The Journal of Systems & Software 173 (2021) 110886

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Dynamic partitioned scheduling of real-time tasks on ARM big.LITTLE architectures☆,☆☆

Agostino Mascitti *, Tommaso Cucinotta, Mauro Marinoni, Luca Abeni

Scuola Superiore Sant'Anna, Via Moruzzi, 1, 56124, Pisa, Italy

ARTICLE INFO

Article history:
Received 13 July 2020
Received in revised form 25 October 2020
Accepted 9 December 2020
Available online 11 December 2020

Keywords:
Real-time scheduling
ARM big.LITTLE
Heterogeneous multicore processing
Energy-efficiency

ABSTRACT

This paper presents Big-LITTLE Constant Bandwidth Server (BL-CBS), a dynamic partitioning approach to schedule real-time task sets in an energy-efficient way on multi-core platforms based on the ARM big.LITTLE architecture. BL-CBS is designed as an on-line and adaptive scheduler, based on a push/pull architecture that is suitable to be incorporated in the current SCHED_DEADLINE code base in the Linux kernel. It employs a greedy heuristic to dynamically partition the real-time tasks among the big and LITTLE cores aiming to minimize the energy consumption and the migrations imposed on the running tasks. The new approach is validated through the open-source RT-Sim simulator, which has been extended integrating an energy model of the ODROID-XU3 board, fitting tightly the power consumption profiles for the big and LITTLE cores of the board. An extensive set of simulations have been run with randomly generated real-time task sets, leading to promising results.

1. Introduction

In recent years, embedded systems faced a relentless growth in the requirements posed by users on their computing capabilities, with a pervasive switch to multi-core architectures. Specifically, in the area of mobile and battery-operated embedded systems, we have witnessed a widespread adoption of novel energy-efficient architectures, first and foremost the ARM big.LITTLE one, nowadays a fundamental component of a plethora of smartphone and tablet devices on the market, where the timeliness of soft real-time applications in domains like multimedia and gaming is becoming more and more important.

The big.LITTLE design deviates from classical symmetric multiprocessing (SMP), in that it introduces two different types of cores sharing the same instruction-set architecture (ISA), so that tasks can be seamlessly migrated among them, as in SMP, but with different frequency vs power consumption curves: *LITTLE* cores specialize in low-energy computing whilst *big* cores specialize in performance. The two types of cores are normally capable of switching among principally different but partially overlapped frequency steps. However, the differences in the internal micro-architecture and pipeline design for the two core types causes a task running on a LITTLE core to take longer for execution and to consume less power than when running on a big core at the same frequency. Nowadays, the Dynamic Voltage and Frequency Scaling (DVFS) capabilities of big.LITTLE architectures are constrained to being able to set a single frequency for each of the two core type islands, albeit the most recent and advanced developments of the architecture, named DynamIQ (ARM, 2019), will remove this constraint.

These hardware characteristics make it quite challenging to design a scheduler within an operating system (OS) that manages to perform an optimum use of the available features. This problem has been tackled in the Linux kernel community introducing the Energy-Aware Scheduling (EAS) framework (Perret, 2018), a feature that introduces an energy model within the kernel, gaining awareness of what are the power consumption and processing capacity figures associated to each frequency on big vs LITTLE cores. This information, made available through the kernel device tree,[1] has been leveraged for the default completely fair scheduling (CFS) class.

However, when dealing with real-time workloads, the heuristics provided by the CFS are not effective in handling properly the available hardware features, causing either applications

☆ This work has received funding from the European Commission through the EU H2020 research project AMPERE (A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization) under the grant agreement no. 871669.

☆☆ Editor: Alexander Chatzigeorgiou.

* Corresponding author.
E-mail addresses: agostino.mascitti@santannapisa.it (A. Mascitti), tommaso.cucinotta@santannapisa.it (T. Cucinotta), mauro.marinoni@santannapisa.it (M. Marinoni), luca.abeni@santannapisa.it (L. Abeni).

[1] More information at: https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt.

https://doi.org/10.1016/j.jss.2020.110886
0164-1212/

to experience unnecessary deadline misses, or forcing the platform into excessively high power consumption. More appropriate solutions are needed, designed in the realm of real-time systems (Balsini et al., 2019, 2016). An interesting feature of the latter kind, made recently available in the Linux kernel, is the SCHED_DEADLINE scheduler, a multi-processor variant of the well-known constant bandwidth server (CBS) (Abeni and Buttazzo, 1998), employing a reservation-based scheduling strategy that can be conveniently configured as using either global or partitioned or clustered EDF scheduling underneath. However, the interesting energy-awareness features of the EAS framework cannot be exploited by SCHED_DEADLINE at the moment.

1.1. Paper contributions

In this paper, we propose a novel energy-aware scheduling strategy for soft real-time tasks running on big.LITTLE architectures, in the context of a complex embedded OS like Linux, where applications cannot be known beforehand, and they can be dynamically started and terminated. The proposed approach combines partitioned EDF scheduling (CBS, actually) with an on-line partitioning heuristics that is activated only at task wake-up (or creation) and suspension (or termination) times, performing a single task placement (or migration) action that: (1) ensures schedulability of the real-time tasks that have been admitted onto the system; (2) achieves the lowest expected power consumption for the execution of the real-time tasks; (3) avoids unnecessary migrations that might degrade the tasks performance.

This work extends a preliminary prior work of ours (Mascitti et al., 2020) along several directions: we clarify and refine several aspects of the proposed mechanism, identify the theoretical conditions under which schedulability is guaranteed, discuss key implementation details related to the performance of the mechanism, and present a much more comprehensive evaluation of the technique under various workload conditions. The performed simulations show that our approach is actually promising, allowing 15% of energy saving in average with respect to the current state of the SCHED_DEADLINE code base.

1.2. Paper organization

This paper is organized as follows: after a brief review of the related research in Section 2, key background concepts related to the adopted real-time task model and CBS scheduling are presented in Section 3, then the computing platform we focus on and its energy model are described in Section 4, along with some accompanying notation that is adopted throughout the rest of the paper. The proposed BL-CBS technique is described in Section 5, along with the main factors driving its design. Section 6 introduces theoretical conditions on the real-time task sets that are schedulable under BL-CBS, i.e., they are guaranteed not to miss any deadlines. Then, after providing a few important details in Section 7 related to an efficient implementation within an in-kernel scheduler, the proposed BL-CBS technique is evaluated by simulation in Section 8. Finally, conclusions are drawn in Section 9, sketching out possible directions for future research on the topic.

