Migrate when necessary: toward partitioned reclaiming for soft real-time tasks

Houssam Eddine Zahaf, Giuseppe Lipari, Luca Abeni, Houssam-Eddine Zahaf

Migrate when necessary: toward partitioned reclaiming for soft real-time tasks

Houssam-Eddine Zahaf, Giuseppe Lipari & Luca Abeni

October 7, 2017

Abstract

This paper presents a new strategy for scheduling soft real-time tasks on multiple identical cores. The proposed approach is based on partitioned CPU reservations and it uses a reclaiming mechanism to reduce the number of missed deadlines. We introduce the possibility for a task to temporarily migrate to another, less charged, CPU when it has exhausted the reserved bandwidth on its allocated CPU. In addition, we propose a simple load balancing method to decrease the number of deadlines missed by the tasks. The proposed algorithm has been evaluated through simulations, showing its effectiveness (compared to other multicore reclaiming approaches) and comparing the performance of different partitioning heuristics (Best Fit, Worst Fit and First Fit).

Keywords: Resource reservations, partitioned scheduling, CPU reclaiming

1 Introduction

Many mobile appliances (smart-phones, tablets, smart-watches, etc.) feature variable workloads which may contain (soft) real-time tasks (audio, video, real-time communication, gaming, etc); they have strong requirements in terms of energy consumption, but also of latency, memory etc. They are equipped with multicore processors, and run with a general purpose OS (e.g. Android, which is based on the Linux kernel, or iOS).

These OSs generally provide some kind of real-time support for time sensitive applications. For example, the Linux kernel provides three real-time scheduling policies: `SCHED_FIFO` and `SCHED_RR` which are based on fixed priorities and are standardized by POSIX; and `SCHED_DEADLINE` [1], which provides resource reservation on top of Earliest Deadline First (EDF). Thanks to the *temporal isolation* property, the latter is particularly useful when mixing real-time and non real-time workloads in the same system, as required by the appliances mentioned above. From our experience, the typical workload executed in these systems consists of a few hard real-time tasks with relatively small utilizations, and a certain number of soft real-time tasks with variable execution time.

These scheduling policies are highly configurable, and can be used to implement either *global scheduling* or *partitioned scheduling* by properly setting the tasks' *CPU affinities* or using the `cpuset` mechanism[1].

From a logical point of view, in global scheduling all tasks are ordered by priority in a single global queue, and the m highest priority tasks are executed on the m available processors. Due to the intrinsic nature of the global queue, tasks may *migrate* from one processor to another one. Task migration enables automatic load balancing across CPUs, however it is a possible source of overhead: in addition to the cost of the context switch, tasks may have to reload the content of at least the L1 cache; in turns this may cause additional contention on the shared memory bus, cache evictions for the other tasks, increased execution times, etc. Moreover, global EDF and global FP algorithms do not always achieve a high schedulable utilization.

In partitioned scheduling, every task is allocated on a processor and it is not allowed to migrate. Partitioned scheduling may achieve a higher schedulable utilization than global EDF and global fixed priorities, and it is simpler to implement in an OS kernel. However, pure (and static) partitioned scheduling performs poorly in presence of highly dynamically workloads, where tasks have variable execution times and can dynamically enter and leave the system. In some extreme cases, a re-allocation of tasks to processors (*load balancing*) might be needed every time the system load changes (a task enters (or leaves) the system).

Recently, it has been shown that semi-partitioned scheduling can achieve in practice very high (quasi-optimal) schedulable utilization [3], even in presence of dynamic workloads [4]. Using semi-partitioned scheduling, tasks are split in two or more sub-tasks, and every sub-task is statically assigned to a CPU / core. This solution has been shown to work well in practice; however, to enforce precedence constraints across sub-tasks of the same task, the task algorithm assigns short relative deadlines to the sub-tasks and imposes that a sub-task cannot start executing before the absolute deadline of the preceding one. This complicates the scheduling algorithm and the schedulability analysis (and hence admission control of newly incoming tasks).

When considering Open Systems running soft real-time tasks, it may be more convenient to avoid the complexity of task splitting altogether, and use other simpler techniques, such as resource reservation and reclaiming. We focus on this proposal because our goal is to support mixed workloads (composed by both hard and soft real-time tasks): resource reservation can be used to schedule both hard and soft real-time tasks [5] and a reclaiming mechanism can improve the performance of soft real-time tasks.

Resource Reservation and reclaiming Resource reservations are used to provide temporal isolation and individual real-time guarantees to hard and soft real-time tasks. In the resource reservation framework [6], every task is assigned

[1] In theory, CPU affinities can even be used to implement even more complex configurations, but this setup is harder to analyze [2].

a reservation period and it is reserved a percentage of the computational bandwidth: if its resource requirements never exceed the reserved bandwidth, and by properly assigning the reservation period, the task is guaranteed to never miss its deadlines.

On the other hand, we can also reserve bandwidth based on the average computational requirements of the (soft) real-time tasks. In this case, a task may occasionally miss its deadline. However, we can further opportunistically reduce the percentage of deadline misses by reclaiming the unused bandwidth in the system.

There is a vast literature on reclaiming for resource reservation systems in single processor systems. The problem has been treated much less extensively in multiprocessor systems. Currently, Linux supports resource reservations with the SCHED_DEADLINE policy (both global and partitioned) and implements the sequential M-GRUB reclaiming strategy [7] that, as will be shown in Section 7.2, can result in a large number of tasks' migrations.

Our approach In this paper, we investigate the use of *partitioned resource reservations* together with a CPU reclaiming mechanism and a limited (temporary) task migration, plus a simple load balancing algorithm. The simplest way to combine CPU reclaiming with partitioned resource reservations is to statically assign tasks/reservations to CPUs, and to perform per-CPU (local) reclaiming only. This is basically what we get by using the single processor GRUB algorithm [8]. Such a simple strategy is ineffective when a soft real-time task needing more than what has been reserved is placed on a CPU where there is no unused bandwidth, whereas unused bandwidth is available on other CPUs.

On the other end of the spectrum, we can use resource reservations with global EDF scheduling and an extension of the GRUB algorithm for global scheduling. This solution has been investigated in [7]: two reclaiming algorithms have been proposed, the *sequential reclaiming* (where unused bandwidth is collected on a per-CPU basis) and *parallel reclaiming* (where unused bandwidth is collected at the global level). The latter is very similar to the M-CASH algorithm proposed by Pellizzoni and Caccamo [9].

The solution investigated in this paper is a trade-off between these two extremes. We use partitioned reservations (tasks / reservations statically assigned to CPUs / cores) and global reclaiming (achieved through a temporary migration mechanism). We complement the temporary migration mechanism with a load balancing algorithm to better tolerate large variations in the workload.

