Adaptive Partitioning of Real-Time Tasks on Multiple Processors

Luca Abeni
Scuola Superiore Sant'Anna
Pisa, Italy
luca.abeni@santannapisa.it

Tommaso Cucinotta
Scuola Superiore Sant'Anna
Pisa, Italy
tommaso.cucinotta@santannapisa.it

ABSTRACT

This paper presents a new algorithm for scheduling real-time tasks on multiprocessor/multicore systems. This new algorithm is based on combining EDF scheduling with a migration strategy that moves tasks only when needed. It has been evaluated through an extensive set of simulations that showed good performance when compared with global or partitioned EDF: a worst-case utilisation bound similar to partitioned EDF for hard real-time tasks, and a tardiness bound similar to global EDF for soft real-time tasks. Therefore, the proposed scheduler is effective for dealing with both soft and hard real-time workloads.

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; *Embedded systems*; • **Software and its engineering → Real-time schedulability**;

KEYWORDS

Real-time Scheduling, Real-Time Operating Systems

ACM Reference Format:
Luca Abeni and Tommaso Cucinotta. 2020. Adaptive Partitioning of Real-Time Tasks on Multiple Processors. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3373937

1 INTRODUCTION

With the evolution of hardware and CPU technologies, many modern CPUs have multiple cores, even in low-power embedded systems. It is hence becoming more and more frequent to schedule real-time applications (characterised by temporal constraints) on multiple cores or CPUs. The approaches generally used to serve these applications are based on *global scheduling*, where the scheduler is free to migrate tasks among cores/CPUs to respect some global invariant, or *partitioned scheduling*, where each task is statically assigned to one CPU on which a uniprocessor scheduler is used.

For example, the real-time scheduling policies provided by the Linux kernel (`SCHED_FIFO`, `SCHED_RR` and `SCHED_DEADLINE` [18]) can be used to schedule tasks globally across the whole platform, to partition the real-time tasks across the available CPUs, or even to partition them across clusters/islands of limited sets of CPUs where global scheduling occurs within each CPU set.

Partitioned scheduling approaches are simpler, easier to analyse (uniprocessor scheduling analysis can be easily applied) and avoid the overheads caused by tasks migrations. On the other hand, they are less robust against temporary overloads: if a core/CPU is overloaded because a task executes for more time than expected, because of an interrupt storm, or because of some other reason, the scheduling algorithm is not able to exploit the idle times possibly available on the other cores to tolerate the overload.

Moreover, partitioned scheduling can be used only if the taskset is *partitionable* (that is, if tasks can be partitioned on the available CPUs so that existing uniprocessor scheduling algorithms can properly serve the tasks on each CPU without missing deadlines).

Global scheduling approaches are more complex and difficult to analyse; for example, the schedulability analysis available for global Earliest Deadline First (EDF) or global fixed priorities is very pessimistic. On the other hand, global algorithms can better tolerate temporary overloads by migrating tasks from overloaded cores/CPUs to idle ones. Moreover, global EDF guarantees that if the total task load is less than 100% then all the tasks will have an upper bound to their tardiness [11, 26].

1.1 Related Work

Previous research focused on supporting either hard real-time systems (where all the deadlines of all the tasks have to be respected) [2, 4, 9, 19, 23] or soft real-time systems (where a controlled amount of missed deadlines can be tolerated) [11, 26].

Regarding hard real-time systems, it is known from literature [1] that global EDF can be modified to have an utilisation bound which is optimal for fixed-job-priority algorithms [4] and that optimal multiprocessor scheduling algorithms (based on global scheduling) exist [2, 9, 19, 23]. However, all these algorithms are not very used in practice, and commonly used Operating Systems focus on partitioned or global fixed-priority or EDF scheduling.

As a result, partitioned scheduling is generally preferred in hard real-time systems (where execution times are more stable and transient overloads are less likely to happen, but respecting all the deadlines is important) while global scheduling (especially global EDF) is more used in soft real-time systems (where the tardiness guarantees provided by global EDF are generally enough, but execution times are less predictable and transient overloads are more likely to happen). Some previous works [5, 17] performed empirical studies comparing the advantages and disadvantages of global vs partitioned scheduling in various contexts.

The previous works considering hard real-time systems generally focused on optimal partitioning, such as achievable via integer

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373937

linear programming techniques [22, 27]. Other works, instead, considered more dynamic real-time systems where on-line partitioning approaches are required. In this context, various authors investigated on the effectiveness of bin-packing heuristics such as first-fit, worst-fit and next-fit, which have been studied at large also in other contexts, such as memory management (see for example the seminal works by Graham [14] and Johnson [15, 16], or more recent works and comprehensive surveys on the topic[8, 25]). Many of these works focus on the *absolute approximation ratio*, the minimum number of bins that are needed to pack a number of items with different weights, when using the above mentioned simple bin-packing heuristics, in comparison to the optimum number that would have sufficed using an optimal approach. Some authors focused on the *asymptotic* value of such approximation ratio, achieved as the size of the problem grows to ∞. For example, one interesting result in the area is the $12/7 \cong 1.7143$ bound for the first-fit heuristic [8]. However, many of these works are not concerned with scheduling of real-time tasks, so they do not study the effectiveness of the mentioned heuristics on the performance, in terms of slack and/or tardiness, obtained when scheduling various real-time task sets.

1.2 Contributions

This paper focuses on the problem of scheduling independent, periodic, real-time tasks on a symmetric multiprocessor through a new migration strategy based on "adaptive partitioning", with the goal of taking the best of global and partitioned approaches. As detailed in Section 3, the resulting *adaptively partitioned EDF* scheduling algorithm uses a migration policy based on well-known heuristics to distribute tasks among cores/CPUs so that partitioned scheduling (and, in particular, partitioned EDF) can be used. This partitioning heuristic is invoked only on job arrival or termination, with the aim of ensuring that the partitioned EDF schedulability condition is met on each core/CPU, while keeping the number of migrations to the bare minimum. The adaptive partition rapidly converges to a static schedulable partitioning when possible, while it provides a bounded tardiness (like the global strategy) whenever a schedulable static task partitioning cannot be found. To achieve this property, when the partitioning heuristic is not able to place a task on a core/CPU without overloading it, the new migration algorithm falls back to a global EDF scheduling policy. However, the migration algorithm tries to restore a partitioned EDF schedulability whenever possible[1].