2. Related work

Energy-efficient scheduling for real-time tasks has been widely investigated in the research literature, starting from some seminal works on uni-processor systems. In particular, real-time schedulability analysis has been combined with DVFS techniques to reduce the CPU frequency as much as possible without breaking guarantees. For example, the RT-DVS algorithms proposed by Pillai and Shin (2001) used an approach exploiting the task unused computation time (with respect to its worst case) to decrease the working frequency dynamically. Aydin et al. (2004) used similar static scaling algorithms to minimize the speed while guaranteeing tasks deadlines, coupling them with dynamic reclaiming of unused computation time both intratask and intertask. While these authors focused on solutions based on dynamic priorities and the Earliest Deadline First (EDF) scheduling policy, Saewong and Rajkumar (2003) presented similar algorithms but focusing on fixed priorities. More dynamic approaches have been proposed, for example, by Zhu and Mueller (2004, 2007), who used a feedback mechanism to maximize energy saving for the average execution time, still guaranteeing the timing constraints in case of worst-case execution times.

When considering multi-core CPUs, minimizing the consumed energy is not as simple as selecting the minimum frequency and different strategies can be used. See Bambagini et al. (2016) for an overview of the solutions presented in the literature (in particular Section 7 considers uniform heterogeneous multi-processors or multi-cores, as defined in Funk, 2004). For example, the placement of tasks on the various cores (or the migration of tasks between cores) has a significant impact on the operating frequencies, and on some platforms (such as ARM big.LITTLE) different cores can be characterized by different power characteristics (and different sets of possible operating frequencies). Hence, frequency scaling, task placement, and migration actions must be strictly coordinated.

Multi-core real-time schedulers are generally classified as *global* schedulers or *partitioned* schedulers: while a global scheduler is free to migrate tasks between different cores according to the scheduling policy (and hence, conceptually, the scheduler uses one single global ready queue that contains all the tasks ready for execution), a partitioned scheduler does not migrate tasks between cores (and tasks are statically assigned to cores by the system designer). As a consequence, when using partitioned scheduling the problem of scheduling tasks on m CPUs is reduced to m scheduling problems on a single CPU, and single-processor DVFS algorithms can be re-used. Hence, the main challenge in partitioned scheduling is the tasks assignment so that the DVFS algorithm can more effectively decrease the consumed energy.

Semi-partitioned scheduling (Andersson and Tovar, 2006; Burns et al., 2012; Casini et al., 2017) represents a trade-off between global and partitioned scheduling, allowing to schedule tasksets that are not partitionable. It relies on splitting a real-time task into two parts with reduced demand that fit into two different cores and execute with a precedence constraint (one after the other), keeping schedulability. These techniques have also been applied to heterogeneous multi-cores, and used to reduce energy consumption (Liu et al., 2016). However, this power saving technique relies on an off-line placement of the tasks (or parts of the tasks) on the various cores. Some authors (Casini et al., 2017) investigated the use of linear-time approximation methods to perform the splittings so that they can be performed on-line, but this technique does not take into account energy consumption.

The present work advocates a simpler utilization-based method where task splitting is not used and a "restricted migrations" approach (Baruah and Carpenter, 2003) is used: a task resides mostly on one core for each job, and it is migrated normally only across subsequent activations.[2] This is easier to compute in an OS kernel (see Section 5 for details). On the other hand, task splitting requires to migrate tasks while running, when one split of the task exhausted its assigned time on one CPU, and the subsequent split continues execution on a different CPU.

[2] As it will become clear in Section 5, a task can also be migrated in the middle of a job to balance workload and bring down the island frequency.

Notice that frequency scaling is not the only technique that can be used to reduce energy consumption. For example, Dynamic Power Management (DPM) allows for reducing the energy consumption by putting the CPU in a "low-power" (or sleep) state whenever it is idle. DPM is not an alternative to DVFS but can be combined with it. For example, these techniques are jointly investigated in the context of real-time scheduling in Bambagini et al. (2013) and Moulik et al. (2019). Other works (Imes and Hoffmann, 2015) investigated the trade-offs between DVFS and DPM, studying whether it is better to execute at the highest possible frequency and then set cores to a sleep state as long as possible (and so relying on DPM only), or finding the lowest frequency needed to make tasks respect their deadlines and never go idle (and so relying on DVFS only).

Several authors reduced the complexity of the scheduling problem by reducing the number of decisions taken at run-time (for example, by statically assigning tasks to cores or core islands, or statically deciding the frequency at which each core or core island executes, by resorting to a static schedule computed off-line). This allows for taking some decisions by solving an optimization problem (Chwa et al., 2015; Thammawichai and Kerrigan, 2018; Qin et al., 2019a,b). Polynomial-time algorithms have been proposed (Liu et al., 2015) to divide real-time streaming applications across DVFS capable islands, where tasks are statically assigned to an island and then globally scheduled inside it via an optimal scheduler. In Colin et al. (2014), it is found that the most efficient way to allocate real-time tasks and save energy is neither balancing the load nor choosing the most power-efficient core. They find off-line the optimal load distribution via integer linear programming (ILP) and try to approximate on-line that result via heuristics. Other authors (Thammawichai and Kerrigan, 2018) divide the scheduling problem into workload partitioning and next task ordering. The first step determines what parts of the tasks should be executed at what frequency within a time interval such that feasibility constraints are satisfied, while the second part establishes how to order the pieces of tasks for each core. Also, an analysis of the task code structure coupled with an ILP returning the minimum frequency and location to be used for each code segment has been proposed (Qin et al., 2019a). In the same work, the authors tried to moderate the use of the *LITTLE-Core-First* principle, according to which one should always fill LITTLE cores while possible before selecting a *big* one. Using the task execution variance (i.e., the ratio between the WCET on LITTLE and on big for a given task) an ILP formulation is used to compute the optimal distribution of the utilization of the tasks between the islands and their minimum frequencies to respect the deadline. A heuristic is then used to assign tasks and set the frequencies. This approach, however, is related to ARM DynamIQ and it makes use of per-core DVFS, which is not feasible for generic big.LITTLE platforms. Optimization methods have also been used in Nogues et al. (2016) for choosing the best frequency for each node of stream processing applications represented as Synchronous Data Flows with end-to-end deadlines, making their parallelism explicit in the model.

While most of the above works focus on optimizing some decisions taken off-line, in this paper, we deal with designing a strategy that can be applied on-line within an OS kernel scheduler. Hence, our algorithm copes with both on-line scheduling of tasks (considering both task migrations and possible task overruns, which we handle by using CBS servers) and dynamic frequency scaling.