Organization The paper is structured as follows. In Section 2, we discuss the state of the art in multiprocessor real-time scheduling. In Section 3 we present the system model. In Section 4 we recall the Greedy Reclamation of Unused Bandwidth (GRUB) algorithm. Section 5 presents our proposed framework and it is the core of our paper: we describe the migration conditions, and we present the proof that the migration will not affect the correctness of temporal isolation. Section 6 describes the partitioning and load balancing techniques used when

temporary migrations are not sufficient to provide good real-time performance. Section 7 is reserved to the experimentation and discussions. We conclude in Section 8.

2 State of the art

Optimality In real-time scheduling theory, all optimal scheduling algorithms for multiprocessor systems (for example, RUN [10], QPS [11], etc.) belong to the class of global scheduling algorithms. They are optimal in the sense that, under certain conditions, they can achieve full utilization of the platform without any deadline miss. From a practical point of view these algorithms result to be complex to implement, because they require tight synchronization among processors. Furthermore, they all produce *frequent task migrations* which may increase tasks' execution times.

Partitioning On the other end of the spectrum we have partitioned scheduling. Partitioned scheduling is simpler to implement as each processor can be considered as a single independent computing resource. Also, if each task is allocated on one processor, it cannot migrate, thus reducing the scheduling overhead. Unfortunately, optimal partitioning is a NP-hard problem (equivalent to knapsack).

Recently, semi-partitioned scheduling has emerged as an interesting alternative to global and partitioned scheduling. In semi-partitioned scheduling, most of the tasks are partitioned among processors, and a few tasks are *split* in multiple parts (with each part executing on on a different processor). In comparison with global scheduling, semi-partitioned scheduling permits to reduce and keep under control the number of migrations; semi-partitioned scheduling also shows a better utilization factor when compared with pure partitioned scheduling. Brandenburg and Gl [3] have shown that the use of semi-partitioned scheduling coupled with a slack reclaiming strategy [12] in practice allows to achieve very high utilization factors.

Task splitting is usually performed on a static task set before run-time, thus this technique cannot be easily used on-line on a dynamically varying task set. Recent work [3, 4] addresses this issues by performing task splitting on dynamically arriving tasks.

Soft Real-Time scheduling Since respecting all of the tasks' deadlines on multi-processor systems is a complex problem, some scheduling algorithms focus on providing tardiness / lateness guarantees to soft real-time tasks. Many of these algorithms are based on global scheduling, that has been shown to have some optimality properties for soft real-time tasks [13, 14].

Scheduling of soft and hard real-time tasks has been mixed by using partitioned scheduling for hard tasks and global scheduling for soft tasks (and scheduling soft tasks in background respect to hard tasks) [15]. In this work,

a reclaiming mechanism similar to CASH has been used to improve the performance of soft real-time tasks.

Finally, it has been noticed that semi-partitioned scheduling is a good choice for soft real-time tasks too [16].

As explained, in this paper we focus on a different approach, based on partitioned scheduling (with some possible load balancing) and CPU reclaiming associated to temporary migrations to reduce the number of migrations.

Resource reclaiming Many reclaiming algorithms exists for reclaiming unused bandwidth on single processors [17, 12, 8]. In particular, we cite here two reclaiming mechanisms based on EDF. The CASH (Capacity Sharing) [12] algorithm utilizes a queue of unused capacities, each capacity is a pair of budget and deadline. When a job of a periodic task finishes, the residual budget together with the task absolute deadline are inserted in the reclaiming queue. When a task executes, in addition to its own budget, it can use all budgets in the reclaiming queue with deadline no greater than the task's deadline. The algorithm is simple and effective, however it can only reclaim the unused budget released by periodic tasks.

GRUB (Greedy Reclamation of Unused Bandwidth) [8] uses a different scheme based on utilization. The algorithm keeps track of the bandwidth of the active tasks in the system: when a task executes, the budget is decreased by an amount proportional to the free available bandwidth. GRUB can be effectively used with periodic and sporadic tasks, however it can only treat periodic reservations with relative deadline equal to period. The techniques based on this paper are based on GRUB (see Section 4.)

M-CASH is a reclaiming mechanism for multicore systems with global EDF scheduling [9]. It uses CASH for periodic tasks, and an utilization based algorithm similar to GRUB for sporadic tasks and for reclaiming the extra free bandwidth in the system.

The authors of [7] propose two different reclaiming mechanism for global EDF based on GRUB: parallel reclaiming is a global reclamation scheme (similar to the sporadic mechanism proposed in M-CASH) where all available unused bandwidth is stored in a single global variable accessible by all cores. Sequential reclaiming removes this constraint by storing the unused bandwidth on a per-core basis. Unfortunately, due to the global nature of the scheduler, both M-CASH, Parallel and Sequential reclaiming schemes suffer from a relatively large amount of task migrations.

3 System model

A real-time task τ_i is a (possibly infinite) sequence of jobs $\mathcal{J}_{i,k}$. Each job $\mathcal{J}_{i,k}$ arrives at time $\mathsf{a}_{i,k}$ and is expected to complete its execution before time $\mathsf{a}_{i,k}+\mathsf{D}_i$, where D_i is the *relative deadline* of the task. A job *misses its deadline* if the job finishing time $\mathsf{f}_{i,k}$ is greater than $\mathsf{a}_{i,k} + \mathsf{D}_i$. A task is *periodic* if the arrival time between two consecutive jobs is fixed to T_i, called period ($\mathsf{a}_{i,k} = \mathsf{a}_{i,0} + K \cdot$

T_i). Sporadic tasks relax the last constraint so the arrival time between two consecutive jobs is equal or greater than the task's period T_i ($\mathsf{a}_{i,k+1} - \mathsf{a}_{i,k} \geq \mathsf{T}_i$). In this paper we assume that all tasks have deadline equal to their period (or their minimum inter-arrival time). Let $\mathsf{C}_{i,k}$ denote the execution time of job $\mathcal{J}_{i,k}$: its exact value is not know before the execution of the job.

A *server* is a scheduling abstraction that is used by the scheduler to provide temporal isolation between different tasks. In the resource reservation framework, each task is assigned a server $\mathcal{S}_i$ characterized by a budget Q_i and a period P_i. The resource reservation algorithm guarantees that the task can execute for a minimum amount of time Q_i every period P_i. The ratio $\mathsf{Q}_i/\mathsf{P}_i$ is the *server utilization* u_i: it represents the fraction of CPU time reserved to τ_i. The total CPU utilization U of a set of servers is simply the sum of all server utilization's $\mathsf{U} = \sum_{i=1}^{n} \mathsf{u}_i$. When using partitioned scheduling, tasks and servers are allocated to CPUs/cores; we denote by $\mathcal{T}_j$ the set of tasks allocated on core j, and U_j the total utilization of $\mathcal{T}_j$.

