EDF Scheduling of Real-Time Tasks on Multiple Cores: Adaptive Partitioning vs. Global Scheduling

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 novel migration algorithm for real-time tasks on multicore systems, based on the idea of migrating tasks only when strictly needed to respect their temporal constraints and a combination of this new algorithm with EDF scheduling. This new "adaptive migration" algorithm is evaluated through an extensive set of simulations showing good performance when compared with global or partitioned EDF: our results highlight that it provides a worst-case utilisation bound similar to partitioned EDF for hard real-time tasks and an empirical tardiness bound (like global EDF) for soft real-time tasks. Therefore, the proposed scheduler is effective for dealing with both hard and soft real-time workloads.

CCS Concepts

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

Keywords

Multi-Core Real-time Scheduling, Real-Time Operating Systems

1. INTRODUCTION

Computing technologies have always been evolving towards higher and higher performance platforms and microprocessors, to meet an always increasing demand due to evolving user requirements, as well as the increasing complexity of the software and the computations to be performed. This evolution has been flanked by an evolution in chip manufacturing technologies so that new generations of processors could have lower and lower power consumption per transistor and per clock cycle, which allowed for higher and higher CPU frequencies as one of the major drivers to scale-up the performance. However, in 2004 we have witnessed this CPU frequency "rush" to come to a dead-end, with Intel cancelling [18] its announced line of CPUs for desktops and servers, to start investing on multi-core technologies. The decision seems to have been facilitated by thermal issues for the new lines of processors, targeting 5GHz–10GHz frequencies, that would have suffered from unprecedented problems in heat dissipation. On the other hand, multiprocessor platforms had been for decades the exclusive domain of expensive high-performance computing systems, but they equally allowed software technologies and operating systems to evolve for supporting parallelism.

As a consequence, the more-than-linear increase in power consumption of microprocessors with the frequency led to a new generation of computing platforms where performance could be scaled-up in an energy-viable way by adding more and more cores. Fast-forward to today, and we have multi-core computing platforms in any computing domain, not only in high-performance and cloud computing but also in personal and mobile computing, as well as many embedded and real-time control systems.

However, software has been struggling at coping with this increase of parallelism in hardware, where technologies like OpenMP [29] gained in popularity with their capability of making parallel software development easier for those used to sequential loops. On the other hand, operating system mechanisms had to dramatically evolve to support the new platforms, not only multi-core SMP platforms but DVFS-capable, non-symmetric and heterogeneous multi-core platforms, recently with GPU and FPGA acceleration [30].

1.1 Problem Presentation

In this context, a particularly nasty problem has become the one of designing efficient and real-time CPU schedulers for multi-core and multi-processor platforms, which is well known to be more cumbersome than for single-CPU systems [5]. For example, real-time schedulers for multi-core platforms are known to be subject to the so called *scheduling anomalies*, where increasing the number of CPUs or the CPU frequency sometimes turns a system non-schedulable causing deadline misses.

However, research and industry have been putting a relentless effort towards increasingly better support for multi-core/processor platforms for real-time tasks, where the correctness of the system depends not only on its functional correctness but also on its capability to let real-time tasks respect their temporal constraints.

We can distinguish two main categories of scheduling algorithms for multi-cores: in *global scheduling*, tasks can be migrated among cores/CPUs according to the internals of the scheduling strategy; in *partitioned scheduling*, tasks are statically partitioned among the available processors, where a single-processor scheduler is used on each CPU. These are both relevant and viable options available in nowadays operating systems.

For example, the Linux kernel provides multiple real-time scheduling policies, based on fixed-priority or deadline-based scheduling [24], which can be configured in a very flexible way when it comes to multi-processors. We can configure the system to use global scheduling across all of the system CPUs, completely partitioned scheduling so that each task is locked onto a specific CPU, or we can even partition the tasks and CPUs so that each task set undergoes global scheduling restricted to a specific set of CPUs.

Partitioned schedulers are normally simpler to realize, easier to analyze thanks to the applicability of uni-processor analysis, and exhibit higher efficiency, as they avoid the overheads caused by migrations across CPUs. However, in presence of temporary overloads due to, e.g., interrupt storms or timing misbehaviors of some tasks, partitioned schedulers are unable to adapt to the situation and deadline misses can occur, despite other CPUs remaining idle. Furthermore, partitioned schedulers can only serve tasksets that are *partitionable*, i.e., where each partition of tasks on each CPU meets the requirements of uniprocessor scheduling algorithms.

Global schedulers for real-time task sets are more complex to design and more difficult to analyse. For example, the schedulability tests for global Earliest Deadline First and global fixed priorities are quite pessimistic. However, global schedulers are in a better position to tolerate temporary overloads as they can migrate tasks among the available CPUs. Furthermore, 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 [16, 34].

In this context, an area of research that undoubtedly deserves more attention is the one of on-line and adaptive approaches that can be efficiently and effectively used to schedule real-time task sets on multi-processor systems, trying to exploit the advantages of both global and partitioned scheduling. A first example of this kind of algorithms has been presented in an earlier version of this paper [1].

1.2 Contributions

This paper describes two scheduling algorithms based on adaptive partitioning, named apEDF and a^{2}pEDF, and provides an evaluation of their performance based on simulations. As detailed in Section 4, the *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 $(M+1)/2$ (where M is the number of cores/CPUs), then all the deadlines are respected, while if the total utilisation is between $(M+1)/2$ 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 ($(M+1)/2$) is known to be optimal for fixed-job-priority algorithms [3].

Respect to the previous version of this paper [1], implementation issues are described in more detail, the hard and soft real-time performance are better defined and evaluated, and some considerations (and experiments) about dynamic task creations and destructions are presented.