Adaptive partitioning can be used in both hard real-time and soft real-time systems, by simply changing the admission control: if the total utilisation is smaller than $\frac{M+1}{2}$ (where M is the number of cores/CPUs), then all the deadlines are respected, while if the total utilisation is between $\frac{M+1}{2}$ and M then the algorithm provides a bounded tardiness (this is an important difference respect to most of the previously designed schedulers for either hard or soft real-time tasks). Finally, the hard schedulability bound for the new algorithm introduced in this paper ($\frac{M+1}{2}$) is known to be optimal for fixed-job-priority algorithms [1].

[1] Although the proposed technique is a form of dynamic partitioning, the name "adaptive partitioning" is used to avoid confusion with a previous work [24] that used the term "dynamic partitioning" to indicate a completely different algorithm (based on making a distinction between real-time cores and non-real-time cores, and using a dedicated scheduling core to perform on-line admission control on the arriving jobs).

2 DEFINITIONS AND BACKGROUND

The system under study consists of a set $\Gamma = \{\tau_i\}$ of real-time tasks τ_i, to be scheduled onto a platform with M identical cores/CPUs. Each real-time task τ_i can be modelled as a stream of jobs $\{J_{i,k}\}$, where each job $J_{i,k}$ arrives (becomes ready for execution) at time $r_{i,k}$ and finishes at time $f_{i,k}$ after executing for an amount of time $c_{i,k}$ ($f_{i,k}$ clearly depends on the scheduler). Moreover, each task is associated with a relative deadline D_i and each job $J_{i,k}$ is required to complete within its absolute deadline of $d_{i,k} = r_{i,k} + D_i$.

If $f_{i,k}$ is smaller than or equal to the job's absolute deadline $d_{i,k}$, then the job respected its deadline, otherwise the deadline is missed. Task τ_i respects all of its deadlines if $\forall k, f_{i,k} \leq d_{i,k}$. Since $d_{i,k} = r_{i,k} + D_i$, this condition is often expressed as $\forall k, f_{i,k} - r_{i,k} \leq D_i$. The tardiness of job $J_{i,k}$ is defined as $\max\{0, f_{i,k} - d_{i,k}\}$.

Real-time tasks are often periodic ($\forall k, r_{i,k+1} - r_{i,k} = P_i$) or sporadic ($\forall k, r_{i,k+1} - r_{i,k} \geq P_i$) with period or minimum inter-arrival time P_i and in this work we assume $D_i = P_i$. The exact job execution times $c_{i,k}$ are not known beforehand, however we assume to know a reasonable Worst-Case Execution Time (WCET) $C_i \geq c_{i,k}\ \forall k$.

The goal of a real-time scheduler is to provide predictability, so that, given a taskset $\Gamma = \{\tau_i\}$ (with each task τ_i characterised by its parameters (C_i, P_i, D_i)) it is possible to check in advance if any deadline will be missed (or, it is possible to provide guarantees about the worst-case tardiness experienced each task τ_i).

In case of single core, fixed-priority (with the Rate Monotonic assignment [20]) or EDF [10] schedulers can provide the guarantee that every task will respect all its deadlines if some schedulability condition is respected (if $P_i = D_i$, then the schedulability condition is $\sum_i \frac{C_i}{P_i} \leq U^{lub}$, with U^{lub} depending on the scheduling algorithm — for EDF, $U^{lub} = 1$).

If the system is composed of multiple cores or CPUs, then different approaches can be used:

- In case of *partitioned scheduling*, all tasks $\Gamma = \{\tau_i\}$ are partitioned across the CPUs so that each partition $\Gamma_j \subset \Gamma$ is statically associated to a single core/CPU j, and EDF (or fixed-priority scheduling) can be used on each core/CPU (in this case, the schedulability analysis can be performed independently on the various cores/CPUs). Of course, this approach requires that it is possible to partition the tasks among cores/CPUs so that every partition has $\sum_{\tau_i \in \Gamma_j} \frac{C_i}{P_i} \leq 1$ (if EDF is used). However, this may not be always possible. For example, the taskset $\Gamma = \{(6, 10, 10), (6, 10, 10), (6, 10, 10)\}$ is not schedulable on 2 CPUs using a partitioned approach
- In case of *global scheduling*, the scheduler dynamically migrates tasks among cores/CPUs so that the m highest priority ready tasks (or the m earliest deadline ready tasks) are scheduled (where m is the minimum between the number of cores/CPUs and the number of ready tasks). In this case, the uniprocessor schedulability analysis cannot be re-used, and new schedulability tests (which turn out to be much more pessimistic) are needed [6, 7, 13]. Looking again at the taskset $\Gamma = \{(6, 10, 10), (6, 10, 10), (6, 10, 10)\}$, it is possible to notice that some deadlines will be missed also when using global EDF (gEDF) scheduling, but in this case the finishing times of all jobs will never be much larger than the absolute

deadlines (in practice, $\forall i,k, f_{i,k} - r_{i,k} \leq 12$). This is a property of the gEDF algorithm which holds when $\sum \frac{C_i}{P_i} \leq M$ (with M being the number of cores/CPUs) which obviously cannot be respected by partitioned EDF (pEDF).

