Migrating Constant Bandwidth Servers on Multi-Cores

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

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

ABSTRACT

This paper introduces a novel admission test for partitioned CBS reservations on multi-core systems, that, on a new reservation arrival, is capable of exploiting better the CPU capacity in cases in which tasks have just recently left the CPU (for example, due to termination or migration to a different CPU). This is particularly useful for highly dynamic scenarios (with frequent arrivals of new tasks or leaves of existing ones) or when adaptive and possibly power-aware partitioning techniques (in which task migrations are triggered quite often to re-balance the workload among the available cores) are used.

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

ACM Reference Format:
Tommaso Cucinotta and Luca Abeni. 2021. Migrating Constant Bandwidth Servers on Multi-Cores. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3453417.3453441

1 INTRODUCTION

To improve computational power without increasing energy consumption too much, many hardware manufacturers are starting to commercialise massively multi-core CPUs. As a result, multi-core architectures which were traditionally employed in the context of high-performance computing (HPC), are starting to see interesting applications in new fields such as, for example, embedded systems. Here, the higher and higher richness of features demanded by modern and future embedded scenarios [23], often including real-time video processing and object recognition or time-series forecasting algorithms based on deep neural networks, as well as possessing non-negligible constraints in terms of *energy efficiency*, needs correspondingly higher complexity in the deployed software stack. This has to be capable of exploiting novel and often highly heterogeneous computing platforms, heavily accelerated using multi-core architectures, as well as graphic processing units (GPUs), tensor processing units (TPUs), or even FPGA fabrics.

This, of course, poses new requirements; first of all, the necessity to respect temporal constraints of real-time tasksets scheduled on multiple cores. Multi-core real-time schedulers can be organised in 2 main categories: *global schedulers* and *partitioned schedulers.* On a multi-core platform with m cores, a global scheduler is free to migrate tasks among the cores to respect some kind of invariant (i.e., the m highest priority tasks are scheduled, the m earliest deadline tasks are scheduled, etc...), while a partitioned scheduler statically assigns tasks to CPU cores and uses multiple instances of a single-core scheduling algorithm to select the tasks to be executed.

When using a partitioned scheduling approach, tasks are assigned to CPU cores so that the taskset bound to each core is schedulable, and the schedulability test, in simple cases, may be conveniently performed by using some kind of utilization-based admission control (the taskset assigned to a CPU core is schedulable if its utilization is smaller than a *least upper bound* U^{lub}). This way, the problem of partitioning a set of tasks among multiple cores can be reduced to an instance of the *bin packing problem* [12]. If real-time tasks can be dynamically generated and terminated, as well as if they are dynamically migrated among cores as often needed for energy efficiency reasons [18], then the utilization of each core can dynamically change (increasing when a task is associated with the core and decreasing if a task leaves a core). Then, an *on-line bin packing algorithm* can assign new tasks to cores so that their utilization is smaller than U^{lub}. In case this condition cannot be immediately respected, it is commonplace to try to move tasks among cores to create space for the newly arrived task(s) (this action is often called *re-packing*).

In other words, high-performance embedded real-time computing scenarios of today and tomorrow, have an increasing need for handling properly situations in which real-time tasks (or reservations) are migrated dynamically and often among cores.

Unfortunately, the mechanisms currently used to keep track of the dynamic utilization of the cores, used by the on-line task placement algorithm that assigns tasks to CPU cores, are often either incorrect (for example, the `SCHED_DEADLINE` scheduling policy provided by Linux immediately decreases the utilization of a core when a task migrates away) or too pessimistic [5]. In the former case, the instantaneous removal of the utilization of a task leaving a core from the overall core utilization may cause deadline misses if the admission logic admits new tasks, under quite common conditions (see Figure 1 later). In the latter case, we have the correct postponement of the removal of the task utilization from the overall core utilization at a later time, either the last deadline of the task

RTNS'2021, April 7–9, 2021, NANTES, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9001-9/21/04...$15.00
https://doi.org/10.1145/3453417.3453441

or, better, its last 0-lag time. However, an admission test that considers the leaving task utilization still occupied and non-available for an arbitrarily long time in the future reduces unnecessarily the chances to admit tasks on the CPU, as it will be shown in Section 5, and is particularly inefficient for highly dynamic scenarios where real-time tasks may need to be migrated often.

1.1 Contributions

This paper presents a new algorithm to keep track of the dynamic utilization of a CPU core, which is both correct and not too pessimistic, improving schedulability with respect to other correct mechanisms. Such an improved utilization tracking algorithm can be used to implement an efficient utilization-based admission control for dynamic real-time systems (allowing dynamic task creation and termination) but is also useful for efficiently supporting online bin-packing algorithms (by handling re-partitioning of real-time tasks or migration of real-time tasks in partitioned systems). In particular, this technique has been designed to improve the effectiveness of an adaptively partitioned variant [2] of the SCHED_DEADLINE policy [14] provided by Linux, which implements the CBS algorithm [1], based on EDF [10], using a utilization-based admission control.

1.2 Paper organisation

This paper is organised as follows: a brief overview of related research in the literature is provided in Section 2. Then, background concepts and notation elements are introduced in Section 3, which will be useful for a better understanding of the rest of the paper. Section 4 introduces some basic concepts related to dynamic partitioning. The proposed new approach for performing an effective utilization-based admission of EDF-scheduled real-time tasks is described in Section 5. Then, a proof of correctness of the technique is provided in Section 6, accompanied by results from an experimental evaluation by simulation, discussed in Section 7, recurring to the RTSim simulator [20]. Finally, conclusions are drawn in Section 8, along with a discussion of possible future work on the topic.

2 RELATED WORK

This paper focuses on dynamically partitioned scheduling of Constant Bandwidth Servers, where each server is bound to a CPU core and servers are migrated as needed to respond to dynamic changes of the tasksets hosted on each core. Different approaches use global scheduling so that explicit per-core utilization tracking is not needed, and no explicit action is needed when the workload changes. For example, the M-CBS algorithm [4] uses a slightly modified version of global EDF (introducing the notion of "high-priority server") to schedule CBSs on multiple processors.

The utilization tracking problem described in this paper is a particular case of *transient analysis* (analysing the schedulability of time-changing tasksets, with focus on the time intervals around the changes).

In previous work, scheduling transients have been mainly analysed in order to cope with mode changes of applications [9, 21, 22, 24], whereas this paper focuses on moving tasks between CPU cores in (dynamically) partitioned scheduling (a similar approach has been previously considered in semi-partitioned scheduling [6]). Most of the previous analysis is based on computing the demanded time[1] of a taskset (either through an exact approach or some efficient approximation to be used online [7, 8]). In some cases, the analysis is simplified (time-demand analysis can be computationally expensive) by considering the effects of a scheduling transient on the taskset to be finished after the latest deadline between the jobs involved in the transient [19] (and this is, of course, a very pessimistic approach). In other cases [13], the schedulability analysis is extended to consider the worst-case transient caused by possible mode changes, but this approach cannot be used in case of repacking of dynamic tasksets (which are not known a-priori).

