An Evaluation of Adaptive Partitioning of Real-Time Workloads on Linux

Andrea Stevanato, Tommaso Cucinotta, Luca Abeni
Scuola Superiore Sant'Anna, Pisa, Italy
{a.stevanato,t.cucinotta,l.abeni}@santannapisa.it

Daniel Bristot De Oliveira
Red Hat, Inc., Pisa, Italy
bristot@redhat.com

***Abstract*—This paper provides an open implementation and an experimental evaluation of an adaptive partitioning approach for scheduling real-time tasks on symmetric multicore systems. The proposed technique is based on combining partitioned EDF scheduling with an adaptive migration policy that moves tasks across processors only when strictly needed to respect their temporal constraints. The implementation of the technique within the Linux kernel, via modifications to the `SCHED_DEADLINE` code base, is presented. An extensive experimentation has been conducted by applying the technique on a real multi-core platform with several randomly generated synthetic task sets. The obtained experimental results highlight that the approach exhibits a promising performance to schedule real-time workloads on a real system, with a greatly reduced number of migrations compared to the original global EDF available in `SCHED_DEADLINE`.**

***Index Terms*—Real-Time Scheduling, Real-Time Operating Systems, Linux Kernel**

I. Introduction

Nowadays, processing platforms stopped evolving towards higher and higher frequencies, taking a tight turn towards multi-core and multi-processor architectures. This allowed manufacturers to keep deploying hardware with consistently higher and higher computing capabilities, overcoming the thermal and heat dissipation problems which led to the non-practicality of releasing microprocessors with frequency beyond 5 GHz for the general public [1].

Parallel computing platforms are not only taking off in personal and general-purpose computing (besides high-performance computing, where they have been the standard for decades). They are also becoming increasingly adopted and playing a key role in embedded and real-time systems, where the growing richness of the required features implies the use of more and more powerful processing platforms [2], often made available through multi-core architectures, GPU and FPGA acceleration [3]. Hence, it is becoming increasingly important to design efficient and practical scheduling techniques for real-time applications on multi-core and multi-processor platforms. However, this is known to be a non-trivial problem [4].

Multi-core real-time schedulers are generally described [5] as *global* or *partitioned* schedulers. In the former case, tasks can be migrated among cores to respect some scheduling invariant, whilst in the latter one tasks are statically partitioned among the available CPU cores, and a single-processor scheduler can be used on each core. A useful trade-off between these two extremes is the one of *clustered* schedulers, where cores are partitioned in clusters, tasks are statically assigned to clusters and scheduled globally within each cluster.

An interesting new approach that tries to reconcile these different scheduling approaches exploiting their advantages is represented by *adaptive partitioning* EDF (apEDF) [6], [7]. This technique consists in combining an EDF-based partitioned real-time scheduler, with a policy for dynamic migration of tasks among cores until a schedulable task partitioning is reached, if possible. At such point, migrations stop and the scheduler converges to a partitioned scheduler. This happens for example if a first-fit allocation policy is used, and the task-set satisfies well-known conditions on its total utilization [8], [9], making the technique viable for *hard real-time* workloads. Even when said conditions are not verified, if a schedulable tasks partitioning does not exist, adaptive partitioning falls back to approximating global EDF [6], so that interesting properties of such an approach can be preserved, like the bounded tardiness, making it suitable for *soft real-time* workloads. However, although the adaptive partitioning approach seems to have promising properties, it has not been implemented in a real scheduler yet, having only been evaluated through simulations, to the best of our knowledge.

In this paper, we present the first implementation of an adaptively partitioned EDF scheduler, based on Linux, an increasingly attractive operating system (OS) for real-time workloads. Currently, Linux supports global EDF scheduling via the `SCHED_DEADLINE` policy [10][1]. Motivated by the results found by simulation in [6], [7], where apEDF has been shown to possess a distinguished advantage over a global scheduling approach, we implemented apEDF in Linux modifying `SCHED_DEADLINE`, replacing its migration mechanism (which implements global EDF by default) with a new mechanism based on apEDF. The new scheduler, which is made available as an open-source patch to `SCHED_DEADLINE` for the community, is thoroughly evaluated by running synthetic randomly generated task sets, as common in the real-time scheduling community, in the simplifying scenario of independent (non-interacting) periodic real-time tasks. The obtained results show that the modified scheduling policy performs better than the

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".

[1]More information about the `SCHED_DEADLINE` can be found in the official Linux kernel documentation at https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt

original one, obtaining a slightly better deadline-miss ratio, with a greatly reduced number of migrations. Note that this paper focuses on an experimental evaluation of the proposed scheduler. Readers interested in schedulability analysis arguments can refer to prior works as detailed next.

II. Related Work

Several works exist in the literature dealing with scheduling of hard or soft real-time task sets on multi-processor systems. In the former case [11]–[15], even a single deadline miss cannot be tolerated and is considered a system failure. In the latter case [16], [17], a few deadline misses can be acceptable if their number and frequency can be kept under control.

Approaches based on partitioned scheduling are known to have the potential of reusing well-known optimal results from the single-processor real-time scheduling literature, like the EDF utilization bound [18] for independent periodic tasks, but they add the burden of having to partition the task set upfront across the CPUs, which is a NP-hard problem when optimality is needed. This is normally tackled via integer linear programming techniques applied off-line [19], [20].

On the other hand, approaches based on global scheduling are easier to adopt, but their capability to saturate the underlying physical resources is generally reduced. Indeed, for example, simple utilization-based tests for schedulability analysis on multi-processors are characterized by poor and quite pessimistic utilization bounds [21]–[23]. However, in soft real-time systems one can load the system beyond said limits, as generally there are techniques to compute the maximum tardiness a task can achieve under certain conditions, like the well-known tardiness bound for global EDF [16], [24].