While conceptually global scheduling requires that all the ready tasks are inserted in a single global queue (ordered by priority or deadline, so that the first M tasks of the queue are scheduled), some OS kernels (such as Linux) implement it by using per-core ready task queues ("runqueues" in Linux) and migrating tasks among them so that the M highest-priority/earliest-deadline tasks are on M different queues. For example, the Linux `SCHED_DEADLINE` scheduling policy [18] implements gEDF through multiple runqueues, using two "pull" and "push" operations to enforce the gEDF invariant. This is done by invoking "push" and "pull" every time the earliest-deadlines tasks change[2]:

- When a task wakes up (becomes ready for execution) and is inserted in the j^{th} runqueue, a "push" operation is performed to check if a task should be pushed from the j^{th} to some other runqueue to respect the global invariant (the M highest-priority/earliest-deadline tasks are on M different runqueues)[3]
- When the task executing on the j^{th} core/CPU blocks (is not ready for execution anymore), a "pull" operation is performed to check if a task queued in some other runqueue should be pulled onto the j^{th} runqueue to respect the global invariant.

Although this mechanism has been designed to implement global scheduling using per-CPU runqueues, it can also be used to implement some kind of trade-off between global and partitioned scheduling. For example, the "push" operation can be modified to control the utilisation U_j of each runqueue (so that it is smaller than U^{lub}). This is the basic idea of the adaptively partitioned scheduling that will be introduced in the next section.

3 ADAPTIVELY PARTITIONED SCHEDULING

The *adaptive partitioning* migration strategy proposed in this paper implements a restricted migration scheduling algorithm based on r-EDF [3]. Since all the runqueues are ordered by absolute deadlines (implementing the EDF algorithm, so that $U^{lub} = 1$), the algorithm is named *adaptively partitioned EDF* (apEDF).

In more details, in a scheduling algorithm based on restricted migrations a task τ_i that starts to execute on core/CPU j cannot migrate until its current job is finished (each job $J_{i,k}$ executes on a single core/CPU, and cannot migrate).

3.1 The Basic Algorithm

To simplify the description of the apEDF algorithm, let $rq(\tau_i)$ indicate the runqueue in which τ_i has been inserted (that is, the core/CPU on which τ_i executes or has executed) and let $U_j = \sum_{\{i:rq(\tau_i)=j\}} \frac{C_i}{P_i}$ indicate the utilisation of the jobs executing on core/CPU j. Moreover, let d^j represent the absolute deadline $d_{h,l}$ of the job currently executing on CPU j, or ∞ if CPU j is idle.

By default, when a task τ_i is created its runqueue is set to 0 ($rq(\tau_i) = 0$) and will be eventually set to an appropriate runqueue when the first job arrives (the task wakes up for the first time).

Data: Task τ_i to be placed with its current absolute deadline being $d_{i,k}$; state of all the runqueues (overall utilisation U_j and deadline of the currently scheduled task d^j for each CPU j)
Result: rq(τ_i)

```
1  if U_rq(τi) ≤ 1 then
       /* Stay on current CPU if schedulable      */
2      return rq(τi)
3  else
       /* Search for a CPU where the task can fit */
4      for j = 0 to M − 1 do /* Iterate over runqueues */
5          if U_j + C_i/P_i ≤ 1 then
6              return j /* First-fit heuristic    */
7          end
8      end
       /* Find the runqueue executing the task with
          the farthest away deadline              */
9      h = 0
10     for j = 1 to M − 1 do /* Iterate over runqueues */
11         if d^j > d^h then
12             h = j
13         end
14     end
15     if d^h > d_{i,k} then
           /* τi is migrated to runqueue h, where it
              will be the earliest deadline one    */
16         return h
17     end
       /* Stay on the current runqueue otherwise  */
18     return rq(τi)
19 end
```

Algorithm 1: Algorithm to select a runqueue for a task τ_i on each job arrival.

When job $J_{i,k}$ of task τ_i arrives at time $r_{i,k}$, the migration strategy selects a runqueue $rq(\tau_i)$ for τ_i (that is, a core/CPU on which τ_i will be scheduled), by using Algorithm 1.

The algorithm uses information about τ_i and the state of the various runqueues. Based on this information, it tries to schedule tasks so that runqueues are not overloaded ($\forall j, U_j \leq 1$) while reducing the number of migrations. If $U_{rq(\tau_i)} \leq 1$ (Line 1), then $rq(\tau_i)$ is left unchanged and the task is not migrated (Line 2). Otherwise (lines 4 — 18), an appropriate runqueue $rq(\tau_i)$ is selected as follows:

- If $\exists j : U_j + \frac{C_i}{P_i} \leq 1$, then select the first runqueue j having this property: $j = \min\{h : U_h + \frac{C_i}{P_i} \leq 1\}$ (Lines 4 — 8). In other words, Lines 4 — 8 implement the well-known First-Fit (FF) heuristic, but other heuristics such as Best-Fit (BF) or Worst-Fit (WF) can be used as well

[2] A similar mechanism is used for the fixed-priority scheduler, invoking "push" and "pull" every time the highest-priority tasks change

[3] Actually, before inserting the task in the runqueue a "`select_task_rq()`" function is invoked to decide in which runqueue the task has to be inserted. This is an optimisation that can make the "push" operation unneeded and can be ignored from a conceptual point of view.

- If the execution arrives to Line 9, this means that $\forall j, U_j + \frac{C_i}{P_i} > 1$ (task τ_i does not fit on any runqueue). Then, select a runqueue j based on comparing the absolute deadlines $d_{i,k}$ and $\{d^j\}$, as done by the gEDF strategy (Lines 9 — 17):
  - If $d^h \equiv \max_j\{d^j\} > d_{i,k}$ (Line 15), select the runqueue h currently running the task with the farthest away deadline into the future (Line 16)[4]
  - Otherwise, do not migrate the task (line 18).

Since $rq(\tau_i)$ is set to 0 when τ_i is created and is updated only when $U_0 > 1$ (the check at Line 1 fails) using the FF heuristic (Lines 4 — 8), it can be seen that if FF can generate a schedulable task partitioning then Algorithm 1 behaves as FF and has the FF properties. Previous work [21] proved that if $U \leq \frac{M+1}{2}$ then FF generates a schedulable partitioning, hence apEDF is able to correctly schedule tasksets with $U \leq \frac{M+1}{2}$ without missing any deadline[5].