In this paper, a simpler utilization-based approach is introduced, which generalises, formally proves and discusses in-depth, what was quickly mentioned in [5] while discussing an example of mode change due to elastic modifications to the periods of tasks. The approach introduced in this paper is both computationally simple and not too pessimistic: when a task leaves a core (either because it terminates or because it is migrated to a different core), the *exact time* at which the task utilization should be removed from the core utilization is computed, instead of simply waiting the end of the task period as done (for example) by r-EDF [3]. At the same time, when admitting a new task with a given period, a straightforward inspection of the scheduled changes of the active core utilization due to tasks that recently left the CPU, possibly falling within the first instance of the new task being admitted, allows for computing a safe maximum runtime for the task, that may be significantly higher than the one considered by traditional pessimistic admission tests that only look at the current instantaneous active utilization.

While some previous works on partitioned real-time scheduling analysed static tasksets, finding utilization-based conditions for the partitionability/schedulability [17], this paper focuses on dynamic tasksets, improving both the capacity of a system to accept new dynamically-created tasks, and its capability to dynamically migrate tasks among CPUs, as needed for example by energy management policies or other reasons.

3 BACKGROUND

3.1 Terminology and definitions

A real-time application is modelled as a set $\Gamma = \{\tau_i\}$ of real-time tasks $\tau_i = (C_i, T_i)$, where C_i is the worst-case execution time (WCET) of the task and T_i is the (minimum) inter-arrival time between task instances (jobs). In this work, the real-time tasks are assumed to be independent and self-suspension is not considered (task dependencies, shared resources, and self-suspension are left to future works). Each task is associated with a dedicated reservation (Q_i, P_i), meaning that τ_i is allowed (and guaranteed) to execute for Q_i time units every P_i. The fraction of CPU time reserved for the task (known as *reservation bandwidth*) is $U_i = Q_i/P_i$.

When the CPU reservation is implemented using the CBS [1] algorithm and a single processor is considered, if $Q_i \geq C_i \land P_i \leq T_i$ and all the reserved bandwidths satisfy

$$\sum_i \frac{Q_i}{P_i} \leq U^{lub} \tag{1}$$

[1]More specifically, the demand bound function

(with $U^{lub} = 1$ as EDF is being used), then τ_i is guaranteed not to miss any deadline.

According to the CBS algorithm, a server $S_i = (Q_i, P_i)$ is used to assign a *current budget* $q_i(t) \leq Q_i$ and a *scheduling deadline* $d_i(t)$, to each task τ_i, so that the tasks can be scheduled according to EDF using their scheduling deadlines.

When the CBS starts handling a task, $q_i(t)$ and $d_i(t)$ are initialised to 0. Then, when task τ_i wakes up at time t, the scheduler checks if the current scheduling deadline $d_i(t)$ can be used (if $q_i(t) < (d_i(t) - t)\frac{Q_i}{P_i}$), otherwise a new value $d_i(t) = t + P_i$ for the scheduling deadline is generated and the runtime $q_i(t)$ is recharged to Q_i. The current budget of a task τ_i executing from time t to time $t + dt$ is consumed as $q_i(t + dt) = q_i(t) - dt$ and when it arrives at 0 the task is depleted. When a task is depleted, it cannot be scheduled until its budget is recharged to Q_i ($q_i(t) = Q_i$) postponing the scheduling deadline by P_i ($d_i(t) = d_i(t) + P_i$). This replenishment can be performed immediately when the task is depleted (as done in the original CBS paper — this is the so-called "soft CBS" behaviour), or later at time $d_i(t)$ (as done by Linux — this is the so-called "hard CBS" behaviour). When task τ_i blocks at time t, its current runtime $q_i(t)$ and scheduling deadline $d_i(t)$ are kept unchanged, to be eventually re-used for the next task activations. Finally, when task τ_i terminates the server S_i associated to it is destroyed, discarding the remaining budget (however, its utilization should be remembered for some time, as we will see later).

When partitioned scheduling is used, each task (and the CBS serving it) is bound to a specific CPU core; the set of tasks (or equivalently reservations) bound to core h is indicated as Γ_h. If a *dynamic* partitioning approach is used, then the tasks partitioning can be occasionally moved between cores (adapting the partitioning to dynamic changes in the workload). In this case, the set of tasks (or reservations) bound to core h is indicated as $\Gamma_h(t)$ as it is a function of the current time t.

The utilization of core h at time t (due to dynamic tasks partitioning) is indicated as $V_h(t)$:

$$V_h(t) = \sum_{\tau_i \in \Gamma_h(t)} U_i. \tag{2}$$

In the simple case of statically partitioned task sets, $\Gamma_h(t)$ and $V_h(t)$ are constant, and, under the assumption of periodic (or sporadic) and independent real-time tasks, in light of Equation (1) the set of tasks $\tau_i \in \Gamma_h(t)$ is schedulable if

$$V_h(t) \leq U^{lub}. \tag{3}$$

3.2 A CBS implementation

An example of a real scheduler implementing a multi-processor variant of the CBS (similar to [4]) is the Linux `SCHED_DEADLINE` policy [14].

Although `SCHED_DEADLINE` implements *global EDF* by default, it is possible to configure it (either using task affinities or configuring exclusive scheduling domains via `cpusets`) to use as a *partitioned EDF* approach (which allows re-using the single-processor schedulability analysis)[2].

The focus of this paper is on making use of `SCHED_DEADLINE` as a partitioned CBS-based scheduler, where a set of real-time tasks can be partitioned across the available CPUs, and tasks can dynamically be created or terminated, or even migrated among CPUs for various reasons, as mentioned in Section 1. Our investigations dig into dynamic and adaptive partitioned CBS-based scheduling of tasks on multi-cores, an area that showed promising results in a prior work of ours [2] including extensive experimentation by simulation. In this context, a simple and efficient online admission test is a fundamental element, where it is interesting to re-use utilization-based tests due to their particular efficiency.

[2]More info at: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt.

4 DYNAMIC PARTITIONING

Dynamically partitioned scheduling is particularly useful for dynamic and open real-time systems, where tasks can dynamically enter and leave the system at any time. In these cases, we may need to occasionally re-arrange the tasksets partitions, for example to make room on a core for a newly arrived task (this is the so-called "re-packing" operation). Therefore, at some time t, we may need to migrate a task τ_i from a core Γ_h to a different core Γ_k.

When a task $\tau_i \notin \Gamma_h(t^-)$ (t^- denotes a time instant just before time t) is admitted onto a core h (due to a migration from another core, or creation of a new task), after verifying that this does not violate Equation (3), the task utilization U_i has to be added instantaneously to $V_h(t)$, therefore $V_h(t) = V_h(t^-) + U_i$.