Modifying global EDF restricting migrations at job-level only [25], it is possible to improve its utilization bound, and prove that it is optimal among fixed-job-priority algorithms [11]. A number of other works exist about optimality of global multiprocessor scheduling algorithms [12]–[15], [26].

However, due to the additional complexity needed in the realization of the above techniques, often requiring higher implementation overheads, common real-time Operating Systems focus mostly on simpler scheduling algorithms, either partitioned or global, and based on either fixed-priority or EDF. Specifically, partitioned fixed-priority scheduling is the preferred choice in hard real-time systems. On the other hand, global EDF-based scheduling is increasingly popular in soft real-time ones, as witnessed by the `SCHED_DEADLINE` policy available in the mainline Linux kernel today [10], often applied to real-time multimedia workloads [27]. Moreover, a number of works can be found with experimental comparisons among the performance of global vs partitioned scheduling techniques under various workload conditions [5], [28].

Linux is a popular platform of choice for evaluating the effectiveness of real-time scheduling algorithms, often prototyped as invasive modifications to the kernel. Indeed, this is the case of the Litmus-RT framework [29], used for empirical comparisons among a number of RT scheduling algorithms and resource handling protocols [30]. Albeit interesting, this platform was not used in the present work, as we tried to adapt directly the global EDF scheduler in `SCHED_DEADLINE`, so to obtain a minimally invasive patch to the mainline code base.

In a number of works, on-line partitioning techniques were investigated, to leverage the advantages of partitioned scheduling, yet being capable of adapting the system configuration depending on dynamic run-time workload conditions.

To this purpose, common heuristics that have been investigated include first-fit, worst-fit and next-fit, which have been studied in depth also in other contexts, such as memory management [31]–[33]. Useful surveys on the topic can be found in [34], [35]. Many of these works focus on the concept of *absolute approximation ratio*: this is the minimum number of bins that are needed to pack a number of items with different weights, when using one of the above mentioned bin-packing heuristics, as compared to the number that would have been sufficient if 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 [35]. 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.

Note that the present study is strongly motivated by [8], where heuristic partitioning techniques are applied to real-time scheduling, analyzing the effectiveness of utilization-based admission tests and demonstrating optimality of the first-fit policy, among the ones usable in the context.

III. Definitions and Background

This section provides some definitions and a quick overview of the adaptive partitioning approach.

A. Task Model

The scheduler selects tasks from a (dynamic) set $\Gamma = \{\tau_i\}$ of real-time tasks τ_i and dispatches them on M (identical) CPU cores. A real-time *task* τ_i is seen as a stream of *jobs* $\{J_{i,k}\}$, each arriving (becoming ready for execution) at time $r_{i,k}$, executing for a time $c_{i,k}$ and then finishing at time $f_{i,k}$. Notice that the finishing time $f_{i,k}$ of each job depends on the scheduling decisions. Each task is also associated with a relative deadline D_i and each job $J_{i,k}$ must finish within its absolute deadline of $d_{i,k} = r_{i,k} + D_i$: if $f_{i,k} \le d_{i,k}$, then the deadline of $J_{i,k}$ is *respected*, otherwise it is *missed*. Task τ_i respects all of its deadlines if $\forall k, f_{i,k} \le d_{i,k} \Rightarrow \forall k, f_{i,k} - r_{i,k} \le D_i$. Finally, the *tardiness* of job $J_{i,k}$ is defined as $\max\{0, f_{i,k} - d_{i,k}\}$.

A real-time task τ_i is often *periodic* with period P_i, if $\forall k, r_{i,k+1} - r_{i,k} = P_i$, or *sporadic* with minimum inter-arrival time P_i among subsequent jobs, if $\forall k, r_{i,k+1} - r_{i,k} \ge P_i$; in this work, we make the simplifying assumption of *implicit deadlines*, i.e., $\forall i,\ D_i = P_i$, and we assume to know a reasonable estimation of the Worst-Case Execution Time (WCET) C_i of each real-time task, respecting the condition: $C_i \ge c_{i,k}\, \forall k$.

Based on these definitions, it is possible to define a task *utilization* C_i/P_i, which is often used to perform admission tests. For example, it is well known [18] that, on single-CPU systems, a set of periodic or sporadic real-time tasks as defined above, scheduled by the EDF algorithm [36], will not miss any deadline if $\sum_i C_i/P_i \leq 1$.

B. Scheduling on Multi-Core Platforms

In what follows, with reference to symmetric multi-processing (SMP) architectures, we use the terms *core*, *CPU* or *processor* interchangeably to refer to a single independent computation unit. Note that dealing with such problems as cache-level or other architecture-level interferences among tasks running on different cores or processors, sharing cache memories or memory controllers on the platform, are out of scope for the present paper. In a simplistic view, any possibe worst-case interference of said kind is assumed to be implicitly accounted for in the WCETs used for the tasks. Moreover, we restrict the discussion to EDF-based scheduling of real-time task sets on multi-processor platforms.

Traditionally, scheduling is performed with either *partitioned* or *global* approaches. The former ones partition all 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 scheduling can be used on each core. This allows analyzing the schedulability of each Γ_i independently on each core, assuming it is possible to partition the tasks among cores in a way that the condition $\sum_{\tau_i \in \Gamma_j} C_i/P_i \leq 1$ holds for each CPU j. Unfortunately, there are a number of well-known cases where this is not possible, like the task set $\Gamma = \{(6, 10, 10),\ (6, 10, 10),\ (6, 10, 10)\}$, where each task is specified as a triplet (C_i, D_i, P_i), which is not schedulable on 2 cores using a partitioned approach.