Focusing on recent developments in the mainline Linux kernel, notable energy-aware features for big.LITTLE have been integrated within the EAS framework, that is mostly focused on the CFS scheduler for general-purpose workloads. On the other hand, for real-time tasks, the SCHED_DEADLINE policy has been recently enriched with an on-line support for DVFS (Scordino et al., 2018, 2019), by integrating a variant of the GRUB-PA algorithm (Scordino and Lipari, 2004). However, the current implementation does not support non-symmetric multi-cores such as ARM big.LITTLE platforms. Also, other investigations of ours on the use of adaptive partitioning schedulers for multi-cores (Abeni and Cucinotta, 2020), albeit in a non energy-aware context, highlighted that these techniques can be effective in scheduling real-time task sets reducing deadline violations compared to the global EDF used by SCHED_DEADLINE. Following this research track, this paper focuses on an on-line scheduling algorithm that dynamically partitions real-time tasks, exploiting job-level migration. It is suitable to be implemented into the existing code-base of the Linux kernel in the scheduling class SCHED_DEADLINE and works with incremental, dynamic task sets, which are not known a priori. Off-line approaches cannot be used in this context since they are not suitable for being used with dynamic workloads. However, off-line partitioning algorithms commonly give better results since all the optimizations can be performed on a task set known a priori, leading to better solutions in terms of real-time guarantees and energy-saving. Conversely, an on-line algorithm, like the one proposed in this paper, is expected to make swift decisions, aiming at achieving a good-enough performance in a reduced computation time.

Some authors deal with the problem of scheduling real-time DAG tasks (Guo et al., 2017, 2019; Li et al., 2019) and digraphs (Zahaf et al., 2019) on ARM big.LITTLE, evaluating the approach either through implementation on real hardware or by simulation. Our work is currently limited to scheduling of real-time independent task sets, albeit extensions along said lines will be considered in future extensions.

Finally, an interesting approach is the one proposed in Balsini et al. (2016), where authors highlight that power consumption may vary in non-negligible way depending also on the workload type being computed, supporting the argument with real data measured on an ODROID-XU3 platform. Possible integration of such a per-application power-consumption model might be an interesting area of future extensions of the present work.

In this paper, the validation is carried out using RTSim (Palopoli et al., 2002, 2001; Scordino and Lipari, 2006), an open-source tool we have been evolving over time and used in several previous research works. Albeit other real-time task simulators (Pillai and Isha, 2013; Thakare and Deshmukh, 2017; Cheramy et al., 2014) were available, RTSim was an easier choice for us, also because its modifications in Balsini et al. (2016), integrating a realistic energy consumption model for the ODROID-XU3 big.LITTLE platform, have been used as a starting point to develop the energy-aware adaptive partitioning technique presented in this paper.

3. Background

In this paper we consider a set of real-time tasks $\{\tau_i\}$ to be scheduled on a number of CPUs. A real-time task τ_i is characterized by a minimum inter-arrival period T_i equal to its relative deadline (*implicit deadline* case) and by the worst-case execution time (WCET), which will be discussed in depth in the following sections for the case of the ARM big.LITTLE architecture. τ_i generates a sequence of jobs $J_{i,j}$ and for a job arriving at time $r_{i,j}$, its finishing time is denoted by $f_{i,j} > r_{i,j}$. Task τ_i respects all of its deadlines if $\forall j, f_{i,j} \leq r_{i,j} + T_i$. Generally, $r_{i,j+1} \geq r_{i,j} + T_i$, but for a *periodic* real-time task, we have $r_{i,j+1} = r_{i,j} + T_i$.

In this paper we consider *soft real-time* tasks, meaning that a job missing its deadline will not cause severe consequences, but rather will lessen the Quality of Service (QoS) of the system. For example, a multimedia application that needs to periodically

perform buffering, decoding and visualization of frames, needs to complete each activation by a precise deadline. However, a relatively small percentage of deadline misses can be tolerated, with the user perceiving a degraded quality as said percentage grows.

As many real-time tasks may be concurrently active on the same CPU, they interfere with each other, jeopardizing their deadlines. Therefore, we make use of *resource reservations* to enforce *temporal isolation* among tasks. Each task τ_i is associated with its own reservation (Q_i, P_i), meaning that τ_i is guaranteed to be scheduled on the CPU for Q_i time units (a.k.a., *budget*) in every time interval of length P_i (a.k.a., *reservation period*).

In this paper, we use the Constant Bandwidth Server (CBS) (Abeni and Buttazzo, 1998) as a resource-reservation mechanism. In CBS, reservations are realized by means of the Earliest Deadline First (EDF) scheduler, which schedules tasks $\{\tau_i\}$ based on their *scheduling deadlines* $\{d_i\}$, assigned by the CBS algorithm. When a new job $J_{i,j}$ arrives, the server checks whether the current deadline is sufficient to schedule it, otherwise it assigns a new deadline equal to $r_{i,j} + P_i$. While the job executes, the budget of the associated CBS is decreased. If it executes for more than Q_i time units, its scheduling deadline is postponed by P_i. Therefore, the job is prevented from executing more than Q_i time units with the same scheduling deadline, and it is guaranteed a computation bandwidth of $B_i = Q_i/P_i$ regardless of the behaviour of the other tasks. This ensures temporal isolation, preventing a misbehaving task to cause deadline misses on jobs of other tasks with farther away deadline. To guarantee the schedulability of each task, the following *schedulability condition* must hold:

$$\sum_i B_i \leq U_{max} \tag{1}$$

with $U_{max} = 1$ in case of EDF.

A CBS server may be associated with many tasks. However, in this paper, each one will have its own CBS server. For the reader's convenience, Table 1 summarizes the notation symbols used throughout this paper.

An energy-aware extension of the CBS server is GRUB-PA (*Greed Reclamation of Unused Bandwidth — Power Aware*), integrating the ability to reclaim unused processor capacity (*bandwidth*) that is not used because some of the servers may have no jobs awaiting execution and exploiting the hardware DVFS capabilities to reduce the cores frequency. GRUB-PA has been implemented in the mainline Linux running SCHED_DEADLINE CBS reservations, starting from version 4.13 released in September 2017. Its energy-related behaviour on multiprocessor platforms could be summarized as follows. When a new task instance $J_{i,j}$ arrives, the first free core is selected if available; otherwise, the core with the latest-deadline task is chosen and the new task is dispatched onto it. Then, in the case of ARM big.LITTLE, the highest-utilization core of each island is used to determine the frequency (or the highest frequency is picked if the busiest core has utilization greater than 1.0). When a task in a server ends, leaving its core idle, the task with closest deadline is pulled onto it and the islands frequencies are adjusted. The reader can find more details about the GRUB-PA in Scordino and Lipari (2004).