However, when a task $\tau_i \in \Gamma_h(t^-)$ leaves its core h at time t, (because it terminates, or because it is moved to a different core), its utilization U_i cannot be immediately removed from $V_h(t)$. Otherwise, the admission test risks to admit real-time tasks that can cause deadline misses.

An example of the issue is depicted in Figure 1, where two tasks τ_1 and τ_2 saturate the CPU they are running onto. Upwards arrows denote arrival times, downwards ones denote absolute deadlines, and grey boxes represent when tasks are scheduled on the CPU. As visible, if task τ_1 dies just after having consumed its whole budget, and its utilization is mistakenly immediately subtracted from the total utilization $V_h(t)$ of the CPU, then any task τ_3 admitted right after, with a deadline earlier than the one of the remaining task τ_2, would cause a deadline miss of τ_2.

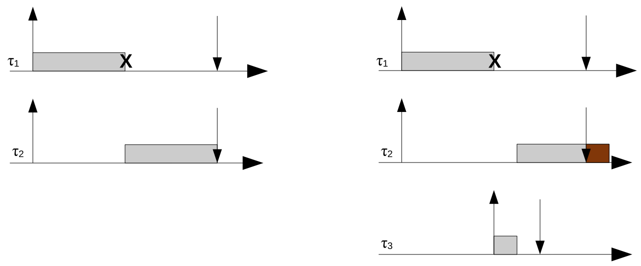


Figure 1: Example of deadline miss when the utilization of a task $t1$ that is removed from the CPU runqueue, is forgotten immediately and a new task $t3$ is mistakenly admitted too early.

A common way to avoid the just mentioned schedulability issue is to subtract U_i from $V_h(\cdot)$ not at time t, but after the next deadline of τ_i has passed (if the CBS is used, only after time $d_i(t)$). However, if τ_i is removed from $\Gamma_h(t)$ when it has consumed an amount of budget $q < Q_i$ smaller than the maximum budget Q_i (hence, the current budget is $q_i(t) = Q_i - q$), then it is well-known that [5] the safe time to subtract U_i from $V_h(t)$ is $\delta_i = d_i(t) - q_i(t)/U_i$[3], where "$t$" is the time when the task leaves the core. This time δ_i is often called *0-lag time* and corresponds to the time instant when $t = v(t)$, where $v(t)$ is the *virtual time* as defined by the GRUB algorithm [15].

Summing up, if a task τ_i leaves $\Gamma_h(t)$ at time t, its utilization U_i has to be remembered on core h until time δ_i, in the form of an *active migrated utilization* $V^m_{i,h}(t)$, defined as:

$$V^m_{i,h}(t) = \begin{cases} U_i \text{ if } \tau_i \notin \Gamma_h(t) \wedge t < \delta_i \\ 0 \text{ otherwise (if } \tau_i \in \Gamma_h(t) \vee t \geq \delta_i). \end{cases} \quad (4)$$

To keep track of the active migrated utilization of the tasks leaving a CPU core, when task τ_i leaves $\Gamma_h(t)$ at time t it is added to a set of *migrated tasks* $\Gamma^m_h(t)$. Task τ_i will be removed from $\Gamma^m_h(t)$ at time δ_i. Based on this, the overall active migrated utilization $V^m_h(t)$ of core h at time t is given by the sum of the contributions due to all previously migrated tasks $\tau_i \in \Gamma^m_h$, and is a piece-wise constant function:

$$V^m_h(t) = \sum_{\tau_i \in \Gamma^m_h(t)} V^m_{i,h}(t). \quad (5)$$

Note that, albeit Equation (5) seems to consider the whole history of tasks that migrated across CPUs, from a practical perspective the active utilization of a migrated and/or dead task expires and can be forgotten after its 0-lag time has passed (τ_i is removed from $\Gamma^m_h(t)$ at time δ_i). Therefore, in a real scheduler implementation, when at time t a task τ_i leaves the CPU h, it is sufficient to post a single-shot timer that, at time δ_i will decrease $V^m_h(t)$ by an amount equal to U_i. Such a timer is already used by `SCHED_DEADLINE` in the Linux kernel to keep track of the inactive utilization of tasks as used by the CPU reclaiming mechanism. This is the so-called task *inactive timer*.

5 PROPOSED APPROACH

The core utilization $V_h(t)$ and the active migrated utilization $V^m_h(t)$ can be used to implement an efficient and correct admission test for dynamic tasksets, where tasks can dynamically move between CPU cores. This new test relies on three main elements:

- the mechanism used to track the instantaneous active utilization of each CPU core, used for admission control;
- the observation that, when a task needs to enter a CPU core, the admission test can keep into account the scheduling deadline of the CBS serving the task to make a precise assessment of the maximum admissible budget;
- the application of this scheme to the dynamic migration of tasks in multi-core systems. This allows ensuring schedulability and precise accounting of CBS budgets without the excess of pessimism typically used in these scenarios.

5.1 Tracking per-core utilizations

Remember that $V_h(t)$ represents the overall utilization of all tasks $\tau_i \in \Gamma_h(t)$ bound to core h at time t, while $V^m_h(t)$ represents the utilization of tasks $\tau_i \in \Gamma^m_h(t)$ that left core h but still contribute to its active utilization at time t. These definitions allow extending the classical utilization-based admission tests shown in Equation (3) to support tasks that can dynamically enter and leave a CPU core by considering both $V_h(t)$ and $V^m_h(t)$. Hence, at time t a new task τ_i can be admitted on core h if

$$V_h(t)+V^m_h(t)+U_i \leq U^{lub} \iff U_i \leq U^{lub} - (V_h(t)+V^m_h(t)) \quad (6)$$

with $U^{lub} = 1$ if EDF is used as a scheduler.

The value of $V_h(t)+V^m_h(t)$ is a step function decreasing at the 0-lag times of the tasks that recently left core h, as shown for example in Figure 2. This example describes the evolution of the total per-core utilization (considering the tasks assigned on the core and the contributions of previously migrated tasks) when 3 tasks leave the core at times t_0, t_1 and t_2 (marked with an "X"). As it is possible to notice, the utilization does not decrease immediately at the times marked with "X", but is decreased later, at the 0-lag times of the 3 tasks. This is an important difference with respect to [5], where the utilization of a core is considered as constant until the 0-lag time of the latest task changing its utilization (in this case, leaving the core). In the figure, the dashed line (named "Delta max") indicates the evolution of the per-core utilization according to this last approach, clearly showing the advantages of the approach proposed in this paper (for example, a new task arriving — or migrating to this core — around time t_2 has a much higher probability to be accepted).

As $V^m_h(t)$ is a piece-wise constant function, it can be modelled as a sequence of (δ_i, U_i) pairs, meaning that U_i must be removed from the utilization of core h at time δ_i. This approach will be used in Section 6 to represent the per-core utilization $V_h(t) + V^m_h(t)$.