On the other hand, *global scheduling* approaches are able to dynamically migrate tasks among cores so that the m earliest-deadline ready tasks are scheduled, where m is the minimum between the number of cores and the number of ready tasks. Looking again at the task set $\Gamma = \{(6, 10, 10), (6, 10, 10), (6, 10, 10)\}$, it is clear that some deadlines will be missed also when using global EDF (gEDF), 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 *bounded tardiness* is a property of the gEDF algorithm which holds when $\sum C_i/P_i \leq M$, with M being the number of cores [16].

The Linux kernel implements a global EDF algorithm (for its `SCHED_DEADLINE` scheduling policy) by using M per-core ready task queues (*runqueues* — `rq`) and by migrating tasks among the runqueues so that the global EDF invariant (at any time, the M earliest deadlines ready tasks are scheduled) is respected. These migrations are performed by three different routines in the scheduler: `select_task_rq()` (invoked every time a task becomes ready for execution), `pull` (invoked every time a task blocks) and `push` (invoked every time a task is inserted in a runqueue). This way, every time the set of the M earliest deadline tasks changes, the kernel has a chance to balance it, so that such tasks are inserted in M different runqueues.

C. Adaptive Partitioning

In this paper, we perform an experimental evaluation of the apEDF and a^{2}pEDF algorithms, originally introduced in [7] where they were accompanied by a simulation-only evaluation. Basically, the essence of these algorithms is to schedule tasks so that runqueues are not overloaded ($\forall j, U_j \leq 1$) while reducing the number of migrations. In apEDF, pull operations are disabled and the `select_task_rq()` function is modified to select a runqueue $rq(\tau_i)$ so that its total utilization $U_{rq(\tau_i)} = \sum_{\{i: \tau_i \in \Gamma_{rq(\tau_i)}\}} C_i/P_i$ is smaller than the least-upper bound guaranteeing schedulability U^{lub} ($U^{lub} = 1$ for EDF). Here, $\Gamma_j = \{\tau_i : rq(\tau_i) = j\}$ is the set of tasks assigned to core j (that is, the set of tasks inserted in runqueue j). As a result, tasks are migrated only at wake-up time. In a^{2}pEDF, pull operations are re-introduced, but are only used when a core becomes idle (so, tasks are not pulled to cores that are already executing a `SCHED_DEADLINE` task) and do not pull tasks from cores having utilization $U_j \leq U^{lub}$. Additional details on how these techniques have been implemented in the Linux kernel are provided in Section IV. Finally, in the definitions of the apEDF and a^{2}pEDF algorithms the symbol d^j represents the absolute deadline $d_{h,l}$ of the job currently executing on core j (if the core is idle, $d^j = \infty$ is assumed).

Based on these definitions, the apEDF algorithm works by assigning each task τ_i to runqueue 0 ($rq(\tau_i) = 0$) when it is created, and by using Algorithm 1 (taken from [6]) to select a more appropriate runqueue for task τ_i when a new job $J_{i,k}$ arrives. Then, tasks are scheduled on each CPU according to EDF (so, $U^{lub} = 1$).

While the complete description of the algorithm can be found in [6], [7], here we just notice a few important facts:

- If τ_i is already assigned to a non-overloaded runqueue ($U_{rq(\tau_i)} \leq 1$), then it is not migrated (see Lines 1 and 2 of the algorithm). This means that *if the tasks' assignment converge to a schedulable task partition, then no further migrations are performed*
- If the First Fit (FF) heuristic is able to find a runqueue j where τ_i is schedulable ($U_j + C_i/P_i \leq 1$), then the task is migrated to core j ($rq(\tau_i) = j$); see Lines $4 - 8$ of the algorithm. This means that *the FF heuristic is used to search for a schedulable task partition*
- If τ_i cannot be assigned to any core without overloading it ($\forall j, U_j + C_i/P_i > 1$), then $rq(\tau_i)$ is selected as in the original `SCHED_DEADLINE` "push" operation, according to global EDF (see Lines $9 - 17$ of the algorithm). This means that the algorithm implements a *fallback to global EDF* if no schedulable task partition can be found.

The a^{2}pEDF algorithm extends apEDF by adding a pull operation that works as in Algorithm 2 (here, "second(j)" is the first non-executing task in runqueue j and d'^j is its absolute deadline — or ∞ if the runqueue contains less than 2 tasks). This is similar to the "pull" operation used by the original

```
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 core j)
Result: rq(τ_i)
1  if U_{rq(τ_i)} ≤ 1 then
       /* Stay on current core if schedulable    */
2      return rq(τ_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 ≤ 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
           /* τ_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(τ_i)
19 end
```