In this paper we consider an identical multicore platform: all cores have the same characteristics (architecture, micro-architecture, ...) and share the same fixed operating frequency.

4 The GRUB algorithm

In this section, we recall the GRUB (Greedy Reclamation of Unused Bandwidth) algorithm [8] and its main properties. Therefore, in this section we restrict to single processor scheduling.

Let τ_i be a task of $\mathcal{T}$. All jobs of task τ_i are executed according to their arriving order: job $\mathcal{J}_{i,k}$ cannot start executing until job $\mathcal{J}_{i,k-1}$ has finished. Each task is assigned to a server $\mathcal{S}_i$ which is characterized by two parameters: the server bandwidth u_i and the server period P_i.

The run-time behavior of each server $\mathcal{S}_i$ is described by two state variables, that are computed and updated at run-time: the server deadline d_i and the virtual time V_i. The urgency of each job of τ_i depends on its server's deadline, hence jobs of $\mathcal{T}$ are scheduled according to their earliest server's deadline (EDF-like policy). Please notice that the task's deadline D_i and the server's deadline d_i are two different entities (D_i is a *relative* deadline, while d_i is an *absolute* deadline). Also, please notice that the absolute deadline of job $\mathcal{J}_{i,k}$ is $\mathsf{a}_{i,k} + \mathsf{D}_i$ and it may or may not coincide with the server's deadline d_i. A global variable U^a stores the current *active load* on the processor. U^a is initialized to 0.

The GRUB algorithm guarantees that all the servers' deadlines are respected (each server $\mathcal{S}_i$ allows its task τ_i to execute for Q_i time units before d_i) if Equation (1) (EDF schedulability) is respected:

$$\sum_i \mathsf{u}_i \leq 1. \tag{1}$$

The dynamic evolution of $\mathcal{S}_i$ is described by the finite state machine shown in Figure 1.

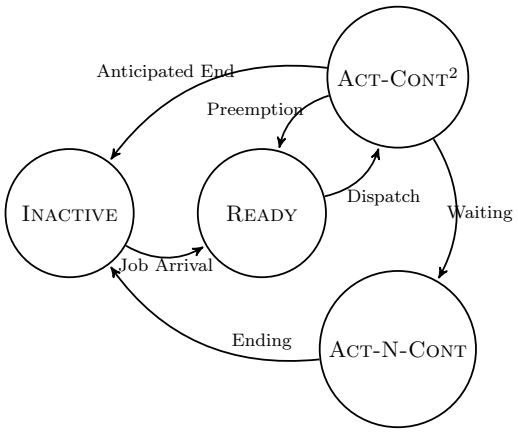

Figure 1: Grub Control automaton

Ready state Server $\mathcal{S}_i$ is in Ready state when there is at least a pending job and server's deadline d_i is not the earliest.

When the server moves from Inactive to Ready because a new job has arrived at time t, the server's virtual time V_i and deadline d_i are updated as follows:

$$\mathsf{V}_i \leftarrow \mathsf{a}_{i,k} \tag{2}$$

$$\mathsf{d}_i \leftarrow \mathsf{V}_i + \mathsf{P}_i \tag{3}$$

Also, its utilization must be added to the utilization of the active tasks:

$$\mathsf{U}^a \leftarrow \mathsf{U}^a + \mathsf{u}_i \tag{4}$$

where U^a is the total utilization of the active servers.

A server $\mathcal{S}_i$ in Ready state goes into Executing state when its deadline is the earliest among all ready servers.

Executing state While server $\mathcal{S}_i$ is in Executing state for Δt units of time, its virtual time V_i is updated as follows:

$$\mathsf{V}_i(t + \Delta t) \leftarrow \mathsf{V}_i(t) + \frac{\mathsf{U}^a}{\mathsf{u}_i}\Delta t \tag{5}$$

If, while in Executing, the server's deadline becomes equal to its virtual time, the deadline is postponed to:

$$\mathsf{d}_i \leftarrow \mathsf{V}_i(t) + \mathsf{P}_i \tag{6}$$

Notice that deadline postponing can decrease the relative priority of the server, that can then be preempted by other more urgent servers that are present in the ready state.

When a job ends its execution and there are no more pending jobs of the same task, its server can move out of the EXECUTING state: if its virtual time is greater than t, it moves to ACT-N-CONT state; otherwise, it goes to INACTIVE.

Act-N-Cont state When the server is in ACT-N-CONT state, even if no job is pending we still consider it as *active*, because its virtual time is in the future, which means that it has already consumed all its reserved bandwidth until time $\mathsf{V}_i t$. Hence, its utilization is still accounted for in the active utilization U^a.

In this state, the server does not update its variables: when t reaches V_i, the server moves to the INACTIVE state.

If a new job arrives while the server is in ACT-N-CONT state, then the server moves to state READY without updating its variables.

Inactive state In this state, the server has no pending job to serve. Every time the server enters this state, the active bandwidth is updated accordingly:

$$\mathsf{U}^a \leftarrow \mathsf{U}^a - \mathsf{u}_i \tag{7}$$

4.1 Properties

In this section we recall some of the properties of the GRUB server that will be useful later for our proof of correctness. We start by defining the notion of *active server*. A server is active if it is in any of the states READY, ACTIVE-CONTENDING, ACTIVE-NON-CONTENDING. Only active servers contribute with their bandwidth to variable U^a and hence can impact the amount of available reclaiming. The following property holds.

Lemma 1. *Given a single processor system scheduled by the GRUB algorithm, and let* $\mathsf{U}^a(t)$ *be the sum of the bandwidth of all* active *servers in the system at time* t*.*

If $\forall t \;\; \mathsf{U}^a(t) \leq 1$*, then no server misses its scheduling deadline, regardless of the behavior of the served tasks.*

Proof. The proof can be found in [18]. □

Notice that Lemma 1 does not require a static set of servers: a server can enter the system at any time, and it will not impact the guarantees on the already existing servers, as long at the total active bandwidth does not exceed 1 at any instant.

A consequence of the previous Lemma is that, to guarantee that no server ever misses its deadline, we can limit the sum of the bandwidths of all servers currently in the system, whether active or not.

Corollary 1. *If the sum of the utilization of all the servers currently assigned to the processor does not exceed 1, no server deadline is missed.*

Proof. Since $\forall t \;\; \sum_i \mathsf{U}_i \leq \mathsf{U}^a$, the corollary simply descends from Lemma 1. □

4.2 Advantages of GRUB

Compared to other reclaiming algorithms (e.g. [12, 17]), GRUB has some advantages:

- It can be used with periodic, sporadic and aperiodic non-real-time tasks;
- It automatically reclaims the unreserved bandwidth;
- It has the same complexity as the CBS algorithm [5];
- It keeps track of the *active bandwidth* on the processor; this property will be used for *temporary migration* (see Section 5);
- It can be used to lower the processor frequency (DVFS) in order to reduce energy consumption [19].