1.3 Paper Organization

This paper is organized as follows. After a review of the most relevant research literature in Section 2, we provide additional background in Section 3 about some fundamental notation and concepts that are used throughout the rest of the paper. In Section 4, we describe our proposed approach to adaptive partitioned scheduling, which is validated through the experimental results presented in Section 5. Finally, conclusions are drawn in Section 6, along with a sketch of possible lines of future research on the topic.

2. RELATED WORK

Previous research on multi-processor real-time scheduling focused on supporting either hard real-time systems (where all the deadlines of all the tasks have to be respected) [7, 4, 14, 31, 25] or soft real-time systems (where a controlled amount of missed deadlines can be tolerated) [16, 34].

Regarding hard real-time systems, it is known from literature [3] that global EDF can be modified to have an utilisation bound which is optimal for fixed-job-priority algorithms [7] and that optimal multiprocessor scheduling algorithms (based on global scheduling) exist [8, 4, 14, 31, 25]. 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 re-

[1]Although the proposed technique is a form of dynamic partitioning, the name "adaptive partitioning" is used to avoid confusion with a previous work [32] 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).

specting 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 [23, 9] 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 off-line partitioning, such as achievable via integer linear programming techniques [28, 35].

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 [20] and Johnson [22, 21], or more recent works and comprehensive surveys on the topic [33, 12]). 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 an approximation ratio, achieved as the size of the problem grows to ∞. For example, one interesting result in the area is the $12/7 \approxeq 1.7143$ bound for the first-fit heuristic [12]. 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.

3. 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 composed of one or more CPUs, with a total of M identical cores. 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 $\forall k, f_{i,k} \le d_{i,k}$. Since $d_{i,k} = r_{i,k} + D_i$, this condition is often expressed as $\forall k, f_{i,k} - r_{i,k} \le D_i$. 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 ($\forall k, r_{i,k+1} - r_{i,k} \ge P_i$) 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) $C_i \ge c_{i,k} \, \forall k$.

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 by each task τ_i).

In case of a single core ($M = 1$), fixed-priority (with the Rate Monotonic assignment [26]) or EDF [15] 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 $\sum_i C_i/P_i \le U^{lub}$, with U^{lub} depending on the scheduling algorithm — for EDF, $U^{lub} = 1$.

If tasks are scheduled on multiple CPU cores, then different approaches can be used, and are generally grouped in two classes of algorithms known as *partitioned scheduling* and *global scheduling*.

Algorithms based on partitioned scheduling partition the tasks $\Gamma = \{\tau_i\}$ across the cores so that each partition $\Gamma_j \subset \Gamma$ is statically associated to a single core j, and EDF (or fixed-priority scheduling) can be used on each core (in this case, the schedulability analysis can be performed independently on the various cores). Of course, this approach requires that it is possible to partition the tasks among cores so that every partition respects $\sum_{\tau_i \in \Gamma_j} C_i/P_i \le 1$ (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 cores using a partitioned approach

Algorithms based on *global scheduling*, instead, dynamically migrate tasks among cores 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 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 [11, 10, 19]. 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, $\forall i, k, f_{i,k} - r_{i,k} \le 12$). This is a property of the gEDF algorithm which holds when $\sum C_i/P_i \le M$ (with M being the number of cores) 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 [24] implements gEDF through multiple runqueues, using three operations to enforce the gEDF invariant: "select task runqueue", "pull" and "push". This is done by invoking one of the three operations every time the earliest-deadlines tasks change[2]:

[2]A similar mechanism is used for the fixed-priority scheduler, invoking "select task runqueue", "push" or "pull" every

- When a task wakes up (becomes ready for execution) a "select task runqueue" (the "`select_task_rq()`" kernel function) is invoked to decide in which runqueue j the task has to be inserted (this choice should be made so that the gEDF invariant is respected, but also takes into account task affinities and migration overheads)

- When (after invoking "`select_task_rq()`") the task is inserted in the j^{th} runqueue, a "push" operation is performed to check if a task should be moved 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). Notice that thanks to the "select task runqueue" optimization the "push" operation is needed only if the task that wakes up preempts another deadline task that was already running on the selected core.

- When the task executing on the j^{th} core 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 originally designed to implement global scheduling using per-core runqueues, it can also be used to implement some kind of trade-off between global and partitioned scheduling. For example, the `select_task_rq()` function can be modified to control the utilisation U_j of each runqueue (so that it is smaller than U^{lub}), completely removing the "push" and "pull" operations (so that tasks are migrated only when they wake up). This is the basic idea of the adaptively partitioned scheduling that will be introduced in the next section.

4. ADAPTIVE PARTITIONING

The *adaptive partitioning* migration strategy, originally proposed in a previous version of this paper [1], implements a restricted migration scheduling algorithm based on r-EDF [6]. 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 detail, in a scheduling algorithm based on restricted migrations a task τ_i that starts to execute on core j cannot migrate until its current job is finished: each job $J_{i,k}$ executes on a single core, and cannot migrate. Hence, using the Linux terminology, the scheduler is based on a "select task runqueue" operation (invoked when a new job arrives), but does not use any "push" nor "pull" operation.