The check at Line 1 of the algorithm ensures that if task τ_i has been previously inserted in a runqueue with $U_i \leq 1$, then it is not migrated; hence, only tasks that have been assigned to overloaded CPUs (potentially suffering because of missed deadlines) are migrated. As a result, tasks can initially migrate, but if Algorithm 1 is able to find a schedulable partitioning then the tasks do not migrate anymore. This is an important difference between apEDF and r-EDF. The latter "forgets" the CPU on which a task has been run at each task deadline $d_{i,k}$, decreasing U_j by $\frac{C_i}{P_i}$ at that time, potentially migrating tasks at each job arrival/activation, even if they are correctly partitioned. Algorithm 1, instead, avoids unneeded migrations by letting tasks stay on the same CPU as long as there are no overloads, updating the runqueues' utilisations only when tasks migrate (and not when they de-activate).

If the FF heuristic is not able to find a schedulable partitioning, apEDF allows to migrate tasks at every job arrival, so that if a schedulable task partitioning exists then Algorithm 1 can converge to it (by only migrating tasks that have been placed on overloaded runqueues — that is, runqueues with $U_j > 1$). In this case, only few deadlines will be missed at the beginning of the schedule (and after a sufficient amount of time no deadlines will be missed anymore).

A formal proof of this property has not been developed yet, but simulations seem to show that if a schedulable partitioning of the tasks exists (that is, if tasks $\tau_i \in \Gamma$ can be assigned to runqueues $0 \ldots M-1$ so that $\forall 0 \leq j < M, U_j \leq 1$), then after a finite number of migrations the tasks' assignments $\{rq(\tau_i)\}$ converge to such a partitioning.

If, instead, a schedulable tasks partitioning does not exist, then Lines 9 — 17 of Algorithm 1 ensure that the M earliest-deadline tasks are either scheduled or placed on non-overloaded cores/CPUs. Intuitively, this mechanism tries to make sure that tasks with small absolute deadlines cannot be starved and the difference between the current time and the absolute deadline is bounded. Hence, it can be conjectured that if $U \leq M$ then each task still experiences a bounded tardiness ($\exists L : \forall \tau_i \in \Gamma, \max_k\{f_{i,k} - d_{i,k}\} \leq L$) even if such a schedulable partitioning does not exist.

[4]Notice that for the sake of clarity Lines 9 — 14 show how to compute $\max_j\{d^j\}$ by iterating on all the runqueues, but the Linux kernel stores all the d^j in a heap, so the maximum can be obtained with a logarithmic complexity in the number of CPUs.

[5]The same work [21] also proves that, if $C_i/P_i \leq \beta \; \forall i$, then the FF utilisation bound is higher: $U \leq \frac{M\beta+1}{\beta+1}$.

In other words, the apEDF is designed to provide the good properties of both pEDF and gEDF.

Data: Runqueue rq where to pull; state of all the runqueues
Result: Task τ_i to be pulled

```
1  if rq is not empty then
2  |  return none
3  else
4  |  τ = none; min = ∞;
   |  /* Search for a task τ to pull                    */
5  |  for j = 0 to M − 1 do /* Iterate over runqueues */
6  |  |  if U_j > 1 then
7  |  |  |  if d'^j < min then
8  |  |  |  |  min = d'^j
9  |  |  |  |  τ = second(j)
10 |  |  |  end
11 |  |  end
12 |  end
13 |  return τ
14 end
```

Algorithm 2: Algorithm to pull a task in a^{2}pEDF.

3.2 Reducing the Tardiness

Although apEDF can provide a bounded tardiness if $U \leq M$, the tardiness bound L can be quite large (much larger than the one provided by gEDF), as it will be shown in Section 4. This is due to the fact that the runqueue on which a job $J_{i,k}$ is enqueued is selected at time $r_{i,k}$ when the job arrives; if task τ_i does not fit on any runqueue, the target runqueue is selected based on the absolute deadlines $\{d^j\}$ of the jobs that are executing on all CPUs at time $r_{i,k}$, so the selection can be sub-optimal after some of these jobs have finished.

This issue is addressed by the improved "a^{2}pEDF" algorithm that runs a "pull" operation each time a job finishes. This operation (described by Algorithm 2) is similar to the "pull" operation currently used by the Linux scheduler, but only pulls tasks on idle CPUs (see the check on Line 1) and only pulls from overloaded runqueues (see the check on Line 6). In the description of the algorithm, "second(j)" indicates the first non-executing task in runqueue j and d'^j indicates the absolute deadline of such a task (or ∞ if the runqueue does not contain any task that is not executing). Notice that, in contrast with apEDF, a^{2}pEDF does not follow a restricted migrations approach, because a "pull" operation can migrate a job after it started to execute on a core/CPU (and has been preempted by an earlier-deadline task).

Finally, it is worth noticing that, although apEDF and a^{2}pEDF might look more complex than the "original" pEDF and gEDF algorithms, they can easily be implemented in the Linux kernel. As previously mentioned, Linux currently implements the gEDF policy for `SCHED_DEADLINE` by storing per-CPU runqueues and using "push" and "pull" operations to make sure that the global deadline ordering is respected. The current "push" operation uses a "`find_later_rq()`" function that implements Lines 9 — 17 of Algorithm 1, and the current "pull" operation implements Lines 4 — 13

of Algorithm 2; hence, apEDF can be easily implemented by modifying the "push" operation (adding Lines 1 — 8) and making "pull" a null operation. The Linux kernel already tracks the runqueue utilisations U_j (named "runqueue bandwidth" and stored in the "`rq_bw`" field of the runqueue structure), hence implementing Lines 1 — 8 of Algorithm 1 is not difficult. The a^{2}pEDF algorithm can then be implemented by re-introducing the "pull" operation with small modifications respect to the current code (the only difference is that the a^{2}pEDF "pull" only pulls tasks if the current CPU is idle).

4 EXPERIMENTAL EVALUATION

The apEDF and a^{2}pEDF algorithms have been implemented in a scheduling simulator, to extensively check their properties and compare their performance with gEDF.