The GRUB algorithm has already been extended to multicore systems. In [9], an algorithm similar to GRUB is used to reclaim global unused bandwidth in the system; in [7], *parallel reclaiming* and *sequential reclaiming* strategies based on GRUB have been proposed. In both cases, the underlying scheduling algorithm is global EDF. In this paper, we address partitioned EDF scheduling.

5 GRUB and Migrations

From now on, we assume a platform consisting of m identical CPUs. We propose to partition a set of GRUB servers into the m available CPUs. Each CPU has its own ready queue and a single-processor scheduler based on EDF. Servers are partitioned using one of the classical bin-packing heuristics available in the literature.

We assume that tasks (and the corresponding servers) may enter and leave the system at any time. Suppose a new task τ_i enters the system at time t, with server parameters $\mathcal{S}_i = (\mathsf{Q}_i, \mathsf{P}_i)$. Then, the heuristic allocation algorithm is run to select the CPU on which the server will be allocated. The heuristic uses an *admission control* test to find the most suitable core where to allocate the server. The allocation algorithm will be described in Section 6.1.

After partitioning, tasks are scheduled on each CPU using the GRUB algorithm. Additionally, tasks can temporarily migrate to other CPUs to reclaim extra bandwidth. For every CPU, the algorithm maintains the following variables:

- U_j is the total utilization of the servers allocated on core j; this *does not* include the tasks that have been temporarily migrated on j.
- $\mathsf{U}_j^m(t)$ is the total bandwidth at time t of the tasks that have been temporarily migrated on core j.
- $\mathsf{U}_j^a(t)$ is the total *active* utilization of all the active servers on core j at time t; this may include the bandwidth of the active tasks that have temporarily been migrated on j.

Each server is assigned an additional parameter, the *migrating utilization* u_i^m. The migrating utilization is used to distribute the unused bandwidth on the destination core to the (possibly many) incoming servers. As we will see in the next section, by setting its migrating utilization to 0, temporary migration is disabled for a given task.

5.1 Temporary Migrations

Suppose that task τ_i is allocated on core j and it is served by $\mathcal{S}_i$. Consider job $\mathcal{J}_{i,k}$: it starts executing on core j according to the original GRUB algorithm. Suppose that at some time t_0 the virtual time $\mathsf{V}_i(t_0)$ of the server becomes equal to the server's deadline d_i: the job has consumed all its budget and can not reclaim the unused bandwidth of the other tasks allocated to core j.

In the original GRUB algorithm, the server deadline is postponed; in the partitioned version the job is declared as *eligible for migration*. This gives $\mathcal{J}_{i,k}$ a chance to continue its execution with the same server deadline (same urgency) on a different core j'. In the following discussion, we denote core j' as the *destination core*.

A job is eligible for migration at time t_0 when the following two conditions are verified:

$$\mathsf{V}_i(t_0) \geq \mathsf{d}_i \tag{8}$$

$$\mathsf{d}_i > t_0 \tag{9}$$

If (8) is true, task τ_i cannot continue to execute on its current core without postponing the current server deadline d_i, hence we may try to migrate it to a different core. If Condition (9) is not verified, the current server deadline d_i has been reached, so there is no point in migrating the task.

When a job is eligible for migration, the algorithm first selects the destination core j' as the one with the smallest active utilization $\mathsf{U}_{j'}^a$, because on j' there is more chance for the task to reclaim extra bandwidth.

Then we must check that the migrating job fits in the destination core with its migrating utilization:

$$\mathsf{u}_i^m + \mathsf{U}_{j'}^m(t) + \mathsf{U}_{j'} \leq 1 \tag{10}$$

Recall that bandwidth $\mathsf{U}_{j'}$ is the total bandwidth of all servers that are allocated on j', whereas bandwidth $\mathsf{U}_{j'}^m(t)$ is the total active bandwidth of the jobs that have temporarily been migrated to j' at time t. In practice, Equation (10) guarantees that we are not overloading the destination core with too many migrating jobs.

If Equation (10) is not respected for the selected core j', we can still try to migrate the job by reducing its migrating utilization u_i^m to:

$$\mathsf{u}_i^{m\prime} = \min\{\mathsf{u}_i^m, 1 - (\mathsf{U}_{j'} + \mathsf{U}_{j'}^m(t)\} \tag{11}$$

The task is migrated by creating a new server $\mathcal{S}_i'$ on the destination core j' with:

- the same server period P_i,
- utilization equal to $\mathsf{u}_i^{m\prime}$ as computed by Equation (11);
- the server state is initialized to READY;
- its virtual time is initialized to t_0 and the server deadline is the same as d_i.

Task τ_i is served by the new temporary server $\mathcal{S}_i'$ until the current job $\mathcal{J}_{i,k}$ completes. This temporary server is managed by the GRUB algorithm like a regular server.

Once the job completes, the task returns immediately to its original processor and the temporary migration is concluded: the next jobs of the task will start executing on the original processor. However, the temporary server $\mathcal{S}_i'$ is not immediately deleted: when the job completes, it follows the rules of the GRUB algorithm, and if $\mathsf{V}_i(t) > t$, the server first moves to state ACT-N-CONT; later, when $\mathsf{V}_i(t) = t$, it moves to INACTIVE and it can be deleted from the system. Notice that a temporary server remains active during its lifespan.

Bandwidth $\mathsf{U}_{j'}^m(t)$ keeps track of the total bandwidth of all tasks that are temporarily migrated to core j': it is incremented by $\mathsf{u}_i^{m\prime}$ when the temporary server is created, and it is decremented by the same amount when the temporary server becomes INACTIVE and it is deleted.

Algorithm 1 modifies the original GRUB algorithm to take into account job migration. When V_i reaches d_i (Line 3 of the algorithm, implementing Condition 8), instead of simply postponing the server deadline as in the original GRUB, Algorithm 1 tries to migrate the task by first selecting a destination core (Line 4). The algorithm uses a boolean flag per each task, denoted as $\texttt{migrated}_i$, initialized to `false`. If the task has not yet been migrated (condition at Line 5), to ensure that the selected core has enough bandwidth to accommodate for the incoming task, we update the migrating utilization (Line 7). At Line 8, we further limit the number of migrations by using an appropriate threshold ϵ to guarantee that the task will be able to execute enough time on the target core, otherwise the overhead of migration can overcome its benefits.

If the current job has already been migrated ($\texttt{migrated}_i$ is true) or if it has not the possibility to execute for enough time on the target core (the test at Line 8 is false), then the server deadline is postponed (Lines 19 and 22). The migration flag is set to limit the number of possible migrations of the same job, and it is reset to `false` once the job completes.