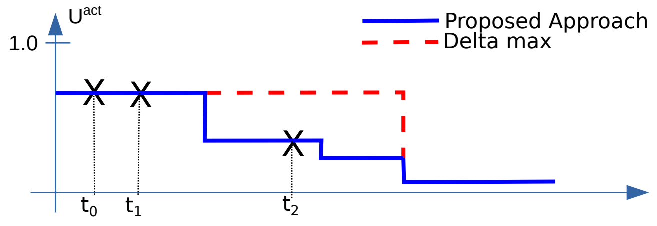


Figure 2: Example of the evolution of the per-core utilization with the proposed approach (continuous line) and according to [5] (dashed line).

5.2 Considering the scheduling deadline of the new server

A second brick of our proposed approach is how the step function $V_h(t) + V^m_h(t)$ is used in the admission test. For now, we make the simplifying assumption that the task needs to be admitted right after having been created, or after a migration occurred synchronously with the arrival of a new job for the task. In the next subsection, we will discuss how to remove this assumption.

The utilization-based test of (6) is improved taking into consideration also the scheduling deadline and the budget of the CBS serving the new task τ_i. Using this information and remembering

[3] In [5] the current job deadline and the remaining execution time $c_i(t)$ are used instead of the current budget and the server deadline, but the result is equivalent.

that if τ_i activates at time t then $d_i(t) = t + P_i$, we can translate the available utilization on the target core at time t (which is $U^{lub} - (V_h(t) + V_h^m(t))$) into a maximum acceptable budget for the incoming task:

$$Q_i \leq P_i \left(U^{lub} - V_h(t) - V_h^m(t)\right). \tag{7}$$

However, in this paper, we claim (and prove in Section 6) that we can exploit more budget coming from the additional bandwidth that is going to be freed due to any upcoming 0-lag times possibly falling between time t and $d_i(t) = t + P_i$.

The maximum amount of budget that can be used by a new task τ_i between time t and $t + P_i$ without breaking the taskset schedulability can be computed as:

$$Q_i \leq P_i \left(U^{lub} - V_h(t)\right) - \int_{s=t}^{t+P_i} V_h^m(s)ds, \tag{8}$$

where the integral can be replaced with a sum made over the steps of the $V_h^m(t)$ function, considering all the changing points falling between t and $t+P_i$. In particular, these points $\delta_1, ...\delta_k$ are caused by the k tasks[4] $\tau_j \in \Gamma_h^m(t) : \delta_j \leq t + P_i$ which recently left the system (are in $\Gamma_h^m(t)$) and have a 0-lag time in $[t, d_i(t)]$. Hence, the integral $\int_{s=t}^{t+P_i} V_h^m(s)ds$ can be computed as $\sum_{\tau_j \in \Gamma_h^m(t):\delta_j \leq t+P_i} (\delta_j - t)U_j$ and Equation (8) becomes:

$$Q_i \leq P_i \left(U^{lub} - V_h(t)\right) - \sum_{\tau_j \in \Gamma_h^m(t)} \min\{\delta_j - t, P_i\}U_j, \tag{9}$$

Visually, Figure 3 compares the maximum runtime Q_{max} admissible according to our new proposed technique in Equation (9), with respect to the lower one that would be admissible according to Equation (7), when evaluating admission of a new task τ_i served by a CBS with period P_i.

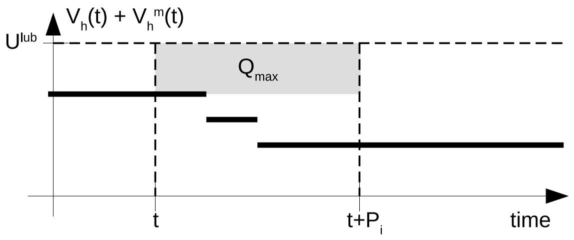
5.3 Handling migrations across CPU cores

When a task needs to be admitted onto a CPU core h because it is migrating from a different core k, two cases need to be considered

- In the first case, the task is migrated at the time of arrival of a new job. This is often done in order not to interfere with the task while it is executing, to avoid to incur in additional migration overheads
- In dynamic scenarios as mentioned above, it may arise the need for migrating a task in the middle of the execution of one of its jobs, for example to re-balance the workload after a task termination on another core. Still, such migration is only recommendable exploiting a moment in which the task is ready-to-run but not being scheduled (in `SCHED_DEADLINE`, the task is *pushable*).

In the second case, a task τ_i that migrates at time t from core k to core h comes with its leftover budget $q_i(t) \leq Q_i$, and its current absolute deadline $d_i(t)$. To guarantee correct completion of the job under execution on the target core h by the deadline $d_i(t)$, we need to deal with the first instance of the task as though it were served by a CBS with a runtime equal to $q_i(t)$, and period equal to $d_i(t) - t$. From the subsequent job onwards, the server exhibits its regular

[4]Notice that the taskset can be reordered so that the k tasks leaving the core are $\tau_1, ...\tau_k$.



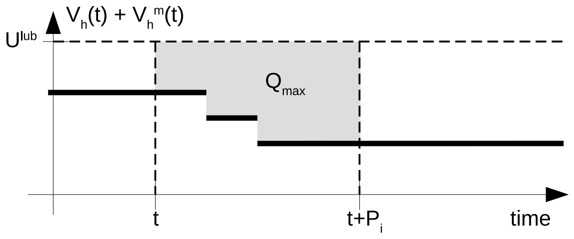


Figure 3: Highlight of the gain in the admitted budget for the proposed technique relying on Equation (8) (highlighted area in the bottom plot), compared to the one admitted using Equation (7) (highlighted area in the top plot).

periodic demand of Q_i time units every period P_i. Therefore, this case requires the task to pass two (simple) tests that need to be jointly performed:

(1) the budget-based admission test from Equation (9), using the current leftover budget $q_i(t)$ as the runtime (budget) to admit, and the value $d_i(t) - t$ as the server period over which to apply the test;
(2) the utilization-based test from Equation (6), (using the static CBS parameters Q_i and P_i) applied at time $d_i(t)$. If there is at least a 0-lag time $\delta_j \leq d_i(t)$, then this check allows admitting more tasks than the simple application of Equation (6) at time t, because $V_h^m(d_i(t)) < V_h^m(t)$.

The first test is useful when the task schedule is "lagging behind" on the source core k compared to a "fluid" execution, i.e., its 0-lag time is still in the past, so the first instance of the task needs higher utilization than the CBS static parameters would require. The second test is useful when the task schedule is "running ahead" of its "fluid" execution, i.e., its 0-lag time is in the future, so the first instance of the task needs temporarily a lower bandwidth to be available for its first instance, but it will need higher bandwidth from the second instance onwards. If the second test is not passed, but there are scheduled decreases of the active migrated utilization $V_h^m(t)$ in correspondence of additional future 0-lag times between $d_i(t)$ and $d_i(t) + P_i$, then it is possible to refine the test applying again the budget-based admission test from Equation (9), but performed as we had to admit at time $d_i(t)$ the full budget Q_i of τ_i.