4.1 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 on which τ_i executes or has executed) and let $U_j = \sum_{\{i:rq(\tau_i)=j\}} C_i/P_i$ indicate the utilisation of the jobs executing on core j. Moreover, let d^j represent the absolute deadline $d_{h,l}$ of the job currently executing on core j, or ∞ if core j is idle.

time the highest-priority tasks change

```
Data: Task $\tau_i$ 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 core $j$)
Result: rq($\tau_i$)
1  if $U_{rq(\tau_i)} \leq 1$ then
       /* Stay on current core if schedulable */
2      return $rq(\tau_i)$
3  else
       /* Search a core where the task fits   */
4      for $j = 0$ to $M - 1$ do /* Iterate over all
        the runqueues                          */
5          if $U_j + C_i/P_i \leq 1$ 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 all
        the runqueueus                         */
11         if $d^j > d^h$ then
12             h = $j$
13         end
14     end
15     if $d^h > d_{i,k}$ then
           /* $\tau_i$ is migrated to runqueue $h$, where
              it will be the earliest deadline
              one                               */
16         return $h$
17     end
       /* Stay on current runqueue otherwise  */
18     return $rq(\tau_i)$
19 end
```

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

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).

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 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 $\exists j : U_j + C_i/P_i \leq 1$, then select the first runqueue j having this property: $j = \min\{h : U_h + C_i/P_i \leq 1\}$ (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

- If the execution arrives to Line 9, this means that $\forall j, U_j + C_i/P_i > 1$ (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)
  - Otherwise, do not migrate the task (line 18).

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 cores.

THEOREM 1. *The apEDF algorithm is able to schedule every taskset with $U = \sum_i C_i/P_i \le (M+1)/2$ without missing any deadline.*

PROOF. 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 [27] proved that if $U \le (M+1)/2$ then FF generates a schedulable partitioning, hence apEDF is able to correctly schedule tasksets with $U \le (M+1)/2$ without missing any deadline[3]. $\square$

THEOREM 2. *If adEDF is able to find a scheduling partitioning, after it is reached the migrations stop.*

PROOF. The check at Line 1 of the algorithm ensures that if task τ_i has been previously inserted in a runqueue with $U_i \le 1$, then it is not migrated; hence, only tasks that have been assigned to overloaded cores (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. $\square$

Theorem 2 shows an important difference between apEDF and r-EDF. The latter "forgets" the core on which a task has been run at each task deadline $d_{i,k}$, decreasing U_j by 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 core as long as there are no overloads, updating the runqueues' utilisations only when tasks migrate (and not when they de-activate).

Notice that Theorem 1 states that if $U < (M+1)/2$ then apEDF is immediately able to find a schedulable partitioning, without overloading and without migrations (only one initial migration from core 0 is needed). If the taskset's utilization is larger than $(M+1)/2$, then some additional migrations might be needed. Theorem 2 then shows that if a schedulable partitioning is reached after these initial migrations, then no further migrations will happen: if the FF heuristic used by apEDF is not able to immediately find a schedulable partitioning, then apEDF migrates a task at every job arrival, until the schedulable partitioning is reached.

[3]The same work [27] also proves that, if $C_i/P_i \le \beta \forall i$, then the FF utilisation bound is higher: $U \le (M\beta + 1)(\beta + 1)$.

It has been conjectured that if a schedulable task partitioning exists then Algorithm 1 is able to converge to it in a finite number of steps, 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 is still under development, but simulations seem to confirm it: in all the simulated tasksets, if a schedulable partitioning of the tasks exists (that is, if tasks $\tau_i \in \Gamma$ can be assigned to runqueues $0...M-1$ so that $\forall 0 \le j < M, U_j \le 1$), 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. 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 has been conjectured that if $U \le M$ then each task still experiences a bounded tardiness ($\exists L : \forall \tau_i \in \Gamma, \max_k\{f_{i,k} - d_{i,k}\} \le L$) even if such a schedulable partitioning does not exist.

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

4.2 Reducing the Tardiness

```
Data: Runqueue $rq$ where to pull; state of all the
      runqueues
Result: Task $\tau_i$ to be pulled
1  if $rq$ is not empty then
2      return none
3  else
4      $\tau$ = none; min = $\infty$;
       /* Search for a task $\tau$ to pull                */
5      for $j = 0$ to $M-1$ do /* Iterate over all
        the runqueues                                   */
6          if $U_j > 1$ then
7              if $d'^j < min$ then
8                  min = $d'^j$
9                  $\tau$ = second($j$)
10             end
11         end
12     end
13     return $\tau$
14 end
```

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

Although apEDF can provide a bounded tardiness if $U \le M$, the tardiness bound L can be quite large (much larger than the one provided by gEDF), as it will be shown in Section 5. 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 cores at time $r_{i,k}$, so the selection can be sub-optimal after some of these jobs have finished. For example, if at time $r_{i,k}$ job $J_{i,k}$ is inserted in the j^{th} runqueue, then it will be scheduled on the j^{th} core even if some other core in the meanwhile becomes idle: in other words, apEDF is not work-conserving.

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 cores (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 (and has been preempted by an earlier-deadline task).

At first glance, Line 1 of the algorithm (pull tasks only to idle cores) might look strange. It is actually motivated by the fact that when a task terminates, its utilization can be reused only after the task's deadline (or, more accurately, after the so called "0-lag time" [2]). Hence, immediately pulling a task from a different core might cause a transient overload, with additional missed deadlines. In theory, a^{2}pEDF could avoid this issue by waiting for the 0-lag time (or the end of the task period) before performing a "pull" operation, but performing a "pull" immediately only when the core becomes idle allows to address the issue in a simpler way (without having to set up timers for pulling).

4.3 Implementing Adaptive Migrations

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-core runqueues and using "select task runqueue", "push" and "pull" operations to make sure that the global deadline ordering is respected (the "select task runqueue" operation is used when a task wakes up, to reduce the number of "push" operations). The current "select task runqueue" and "push" are based on 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 implemented by modifying the "`find_later_rq()`" function (adding Lines 1 — 8) and disabling the "push" and "pull" operations. Push operations are not needed because apEDF migrates tasks only on wake-up, hence `select_task_rq()` is enough. Pull operations are obviously not needed because apEDF does not have a "pull" phase.