5.2 Proofs of correctness

In this section we discuss the correctness of our partitioned GRUB algorithm. In particular, we prove that the temporal isolation property is still valid when we introduce the temporary migration mechanism.

We start by observing that, on each core, we execute an instance of the GRUB algorithm independent of the others. Therefore, as long as no migration

Algorithm 1 Temporary Migration

```
1: Classic_Grub_scheduling_Code
2:                 ...
3: if (V_i ≥ d_i) then
4:     P^m = selectDestinationCore()
5:     if (not migrated_i) then
6:         U_j = U_{P^m} + Σ u_a^m
7:         u_i^m = min(u_i^m, 1 − U_j)
8:         if (u_i^m(d_i − t)/(u_i^m + U_m^a) > ε) then
9:             Create temporary server S'_i = (u_i^m, P_i)
10:            Assign task τ_i to S'_i
11:            migrated_i = true
12:            d'_i = d_i
13:            V'_i(t) = t
14:            State of S'_i = READY;
15:            DoMigration();
16:        else
17:            migrated_i = false
18:            d_i = V_i(t) + P_i
19:        end if
20:    else
21:        migrated_i = false
22:        d_i = V_i(t) + P_i
23:    end if
24: end if
25:                ...
26: Classic_Grub_scheduling_Code
```

is allowed, we can use Equation (1) as an admission control for every core. Let us now consider the case of a temporary migration.

Lemma 2. *Assume that task τ_i is migrated from core j to core j' at time t_0. Then,*

$$\forall t > t_0, \;\; \mathsf{U}_j^a(t) \leq 1 \;\; \wedge \;\; \mathsf{U}_{j'}^a(t) \leq 1$$

Proof. Assume that, before migration, the active bandwidths on core j and j' are not greater than 1:

$$\forall t < t_0, \;\; \mathsf{U}_j^a(t) \leq 1 \;\; \wedge \;\; \mathsf{U}_{j'}^a(t) \leq 1$$

This property is clearly true before the first migration: we want to prove that it remains true after each migration.

The lemma is trivially true on core j: in fact, the original server of the task is not deleted: it remains initially in ACTIVE-NON-CONTENDING and then it may move to INACTIVE: in any case, the active bandwidth on core j cannot not increase due to the migration.

Regarding core j', by definition:

$$\forall t < t_0, \;\; \mathsf{U}^a_{j'}(t) \leq \mathsf{U}_{j'} + \mathsf{U}^m_{j'}(t) \tag{12}$$

i.e. the total active bandwidth does not exceed the sum of the bandwidth of the tasks allocated on j' and the bandwidth of the temporary servers on j'.

At time t_0, a *temporary server* is created on core j' with server utilization calculated according to Equation (11). Notice that the temporary server is active for the duration of its lifespan, and hence its utilization is immediately summed to the active utilization on core j'. Let us denote by t_0^- the instant before the migration takes place:

$$\mathsf{U}^a_{j'}(t_0) = \mathsf{U}^a_{j'}(t_0^-) + \mathsf{u}_i^{m\prime} \leq \mathsf{U}^a_{j'}(t_0^-) + 1 - (\mathsf{U}_{j'} + \mathsf{U}^m_{j'}(t_0^-)) \leq 1,$$

the last inequality is verified by substituting Equation (12).

Furthermore, $\mathsf{U}^m_{j'}(t_0) = \mathsf{U}^m_{j'}(t_0^-) + \mathsf{u}_i^{m\prime}$, and $\mathsf{U}_{j'} + \mathsf{U}^m_{j'}(t_0) \leq 1$. By definition of the GRUB algorithm, $\mathsf{U}^a_{j'}(t) \leq \mathsf{U}_{j'} + \mathsf{U}^m_{j'}(t)$, so the active bandwidth cannot exceed 1 after the migration.

Finally, notice that, once the migrated job has finished, the server is deleted only when its state becomes INACTIVE: at that point, its bandwidth is subtracted both from $\mathsf{U}^m_{j'}(t)$ and from $\mathsf{U}^a_{j'}(t)$, so $\mathsf{U}^a_{j'}(t) \leq \mathsf{U}_{j'} + \mathsf{U}^m_{j'}(t)$ even after the migration is over.

Hence the lemma is proved. □

Theorem 1. *Let us assume a multicore platform with m cores, and a set of servers partitioned on the m cores such that:*

$$\forall j = 1, \ldots, m \quad \mathsf{U}_j \leq 1$$

Then, when scheduled by partitioned GRUB with temporary migration, no server misses its scheduling deadline.

Proof. It descends from Lemma 2 and from Lemma 1. □

5.3 Short server periods

In general, job deadline $\mathsf{a}_{i,k} + \mathsf{D}_i$ can be different than the server deadline d_i. In fact, in many cases it is useful to set the server period P_i equal to a divisor of the task's period T_i, $\mathsf{T}_i = k\mathsf{P}_i$. For example, in the adaptive reservation framework described in [20, 21] the authors suggest to use such technique to improve the performances of the adaptive control mechanism.

In this case, when the virtual time $\mathsf{V}_i(t_0)$ reaches the current server deadline d_i, the job deadline $\mathsf{a}_{i,k} + \mathsf{D}_i$ can still be far in the future and a migration may be not strictly needed: the job has still a chances to meet its deadline without migrating.

For the sake of simplicity in this paper we only describe the case when the server's period is equal to the task period. However, it is easy to modify our algorithm to account for this case by adding an additional server parameter MINPOST that imposes a minimum number of deadline postponing before making the task eligible for migration. This will be the subject of a future work.

6 Permanent Migrations

Temporary migration is useful for reclaiming extra bandwidth. However, it may happen that a task still suffers too many deadline misses even with temporary migration enabled. When a task consistently misses too many deadlines, it can be useful to permanently migrate it to a different core, using some kind of load balancing mechanism. More specifically, if more than a specified amount of jobs have missed their deadlines in an interval of time of size w_i, it may be necessary to *permanently* migrate the task in the hope to improve its quality of service.

In contrast to temporary job migration which happens while the job is running, the condition for permanent task migration is checked at the end of the execution of a job. If a task is permanently migrated, all its future jobs will start executing on the new processor.

First, we select a destination core for the task. We distinguish two cases:

- The task seeks to migrate immediately, thus the migration core must verify the following condition:

  $$\mathsf{u}_i \leq 1 - \mathsf{U}_j - \mathsf{U}_j^m$$

  If this is verified, we can immediately migrate the task with its server in the new core (see below).