4. Notation and energy model

This section presents the models and notation used throughout the paper to represent the underlying processing platform, the energy model, and the scheduled task sets.

4.1. Platform model

The processing platform under study is composed of two core islands, where each island s ($s \in \mathcal{I} \triangleq \{L, B\}$) has m_s identical cores that can (all together) be switched among a set of k_s possible operational performance points (OPPs) with different frequencies $\{f_{s,1}, \ldots, f_{s,k_s}\}$, ordered from the minimum to the maximum one. The per-core power consumption at frequency $f_{s,j}$ is $p_{s,j}$ when computing, or $p^{idle}_{s,j} < p_{s,j}$ when staying idle, where $\forall j_1, j_2 \in \{1, \ldots, k_s\}$, $j_1 < j_2 \implies p_{s,j_1} < p_{s,j_2} \wedge p^{idle}_{s,j_1} < p^{idle}_{s,j_2}$. For the sake of simplicity, in this work, we ignore the existence of multiple deep-sleep idle modes of the CPU(s) with different associated power consumptions, postponing their proper integration in our technique to future work.

We define the *maximum speed* $x_s \leq 1$ of a core of an island s to be the ratio between the processing time C of a task deployed on a core of the big island at maximum frequency f_{B,k_B} and its processing time C_{s,k_s}, when deployed on that core at maximum frequency f_{s,k_s} : $x_s = \frac{C}{C_{s,k_s}}$.

A core of the island s, running at OPP $j < k_s$, has a *reduced speed* $x_{s,j} < x_s$, where a common assumption is that $x_{s,j} = x_s \frac{f_{s,j}}{f_{s,k_s}}$, albeit the approach of this paper relies on arbitrary processing speed factors $\{x_{s,j}\}$ (monotonically increasing with j), being inspired by the capacity matrix as available in the device tree in the Linux kernel supporting EAS.

4.2. Task model

Each task τ_i is assumed to be a *periodic* task with known period T_i, or equivalently a *sporadic* task with minimum inter-arrival time among two subsequent activations of T_i, and with a known *nominal WCET* C_i, being the WCET of the task when running on a big core at maximum frequency. Similarly, the *nominal utilization* U_i is defined as $U_i \triangleq \frac{C_i}{T_i}$. Whenever the task is running on a core of an island s at an OPP j, its timing is characterized by the *scaled WCET* $\tilde{C}_{i,s,j}$ and *scaled utilization* $\tilde{U}_{i,s,j}$, defined as:

$$\tilde{C}_{i,s,j} = \frac{C_i}{x_{s,j}} \left[= \frac{C_i}{x_s} \frac{f_{s,k_s}}{f_{s,j}} \right]; \quad \tilde{U}_{i,s,j} \triangleq \frac{\tilde{C}_{i,s,j}}{T_i} \equiv \frac{U_i}{x_{s,j}}. \tag{2}$$

Our task model comprises a few common task states: a task *arrives* or *activates*, becoming *ready* to run, periodically (or with a minimum periodicity), generating a sequence of *jobs*. When the task is selected for execution on a CPU then it becomes *running*, and its current job starts being executed. While running, a task may be *preempted* by another task just arrived (or migrated from another CPU) with an earlier deadline, causing the former task to go back to the ready state. Each job completes with the task *suspending*, waiting for the next activation, when the task will *wake-up* and become again ready to run, then its next job will start executing when it is scheduled.

This paper focuses on approaches where the set of scheduled tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ can be partitioned among the available cores, so that, at any time, each core h of an island s hosts a subset of tasks $\Gamma_{s,h} \subseteq \Gamma$ meeting some schedulability condition. For example, if EDF is used, the well-known EDF schedulability condition has to be met *on the scaled utilizations*, considering the OPP j at which the island is configured:

$$\sum_{i \in \Gamma_{s,h}} \tilde{U}_{i,s,j} \leq 1 \iff \sum_{i \in \Gamma_{s,h}} U_i \leq x_{s,j} \ \forall h \in \{1..m_s\}. \tag{3}$$

However, the proposed method assumes to have no prior knowledge of what real-time tasks will appear in the scheduler queue over time; thus, it focuses on an adaptive, on-line approach (details in Section 5).

Table 1
Summary of the symbols used in the paper. Some terms are introduced in the following sections.

Symbol	Meaning
τ_i	ith task ($i = 1, \ldots, n$)
Γ	Set of tasks $\Gamma = \{\tau_i\}_{i=1,\ldots,n}$
C_i	Nominal worst-case execution time (WCET) of τ_i
T_i	Period of the instances of τ_i
d_i	Scheduling deadline of τ_i
$U_i = C_i/T_i$	Nominal utilization of task τ_i with WCET C_i and period T_i
k_s	Number of OPPs for island s
$f_{s,j}$	CPU frequency for island s when running at OPP j ($j = 1, \ldots, k_s$)
$p_{s,j}$	Per-core power consumption at frequency $f_{s,j}$
$x_{s,j}$	Speed of island s at OPP j
x_s	Maximum speed each core of island s can reach
$\tilde{U}_{i,s,j}$	Scaled utilization of τ_i on island s with OPP j
$\tilde{C}_{i,s,j}$	Scaled WCET of τ_i on island s with OPP j
$\tilde{E}_{i,s,j}$	Energy consumption over its period T_i of τ_i on island s with OPP j
$\tilde{E}_{\Gamma,s,j}$	Energy consumption over the hyperperiod of tasks in Γ on island s with OPP j
$\tilde{P}_{\Gamma,s,j}$	Power consumption over the hyperperiod of tasks in Γ on island s with OPP j
$\tilde{U}_{s,j_s}$	Scaled utilization of the tasks on all cores of island s running at OPP j_s
m_s	Number of cores in island s
$\Gamma_{s,h}$	Set of tasks on core h of island s
$V_{s,h} = \sum_{i \in \Gamma_{s,h}} U_i$	Overall nominal utilization of tasks on core h of island s
$V_s = \sum_{h=1}^{m_s} V_{s,h}$	Overall nominal utilization of tasks on all cores of island s