Algorithm 1: Algorithm to select a runqueue for a task τ_i on each job arrival (from [6]).

```
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 all the
        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 (from [6]).

`SCHED_DEADLINE` policy, but only pulls tasks from overloaded cores to idle cores.

The basic idea of a^{2}pEDF is to solve the issue of apEDF of not being *work-conserving* (a core might be idle while some real-time tasks are ready for execution but not scheduled). This issue happens because with apEDF the runqueue $rq(\tau_i)$ on which a job $J_{i,k}$ is enqueued is decided at time $r_{i,k}$ and a core different from $rq(\tau_i)$ might become idle later (so the selection can be sub-optimal).

Note that the apEDF algorithm follows the restricted migration approach [25], while a^{2}pEDF does not (as "pull" can migrate jobs after they started to execute on a core if they have been preempted by earlier-deadline jobs).

D. Properties of the Algorithms

A preliminary schedulability analysis of the apEDF algorithm[2] has already been performed [6], [7]. In particular, two important properties have been proved:

- apEDF is able to schedule every task set with $U = \sum_i C_i/P_i \leq (M+1)/2$ without deadline misses (derives from [8] assuming a maximum task utilization of 1);
- if a schedulable task partition is found, after it is reached the migrations stop.

Based on these properties, it is already possible to provide some basic schedulability guarantees.

Other properties have been conjectured, and seem to be valid according to simulations. These conjectures (which will be confirmed by the experimental results presented in this paper) allow to provide soft real-time guarantees and more accurate hard real-time guarantees. In particular:

- if a schedulable task partition exists, then Algorithm 1 converges to it in a finite number of steps, by only migrating tasks that have been assigned to cores j with $U_j > 1$;
- hence, if an optimal partitioning algorithm can find a schedulable task partition, then only a few deadlines will be missed (in the first jobs of each task). After an initial transient, no deadlines will be missed;
- if $\sum_j U_j \leq M$, then each task τ_i experiences a bounded tardiness: $\exists L : \forall \tau_i \in \Gamma, \max_k\{f_{i,k} - d_{i,k}\} \leq L$.

These theoretical and experimental results show that apEDF and a^{2}pEDF are designed to provide the best properties of partitioned EDF and global EDF.

IV. Implementation

This section describes our modifications to the Linux kernel version 5.7.0, to implement apEDF and a^{2}pEDF as described in Section III-C. The patch is publicly available at: https://github.com/thymbahutymba/linux/tree/v5.7-apedf. Our implementation relies conveniently on the kernel scheduler

[2] The analysis applies to a^{2}pEDF too.

design based on multiple per-core runqueues, as described in Section III-B. The patch modifies `SCHED_DEADLINE` with additional code that can be used to switch at run-time the scheduler behavior among the original gEDF policy, apEDF or a^{2}pEDF.

The main functions implementing the migrations for gEDF that have been modified in this work are:

- `push_dl_task();`
- `pull_dl_task();`
- `select_task_rq_dl();`
- `enqueue_task_dl().`

The `push_dl_task()` function has the purpose to find a suitable CPU where to possibly push ready-to-run tasks (the "pushable" tasks) from the current runqueue. In the original code base, this is done either to a CPU that is not running any `SCHED_DEADLINE` task, or to the one that is running the `SCHED_DEADLINE` task with the least urgent deadline.

In apEDF, the original `push_dl_task()` function has been renamed as `push_dl_task_global_edf()`, while a new function `push_dl_task_xf()` has been added to realize the apEDF logic in Algorithm 1. In the end, the `push_dl_task()` has been rewritten as a wrapper that calls either one or the push functions, depending on the kernel configuration. Additionally, for a^{2}pEDF, the original `pull_dl_task()` function has been renamed as `pull_dl_task_global_edf()`, and a new function `pull_dl_task_xf()` has been added to implement the a^{2}pEDF logic in Algorithm 2. Finally, the `pull_dl_task()` function has become a simple wrapper that calls either one of the pull functions, depending on the kernel configuration.

In gEDF, when all cores are running deadline tasks, push and pull use the following functions to find the runqueue running the task with the latest deadline:

- `find_lock_later_rq();`
- `find_later_rq();`
- `cpudl_find().`

Therefore, in the implementation of apEDF the same scheme has been reused for the push operation. In `push_dl_task_xf()` the functions that are exploited, to find the first runqueue where the pushable task fits, are:

- `find_lock_xf_suitable_rq();`
- `find_xf_suitable_rq();`
- `first_fit_cpu_find().`