- If no processor guarantees the above condition, then we can try to delay the migration to some time in the next future. In this case, we first look for a processor in which:

  $$\mathsf{u}_i \leq 1 - \mathsf{U}_j$$

  Then, we disable incoming migrations on the destination core, and we wait for $\mathsf{U}_j^m \leq 1 - (\mathsf{u}_i + \mathsf{U}_j)$. We have to wait at most for the longest response time of any migrating job: this depends on the job execution time and cannot be easily bounded a-priori. Therefore, we additionally start a timer, and if the migration bandwidth has not decreased enough within the timeout, we abort the load balancing operation.

- If none of the conditions above is verified, permanent task migration is not possible.

If permanent migration is possible, the task and the corresponding server are migrated on the destination core j' with their own variables, and $\mathsf{U}_{j'}$ is incremented. We can decrease the U_j on the original core only once the original server becomes inactive (i.e. the server virtual time $\mathsf{V}_i = t$). This can be easily achieved by setting an appropriate timer.

If multiple target cores are available as destination for permanent migrations, some "classical" heuristics such as Best Fit (BF), First Fit (FF) and Worst Fit (WF) can be used to select the destination core. The same heuristics can be used for the *allocation strategy* presented in the next section, and will be used for the simulations reported in Section 7.

The proof of correctness of this mechanism is similar to the proof of Corollary 1, and it is not reported here because of space constraints.

6.1 Allocation strategy

When a new task τ_n enters the system, the allocation strategy used to schedule it on the most suitable core is similar to the strategy used for load balancing. We first try to immediately allocate the task on one of the cores were $1 - (\mathsf{U}_j + \mathsf{U}_j^m) \geq \mathsf{u}_n$. If no core meets the condition, we select one of the cores where $1 - \mathsf{U}_j \geq \mathsf{u}_n$, we disable migration onto the selected core and we wait for the migrating utilization to decrease. If more than one possible target core is found, we select one according to one of the heuristics mentioned above (BF, FF, or WF). When a task leaves the system, its servers utilization is subtracted from the processor utilization only when the server becomes INACTIVE.

If none of the cores guarantees that $1 - \mathsf{U}_j \geq \mathsf{u}_n$, or if the maximum timeout for decreasing the migrating utilization expires, the incoming task is rejected. In this case, the user may decide to assign the task a smaller u_n, of course losing in quality of service, or to kill some other running task to make place for the new one.

7 Experimental Evaluation

The performance of the proposed solution have been evaluated through an extensive set of simulations, and the most interesting results are reported in this section. The simulations have been performed by using a discrete time simulator built in SCALA that adopts a functional programming paradigm. The simulator supports both partitioned and global scheduling and can simulate both sporadic and periodic tasks. It implements the GRUB reclaiming algorithm in all its known incarnations: single-core [8], partitioned (as proposed in this paper, using various heuristics for load balancing and tasks allocation) and global [7] (either with sequential or parallel reclaiming). Global GRUB with sequential and parallel reclaiming will be referred as "G-Seq" and "G-Par" in the following.

Thanks to the usage of the functional programming paradigm and to its modular design, the simulator code can be easily extended with new reclaiming policies.

The simulator will be soon available online at `https://github.com/zahoussem/reclaiming_simulator`

7.1 Experimental Setup

Each simulation scenario consists of randomly generated task sets. First we fix the number of tasks to generate to a fixed number n and the number of cores m, and we set a target total utilization level. Then we generate a set of n servers' utilization $\{\mathsf{u}_i\}$ by using the UUNIFAST-discard algorithm [22].

Then, we generate the execution times of the jobs. Execution times are generated according to a certain probability distribution between a minimum and a maximum value. First we randomly select a minimum execution time minexec and a maximum execution time maxexec for each task as random samples from a

random variable $\mathsf{Unif}(5, 200)$ with uniform distribution. Then a sequence of job execution time is generated from a random variable between $[\mathsf{minexec}, \mathsf{maxexec}]$. In this paper we considered two different distributions: the two-level uniform distribution and the Weibull distribution.

Two-level uniform distribution takes as an additional parameter a desired budget B and a probability PM (set to 0.75 in the experiments). Then, the execution time is chosen with an uniform distribution between $[\mathsf{minexec}, B]$ with probability PM, and with an uniform distribution between $[B+1, \mathsf{maxexec}]$ with probability $1 - \mathsf{PM}$. In this way we can precisely and easily control the number of jobs with execution time larger than B. However, the two-level uniform distribution does not represent realistic workloads.

The Weibull distribution is known to be a good model of the execution time of many real-time workloads [23] because it effectively represents the statistical behavior of typical execution time profiles with long tails. The Weibull probability density function can be expressed as:

$$f(x) = \frac{k}{\lambda}\left(\frac{x-\theta}{\lambda}\right)^{k-1} e^{-\left(\frac{x-\theta}{\lambda}\right)^k}$$

where k is the *shape* parameter, λ is the *scale* parameter, and θ is the location (or displacement) parameter.

Once execution times are generated, the budget B for the task is selected so to ensure that approximately a certain percentage PM of jobs will have execution time greater than B. In both cases, the task period and the server period are computed as B/u_i. For every server, the migrating utilization has been set to 0.1.

In the simulation experiments that we performed, we did not observe statistically relevant differences between the two-levels uniform distribution and the Weibull distribution for the migration ratio and the deadline miss rate ratio. Therefore, in this paper we discuss only the results obtained using the two-level distribution.

Once the set of server is generated, we test schedulability and allocation. In particular, we test that the set can be partitioned using FF, BF and WF; and we test the schedulability of the servers with the schedulability tests for global EDF described in [7]. If any of the allocation algorithms fails, or any of the scheduling tests fails, we discard the generated task set.

In the rest of this section, the results of simulations are presented and discussed. We compare the performances of different partitioning heuristics and the parallel and sequential reclaiming techniques described in [7] according to the number of deadlines missed and the number of job and task migrations.

7.2 Impact of Temporary Migrations

To evaluate the effects of job migration, we set the number of cores to $m = 4$ and the number of tasks to $n = 25$. The total utilization ranges between $[0.5, 3]$. In fact, it is very difficult to generate task sets with high total utilization that

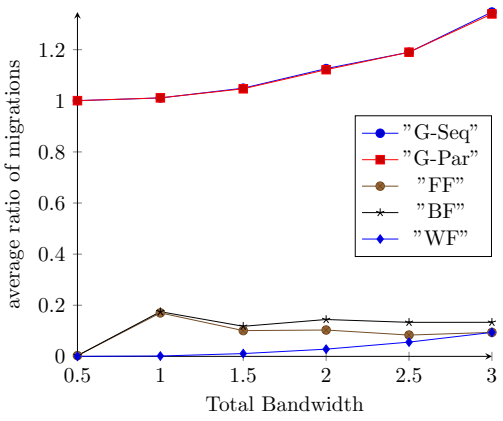

Figure 2: Migrations per job as a function of the system load.

are schedulable by all the scheduling methods considered in this comparison. For each utilization level, 100 scenarios have been generated. For our partitioned reclaiming scheme, in the figures presented in this section job migration is enabled and load balancing is disabled.