Finally, a particular case of mention is deserved to the special (but very unlikely) case of a task τ_i that migrates from core h to core k at a time t and comes back to core h at a later time $t' \in]t, \delta_i]$ with a leftover budget $q_i(t') \leq q_i(t)$ and the same deadline $d_i(t)$. Assuming no other tasks were admitted on core h in the interval

$[t, t']$, with the proposed admission test, the task can resume execution on core h with no need for any special handling. Since the task left core h at time t with runtime $q_i(t)$ and deadline $d_i(t)$, its utilization $U_i = Q_i/P_i$ is scheduled to be removed from the core utilization at the 0-lag time $\delta_i = d_i(t) - \frac{q_i(t)}{U_i}$. In the worst case of the task coming back at the same time $t' = \delta_i$, with still its full residual budget $q_i(t)$, the task can be admitted because the maximum budget available till the deadline $d_i(t)$ contains at least the budget that is scheduled to be subtracted starting from the 0-lag time just mentioned. More formally, the additional runtime exploitable till the deadline $d_i(t)$ in the admission test from Equation (9), due to τ_i having left, is:

$$U_i(d_i(t) - \delta_i) = U_i\left(d_i(t) - \left(d_i(t) - \frac{q_i(t)}{U_i}\right)\right) = U_i\left(\frac{q_i(t)}{U_i}\right) = q_i(t)$$

which is exactly what is needed to admit the task back.

On the other hand, if the task comes back at a later time $t' > \delta_i$, then the proposed test may generally require a higher bandwidth to be made available for the first instance of the task (i.e., if it did not run while away from core h). This includes the case in which the task may not be admitted back.

The task can also be handled by remembering that it left core h earlier and modelling the time executed on other cores as a suspension of the task on core h. In this case, the regular CBS wake-up rule (checking if $q_i(t) < (d_i(t) - t)\frac{Q_i}{P_i}$) is applied to decide whether keeping the current deadline and budget is safe or not. Notice that this is consistent with the approach presented above since the CBS wake-up rule can be re-formulated as "if the task wakes up before the 0-lag time δ_i, then the current budget and scheduling deadline can be re-used, otherwise a new scheduling deadline must be generated". This option might still need to be further investigated and is not discussed further here for the sake of brevity.

6 SCHEDULABILITY ANALYSIS

As previously discussed, when a new task τ_{n+1} arrives on core h at time t (because it is created or migrated from a different core) the new task must pass an admission test. This admission test is based on the core's total utilization $V_h(t)$ (the utilization of currently hosted tasks) and active migrated utilization $V_h^m(t')$ (for $t' > t$ in the future), which is the utilization of k tasks that previously left the core (due to migration to other CPUs or termination) and have the 0-lag time in the future.

The evolution over time of $V_h^m(t')$ for $t' > t$ can be modelled through a sequence of pairs $(\delta_1, U_1), \ldots, (\delta_k, U_k)$ indicating that U_j contributes to $V_h^m(t')$ until time δ_j (U_j is the utilization of the j^{th} task $\tau_j \in \Gamma_h^m(t) : \delta_j \leq t + P_i$ that recently left the CPU and will expire at time δ_j — its 0-lag time).

If the decrements in $V_h^m()$ happening at times δ_j are not considered, then τ_{n+1} can be accepted onto core h if its CBS has scheduling deadline $d_i(t) = t + P_i$ and maximum budget smaller than the one computed by Equation (7). However, this admission test can be pessimistic.

In the following, we prove the correctness of the more advanced test considering the $V_h^m()$ evolution presented in Equation (9).

To prove the core result, the following lemma will be useful:

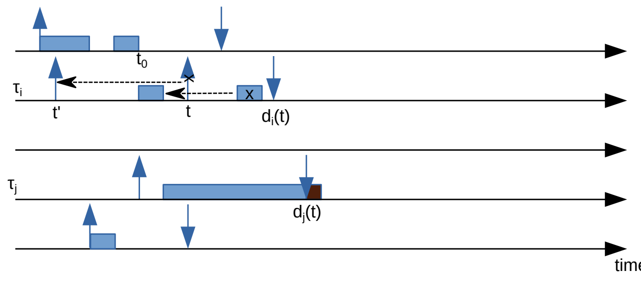


Figure 4: Visualization useful for a better understanding of the proof of Lemma 6.1.

Lemma 6.1. *If a taskset $\Gamma = \{\tau_1, \ldots \tau_n\}$ is schedulable on a core h when task $\tau_i \in \Gamma$ is activated at time t with deadline $d_i(t)$ and budget $q(t)$, the activation time can be moved to $t' < t$ (leaving $d_i(t)$ unchanged) without breaking the taskset's schedulability.*

Proof. Let us denote with $\sigma()$ the schedule, indicating with $\sigma(t)$ which task is scheduled at time t. Now, if we move the release time of this task τ_i from time t back to time $t' < t$, (see Figure 4) but we leave its absolute deadline at time $d_i(t)$, then we obtain a different schedule $\sigma'()$.

Since EDF is used and $d_i(t)$ is not changed, τ_i's priority is the same in $\sigma()$ and $\sigma'()$. Hence, if τ_i does not miss its deadline in $\sigma()$, then it does not miss the deadline in $\sigma'()$ too. So, we just need to prove that anticipating the activation time from t to t' does not break the schedulability of the other tasks in Γ.

According to EDF, τ_i cannot be scheduled before other tasks having earlier deadlines. Assuming that all the tasks with earlier deadlines become idle at some time t_0, we need to consider two cases. If $t_0 \geq t$, then τ_i is scheduled after time t also in $\sigma'()$; hence, releasing τ_i at time $t' < t$ instead of time t does not change the schedule ($\forall t, \sigma'(t) = \sigma(t)$). If $t_0 < t$, then τ_i can execute earlier in $\sigma'()$ than in $\sigma()$, risking to affect the schedulability of tasks with deadlines later than $d_i(t)$. As a result, if there are missed deadlines they cannot occur at a time earlier than $d_i(t)$.

Now we prove that no tasks can have deadline misses in $\sigma'()$ by contradiction. Let τ_j be the first task to miss a deadline, at time $d_j(t) > d_i(t)$. In the original schedule $\sigma()$ without the anticipated execution of τ_i, τ_j would have executed all of its budget $q(t'')$ (where t'' is the activation of the τ_j's instance missing a deadline) before its deadline $d_j(t)$, but let us assume that with the anticipated release of τ_i this does not happen. Such a deadline miss is caused by the fact that in $\sigma'()$ τ_i executes before time t instead of τ_j or instead of different tasks that have deadline earlier than $d_j(t)$. However, the fact that τ_j is not allowed to execute for more than $q_i(t)$ time units before $d_i(t)$ implies that if in $\sigma'()$ it executes for an amount of time $q0$ before time t, then in the time interval $(t, d_i(t))$ it will execute for $q0$ time units less respect to $\sigma()$. Hence, the tasks that have been preempted for a time $q0$ before time t in $\sigma'()$ will have $q0$ time units for executing in $(t, d_i(t))$ (which is before their deadlines).