Multiple sets of simulations have been performed by using random tasksets (generated by using the Randfixedsum algorithm [12]) with different sizes and utilisations (for each configuration of the taskset's parameters, 30 different tasksets have been simulated). Each taskset has been simulated from time 0 to $2 * H$, where $H = lcm_i\{P_i\}$ is the taskset's hyperperiod.

First of all, the apEDF hard schedulability property (if $U \leq \frac{M+1}{2}$, then no deadline is missed) has been verified through simulations. A large number of tasksets has been generated and simulated on 2, 4, 8 and 16 CPUs, setting $U = \frac{M+1}{2}$ and using a number of tasks ranging from $2M$ to $3M$. In all the simulations, no deadlines were missed, confirming the property.

Then, the next sets of experiments compared the performance of apEDF (and a^{2}pEDF) with the performance of gEDF. Simulating the previously generated tasksets with gEDF it turned out that gEDF misses deadlines with many of them. To perform a more systematic comparison, more tasksets have been generated, varying their utilisation U from a little bit less than $\frac{M+1}{2}$ to almost M. The tasksets have been simulated using both gEDF and apEDF and measuring the percentage of deadlines missed by the two algorithms. For example, Figures 1, 2 and 3 show the percentage of missed deadlines and the average number of migrations per job when $M = 2, 4, 8$ (the figures show the results for tasksets composed by $N = 16$ tasks, but similar results have been obtained with different values of N). Notice that the plots about missed deadlines use a logarithmic scale on the Y axis to make the figure more readable. The results presented in the figures indicate that apEDF performs better than gEDF in most of the cases, but has some issues (resulting in a very large percentage of missed deadlines — for example, more than 30% for $M = 4$ and $U = 3.9$, or more than 20% for $M = 8$ and $U > 7.1$) for "extreme" values of the utilisation U.

Looking at the average migrations per job, it is interesting to see how for gEDF this number increases with the utilisation, while it stays to almost 0 for apEDF and low utilisations[6]. When the utilisation increases, and some of the generated tasksets are not partitionable (it is not possible to find a schedulable partitioning for them), the average number of migrations per job in apEDF increases (because Lines 9 — 17 of Algorithm 1 are used), but it is always very small compared to gEDF.

[6]The only measured migrations are the ones on the first job, from runqueue 0 to an appropriate runqueue.

An analysis of the problematic tasksets for which apEDF results in a high percentage of missed deadlines revealed that the issue is caused by the small number of migrations used by apEDF. For these tasksets it is not possible to find a schedulable partitioning, hence apEDF tries to respect the gEDF invariant when selecting a runqueue, but does not perform any "pull" operation (notice, again, that the number of migrations per job is small even if the tasksets are not partitionable). In contrast, gEDF uses a "pull" operation when jobs terminate (increasing the number of migrations per job), exploiting cores that would become idle. This suggests that introducing a "pull" phase in apEDF can fix the issue, and in fact in these situations a^{2}pEDF performs much better and can again outperform gEDF, as shown in the figures: for example, the percentage of deadlines missed by a^{2}pEDF for $M = 4$ and $U = 3.9$ is 7% — notice that it is 9% for gEDF (this result is confirmed by other experiments presented later). On the other hand, the figures plotting the average number of migrations per job show that a^{2}pEDF causes less migrations than gEDF. Figure 3 is even more interesting, showing that for very high values of the utilisation a^{2}pEDF misses a percentage of deadlines similar to gEDF (so, for non partitionable tasksets the a^{2}pEDF performance and the gEDF performance are similar).

Finally, notice that for apEDF the percentage of missed deadlines is 0 up to $U = 1.8$ for $M = 2$, $U = 3.3$ for $M = 4$ and $U = 6.2$ for $M = 8$.

Next, the impact of the number of CPUs on the apEDF performance has been evaluated. Multiple tasksets with a fixed utilisation and number of tasks have been generated and simulated on 2, 4 and 8 CPUs. In general, apEDF performed better than gEDF, both in terms of percentage of missed deadlines (soft real-time metric) and in terms of number of tasksets missing at least a deadline (hard real-time metric). Since the most interesting results were obtained with high utilisations, here the results for $U = 0.8M$ (and a number of tasks fixed to $N = 16$) are reported. Figure 4 shows the percentage of tasks missing at least one deadline for gEDF and apEDF with two kinds of tasksets: in the first case ("global" tasksets, on the left of the figure) the tasksets with utilisation $U = 0.8M$ and $N = 16$ tasks were directly generated using the Randfixedsum algorithm, while in the second case ("part" tasksets, on the right of the figure) M tasksets with utilisation U and N/M tasks were generated and merged to create a single larger taskset (the second kind of tasksets has been generated to check that apEDF can converge to the schedulable task partitioning).

As it can be noticed from the figure, the percentage of tasksets missing at least a deadline with apEDF is always smaller than the one with gEDF, showing that apEDF has better hard real-time performance than gEDF. The only case in which deadlines are missed in a relevant percentage of tasksets is the one with "global" tasksets, $M = 8$, $N = 16$ and $U = 6.4$. This happens because the number of tasks is relatively small respect to the number of CPUs ($N = 2M$) while the utilisation is quite high; in this situation, the taskset is likely not partitionable (hence, apEDF falls back to something similar to gEDF). The small percentage of tasksets missing at least a deadline with the same configuration ($M = 8$, $U = 6.4$) and "part" tasksets has been investigated and it turned out that it is due to the initial deadline misses of some tasksets, happening before the (stable) schedulable task partitioning has been reached (this has been verified by checking that increasing the simulation length the

