computer Society, 2000.

[18] L. Nogueira and L. M. Pinho. Capacity sharing and stealing in dynamic server-based real-time systems. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[19] R. Pellizzoni and M. Caccamo. M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms. *Real-Time Systems*, 40(1):117–147, 2008.

[20] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Photonics West'98 Electronic Imaging*, pages 150–164. International Society for Optics and Photonics, 1997.