Hence, we can infer that if τ_j misses its deadline $d_j(t)$ in $\sigma'()$ then it must do the same also in $\sigma()$ which is a contradiction. □

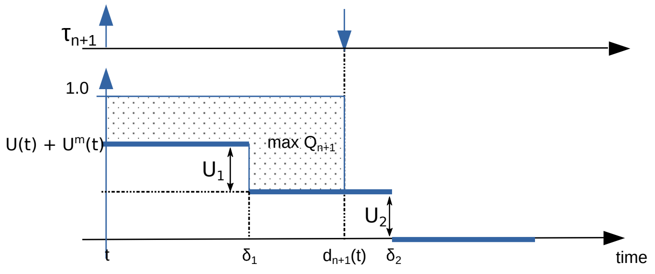


Figure 5: Visualization of the maximum admissible budget Q_{n+1} for the task τ_{n+1} under admission.

Theorem 1. *A new task τ_{n+1} served by a CBS (Q_{n+1}, P_{n+1}) can be admitted on core h at time t if the following condition is respected:*

$$Q_{n+1} \leq \left(U^{lub} - V_h(t) - \sum_{\tau_j \in \Gamma_h^m(t)} U_j \right) P_{n+1} + \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j \leq t+P_i}} U_j \left(t + P_{n+1} - \delta_i \right).$$

Proof. According to utilization-based analysis (see Equation (7)), the schedulability of the taskset is not broken by accepting a new task τ_{n+1} served by a CBS $S(Q, P_{n+1})$ with period equal to P_{n+1}, (thus its first deadline equal to $d_1(t) = t + P_{n+1}$), and budget $Q \leq (U^{lub} - V_h(t) - \sum_{\tau_j \in \Gamma_h^m(t)} U_j) P_{n+1}$. However, a larger budget can also be admitted, because at time δ_1 the utilization U_i of a task that left will be removed from $V_h^m()$. At said time, it would be possible to accept a new CBS $S' = (Q', t + P_{n+1} - \delta_1)$ with period equal to $t + P_{n+1} - \delta_1$, first deadline $\delta_1 + t + P_{n+1} - \delta_1 = t + P_{n+1} = d_i(t)$, and maximum budget Q' equal to:

$$Q' = U_1 \left(t + P_{n+1} - \delta_1 \right).$$

Remember from Lemma 6.1 that the arrival time of the new server S' can be moved backwards from δ_1 to the current time t without impacting on the schedulability. Therefore, S' is released synchronously with the server S serving τ_{n+1}, and it has the same deadline, then its budget Q' can be added to the base budget Q of S.

The theorem statement is obtained by reiterating the reasoning made above for all the other 0-lag times $\delta_2 ... \delta_k$ between time t and $t + P_i$. The result is that we can admit, synchronously with the server S serving τ_{n+1}, several other servers with the same deadline. Therefore, their budgets can add up to the base budget of S given by Equation (7), obtaining the theorem claim. □

Corollary 6.2. *A new task τ_{n+1} served by a CBS (Q_{n+1}, P_{n+1}) can be admitted on core h at time t if the condition expressed by Equation (9) is respected.*

Proof. The condition presented in Theorem 1 is equivalent to Equation (9) and can be obtained from it as follows:

$$P_i \left(U^{lub} - V_h(t) \right) - \sum_{\tau_j \in \Gamma_h^m(t)} \min\{\delta_j - t, P_i\} U_j =$$
$$P_i \left(U^{lub} - V_h(t) - \sum_{\tau_j \in \Gamma_h^m(t)} U_j \right) +$$
$$P_i \sum_{\tau_j \in \Gamma_h^m(t)} U_j - \sum_{\tau_j \in \Gamma_h^m(t)} \min\{\delta_j - t, P_i\} U_j$$

The first part of this expression is the same as in the theorem, and the second part can be modified as follows:

$$P_i \sum_{\tau_j \in \Gamma_h^m(t)} U_j - \sum_{\tau_j \in \Gamma_h^m(t)} \min\{\delta_j - t, P_i\} U_j =$$
$$\sum_{\tau_j \in \Gamma_h^m(t)} P_i U_j - \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j < t+P_i}} \min\{\delta_j - t, P_i\} U_j - \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j \geq t+P_i}} \min\{\delta_j - t, P_i\} U_j =$$
$$\sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j < t+P_i}} P_i U_j + \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j \geq t+P_i}} P_i U_j - \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j < t+P_i}} (\delta_j - t) U_j - \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j \geq t+P_i}} P_i U_j =$$
$$\sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j < t+P_i}} P_i U_j - \sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j < t+P_i}} (\delta_j - t) U_j =$$
$$\sum_{\substack{\tau_j \in \Gamma_h^m(t): \\ \delta_j < t+P_i}} (P_i + t - \delta_j) U_j$$

which gives the condition presented in the theorem. □

Notice that the condition of Equation (9) has an intuitive interpretation represented in Figure 5.

7 EXPERIMENTAL EVALUATION

In this section, an experimental validation of the correctness and evaluation of the performance of the admission test introduced in Section 5 is performed by simulation. To this purpose, the RT-Sim [20] simulator is used. RTSim is an open-source tool with the ability to simulate the timing of simple real-time tasks. It supports a number of different real-time scheduling policies and a few resource-sharing protocols, and platforms with multiple processors, including the recent addition of asymmetric multi-cores [18].

First, the consistency of the simulated model with the the admission test proposed in this paper has been validated by checking that when it is used no deadline is missed in the simulations. A large number of dynamic tasksets (see below for the details) has been generated and simulated through RTSim (accepting new tasks only if the new admission test allowed them), verifying that the generated schedules do not contain any deadline miss. Then, the improvements in schedulability introduced by the presented algorithm have been evaluated.

The overall set of results presented in this section can be exactly reproduced using the software available at:
http://retis.sssup.it/~tommaso/papers/rtns21.php.

7.1 Experimental setup

All the experiments are based on random scenarios where tasks dynamically leave a CPU (it is unimportant whether due to termination or migration to a different CPU), and other tasks need to dynamically enter the CPU. The technique introduced in this paper is expected to allow admitting more tasks than without it.