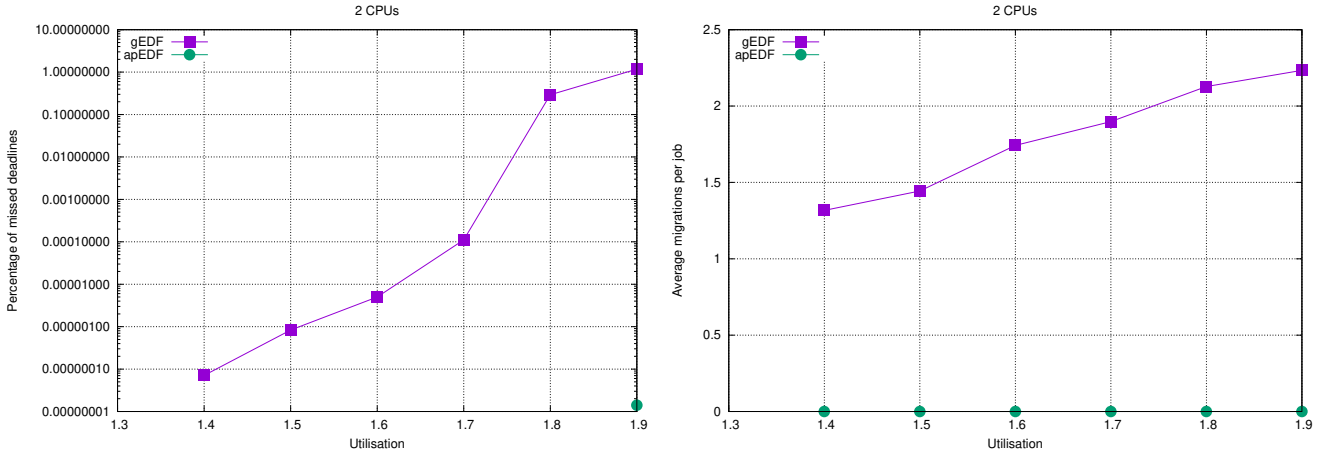

Figure 1: Percentages of missed deadlines and average migrations per job (as a function of U) with 2 CPUs and 16 tasks.

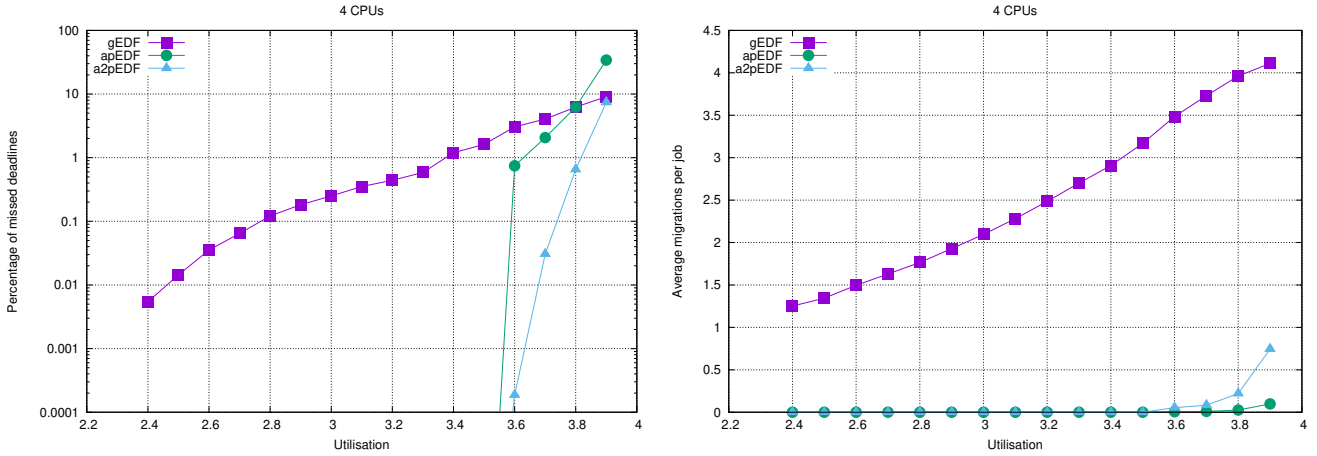

Figure 2: Percentages of missed deadlines and average migrations per job (as a function of U) with 4 CPUs and 16 tasks.

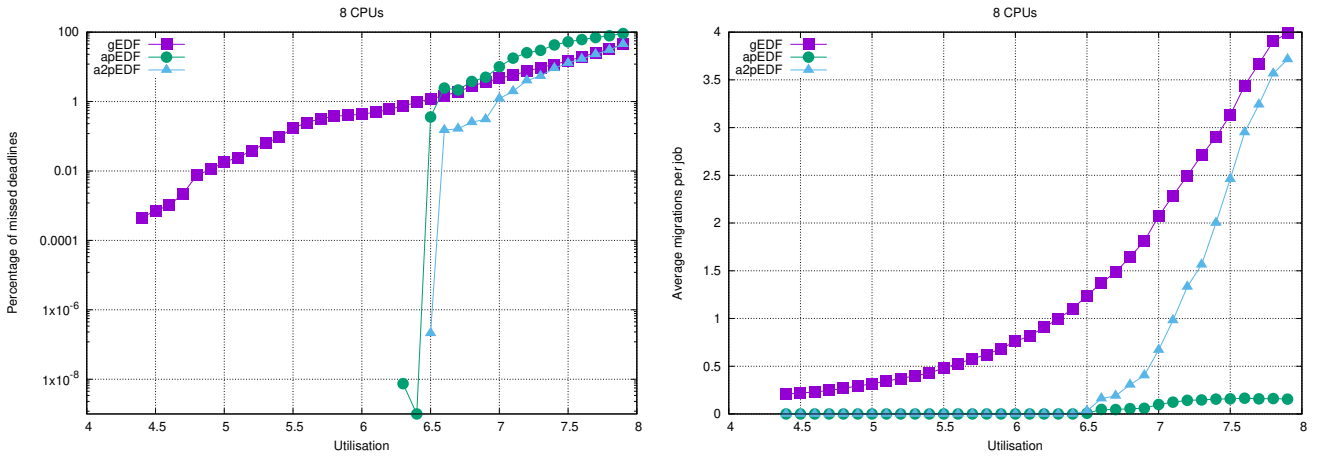

Figure 3: Percentages of missed deadlines and average migrations per job (as a function of U) with 8 CPUs and 16 tasks.