4.3. Energy consumption model

A task τ_i deployed alone on a core of an island s at frequency j keeps the core busy for a time $\tilde{C}_{i,s,j}$ and idle for a time $T_i - \tilde{C}_{i,s,j}$ in each time window with a duration of its period T_i, resulting in an overall energy consumption over its period equal to:

$$E_{i,s,j} = p_{s,j}\tilde{C}_{i,s,j} + p^{idle}_{s,j}\left(T_i - \tilde{C}_{i,s,j}\right).$$

Similarly, for a schedulable set of tasks Γ deployed on a core of island s at frequency j, we can compute the overall energy consumption over the hyperperiod $H_\Gamma \triangleq LCM_i\{T_i \mid \tau_i \in \Gamma\}$, considering that each task τ_i will have $\frac{H_\Gamma}{T_i}$ instances over a time duration of H_Γ (with reference to a schedule with null initial offsets):

$$E_{\Gamma,s,j} = p_{s,j}\sum_{i\in\Gamma}\tilde{C}_{i,s,j}\frac{H_\Gamma}{T_i} + p^{idle}_{s,j}\left(H_\Gamma - \sum_{i\in\Gamma}\tilde{C}_{i,s,j}\frac{H_\Gamma}{T_i}\right).$$

For any practical calculation, it is convenient to divide the above equation by H_Γ, obtaining the *average power consumption* $P_{\Gamma,s,j}$ (over the hyperperiod) of a schedulable task set Γ deployed on a core of an island s at frequency j:

$$P_{\Gamma,s,j} \triangleq \frac{E_{\Gamma,s,j}}{H_\Gamma} = p_{s,j}\sum_{i\in\Gamma}\tilde{U}_{i,s,j} + p^{idle}_{s,j}\left(1 - \sum_{i\in\Gamma}\tilde{U}_{i,s,j}\right).$$

Considering a system where each island s is running at OPP j_s and each core h on island s is hosting a task set $\Gamma_{s,h}$, the *overall average power consumption* for the system is defined as:

$$\begin{aligned} P &\triangleq \sum_{s\in\mathcal{I}}\sum_{h=1}^{m_s} P_{\Gamma_{s,h},s,j_s} \\ &= \sum_{s\in\mathcal{I}}\sum_{h=1}^{m_s}\left[p_{s,j_s}\sum_{\tau_i\in\Gamma_{s,h}}\tilde{U}_{i,s,j_s} + p^{idle}_{s,j_s}\left(1 - \sum_{\tau_i\in\Gamma_{s,h}}\tilde{U}_{i,s,j_s}\right)\right] \\ &= \sum_{s\in\mathcal{I}}\left[p_{s,j_s}\tilde{U}_{s,j_s} + p^{idle}_{s,j_s}\left(m_s - \tilde{U}_{s,j_s}\right)\right], \end{aligned} \tag{4}$$

where $\tilde{U}_{s,j_s} \triangleq \sum_{h=1}^{m_s}\sum_{\tau_i\in\Gamma_{s,h}}\tilde{U}_{i,s,j_s}$ is the overall scaled utilization of the tasks hosted on all cores of island s when running at OPP j_s.

The metric P defined in Eq. (4) is the main driver for our proposed task placement algorithm that will be presented next.

5. Proposed approach

In the proposed approach, real-time tasks are dynamically partitioned among the available cores and they are scheduled on the assigned cores using the CBS scheduling policy, based on EDF. In the rest of the paper, a common practice is applied that assigns a single task to each CBS server, setting the server budget equal to the task WCET C_i and the server period equal to the task minimum inter-arrival time T_i. Migrations of tasks among CPUs can dynamically occur at job-level, whenever a task suspends (i.e., its current job ends), and its active utilization expires, or a task wakes up (i.e., a new job begins).

In order to decide the CPU on which to place a new task that becomes ready-to-run, and at what CPU frequency, we propose a *greedy algorithm aiming at minimizing the average power consumption P* as defined in Eq. (4). The heuristic is based on choosing on a task wake-up (bringing it back into the scheduler queue of ready tasks) a core placement decision that causes the minimum P increase, among all the possible moves that keep the schedulability of all the tasks. Also, whenever the active utilization of a task expires on a core, the frequency of the corresponding island is lowered to the minimum one that keeps the schedulability of all the tasks on the island.

Whenever multiple choices are available, bringing the same difference in the average power consumption, we give preference to spreading and balancing the workload across the available cores, adopting a *worst-fit* strategy.

In the following, we start from important observations on the power consumption of a single task in Section 5.1, a set of tasks across single-core islands in Section 5.2 and multi-core islands in Section 5.3. Then, the overall placement algorithm is presented in Section 5.4. Additional implementation details and observations are discussed in Section 7, including the discussion of possible efficiency issues and the computational complexity of the proposed algorithm.

5.1. Single-task placement

Proposition 1. *A set of tasks $\Gamma_{s,h}$ can be hosted on any core of an island s at a given OPP j only if their overall nominal utilization does not exceed a maximum value $U^{max}_{s,j}$ equal to the speed $x_{s,j}$ corresponding to the OPP j.*

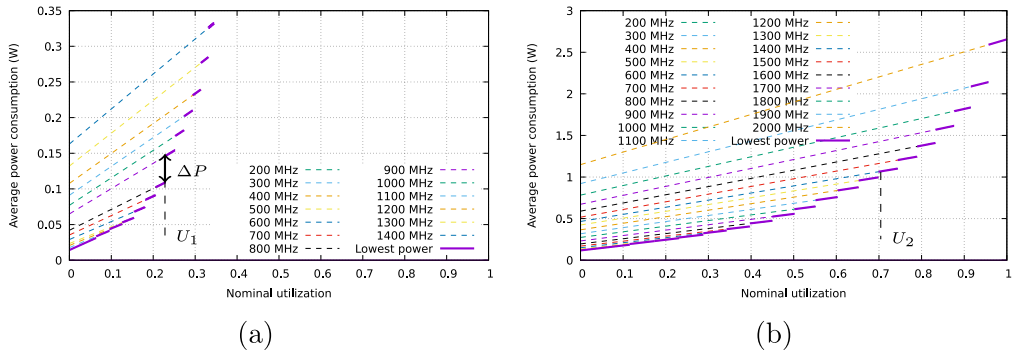


Fig. 1. Contribution to the average power consumption (on the Y axis) due to a single core as a function of the aggregated nominal utilization of the hosted tasks (on the X axis), for LITTLE cores (left plot) or big ones (right plot).