The Linux kernel already tracks the runqueue utilisation 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 reintroducing 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 core is idle).

4.4 Dynamic Task Arrivals and Terminations

Although most of the discussion up to this point focused on static tasksets (all tasks starts at time 0 and never terminates, there is no dynamic admission control) adaptively partitioned scheduling can support dynamic tasksets (with tasks created and terminated at arbitrary time instants).

The dynamic admission control mechanism can still be based on the taskset's utilization U (that can be time-dependent, increasing when new tasks are created and accepted in the system, and decreasing when an existing task terminates). However, Theorem 1 is only valid for static tasksets, hence even using $U < (M + 1)/2$ as an admission test a dynamic taskset can experience some transient deadline miss. This can happen because Theorem 1 is based on the FF property that if a task is assigned to core j then it cannot fit in any core $h < j$. When a task terminates on core h, this property might be broken (some of the tasks assigned to core $j > h$ that previously did not fit on core h can fit in it after the task terminates and the core's utilization decreases). Hence, one of the assumptions of the theorem is not respected, and the theorem does not hold.

As an example, consider a 2-cores system, with 3 tasks τ_0, τ_1 and τ_2 having utilizations 0.4 each. The total utilization is $U = 0.4 + 0.4 + 0.4 \leq (2 + 1)/2$, and Algorithm 1 is able to schedule the three tasks placing τ_0 and τ_1 on core 0 (with $U_0 = 0.8$), and τ_2 on core 1 (with $U_1 = 0.4$). At time t', τ_0 terminates, leaving the system; now, $U_0 = 0.4$, $U_1 = 0.4$ and the FF property mentioned above is not respected: τ_2 is scheduled on core 1 while it could fit on core 0. If at time $t'' > t'$ a new task τ_3 with utilization 0.65 arrives, it cannot fit on any core even if $U = 0.4+0.4+0.65 = 1.45 < (2+1)/2$! Of course, since one of the two cores becomes overloaded apEDF will soon migrate a task so that τ_1 and τ_2 will be on the same core, and τ_3 will be on the other one. Hence, some deadlines can be missed, but only during a short transient when a new task is created and passes the admission test.

5. 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 (as currently implemented by `SCHED_DEADLINE`). First, the properties and performance of adaptive partitioning algorithms have been tested using *static* tasksets, that do not change over time (no dynamic task creation or termination: all the tasks are created at time 0 and run until the end of the simulation). Then, some experiments have been performed to evaluate the behaviour of adaptive partitioning algorithms in presence of dynamic task arrivals and terminations.

5.1 Adaptive Partitioning and Schedulability

Before comparing adaptive partitioning with global scheduling, the hard schedulability property of apEDF (as stated by Theorem 1: if $U \leq (M+1)/2$, then no deadline is missed) has been verified through extensive simulations, using static tasksets. These tasksets have been generated by using the Randfixedsum algorithm [17], using different numbers of tasks and utilisations (for each configuration of the taskset's parameters, 10 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.

A large number of tasksets has been generated and simulated on 2, 4, 8 and 16 CPUs, setting $U = (M+1)/2$ and using a number of tasks ranging from $2M$ to $3M$. In all simulations, no deadlines were missed, confirming the property.

5.2 Soft Real-Time Performance

Then, the performance of apEDF (and a^{2}pEDF) have been compared with the performance of gEDF, starting from the observation that gEDF missed deadlines on many of the previously generated tasksets. To perform a more systematic comparison, it is important to define some meaningful metrics. For example, in *soft* real-time systems deadlines can be missed and the percentage of jobs missing their deadline can be a significant performance metric. On the other hand, in *hard* real-time systems a single job missing a deadline can compromise the correctness of the whole application (modelled as a taskset); hence, the percentage of the tasksets missing at least one deadline is a more meaningful performance metric.

In the following experiments, more tasksets have been generated, varying their utilisation U from slightly less than $(M+1)/2$ to almost M. The tasksets have been simulated using both global scheduling and adaptive partitioning, measuring both soft real-time metrics (percentage of jobs missing their deadline) and hard real-time metrics (percentage of tasksets containing at least a task missing a deadline).

First, the soft real-time performance is reported and discussed. For example, Figures 1, 2 and 3 compare the soft real-time performance by showing 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 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[4]. 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

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

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	8e−9	6.7e−5	0	0
4	8.264e−6	0.8936	0	4e−9
8	2.2046e−5	1.5759	4e−9	2.09e−7

compared to gEDF.

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. As expected, 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 fewer 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 $M = 8$.

Then, 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 4, 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.

Again, 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 5, 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.