In order to implement the first-fit policy, we exploited the total utilization U_j already tracked by the Linux kernel, in the form of the "runqueue bandwidth" field stored in the `this_bw` field of the `dl_rq` data structure of a runqueue, implementing Lines $1 - 8$ of Algorithm 1.

For the implementation of a^{2}pEDF, the function `pull_dl_task_xf()` has been added so that, if the a^{2}pEDF is the current scheduler, the pull is started only from an idle runqueue and pulls only from an *overloaded* runqueue, i.e., one with a total bandwidth greater than 1.

The aim of the `select_task_rq_dl()` function is to search the right CPU for a task when it is created, or when it wakes up after suspension. A criticality we faced in this regard, is due to the fact that, when changing scheduling policy from CFS to `SCHED_DEADLINE`, the `select_task_rq_dl()` was not called. In our apEDF implementation, in the worst-case a new `SCHED_DEADLINE` task is pushed to the right CPU, according to the apEDF policy, when it wakes up after its first block/sleep. This will be fixed in a future revision of the patch.

In order to choose whether to use gEDF with respect to apEDF or a^{2}pEDF, a few sysctl tunables have been added. These tunables may be found in the `/proc/sys/kernel` directory, all with a common prefix of `sched_dl`:

- `sched_dl_policy` allows for choosing the policy to use, setting it to 0 for gEDF, or 1 for apEDF or a^{2}pEDF;
- `sched_dl_xf_pull` allows to enable the pull of a^{2}pEDF, when set to 1;
- `sched_dl_fallback_to_gedf` allows to enable the fallback to gEDF in the push operation, when set to 1 (lines $9 - 17$ in Algorithm 1).

Note that real-time tasks often implement a periodic behaviour using the `clock_nanosleep()` system call (or similar). In this case, if a job of the periodic task finishes after the end of the period the task does not block waiting for the next activation (because when `clock_nanosleep()` is invoked to wait until a time in the past, it returns immediately without blocking the task). Therefore, in this case, the task continues execution straight into the new job, and it does not wake-up at its beginning. However, there is no way the kernel can distinguish the beginning of the new job, unless specialized APIs are introduced for the purpose[3]. We deal with this scenario by recurring to a push operation when the reservation is replenished.

Finally, when a task changes its scheduling policy to `SCHED_DEADLINE` without blocking, the kernel can continue to execute it on its previous core, breaking the apEDF policy. To fix this issue, the first time that the task blocks and wakes-up the push function has to be invoked even if the task is not executing on an overloaded core (in other words, the first time that Algorithm 1 is invoked for a task, the check at Line 1 must be skipped).

V. Experimental Evaluation

The apEDF and a^{2}pEDF algorithms, implemented modifying the `SCHED_DEADLINE` code base as described in Section IV, have been experimentally validated, comparing their performance with the one of the mainline gEDF of Linux. To this purpose, a number of task sets have been randomly generated, and run to the hyperperiod, dumping on disk the experienced response-time of all the jobs at the end of each run, as well as the number of migrations across CPUs experienced by each task throughout each run. This allowed us to build useful statistics by which the mentioned comparison is carried out, through the text that follows.

[3]If the application uses the `sched_yield()` function to terminate the current job and wait for the next period, then it always blocks, however the use of this capability of `SCHED_DEADLINE` is discouraged because of the risk of skipping a whole period needlessly.

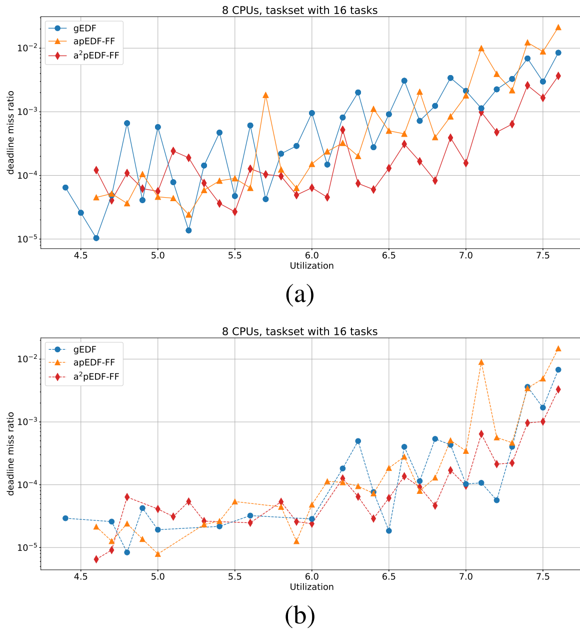


Fig. 1. Mean (a) and median (b) of the deadline-miss ratio (on the Y axis) achieved with various values of U (X axis), in the case of 8 CPUs and 16 tasks. Note that points that are not marked in correspondence of X ticks imply a value of 0 on the Y axis, which is logarithmic.

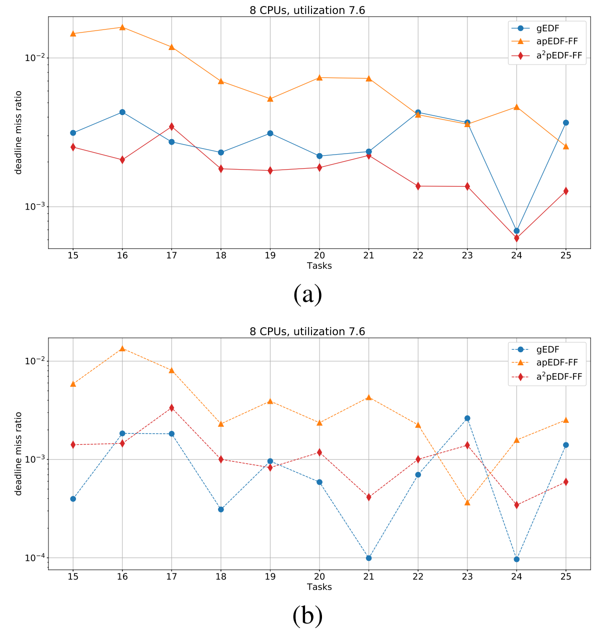


Fig. 2. Mean (a) and median (b) of the deadline-miss ratio (on the Y axis) achieved with various numbers of tasks (X axis), in the case of 8 CPUs and $U = 7.6$. Note that points that are not marked in correspondence of X ticks imply a value of 0 on the Y axis, which is logarithmic.

A. *Taskset Generation and Experiments Set-up*

To carry out the experiments, many task sets have been generated by using the well-known Randfixedsum algorithm [37], using different numbers of tasks, from 2 to 8 tasks per core, a total utilization ranging from $0.4 * M$ to $0.95 * M$, with a number of cores M ranging from 2 to 8. Note that the range of tried utilizations goes well beyond the limit of $(M+1)/2$, where apEDF is known to behave well from theory [8]. For each configuration, 10 different task sets have been generated, and each run lasted for a time equal to twice the hyperperiod $2 * H$.