Proof. This follows easily from the schedulability condition in Eq. (3) and the scaled WCET definition in Eq. (2):

$$\sum_{i\in\Gamma_{s,h}} \tilde{U}_{i,s,j} \leq 1 \iff \sum_{i\in\Gamma_{s,h}} \frac{C_i/x_{s,j}}{T_i} \leq 1$$

$$\sum_{i\in\Gamma_{s,h}} \frac{C_i}{T_i} \equiv \sum_{i\in\Gamma_{s,h}} U_i \leq x_{s,j} \triangleq U_{s,j}^{max}. \quad \square$$

Note that the $x_{s,j} \equiv U_{s,j}^{max}$ are monotonically increasing with the OPP j.

Highlighting the role of the nominal utilizations in Eq. (4), we have:

$$\begin{aligned} P &= \sum_{s\in\mathcal{I}} \left[p_{s,j_s} \tilde{U}_{s,j_s} + p_{s,j_s}^{idle} \left(m_s - \tilde{U}_{s,j_s} \right) \right] \\ &= \sum_{s\in\mathcal{I}} \left[m_s p_{s,j_s}^{idle} + \left(p_{s,j_s} - p_{s,j_s}^{idle} \right) \frac{V_s}{x_{s,j_s}} \right] \end{aligned} \tag{5}$$

where we introduced the *aggregated* nominal utilization V_s of the tasks $\cup_{h=1}^{m_s} \Gamma_{s,h}$ in all cores of an island s: $V_s \triangleq \sum_{h=1}^{m_s} \sum_{\tau_i\in\Gamma_{s,h}} U_i$. Similarly, we use $V_{s,h}$ to refer to the aggregated nominal utilization of the tasks $\Gamma_{s,h}$ currently on core h: $V_{s,h} \triangleq \sum_{\tau_i\in\Gamma_{s,h}} U_i$.

Eq. (5) states that the power consumption of an island running at a given OPP is linearly dependent on the overall nominal utilization deployed across the cores in the island. This is highlighted in Fig. 1, reporting, for each available OPP (different curves) of either the big or the LITTLE island (different plots) of an ODROID-XU3 board, the contribution to the average power consumption of each core (Y axis) as a function of the aggregated nominal utilization hosted on the core (X axis). Due to Proposition 1, the maximum nominal utilization that can be hosted on a single core of island s at OPP j cannot exceed the associated speed $x_{s,j}$, thus each line in Eq. (5) is only displayed in the range $[0, x_{s,j}]$ of the X axis.

Looking at the power curves in Fig. 1, it is clear that, given a single real-time task with a given nominal utilization U_i, the minimum average power consumption on each island is attained by using the minimum OPP with speed $x_{s,j} \geq U_i$. By taking such a choice for each possible U_i, we obtain the thick power-utilization curves labelled as "Lowest power".

However, optimum placement decisions need to take into account the multitude of tasks and cores in the platform, as discussed next.

5.2. Multi-task placement

The minimum among the thick curves in the plots in Fig. 1 allows us to decide, given a single task with a utilization U_i, where to place and schedule it in the most energy-efficient way, so that no deadlines can be missed.

However, our real system has a number of real-time tasks. In order to highlight how the problem becomes more complex, consider a simple conceptual example. Imagine a system with just one core for the big and one core for the LITTLE islands; we have already placed one lightweight task τ_1 on the LITTLE core, with a nominal utilization of $U_1 = 0.2296$, which is just a tiny bit below a relatively big power consumption jump for LITTLE cores, as shown in Fig. 1(a); also, we already placed a heavyweight task τ_2 with nominal utilization above the LITTLE maximum speed of $x_L = 0.345328$, for example $U_2 = 0.71$, as shown in Fig. 1(b). Clearly, both cores are running at the minimum frequency guaranteeing schedulability, i.e., the big core at a frequency of 1400 MHz, and the LITTLE one at a frequency of 800 MHz.

Under said conditions, if a very lightweight task τ_3 with a small U_3 arrives, placing it on the LITTLE core is not optimal. Indeed, to preserve schedulability, such a placement would force to bump the LITTLE core frequency up one step, adding to the power consumption P a fixed term $\Delta P = 0.0372$ W, plus an additional increment per nominal utilization unit of 0.355 W. On the other hand, the big core is in a status in which it would be able to host τ_3 without any frequency change, causing an increase in P of just a (steeper) increment per nominal utilization unit of 0.855 W. Therefore, with sufficiently small U_3 values, the most energy efficient action is achieved by placing τ_3 on the big core, alongside τ_2. Fig. 2 shows the total increase in the average power consumption achieved by placing τ_3 on either the big or the LITTLE core, as a function of the new task utilization U_3. As visible, it is more convenient to place τ_3 on the big core if $U_3 \leq \sim 0.034$, while the LITTLE core is a better choice for $U_3 > \sim 0.034$ and $U_3 < \sim 0.116$. Beyond this last utilization, it cannot fit on the LITTLE core and the only choice is an assignment on the big one.

The above reasoning introduces the motivations behind our scheduling strategy design, based on two points. First, every time a task enters the ready queue (i.e., at task creation or wake-up time) it is placed on the core causing the minimum possible increase in the overall average power consumption P as defined in Eq. (5), also considering the potential need for increasing the operating frequency of the destination island, in order to preserve schedulability. Second, every time a task exits the ready queue (i.e., it goes to sleep or terminates), a pull operation is scheduled at the task active utilization expiry time (the task virtual time if other tasks are ready on the CPU, or the current time if the CPU remains idle — this ensures that schedulability is preserved, Scordino et al., 2019), which migrates a task from another CPU determining the move with the maximum possible decrease in P, also considering the potential need for switching to a higher OPP for the destination island, and possibly the opportunity to switch the big island to a lower OPP, in case we can pull from there.

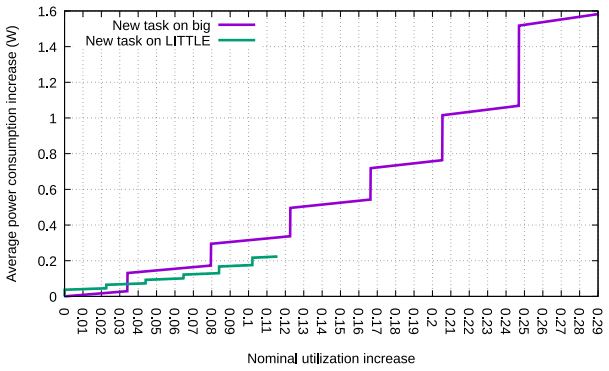


Fig. 2. Increase in average power consumption (on the Y axis) due to placing a new task on the big vs LITTLE core (different curves) as a function of its utilization U_3 (on the X axis), in a sample scenario.