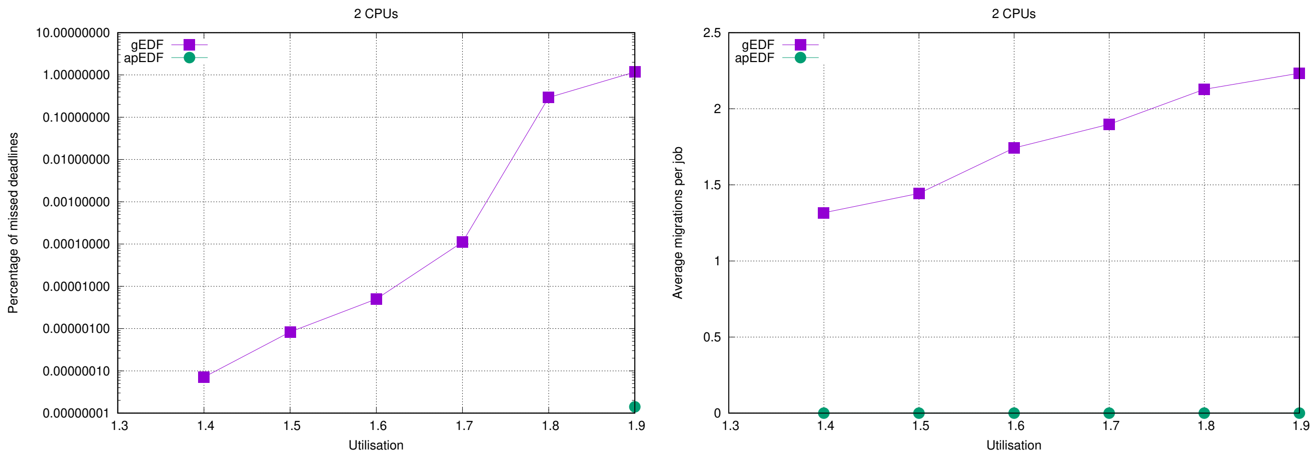

Figure 1: Percentages of missed deadlines and average migrations per job (as a function of U) with 2 CPUs and 16 tasks.

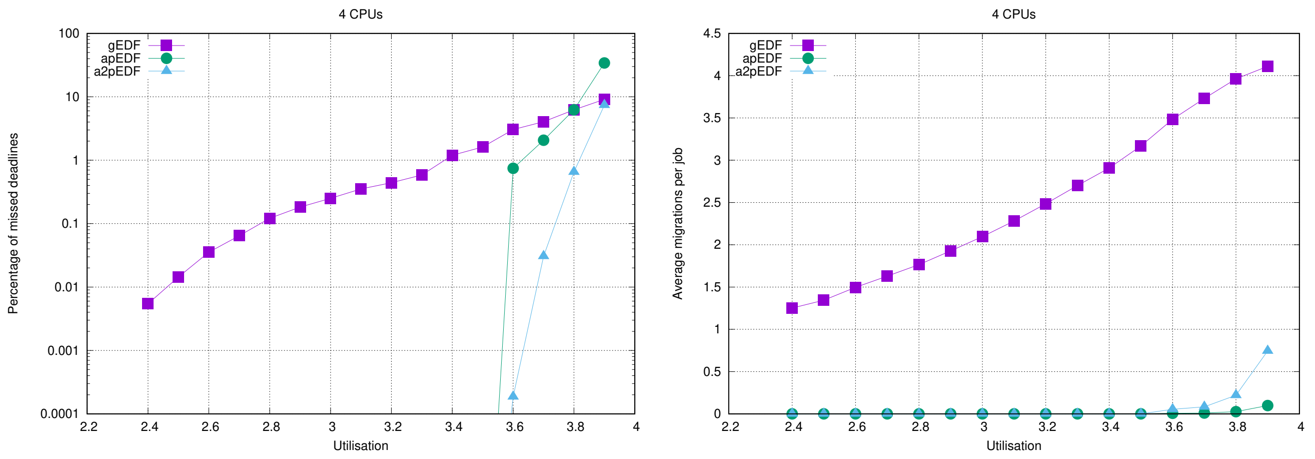

Figure 2: Percentages of missed deadlines and average migrations per job (as a function of U) with 4 CPUs and 16 tasks.

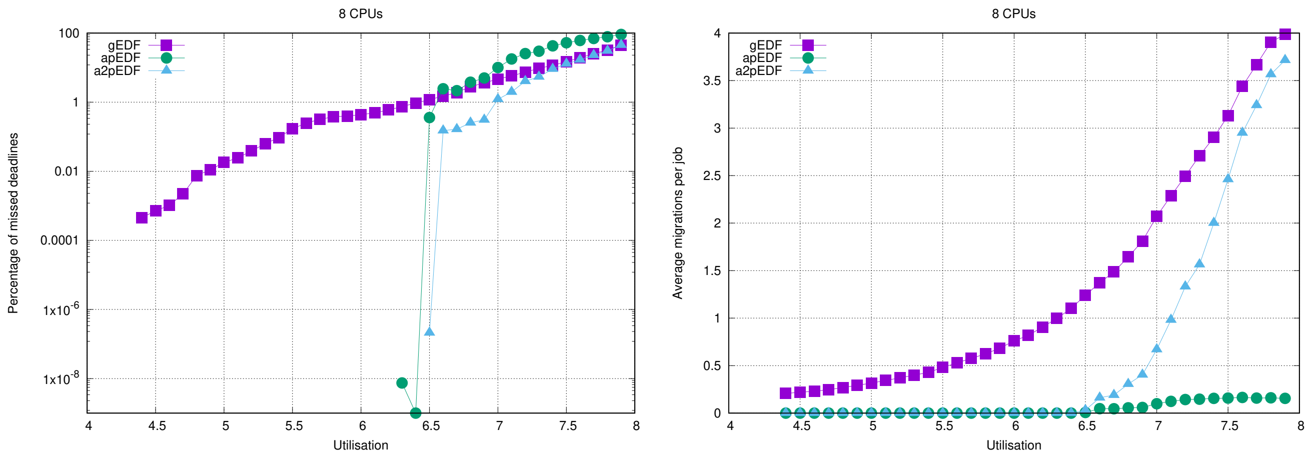

Figure 3: Percentages of missed deadlines and average migrations per job (as a function of U) with 8 CPUs and 16 tasks.