The simulation set-up is as follows: we generate randomly tasksets with a random number n of tasks between 4 and 10, and a pre-configured total utilization of U_{tot}, using the well-known `taskgen.py` from [11]. The generated tasks have periods with loguniform distribution between 1000 and 2000 time units, with an enforced period granularity of 100 units. The tasks are scheduled using EDF on a single processor. The simulation is paused at random times, until we find a time in which there are at least a prefixed number k of tasks with 0-lag time in the future. Then, k of said tasks are killed, and a new task is instantiated with period uniformly distributed within the earliest and twice the latest future 0-lag times of the killed tasks, and runtime computed according to the maximum allowed values as from Section 5. The simulation is then continued, verifying that no tasks missed their deadlines after the instantiation of the new task. This is done for a time equal to 10 times the biggest period among the tasks. Indeed, once every task has started a new instance, the "transient" due to the tasks having been killed and the new one having been started is over, and the taskset remains schedulable due to the classic single-processor test from Liu & Layland [16].

The above scenario has been repeated 1000 times for each of the configurations of the parameters U_{tot} and k, as shown in Table 1.

7.2 Validation of the model

Table 1: Obtained maximum relative response times and average bandwidth gain for the simulated scenarios.

U_{tot}	k	maximum resptime/T	average bw gain
0.90	1	0.9000	2.03741
0.90	2	0.8973	2.99117
0.90	3	0.8973	4.21386
0.95	1	0.9500	3.23395
0.95	2	0.9467	5.18756
0.95	3	0.9467	7.77282
0.99	1	0.9900	12.8519
0.99	2	0.9867	22.8740
0.99	3	0.9867	35.3014

The table highlights in the third column the highest obtained ratio between the response-time (difference between finishing time and arrival time) and the period of tasks in all 1000 tasksets used for each configuration. The reported numbers are all strictly lower than 1.0, highlighting that no task underwent any deadline miss during the simulated scenarios.

For completeness, we also report in Figure 6 the cumulative distribution function (CDF) obtained for the response-times relative to the periods of all the instances of the tasks from all the runs for each pair of parameters U_{tot} and k.

7.3 Performance evaluation

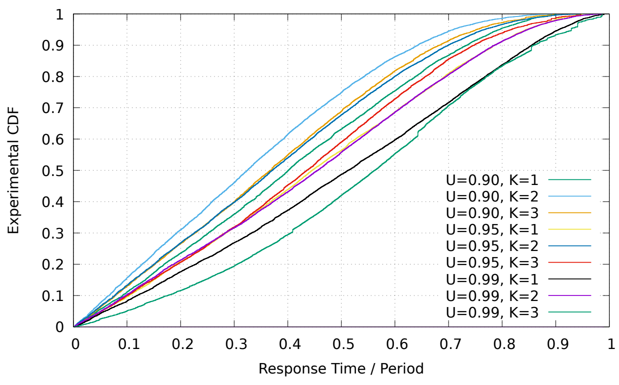


Figure 6: Experimental CDF of the response times relative to the periods in various configurations.

To evaluate the performance of the proposed admission test and highlight its advantages, when instantiating the new task in each simulation we compare the bandwidth U^{new} admitted for the new task (equal to the maximum allowed according to Section 5), with the maximum $U^{old} = U^{lub} - (V(t) + V^m(t)) \equiv 1.0 - U_{tot}$ that would have been allowed considering only the available active utilization at that time. We define the *bandwidth gain* as: $\frac{U^{new} - U^{old}}{U^{old}}$. The fourth column in Table 1 shows the average bandwidth gain obtained among the 1000 scenarios simulated for each configuration of the parameters U_{tot} and k. Consider that, by construction of the simulated scenarios, the residual available bandwidth at the time of arrival of the new task for each configuration is given by $1.0 - U_{tot}$, because the k tasks that have just been killed still have their active utilization counted in, till their future 0-lag times. This value is equal to 0.10, 0.05 and 0.01 for the first, second and third triplet of 3 rows in the table.

As evident from the reported numbers, the average bandwidth gain throughout the simulations goes from a minimum of 2.037 with $U_{tot} = 0.90$ and only $k = 1$ task that left with expiring utilization falling in the task period, to a maximum of 35.30 with $U_{tot} = 0.99$ and 3 tasks that recently left, with expiring utilization falling in the new task period. However, consider that the new task to be admitted has been picked with some randomness in its parameters, but nonetheless its period has been chosen at random in a window useful to show the benefits of the proposed new admission test.

7.4 Relevance of the test

The technique introduced in this paper deals specifically with the problem of admitting a new task to a core during a transitory time window in which a few tasks have just been migrated to other cores (or have terminated), in a moment in which their 0-lag time was in the future. In order to gain an idea of how likely such a situation might occur, we performed an additional experiment where we considered task sets generated similarly to the previous section, but with overall utilization ranging from 10% to 90% in step increments of 10%, and number of tasks being 4, 8 or 12. Each simulation run has been paused at random times for ten thousand times, reporting how many tasks have been found with 0-lag time in the future during the pause.

The obtained results are shown in Figure 7, where the experimental probability of finding a certain number of tasks with 0-lag time in the future are reported, for the various simulated task sets. As visible, depending on the conditions (number of tasks and overall utilization), we can easily see a significant probability of finding a non-null number of tasks with 0-lag time in the future. A possible migration of any of those tasks at some time t would require the technique presented in this paper to perform the admission of some other tasks on the CPU shortly after time t.

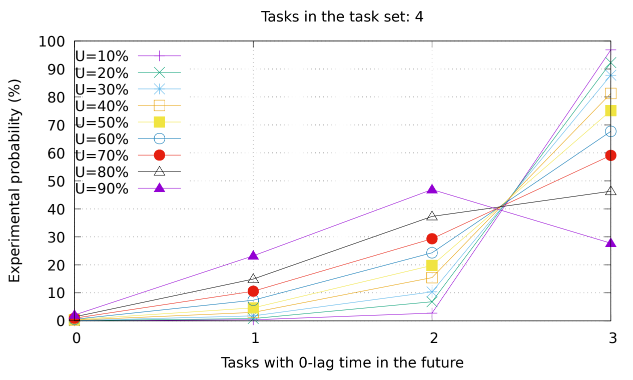
8 CONCLUSIONS AND FUTURE WORK

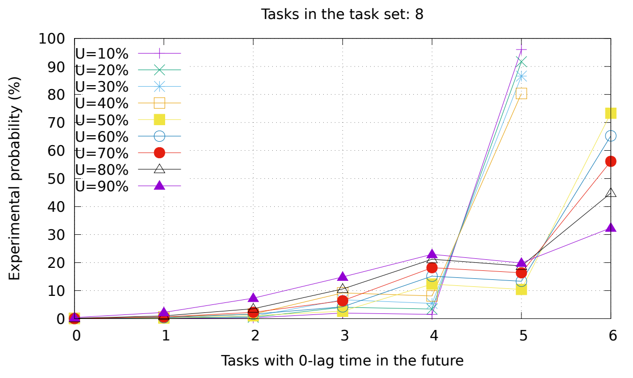
In this paper, a novel admission test for dynamically partitioned CBS reservations on multi-core systems has been proposed. The test keeps fundamentally the simplicity of utilization-based tests, but it leverages knowledge of the future evolution of the active utilization within each CPU due to tasks that recently left, due to either termination or migration, which is tracked by any correct scheduler implementation. This knowledge is combined with the one about the period of the new task to admit, to obtain the maximum admissible budget (or task WCET) that can be admitted. In highly dynamic scenarios where tasks can be migrated among CPU cores, the proposed technique is able to cope correctly with tasks executing partially on a CPU and partially on another CPU, with a contained amount of pessimism.