Figure 2 shows the average number of migrations per job as a function of total utilization. The figure compares the results of global parallel reclaiming (G-Par), global sequential reclaiming (G-Seq), and our reclaiming scheme with Worst-Fit (WF), First-Fit (FF) and Best-Fit (BF). As expected, the number of migrations in global algorithms G-Par and G-Seq is much larger compared to partitioned schemes.

Notice that for utilization 0.5, the reclaiming algorithm is able to respect the tasks constraints without postponing the scheduling deadlines; hence, the partitioned algorithms do not produce any migration. When the load increases, BF and FF exhibit a different behavior compared to WF: FF and BF tend to allocate the maximum number of tasks on the least possible number of cores, so this may lead to cores that are very packed while other cores are less loaded. In contrast, WF tends to spread the workload evenly across cores. Thus, in BF and FF many jobs have less opportunities to reclaim the local unused bandwidth and are more eligible to migrate compared to WF.

For example, when the utilization is 1.0, FF and BF allocate all the jobs on the first core, leaving all the other cores completely unloaded. Hence, all the tasks will migrate to reclaim CPU time from the unloaded cores. This is why for utilization 1.0 BF and FF show a large increment in the number of migrations per job. When the load increases, BF and FF start to assign tasks to the second core (then to the third and to the fourth). Since the number of tasks is fixed, when increasing the total utilization, the average utilization of every single task also increases, so BF and FF tend to distribute the load more uniformly. As a result, the number of migrations per job decreases and for utilization 3.0 the numbers of migrations for BF, FF and WF are similar.

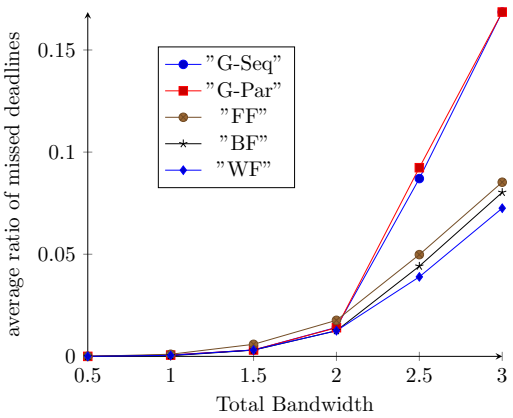

Figure 3: Average missed deadlines ratio.

For the same simulation experiment, Figure 3 shows the average ratio between number of deadline misses and total number of jobs as a function of total utilization.

At low total utilization, all scheduling techniques allow all tasks to respect their deadlines. The higher the total bandwidth, the higher is the probability of missing the deadline. However, partitioned schemes present significantly better performance. While all of the algorithms show almost zero deadline miss for utilizations up to 1.0, the partitioned schedulers cause much less migrations (in particular, WF causes 0 migrations up to utilization 1.0). This shows that with a proper partitioning the local reclaiming rule of the GRUB algorithm can reclaim enough bandwidth for all jobs to complete within their deadlines. Moreover, when the utilization grows above 2.0 the number of deadline misses with the two global schedulers is much higher than with the partitioned schedulers.

When comparing partitioned scheme against each other, again WF shows a slightly better behavior, because it can more effectively balance the load across all processor.

The confidence interval for partitioned techniques is less than ± 0.0025 and less than ± 0.0047 for the global techniques. Similar values of the confidence interval has been noticed for all figures presented in this section.

7.3 Impact of Permanent Migrations

In this section, we evaluate the performances of our heuristics when both job migration and load balancing are activated. Load balancing is useful only in partitioned schemes, so in this section we only compare the three partitioning heuristic against each other and against WF with load balancing disabled. The total utilization is varied between 0.5 to 3.5 in steps of 0.5.

Figure 4 presents the average deadline miss ratio as a function of total bandwidth. The line labeled WF-a presents the results when task migration is dis-

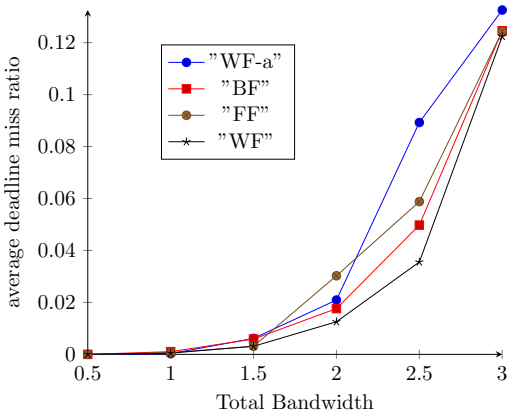

Figure 4: Average deadline miss ratio when load balancing is active ($w = 20$ jobs).

abled. In all others, the task migration is activated, monitoring the deadline miss ratio on time intervals of size equal to 20 jobs. As you can see, load balancing helps to reduce the number of deadline misses for all partitioning techniques, and once again WF is the best heuristic. When the load is very high, all cores are heavily loaded and it is more difficult to apply load balancing, so all heuristics present the same number of migrations.

Figure 5 shows the average number of migrations per job for the same set of experiments and it confirms the results of the previous figures. In particular, FF and BF show a bad behavior even at low workloads, because they tend to concentrate all load on a few processors, and hence reduce the possibility of local reclaiming, whereas WF ensures an already good load balancing in the initial allocation.

The number of migrations is greater when total bandwidth is between 1.5 and 2.5. This is due to the fact that at low utilization, all deadlines are respected. At high utilizations, all core are highly loaded and task migrations are not possible, hence all heuristics behave in the same way. At average load, tasks miss their deadline, and they have the possibility to move from a core to another one, hence the number of task migrations number is higher.

7.4 Dynamic Workloads

In this subsection we investigate the dynamic behavior of the load balancing algorithm. The scenario is generated for a platform with 2 cores (marked with a blue and a pink line, respectively). In the presented scenario, the task set has a total utilization of 1.2. The initial allocation is done using Worst Fit heuristics. A new task enters the system at time $t = 2000$ and leave it at time $t = 6000$. The new task has a high processor demands. The server utilization for the new task is set to 0.3.

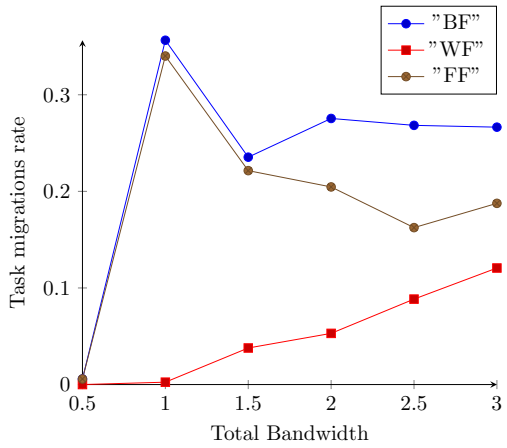

Figure 5: Migrations per job when load balancing is active.