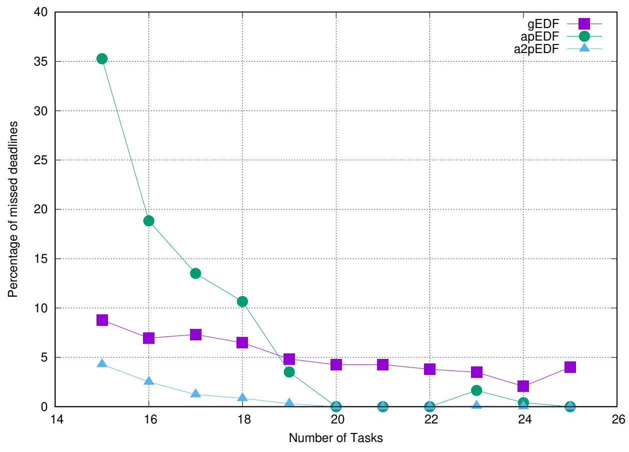

Figure 4: Percentage of missed deadlines with $M = 8$ $U = 7.6$ and N ranging from 15 to 25.

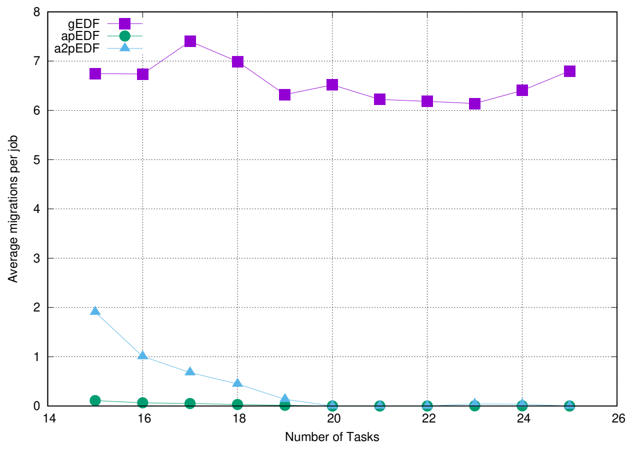

Figure 5: Average migrations per job with $M = 8$ $U = 7.6$ and N ranging from 15 to 25.

Another set of experiments has been performed to check how the taskset generation algorithm can impact on the measured performance. Multiple tasksets with a fixed utilisation and number of tasks have been generated and simulated on 2, 4 and 8 CPUs. The generation has been performed in two different ways: in the first case (referred to as "global" tasksets) the tasksets with utilisation $U = 0.8M$ and $N = 16$ tasks were directly generated using the Randfixedsum algorithm, while in the second case (referred to as "part" tasksets) M tasksets with utilisation U and N/M tasks were generated and merged to create a single larger taskset. The second kind of tasksets is partitionable by construction and it is interesting to see that apEDF converges to a schedulable task partitioning whenever possible).

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. 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: 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	5.6e−9	4.8e−9
4	3.157379	3.188243	4.1e−9	4.8e−9
8	3.655992	2.795994	6.0e−9	4.8e−9

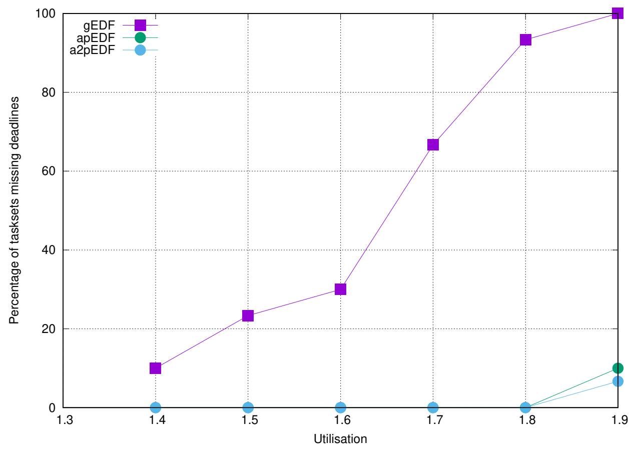

Figure 6: Percentages tasksets missing a deadline (as a function of U) with 2 CPUs and 16 tasks.

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 affected much by variations in the number of CPUs as, for most of the generated tasksets, apEDF is able to find a schedulable partitioning after a few migrations).

5.3 Hard Real-Time Performance

After comparing the soft real-time performance of the algorithms, their hard real-time performance has been compared, by considering the percentage of tasksets containing at least one missed deadline (remember that for each configuration 30 different tasksets have been generated).

Figures 6, 7 and 8 compare the hard real-time performance of the tasksets already reported in Figures 1, 2 and 3 (showing the percentage of tasksets missing at least a deadline instead of the percentage of missed deadlines). The results presented in the figures indicate that when we consider hard real-time performance apEDF always performs better than gEDF. Moreover, for this metric, differences between apEDF and a^{2}pEDF are not very large. From Figure 6 it can be seen that on 2 cores adaptive partitioning can provide good hard real-time performance also with very high utilizations (up to 1.9), while from Figures 7 and 8 it can be seen that adaptive partitioning can provide good hard real-time per-

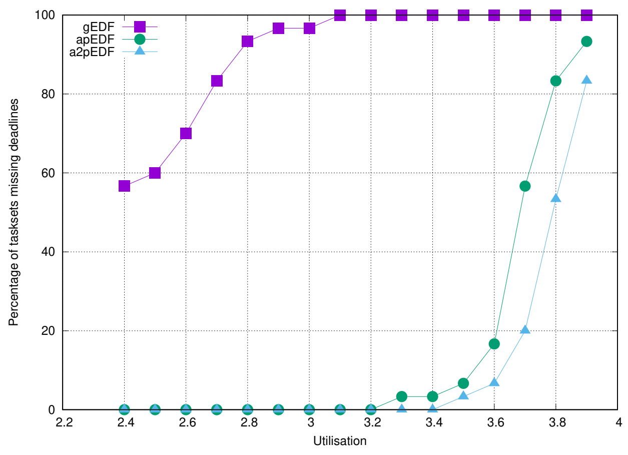

Figure 7: Percentages tasksets missing a deadline (as a function of U) with 4 CPUs and 16 tasks.