The experiments have been carried out on a Dell R630 dual-socket server equipped with two Intel(R) Xeon(R) E5-2640 v4 CPUs at 2.40GHz, with frequency blocked at 50% and hyperthreading disabled (thus 20 cores are available, albeit we used only up to 8 cores as the focus of this paper is on the comparison with simulation-based results reported in [6]).

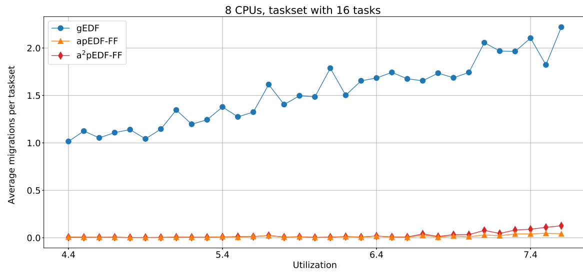
B. *Experimental Results*

Figure 1 (a) shows the average deadline-miss ratio (on the Y axis) obtained at a total utilization U of the task sets (on the X axis) varying in the range from 4.4 to 7.6, with $M = 8$ CPUs and 16 tasks. As visible, a^{2}pEDF behaves noticeably better than both apEDF and gEDF at high utilization values. This result is also confirmed by the median of the deadline-miss ratio experienced by the various task sets of each utilization, as reported in Figure 1 (b).

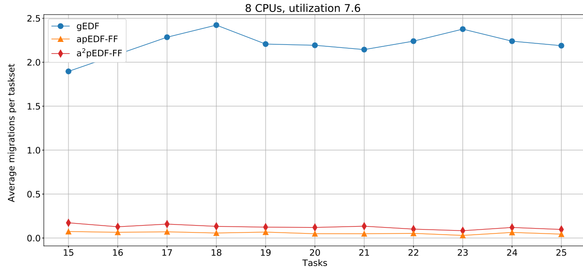
Figure 2 (a) shows the mean (on the Y axis) of the deadline-miss ratio achieved with a variable number of tasks, in the range from 15 to 25, at a fixed total utilization of the task sets at $U = 7.6$, and still on $M = 8$ CPUs. Even in this case, the results show that a^{2}pEDF behaves quite well with respect to the other two. However, the median of the deadline-miss ratio, visible in Figure 2 (b), highlights that often gEDF behaves slightly better than a^{2}pEDF, but there are task sets for which it behaves a lot worse, thus the apparent discrepancy between the mean and median deadline-miss ratio plots.

Figure 3 (a) depicts the average number of migrations per job experienced by the task sets with each algorithm, with 16 tasks and at varying total utilizations in the range from 4.4 to 7.6. Note that gEDF has the highest number of migrations which increase at higher utilizations. a^{2}pEDF has a higher number of migrations than apEDF due to the pull operation, however the migrations are close to zero. The fact of having fewer migrations helps in reducing the number of missed deadlines. Similarly, Figure 3 (b) reports the same information fixing $U = 7.6$ but with a number of tasks varying in the range from 15 to 25. It is evident that, as expected, gEDF suffers of the highest number of migrations, whilst a^{2}pEDF has a slightly higher number of migrations than apEDF, again expected due to the exploitation of the pull opportunities. Considering that task migrations risk to have a non-negligible impact on the execution times of the tasks, we can conclude that, also from this viewpoint, a^{2}pEDF exhibits a very good behavior.

Figure 4 (a) shows the average deadline-miss ratio (on the Y axis) obtained at a total utilization U of the task sets (on the X axis) varying in the range from 2.4 to 3.8, with $M = 4$ CPUs and 16 tasks. As visible, a^{2}pEDF behaves better than both apEDF and gEDF almost always, indeed for $U < 3.6$ the average deadline-miss ratio is equal to 0. However, for



(a)



(b)

Fig. 3. Average number of migrations per job (on the Y axis), averaged through all the task sets, in the case of 8 CPUs, with: (a) various values of U (X axis) and 16 tasks; (b) various numbers of tasks (X axis) and $U = 7.6$.

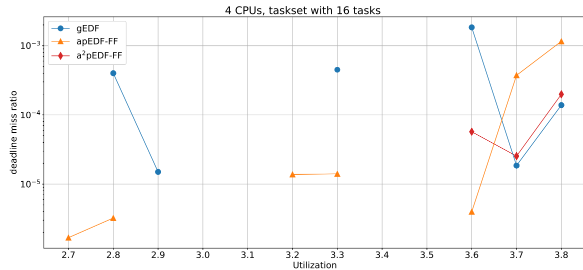
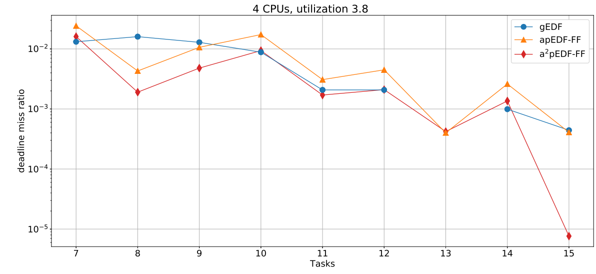


Fig. 4. Mean of the deadline-miss ratio (on the Y axis) achieved with various values of U (X axis), in the case of 4 CPUs and 16 tasks. Note that points that are not marked in correspondence of X ticks imply a value of 0 on the Y axis, which is logarithmic.

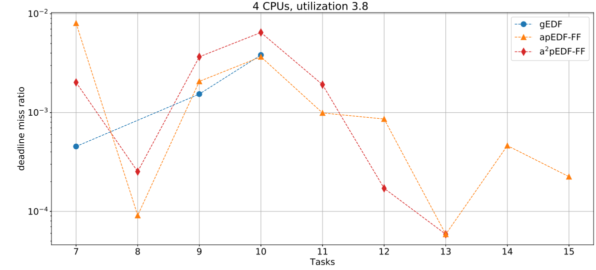
utilization 3.7 and 3.8, a^{2}pEDF and gEDF are almost similar.

Figure 5 (a) shows the mean (on the Y axis) of the deadline-miss ratio achieved with a variable number of tasks, in the range from 7 to 15, at a fixed total utilization of the task sets of $U = 3.8$, and $M = 4$ CPUs. Even in this case, the results show that a^{2}pEDF behaves quite well with respect to the other two. However, the median of the deadline-miss ratio, visible in Figure 5 (b), for a number of tasks greater than 10, shows again that gEDF often performs slightly better than a^{2}pEDF, being a work-conserving policy, but in some cases it performs a lot worse (similarly to what reported in Figure 2).

Figure 6 (a) depicts the average number of migrations per job experienced by the task sets with each algorithm, with 16 tasks and varying total utilizations in the range from 2.4 to 3.8. Note that, even in this case, gEDF has the highest number of migrations which increase at higher utilizations. a^{2}pEDF has a higher number of migrations than apEDF due to the pull operation, however the migrations are close to zero confirming

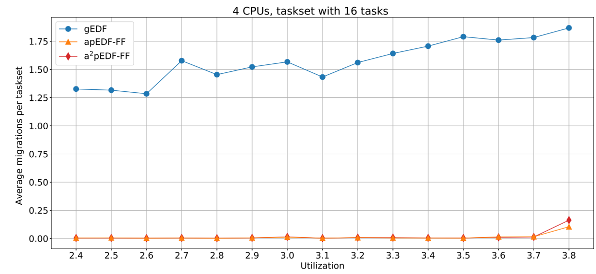


(a)

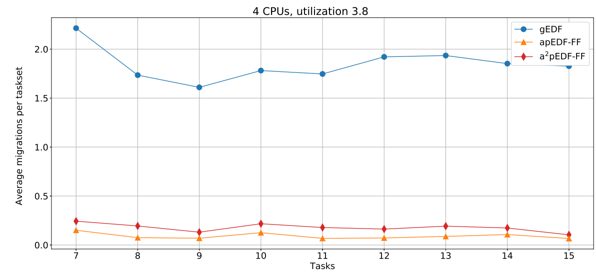


(b)

Fig. 5. Mean (a) and median (b) of the deadline-miss ratio (on the Y axis) achieved with various numbers of tasks (X axis), in the case of 4 CPUs and $U = 3.8$. Note that points that are not marked in correspondence of X ticks imply a value of 0 on the Y axis, which is logarithmic.