We provided formal proof of correctness of the proposed new test, as well as an experimental validation by simulation of the technique, recurring to the open-source RTSim real-time systems simulator. The performed simulations highlight that the new admission test introduced in this paper for dynamic partitioned CBS reservations may lead to a great increase of the chances to admit tasks on a CPU (and to migrate tasks across CPUs), keeping correctness of the schedule as no admission is done at the expense of schedulability of the already accepted and running tasks. The introduced test is sufficiently simple to constitute an excellent candidate to be incorporated in the kernel-level admission logic of an operating system, and particularly in a dynamic partitioning modification to `SCHED_DEADLINE`.

Indeed, as to possible future work on the topic, we plan to combine this test with adaptive partitioning as proposed in [2], and to prototype the effectiveness of the resulting scheduler on a real platform, modifying `SCHED_DEALINE` within the Linux kernel. Also, the technique is planned to play a key role in energy-aware multi-core scheduling policies that may need adaptive placement and replacements of tasks among cores for achieving the maximum possible energy saving on asymmetric multi-core platforms, as investigated by simulation in [18].

The implementation of the technique as a minimally-invasive set of modifications to `SCHED_DEADLINE` is planned to be made publicly available as a patch to the kernel, and submitted to the Linux Kernel Mailing List (LKML) for consideration and discussion for possible mainline inclusion. The implementation will be applied to a number of high-performance embedded use-case scenarios, including those considered by the AMPERE EU project [23].





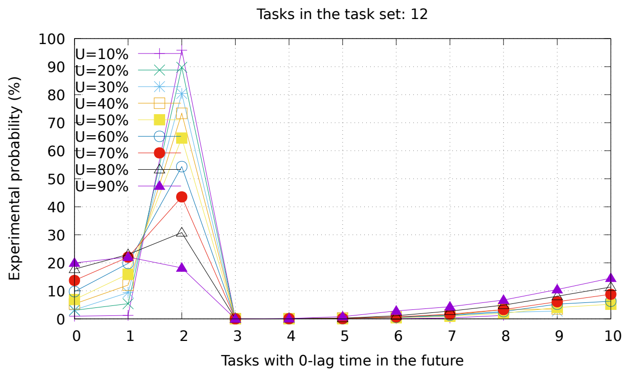


Figure 7: Experimental probability (on the Y axis) of finding a given number of tasks (on the X axis) with 0-lag time in the future, when pausing the simulation at random times, in the cases of 4, 8 and 12 tasks (top, middle and bottom plots, respectively) and total utilization ranging from 10% to 90% (various curves in each plot).

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".

REFERENCES

[1] L. Abeni and G. Buttazzo. 1998. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symp.* Madrid, Spain.

[2] Luca Abeni and Tommaso Cucinotta. 2020. Adaptive Partitioning of Real-Time Tasks on Multiple Processors. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 572–579. https://doi.org/10.1145/3341105.3373937

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

[4] S. Baruah, J. Goossens, and G. Lipari. 2002. Implementing constant-bandwidth servers upon multiprocessor platforms. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. 154–163. https://doi.org/10.1109/RTTAS.2002.1137390

[5] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. 2002. Elastic scheduling for flexible workload management. *IEEE Trans. Comput.* 51, 3 (2002), 289–302. https://doi.org/10.1109/12.990127

[6] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. 2017. Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 76)*, Marko Bertogna (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:23. https://doi.org/10.4230/LIPIcs.ECRTS.2017.13

[7] D. Casini, A. Biondi, and G. Buttazzo. 2019. Handling Transients of Dynamic Real-Time Workload Under EDF Scheduling. *IEEE Trans. Comput.* 68, 6 (June 2019), 820–835. https://doi.org/10.1109/TC.2018.2882451

[8] D. Casini, A. Biondi, and G. C. Buttazzo. 2020. Task Splitting and Load Balancing of Dynamic Real-Time Workloads for Semi-Partitioned EDF. *IEEE Trans. Comput.* (2020), 1–1. https://doi.org/10.1109/TC.2020.3038286

[9] T. Chen and L. T. X. Phan. 2018. SafeMC: A System for the Design and Evaluation of Mode-Change Protocols. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 105–116. https://doi.org/10.1109/RTAS.2018.00021

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

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

[12] R. L. Graham. 1971. Bounds on Multiprocessing Anomalies and Related Packing Algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '72 (Spring))*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/1478873.1478901

[13] J. Lee and K. G. Shin. 2013. Schedulability Analysis for a Mode Transition in Real-Time Multi-core Systems. In *2013 IEEE 34th Real-Time Systems Symposium*. 11–20. https://doi.org/10.1109/RTSS.2013.10

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

[15] G. Lipari and S. Baruah. 2000. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*. 193–200. https://doi.org/10.1109/EMRTS.2000.854007

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

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

[18] Agostino Mascitti, Tommaso Cucinotta, and Mauro Marinoni. 2020. An Adaptive, Utilization-Based Approach to Schedule Real-Time Tasks for ARM Big.LITTLE Architectures. *SIGBED Rev.* 17, 1 (July 2020), 18–23. https://doi.org/10.1145/3412821.3412824

[19] V. Nelis, B. Andersson, J. Marinho, and S. M. Petters. 2011. Global-EDF Scheduling of Multimode Real-Time Systems Considering Mode Independent Tasks. In *2011 23rd Euromicro Conference on Real-Time Systems*. 205–214. https://doi.org/10.1109/ECRTS.2011.27

[20] Luigi Palopoli, Giuseppe Lipari, Gerardo Lamastra, Luca Abeni, Gabriele Bolognini, and Paolo Ancilotti. 2002. An object-oriented tool for simulating distributed real-time control systems. *Software: Practice and Experience* 32, 9 (2002), 907–932. https://doi.org/10.1002/spe.467 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.467

[21] P. Pedro and A. Burns. 1998. Schedulability analysis for mode changes in flexible real-time systems. In *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168)*. 172–179. https://doi.org/10.1109/EMWRTS.1998.685082

[22] L. T. X. Phan, I. Lee, and O. Sokolsky. 2011. A Semantic Framework for Mode Change Protocols. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 91–100. https://doi.org/10.1109/RTAS.2011.17

[23] 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. 2020. 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)*. IEEE, Nashville, Tennessee (turned to a virtual event).

[24] Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Krithi Ramamritham. 1989. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems* 1, 3 (1989), 243–264.