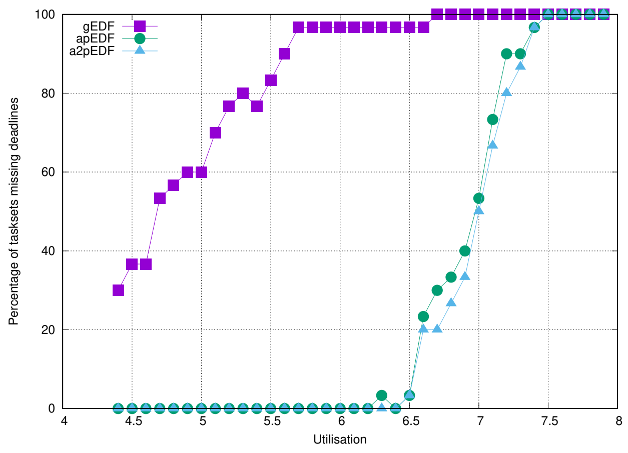

Figure 8: Percentages tasksets missing a deadline (as a function of U) with 8 CPUs and 16 tasks.

formance for $U \leq 3.5$ on 4 cores, and for $U \leq 6.5$ on 8 cores. In general, both apEDF and a^{2}pEDF perform well for utilzations up to about $0.8M$.

The results obtained when simulating the same tasksets used for Figure 4 are reported in Figure 9. It is possible to see that both apEDF and a^{2}pEDF provide better hard real-time performance than gEDF (even if they miss deadlines in a consistent fraction of tasksets). Finally, for this kind of tasksets the difference between apEDF and a^{2}pEDF is relevant: "pull" migrations are needed to respect all the deadlines in at least 50% of the tasksets.

Finally, Figure 10 shows the percentage of tasks missing at least one deadline for gEDF and apEDF with the tasksets used for Table 1. 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,

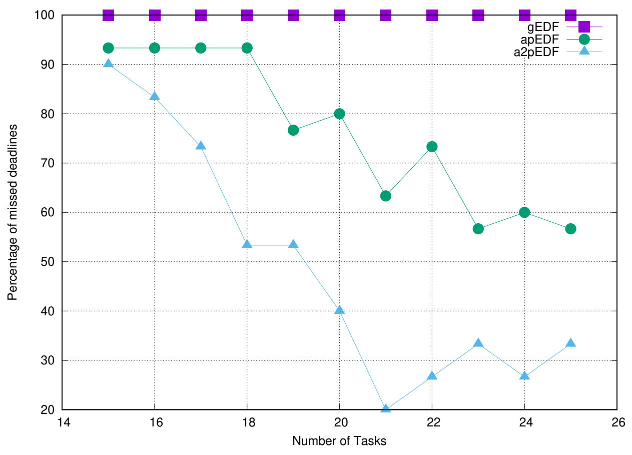

Figure 9: Percentage of tasksets missing a deadline with $M = 8$ $U = 7.6$ and N ranging from 15 to 25.

$M = 8$, $N = 16$ 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 have 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 total number of deadlines missed by apEDF did not increase — of course, it increased using gEDF).

5.4 Experiments with Dynamic Tasksets

After verifying how the algorithms work when tasks are not dynamically created or terminated, some additional experiments with more dynamic tasksets have been performed. The dynamic tasksets have been generated as sequences of "arrival" (creation of a new real-time task, with a utilization following the β distribution) and "exit" (termination of one of the existing tasks) events [13] (the same algorithm used in [13] has been used).

A total of 100 arrival/exit traces have been generated in this way, simulating the scheduling of these dynamic tasksets with gEDF, apEDF and a^{2}pEDF with an admission control $U \leq (M+1)/2$. In all these experiments, apEDF did not miss deadlines, showing that in this situation the issue highlighted in Section 4.4 does not happen very likely.

Hence, the experiment has been repeated changing the admission control; for example, using $U \leq 3$ as an admission control when the tasksets are scheduled on 4 cores, gEDF misses deadlines on 75% of the dynamic tasksets, apEDF misses deadlines on 33% of them and a^{2}pEDF misses deadlines on 13% of them. Looking at the soft real-time performance, all the three algorithms perform quite well: the percentage of jobs missing their deadline is 9.84389e$-$7% for gEDF, 0.04203% for apEDF and 6.515e$-$11% for a^{2}pEDF. In this case, gEDF performs better (for soft real-time perfor-