5.3. Placement for multi-core islands

Focus on an island s with m_s cores. First, we observe that, as highlighted in Eq. (5), a multi-core island s has a power consumption that depends on the OPP j_s currently being used, and the aggregated nominal utilization V_s of all cores in the island. Therefore, while a single-core contribution to the overall power consumption P is well captured by a point on one of the segments in Fig. 1, the contribution of a whole island to P is captured by a point that stays on the same curves, but likely residing on the extension of the segment beyond the maximum speed $x_{s,j}$ of the OPP j_s, up to a maximum per-island utilization V_s of $m_s x_{s,j}$ (e.g., 4 times as much, in the case of our ODROID-XU3 platform with 4 cores per island). The average power consumption curve, for each island s and OPP j, as a function of the overall nominal utilization hosted on the island, will be denoted by $P_{s,j}(\cdot)$.

Whenever a task τ_i with nominal utilization U_i becomes ready, we need to compare what increase in the power consumption P would arise from hosting the task on each of the islands.

On an island s, τ_i can easily be hosted at the same frequency if there is at least a core h with nominal utilization that, increased by U_i, does not exceed the CPU speed x_{s,j_s} due to the current island frequency j_s (see Proposition 1). However, whenever more than one core satisfies this requirement, we propose to adopt a *worst-fit* (WF) placement strategy, that tries to fit the task into the least-loaded core, i.e., the one with the minimum overall nominal (and scaled) utilization among the hosted tasks. This choice is motivated by the need to try to keep each island OPP at a value that is as small as possible, and this is achieved by spreading the workload as evenly as possible across the cores of each island. Note that such a placement is also the one that minimizes the variance in the nominal utilization among cores in the same island. However, any placement over any other core in such a way that the total achieved nominal utilization on the core does not exceed x_{s,j_s} brings the same identical increase in power consumption, so these are all equivalent solutions, including widely known alternatives like: first-fit (FF), choosing the first core with enough free utilization; or best-fit (BF), choosing the core with the smallest free utilization greater than, or equal to, the one of the task to be placed.

If we cannot place the task preserving the current island OPP j_s, we need to factor the increase in power consumption P due to each possible OPP switch. Here, our WF-based choice allows us to search for the minimum OPP j_s^* with an associated speed greater than or equal to the sum of the nominal utilization $U_{h'_s}$ of the least-loaded core h'_s, plus the one of τ_i:

$$j_s^* = \min_{j > j_s \wedge j \le k_s} \left\{ j \mid U_{h'_s} + U_i \le x_{s,j} \right\}. \tag{6}$$

Then, in order to decide what island to deploy τ_i to, we compare the increase ΔP_s in the power consumption P due to a deployment as from Eq. (6), for each island $s \in \mathcal{I}$:

$$\Delta P_s = P_s(V_s + U_i) - P_s(V_s) \tag{7}$$

where $P_s(\cdot)$ is the min-power curve obtained by combining together all the "lowest power" segments highlighted in the plots of Fig. 1, computed according to Eq. (5). When implementing the $P_s(\cdot)$ function in software, a few alternatives are possible achieving different speed vs memory consumption trade-offs, as discussed later in Section 7.

5.4. Placement algorithm

Putting together the building blocks from the above sections, we obtain the overall algorithm that is summarized in Algorithm 1, to be run every time a task enters the ready queue (task creation time or new job arrival). The procedure finds the core where the task fits according to Eq. (3) and for which ΔP_s is minimum. Notice that this pseudo-code is not efficient and it is presented this way for the sake of clarity, while its efficient implementation is discussed in Section 7.

Algorithm 1 Placement of a new task τ_i.

```
1: // Return minimum OPP j of island s needed for per-core utilizations {U_i}
2: procedure MINOPP(island s, utilizations {U_i})
3:     return min{ j ≤ k_s | x_{s,j} ≥ max_i{U_i} }
4: end procedure
5:
6: // Return island and core where to place τ_i with nominal utilization U_i
7: procedure PLACE(utilization U_i)
8:     for each island s ∈ I do
9:         choose h*_s | V_{s,h*_s} = min_h{V_{s,h}}
10:        if V_{s,h*_s} + U_i ≤ x_s then
11:            set W := {V_{s,h}}_h where V_{s,h*_s} is increased by U_i
12:            set j* := MINOPP(s, W)
13:            set ΔP_s := P_{s,j*}(V_s + U_i) − P_{s,j_s}(V_s)
14:        else
15:            set ΔP_s := +∞
16:        end if
17:    end for
18:    choose s* | ΔP_{s*} = min_{s∈I}{ΔP_s}
19:    return (s*, h*_{s*})
20: end procedure
```

When a task running on a core c_{empty} goes to sleep (or terminates) and its virtual time expires, a pull operation is needed. We distinguish pull operations into (i) pull of a task from another core of the same island; and (ii) pull of a task from the big to the LITTLE island. At the moment, a pull operation is attempted only if c_{empty} is left idle. The possibility to pull tasks also if c_{empty} is not idle is left as future work. Running tasks are not pulled to avoid compromising their real-time guarantees, since pulling a task in the middle of a job may imply a non-negligible increase of its execution time, especially for a migration across islands.

In our proposed strategy, summarized in Algorithm 2, if c_{empty} is a LITTLE core, then we try first to pull a task from the big island. If this is not possible, or c_{empty} is a big core, then we try to pull from the same island (Lines 2–9). Assume by now that $U_i^{infl} \equiv U_i$, this will be clarified at the end of this section.

While pulling a task from the big island to the LITTLE core c_{empty} (Lines 10–29), we reduce as much as possible the big island utilization, while trying to leave the overall load across the cores balanced. To this end, we pull the ready task τ_i with the highest utilization U_i from the busiest big core c_{max} that fits inside the target CPU c_{empty} and that produces energy saving.