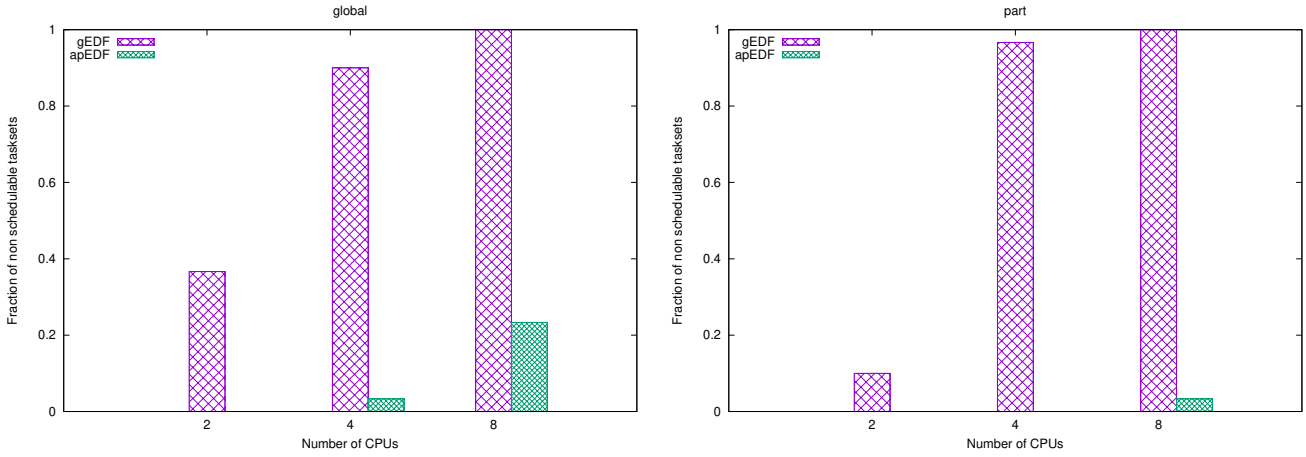

Figure 4: Percentages of tasksets missing at least a deadline with 2, 4 and 8 CPUs, $U = 0.8M$ and scheduling 16 tasks.

Table 1: Percentage of missed deadlines with 2, 4 and 8 CPUs, $U = 0.8M$ and scheduling 16 tasks.

CPUs	gEDF		apEDF	
	part	global	part	global
2	0.000000008	0.0000067	0	0
4	0.000008264	0.8935803	0	0.000000004
8	0.000022046	1.5758920	0.000000004	0.000000209

Table 2: Average number of migrations per job with 2, 4 and 8 CPUs, $U = 0.8M$ and scheduling 16 tasks.

CPUs	gEDF		apEDF	
	part	global	part	global
2	1.887913	1.965759	0.0000000056	0.0000000048
4	3.157379	3.188243	0.0000000041	0.0000000048
8	3.655992	2.795994	0.0000000060	0.0000000048

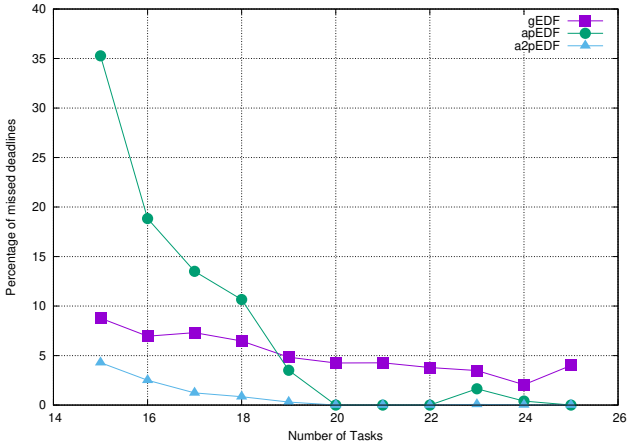

Figure 5: Percentage of missed deadlines with $M = 8$ $U = 7.6$ and N ranging from 15 to 25.

total number of deadlines missed by apEDF did not increase — of course, it increased using gEDF).

Table 1 compares the soft real-time performance of apEDF and gEDF, by showing the percentage of missed deadlines. The table confirms that gEDF can provide good soft real-time performance (even if deadlines are missed in many tasksets, the percentage of missed deadlines is small). However, apEDF still performs better than gEDF even for this metric.

Table 2, instead, compares the average number of migrations per job measured in the previous simulations. Again, apEDF results in a very small number of migrations compared to gEDF (notice that the number of migrations for apEDF is not much affected by variations in the number of CPUs because for most of the generated tasksets apEDF is able to find a schedulable partitioning after a few migrations).

Finally, some experiments have been performed to better investigate the situations in which the apEDF algorithm does not perform well (because of non-partitionable tasksets). As seen, this happens when the utilisation is close to the number of CPUs and the number of tasks is small compared to the number of CPUs. Figure 5, plotting the percentage of missed deadlines for $M = 8$, $U = 7.6$ and N ranging from 15 to 25, shows that for $N < 19$ apEDF misses more deadlines than gEDF. (however, looking at the response times it was possible to see that the tardiness is always limited and does not increase with simulations of longer durations).

Introducing a "pull" phase similar to the gEDF one, as done in the a^{2}pEDF algorithm, solves the issue: as shown in the figure, a^{2}pEDF always misses fewer deadlines than gEDF, even for small values of N. As previously mentioned, the a^{2}pEDF algorithm is not based on restricted migrations, and the average number of migrations per job is higher than the one for apEDF; however, as shown in Figure 6, the number of migrations is still small respect to gEDF. Also notice that when the number of tasks increases the average numbers of migrations for apEDF and a^{2}pEDF are almost equal.