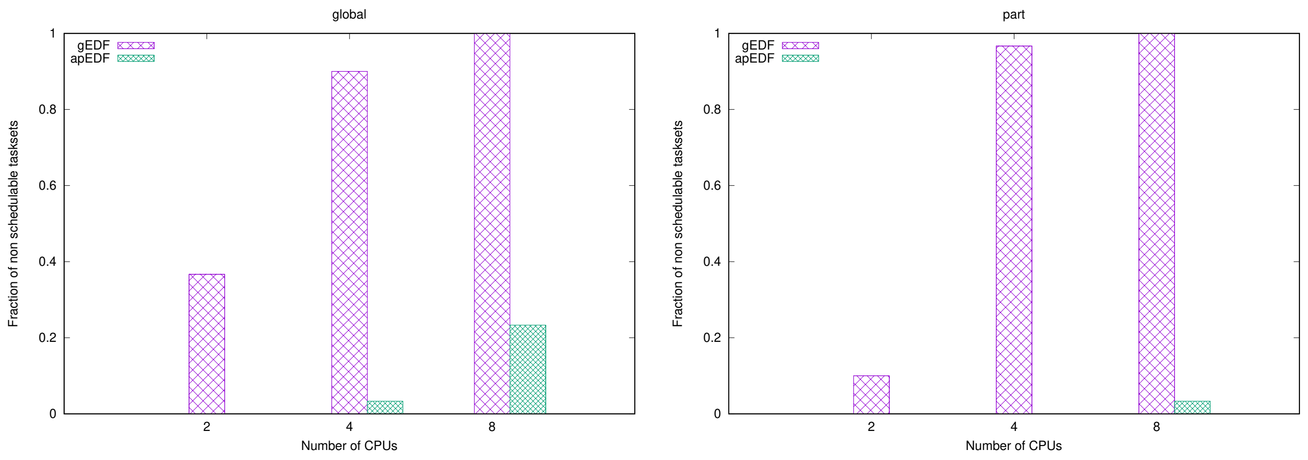

Figure 10: Percentages of tasksets missing at least a deadline with 2, 4 and 8 CPUs, $U = 0.8M$ and scheduling 16 tasks.

mance) than apEDF, while a^{2}pEDF performs again better than all the other algorithms.

6. CONCLUSIONS AND FUTURE WORK

This paper presented a study on the properties and performance of adaptive partitioning strategies [1]. In particular, it focused on the apEDF and a^{2}pEDF scheduling algorithms, that allow for a schedulability analysis less pessimistic than the one known in the literature for gEDF while allowing to handle non-partitionable tasksets that pEDF could not schedule. This results in better hard real-time performance than gEDF and better soft real-time performance than pEDF.

The hard and soft real-time performance of the two algorithms have been evaluated through an extensive set of simulations, reaching the conclusion that 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.

An implementation of the new migration policy in the Linux kernel (replacing the global EDF algorithm used for the `SCHED_DEADLINE` policy) is being developed and will be used as a future work to verify the advantages of adaptive partitioning through a real implementation.

Finally, the theoretical properties of apEDF and a^{2}pEDF that have been previously conjectured will be formally proved and the algorithms will be extended to support arbitrary affinities. The possibility to use the proposed technique to support the coexistence of soft and hard real-time tasks will also be investigated as a future work.

7. ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 871669 AMPERE – "A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation".

8. REFERENCES

[1] L. Abeni and T. Cucinotta. Adaptive partitioning of real-time tasks on multiple processors. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 572–579, New York, NY, USA, 2020. Association for Computing Machinery.

[2] L. Abeni, G. Lipari, A. Parri, and Y. Sun. Multicore cpu reclaiming: parallel or sequential? In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1877–1884, 2016.

[3] J. H. Anderson, V. Bud, and U. C. Devi. An edf-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Systems*, 38(2):85–131, Feb 2008.

[4] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proceedings onf the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*, pages 322–334, Sydney, Qld., Australia, Aug 2006.

[5] S. Baruah, M. Bertogna, and G. Buttazzo. *Multiprocessor Scheduling for Real-Time Systems.* Springer Publishing Company, Incorporated, 2015.

[6] S. Baruah and J. Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 195–202, Porto, Portugal, July 2003. IEEE.

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

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in fesource allocation. *Algorithmica*, 15(6):600–625, 1996.

[9] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *2010 31st IEEE Real-Time Systems Symposium*, pages 14–24, San Diego, CA, USA, Nov 2010. IEEE.

[10] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 149–160, Tucson, AZ, USA, December 2007. IEEE.

[11] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*, pages 209–218, Balearic Islands, Spain, July 2005. IEEE.

[12] J. Boyar, G. Dósa, and L. Epstein. On the absolute approximation ratio for first fit and related results. *Discrete Applied Mathematics*, 160(13):1914 – 1923, 2012.

[13] D. Casini, A. Biondi, and G. Buttazzo. Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing. In M. Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS 2006)*, pages 101–110, Rio de Janeiro, Brazil, Dec 2006. IEEE.

[15] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974.

[16] U. C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, February 2008.

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

[18] L. J. Flynn. Intel Halts Development Of 2 New Microprocessors. https://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html, May 2004.

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

[20] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 205–217, New York, NY, USA, 1972. ACM.

[21] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[22] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256 – 278, 1974.

[23] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari. An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software*, 85(10):2405 – 2416, 2012. Automated Software Evolution.

[24] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, June 2016.

[25] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *Proceesings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, pages 3–13, Brussels, Belgium, July 2010. IEEE.

[26] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[27] J. M. López, M. García, J. L. Diaz, and D. F. Garcia. 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)*, pages 25–33, Stockholm, Sweden, June 2000. IEEE.

[28] T. Megel, R. Sirdey, and V. David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *2010 31st IEEE Real-Time Systems Symposium*, pages 37–46, San Diego, CA, USA, Nov 2010. IEEE.

[29] OpenMP ARB. OpenMP Application Program Interface, version 5.0, 2018.

[30] E. Quiñones, S. Royuela, C. Scordino, L. M. Pinho, T. Cucinotta, B. Forsberg, A. Hamann, D. Ziegenbein, P. Gai, A. Biondi, L. Benini, J. Rollo, H. Saoud, R. Soulat, G. Mando, L. Rucher, and L. Nogueira. The AMPERE Project: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization. In *Proceedings of the 23rd IEEE International Symposium on Real-Time Distributed Computing (IEEE ISORC 2020)*, Nashville, Tennessee (turned to a virtual event), 2020. IEEE.

[31] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proceesings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, pages 104–115, Vienna, Austria, Nov 2011. IEEE.

[32] N. Saranya and R. C. Hansdah. Dynamic partitioning based scheduling of real-time tasks in multicore processors. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 190—197, Auckland, New Zealand, April 2015.

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

[34] P. Valente and G. Lipari. 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), pages 10 pp.–320, Miami, FL, USA, December 2005. IEEE.

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