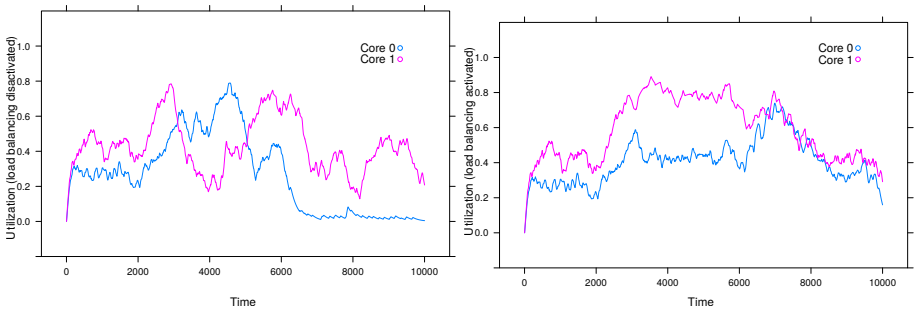

Figure 6: Active utilization when a new task is inserted at $t = 2000$, and leaves at $t = 6000$.

Figure 6 presents active utilization for every core, with load balancing enabled (left) and disabled (right). For clarity of presentation, in the plots we show the exponential average of the active utilization with ratio $1/200$.

In the case the load balancing is activated, at the beginning tasks migrate between the two cores thanks to load balancing, and the load is evenly distributed between cores (left plot). This does not happen in the right scenario where load balancing is disabled, so the pink core remains more loaded than the blue one. At $t = 2000$, when the new task enters the system, in the left scenario the load of the pink core is increased by an amount equal to the utilization of the new task, and remains high during the interval $[2000, 6000]$. On the other hand, when task migration is activated, cores start exchanging tasks in order to miss less deadlines.

In Figure 7, we present the deadline miss ratio (fraction of jobs missing their deadline in an interval of time of size w starting at time t) when the load balancing is enabled (left) and disabled (right) at each instant of time t. Before

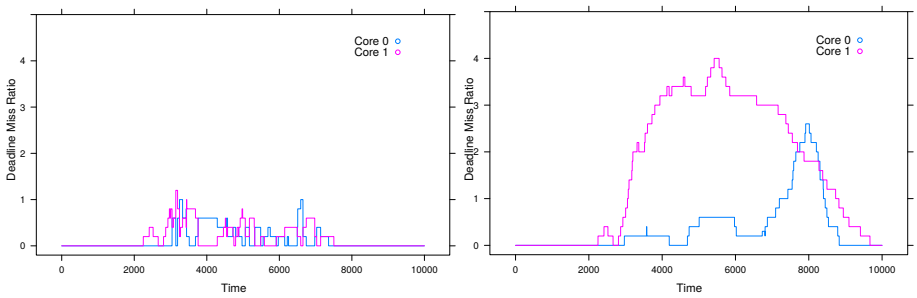

Figure 7: Deadline miss ration on each core for the task inserting scenario

task inserting, both cores have a deadline miss ratio equals to 0. When the task arrives, the deadline miss ratio increases on both cores. When the task migration is disabled, the pink core has a higher load and the tasks that are allocated there miss more deadlines compared to the blue core which has smaller load. On the other hand, when load balancing is activated, tasks can permanently migrate from the blue core to the pink one and vice-versa. Again, the load is spread more evenly, and this explains the fact that we have a smaller deadline miss ratio compared to the one where the task migration is disabled. This does not come for free: the tasks on the blue core miss slightly more deadlines compared to the case of disabled migration (figure on the right). However, as the total bandwidth is balanced in the activated task migration scenario, the gain on the blue core is quantitatively higher than the loss on the pink core.

8 Conclusion

We presented a partitioning scheduling algorithms with temporary migration for soft real-time tasks. When a task exhausts its budget, it first reclaims the unused bandwidth on the local core, and migrates only if it is necessary to reclaim extra bandwidth on the other cores. Simulation experiments show that our technique permits to greatly reduce the number of migrations with respect to global scheduling without increasing the number of deadline misses. The Worst-Fit heuristic seems to be the most effective for balancing the load across cores and to distribute the extra bandwidth.

As a future work, we plan to implement our algorithm on Linux to evaluate the overhead of the scheduler and the complexity of the temporary migration mechanism. We also plan to extend our technique to heterogeneous multi-core processors, such as the ARM big-Little, and complement it with DVFS and power management schemes, in order to reduce the energy consumption of modern mobile appliances.

9 Acknowledgement

This work has been supported in part by the Institut de recherche sur les composants logiciels et matriels pour l information et la communication avancee de Lille, IRCICA, UMR3380.

References

[1] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, Jun 2016.

[2] Arpan Gujarati and Björn B. Brandenburg. Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. *Real-Time Systems*, 51(4):440–483, 2015.

[3] B. B. Brandenburg and M. Gul. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110, Nov 2016.

[4] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. Semi-partitioned scheduling of dynamic real-time workload: A practical approach based on analysis-driven load balancing. In *To be presented at the 29th Euromicro Conference on Real-Time Systems (ECRTS17)*, June 2017.

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

[6] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *1994 Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[7] Luca Abeni, Giuseppe Lipari, Andrea Parri, and Youcheng Sun. Multicore cpu reclaiming: parallel or sequential? In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1877–1884. ACM, 2016.

[8] Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 193–200. IEEE, 2000.

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

[10] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 104–115. IEEE, 2011.

[11] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt. Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 291–300, July 2014.

[12] M. Caccamo, G. C. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, Feb 2005.

[13] U. M. C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 12 pp.–341, Dec 2005.

[14] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–320, Dec 2005.

[15] B. B. Brandenburg and J. H. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 61–70, July 2007.

[16] J. H. Anderson, J. P. Erickson, U. C. Devi, and B. N. Casses. Optimal semi-partitioned scheduling in soft real-time systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2014.

[17] Caixue Lin and Scott A 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.

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

[19] Claudio Scordino and Giuseppe Lipari. A Resource Reservation Algorithm for Power-Aware Scheduling of Periodic and Aperiodic Real-Time Tasks. *IEEE Transactions on Computers*, 55(12):1509–1522, 2006.

[20] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and John Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. 23rd IEEE Real-Time Systems Symp. RTSS 2002*, pages 71–80. IEEE Computer Society, 2002.

[21] Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, Luca Marzario, and Luigi Palopoli. QoS management through adaptive reservations. *Real-Time Systems*, 29(2):131–155, 2005.

[22] Paul Emberson, Roger Stafford, and Robert 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, 2010.

[23] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, July 2012.