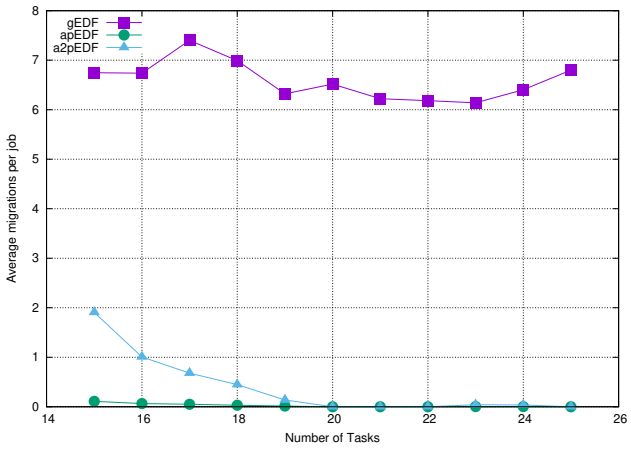

Figure 6: Average migrations per job with $M = 8$ $U = 7.6$ and N ranging from 15 to 25.

5 CONCLUSIONS AND FUTURE WORK

This paper presented a new migration strategy for EDF-based multiprocessor real-time schedulers, leading to the apEDF and a^{2}pEDF scheduling algorithms. Respect to the traditional gEDF, these algorithms allow for a less pessimistic schedulability analysis (leading to better hard real-time performance), while respect to pEDF they allow for a better handling of non-partitionable tasksets. The a^{2}pEDF algorithm provides smaller tardiness than apEDF when the utilisation is high (and the number of tasks is small). In this situation, apEDF performs slightly worse than gEDF while a^{2}pEDF provides better performance than gEDF. On the other hand, apEDF is simpler than a^{2}pEDF, and can be implemented more efficiently, because it does not require any "pull" operation when jobs finish.

As future work, the new migration policy will be implemented in the Linux kernel, replacing the global EDF algorithm used for the `SCHED_DEADLINE` policy, to verify its advantages through a real implementation. Moreover, the theoretical properties will be formally proved and the algorithms (with corresponding schedulability analysis) will be extended to support arbitrary affinities. The possibility to use the adaptive partitioning approach to support the coexistence of soft and hard real-time tasks will also be investigated.

REFERENCES

[1] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. 2008. An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Systems* 38, 2 (01 Feb 2008), 85–131.

[2] B. Andersson and E. Tovar. 2006. Multiprocessor Scheduling with Few Preemptions. In *Proceedings onf the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*. IEEE, Sydney, Qld., Australia, 322–334.

[3] Sanjoy Baruah and John Carpenter. 2003. Multiprocessor Fixed-Priority Scheduling with Restricted Interprocessor Migrations. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*. IEEE, Porto, Portugal, 195–202.

[4] S. K. Baruah. 2004. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Trans. Comput.* 53, 6 (June 2004), 781–784.

[5] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. 2010. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers. In *2010 31st IEEE Real-Time Systems Symposium*. IEEE, San Diego, CA, USA, 14–24.

[6] Marko Bertogna and Michele Cirinei. 2007. Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE, Tucson, AZ, USA, 149–160.

[7] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. 2005. Improved Schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*. IEEE, Balearic Islands, Spain, 209–218.

[8] Joan Boyar, György Dósa, and Leah Epstein. 2012. On the absolute approximation ratio for First Fit and related results. *Discrete Applied Mathematics* 160, 13 (2012), 1914 – 1923.

[9] H. Cho, B. Ravindran, and E. D. Jensen. 2006. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS 2006)*. IEEE, Rio de Janeiro, Brazil, 101–110.

[10] M. L. Dertouzos. 1974. Control Robotics: The Procedural Control of Physical Processes. *Information Processing* 74 (1974), 807–813.

[11] UmaMaheswari C. Devi and J. H. Anderson. 2008. Tardiness Bounds under Global EDF Scheduling on a Multiprocessor. *Real-Time Systems* 38, 2 (01 February 2008), 133–189.

[12] Paul Emberson, Roger Stafford, and Robert I Davis. 2010. 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)*. Brussels, Belgium, 6–11.

[13] Joël Goossens, Shelby Funk, and Sanjoy Baruah. 2003. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems* 25, 2 (September 2003), 187–205.

[14] R. L. Graham. 1972. Bounds on Multiprocessing Anomalies and Related Packing Algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference (AFIPS '72 (Spring))*. ACM, New York, NY, USA, 205–217.

[15] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. 1974. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM J. Comput.* 3, 4 (1974), 299–325.

[16] David S. Johnson. 1974. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.* 9, 3 (1974), 256 – 278.

[17] Juri Lelli, Dario Faggioli, Tommaso Cucinotta, and Giuseppe Lipari. 2012. An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software* 85, 10 (2012), 2405 – 2416. Automated Software Evolution.

[18] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. 2016. Deadline Scheduling in the Linux kernel. *Software: Practice and Experience* 46, 6 (June 2016), 821–839.

[19] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. 2010. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *Proceesings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*. IEEE, Brussels, Belgium, 3–13.

[20] Chung Laung Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment. *Journal of the Association for Computing Machinery* 20, 1 (Jan. 1973), 46–61.

[21] Jose Maria López, Manuel García, José Luis Diaz, and Daniel F Garcia. 2000. Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000)*. IEEE, Stockholm, Sweden, 25–33.

[22] T. Megel, R. Sirdey, and V. David. 2010. Minimizing Task Preemptions and Migrations in Multiprocessor Optimal Real-Time Schedules. In *2010 31st IEEE Real-Time Systems Symposium*. IEEE, San Diego, CA, USA, 37–46.

[23] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. 2011. RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. In *Proceesings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*. IEEE, Vienna, Austria, 104–115.

[24] N. Saranya and R. C. Hansdah. 2015. Dynamic Partitioning Based Scheduling of Real-Time Tasks in Multicore Processors. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, Auckland, New Zealand, 190—197.

[25] David Simchi-Levi. 1994. New worst-case results for the bin-packing problem. *Naval Research Logistics (NRL)* 41, 4 (1994), 579–585.

[26] Paolo Valente and Giuseppe Lipari. 2005. An Upper Bound to the Lateness of Soft Real-Time Tasks Scheduled by EDF on Multiprocessors. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE, Miami, FL, USA, 10 pp.–320.

[27] A. Wieder and B. B. Brandenburg. 2013. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, Porto, Portugal, 49–58.