(a)



(b)

Fig. 6. Average number of migrations per job (on the Y axis) experienced by the task sets, in the case of 4 CPUs, with: (a) various values of U (X axis) and 16 tasks; (b) various numbers of tasks (X axis) and $U = 3.8$.

again that, having fewer migrations helps in reducing the number of missed deadlines. Similarly, Figure 6 (b) reports the same information fixing $U = 3.8$ but with a number of tasks varying in the range from 7 to 15. It is evident that, as expected, gEDF suffers of the highest number of migrations, whilst a^{2}pEDF has a slightly higher number of migrations than apEDF, again expected due to the exploitation of the pull opportunities.

Note that, among the experiments mentioned above, we measured an overhead due to the apEDF and a^{2}pEDF algorithms below $3\mu s$. This was the maximum duration of the select runqueue and push kernel functions, among all invocations observed over roughly 12 hours of experimentation with the above mentioned scenarios.

VI. Conclusions and Future Work

This paper presented a study on the properties and performance of EDF-based adaptive partitioning strategies [6] for scheduling real-time tasks on multi-processor platforms. In particular, it focused on the apEDF and a^{2}pEDF scheduling algorithms, that allow for a schedulability analysis that is less pessimistic than the one known in the literature for gEDF, while allowing to handle in a soft real-time sense non-partitionable task sets that partitioned EDF could not schedule. This results in better performance of the scheduled task sets, from the viewpoint of both the experienced deadline-miss ratio, and the number of migrations enforced by the scheduler. This has been highlighted through an extensive set of experiments conducted using an implementation of the technique within the `SCHED_DEADLINE` code base in the Linux kernel, and applied on synthetic randomly-generated task sets.

The presented results let us conclude that the a^{2}pEDF algorithm provides a smaller tardiness than apEDF at high utilizations, and with a small number of tasks. In this situation, apEDF performs slightly worse than gEDF while a^{2}pEDF provides better performance than gEDF. The improvements of a^{2}pEDF come at a reasonable cost, due to the additional migrations performed in pull operations, which are designed to impact only tasks that are running (both apEDF and a^{2}pEDF never migrate a running task).

In the future, it would be interesting to evaluate the new scheduler with a number of additional and more realistic workload scenarios: for example, data-intensive applications for which the impact of the reduced migrations could have a noticeable impact, as well as audio and video processing pipelines on Android platforms, considering that in this context there has already been some interest in exploiting real-time scheduling [27]. However, such real scenarios need at least to extend the technique to make it applicable, for example, to the general case of real-time DAGs [38], and with an energy-awareness policy [39], for example leveraging the Energy Aware Scheduling framework in the Linux kernel.

Finally, additional work is needed to demonstrate what theoretical properties apEDF and a^{2}pEDF possess, and under what conditions exactly task sets with various characteristics can be guaranteed to be scheduled without deadline misses. A challenging practicality point in this area is the one to consider support for arbitrary affinity task sets, as unavoidable in a complex OS like Linux. A number of promising works [40], [41] exist in this area that might be merged into the present line of research.

References

[1] 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.

[2] V. Nélis, P. M. Yomsi, L. M. Pinho, J. C. Fonseca, M. Bertogna, E. Quiñones, R. Vargas, and A. Marongiu, "The Challenge of Time-Predictability in Modern Many-Core Architectures," in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), H. Falk, Ed., vol. 39. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 63–72. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4605

[3] 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 *23rd IEEE International Symposium on Real-Time Distributed Computing*, Nashville, Tennessee (virtual), 2020.

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

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

[6] L. Abeni and T. Cucinotta, "EDF Scheduling of Real-Time Tasks on Multiple Cores: Adaptive Partitioning vs. Global Scheduling," *SIGAPP Appl. Comput. Rev.*, vol. 20, no. 2, p. 5–18, Jul. 2020.

[7] ——, "Adaptive partitioning of real-time tasks on multiple processors," in *35th Annual ACM Symposium on Applied Computing*, New York, USA, 2020, p. 572–579.

[8] 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 *12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, 2000, pp. 25–33.

[9] A. Mascitti, T. Cucinotta, and L. Abeni, "Heuristic partitioning of real-time tasks on multi-processors," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, 2020, pp. 36–42.

[10] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, June 2016.

[11] S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 781–784, June 2004.

[12] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, Aug 2006, pp. 322–334.

[13] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *27th IEEE International Real-Time Systems Symposium*, Rio de Janeiro, Brazil, 2006, pp. 101–110.

[14] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, 2011, pp. 104–115.

[15] 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*, Brussels, Belgium, July 2010, pp. 3–13.

[16] U. C. Devi and J. H. Anderson, "Tardiness bounds under global edf scheduling on a multiprocessor," *Real-Time Systems*, vol. 38, no. 2, pp. 133–189, February 2008.

[17] P. Valente and G. Lipari, “An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors,” in *26th IEEE International Real-Time Systems Symposium*, Miami, FL, USA, 2005, pp. 311–320.

[18] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

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

[20] A. Wieder and B. B. Brandenburg, “Efficient partitioning of sporadic real-time tasks with shared resources and spin locks,” in *8th IEEE International Symposium on Industrial Embedded Systems*, Porto, Portugal, June 2013, pp. 49–58.

[21] M. Bertogna, M. Cirinei, and G. Lipari, “Improved schedulability analysis of edf on multiprocessor platforms,” in *17th Euromicro Conference on Real-Time Systems*, Balearic Islands, Spain, July 2005, pp. 209–218.

[22] M. Bertogna and M. Cirinei, “Response-time analysis for globally scheduled symmetric multiprocessor platforms,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. Tucson, AZ, USA: IEEE, December 2007, pp. 149–160.

[23] J. Goossens, S. Funk, and S. Baruah, “Priority-driven scheduling of periodic task systems on multiprocessors,” *Real-Time Systems*, vol. 25, no. 2, pp. 187–205, September 2003.

[24] J. Erickson, U. Devi, and S. Baruah, “Improved tardiness bounds for global edf,” in *22nd Euromicro Conference on Real-Time Systems*, 2010, pp. 14–23.

[25] 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*, vol. 38, no. 2, pp. 85–131, Feb 2008.

[26] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate progress: A notion of fairness in fesource allocation,” *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[27] A. Balsini, T. Cucinotta, L. Abeni, J. Fernandes, P. Burk, P. Bellasi, and M. Rasmussen, “Energy-efficient low-latency audio on android,” *Journal of Systems and Software*, vol. 152, pp. 182–195, 2019.

[28] 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*, vol. 85, no. 10, pp. 2405–2416, 2012.

[29] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, “Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers,” in *27th IEEE International Real-Time Systems Symposium*, 2006, pp. 111–126.

[30] B. B. Brandenburg and J. H. Anderson, “A Comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT,” in *Principles of Distributed Systems*, T. P. Baker, A. Bui, and S. Tixeuil, Eds. Springer, 2008, pp. 105–124.

[31] R. L. Graham, “Bounds on multiprocessing anomalies and related packing algorithms,” in *May 16-18, 1972, Spring Joint Computer Conference*, New York, NY, USA, 1972, pp. 205–217.

[32] D. S. Johnson, “Approximation algorithms for combinatorial problems,” *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 256–278, 1974.

[33] 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*, vol. 3, no. 4, pp. 299–325, 1974.

[34] D. Simchi-Levi, “New worst-case results for the bin-packing problem,” *Naval Research Logistics (NRL)*, vol. 41, no. 4, pp. 579–585, 1994.

[35] J. Boyar, G. Dósa, and L. Epstein, “On the absolute approximation ratio for first fit and related results,” *Discrete Applied Mathematics*, vol. 160, no. 13, pp. 1914–1923, 2012.

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

[37] 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*, Brussels, Belgium, 2010, pp. 6–11.

[38] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, “Response time analysis of sporadic dag tasks under partitioned scheduling,” in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016, pp. 1–10.

[39] A. Mascitti and T. Cucinotta, “Dynamic Partitioned Scheduling of Real-Time DAG Tasks on ARM big.LITTLE Architectures,” in *29th International Conference on Real-Time Networks and Systems (RTNS 2021)*, Nantes, France, April 2021.

[40] A. Gujarati, F. Cerqueira, and B. B. Brandenburg, “Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities,” in *25th Euromicro Conference on Real-Time Systems*, 2013, pp. 69–79.

[41] V. Bonifaci, B. Brandenburg, G. D’Angelo, and A. Marchetti-Spaccamela, “Multiprocessor real-time scheduling with hierarchical processor affinities,” in *28th Euromicro Conference on Real-Time Systems*, 2016, pp. 237–247.