In case there is no big task satisfying the just mentioned condition, or c_{empty} is a big core, then we pull from the busiest

Algorithm 2 Pull algorithm onto a core c_{empty} that was left idle.

```
1: // B denotes big cores, L denotes LITTLE ones.
2: // Find a task to pull onto the empty core c_empty from another core
3: procedure PULL(c_empty)
4:     set result := PULLFROMBIG(c_empty)
5:     if result is NULL then
6:         set result := PULLFROMSAMEISLAND(c_empty)
7:     end if
8:     return result
9: end procedure
10: // Return task to be migrated onto c_empty and big core where to pull it from, or NULL
11: procedure PULLFROMBIG(c_empty)
12:     if c_empty ∈ L then
13:         choose c_max | U_{c_max} = max_{c∈B} {U_c}
14:         set R := List of ready tasks τ_i in c_max, sorted by decreasing utilization U_i.
15:         for each τ_i ∈ R do
16:             if U_i^infl ≤ x_L then
17:                 set W_L := {V_{L,h}} where V_{L,c_empty} is set to U_i^infl
18:                 set j_L^* := MINOPP(L, W_L)
19:                 set W_B := {V_{B,h}} where V_{B,c_max} is decreased by U_i
20:                 set j_B^* := MINOPP(B, W_B)
21:                 set ΔP_s := P_{B,j_B^*}(V_B − U_i) + P_{L,j_L^*}(V_L + U_i) − (P_{B,j_B}(V_B) + P_{L,j_L}(V_L))
22:                 if ΔP_s < 0 then
23:                     return (τ_i, c_max)
24:                 end if
25:             end if
26:         end for
27:     end if
28:     return NULL
29: end procedure
30: // Return task to be migrated onto c_empty and same-island core where to pull it from
31: procedure PULLFROMSAMEISLAND(c_empty)
32:     set s := island of core c_empty
33:     set S := cores in island s
34:     choose c_max | U_{c_max} = max_{c∈S} {U_c}
35:     if c_max == c_empty then
36:         return NULL
37:     end if
38:     set R := List of ready tasks τ_i in c_max with U_i < U_{c_max}/2, sorted by decreasing U_i.
39:     for each τ_i ∈ R do
40:         if U_i^infl ≤ x_L then
41:             set W := {V_{s,h}} where V_{s,c_empty} = U_i^infl and V_{s,c_max} is decreased by U_i
42:             set j^* := MINOPP(s, W)
43:             if j^* < j_s then
44:                 return (τ_i, c_max)
45:             end if
46:         end if
47:     end for
48:     return NULL
49: end procedure
```

core of the same island as c_{empty} (Lines 30–49). This is done aiming at balancing the overall nominal utilization hosted on the source core (whose utilization decreases) and the destination one (whose utilization increases) of the same island, in order to maximize the possible OPP reduction. Therefore, as highlighted in Line 38, Algorithm 2 tries to pull the biggest task with $U_i < \frac{U_{c_{max}}}{2}$ in this case.

Note that, at any time t when we evaluate a pull operation, it is necessary to consider that the task being pulled might have already partially executed on the source CPU, so it has a leftover (nominal) WCET $c_i \leq C_i$ that has to be scheduled on the destination CPU within the scheduling deadline $d_i > t$ already in place for the task. Therefore, it is safe to migrate the task only if its *inflated* utilization fits on c_{empty}, which also impacts on the minimum OPP needed on c_{empty} in order to perform correctly the migration:

$$U_i^{infl} \triangleq \frac{c_i}{d_i - t} \geq \frac{C_i}{T_i}. \tag{8}$$

This is why we need to use U_i^{infl} instead of U_i in the pseudo-code following Lines 16 and 40, to determine whether pulling τ_i is safe. Estimating whether pulling a task is energetically convenient, either from the same and the big island, is done by finding the new speed of both islands after the pull and seeing if there is a reduction of energy consumption in the whole system. In these power consumption calculations, the real task utilization is used (as argument to $P_{s,j}(\cdot)$), but the actual OPP to be used on the island of c_{empty} needs to account for the inflated nominal utilization.

Additional notes about the above algorithm are reported in Section 7, along with some implementation details.

6. Schedulable task sets

In this section, we present some of the theoretical conditions under which a set of real-time tasks scheduled according to the technique presented in Section 5 is expected to be schedulable, namely to never miss a deadline. We anticipate that our set of identified conditions is *sufficient*, not *necessary*, for schedulability of the task sets. This allows us to have quick and easy-to-compute admission tests to understand whether a new real-time task can be safely admitted into the system guaranteeing timeliness of execution for all the admitted tasks.

First, we need a preliminary result that can be formulated with reference to a single island s of identical cores. Also, we start enumerating these conditions under simplifying assumptions:

A1 the operation of frequency switch for a given island takes a negligible time;

A2 the operation of context switch among tasks and with the idle task takes a negligible time on any of the cores;

A3 tasks are only pushed according to the rule in Algorithm 1, they are never pulled while running or ready to run.

At the end, we will discuss how to properly relax these assumptions without breaking the schedulability properties. In the following theorems and propositions, notice that the taskset Γ is ordered in decreasing nominal utilization order and, for instance, the highest-utilization task has utilization $U_1 \equiv U^{max}$, while the second highest-utilization task has utilization $U_2 \equiv U^{max2} \leq U_1$.

Proposition 2. *A set of n tasks $\Gamma = \{1, \ldots, n\}$, ordered from the maximum to the minimum nominal utilization, is schedulable by the technique presented in Section 5 on a platform having a multi-core island s with m_s cores if:*

$$n \leq m_s \left\lfloor \frac{x_s}{U^{max}} \right\rfloor \tag{9}$$

where $U^{max} \triangleq \max_{\tau_i \in \Gamma}\{U_i\}$ is the maximum nominal utilization among all the tasks in Γ, and x_s is the speed of a core at the maximum island OPP k_s.

Proof. Whenever evaluating where to place a task, our proposed heuristic applies a worst-fit strategy, where each placement takes place on one of the least loaded cores h_s^* of the island (see Algorithm 1). If the current frequency does not give h_s^* a sufficient computational capacity, we are ready to raise the island frequency to the minimum OPP such that the new task fits (its nominal utilization, plus the one of the core, do not exceed the maximum limit due to the OPP as from Proposition 1). Thanks to the assumptions $A1$ and $A2$, this can all be done in zero time, so a task is certainly schedulable if its utilization fits within the least-loaded core when bumped-up at the maximum island OPP k_s. As the maximum utilization that can be safely hosted on any core at max OPP is x_s, our proposition proof is reduced to proving that this is always the case under the assumptions in Eq. (9).

This is trivially done by contradiction: assume we have a task τ_i that does not fit on any of the m_s cores of the island. This means that $\forall h \in [1, \ldots, m_s], U_{s,h} + U_i > x_s$. However, as each task has a maximum utilization of U^{max}, then $x_s < \sum_{j \in \Gamma_{s,h}} U_j + U_i \leq U^{max}\left(\left|\Gamma_{s,h}\right| + 1\right)$, therefore $\frac{x_s}{U^{max}} < \left|\Gamma_{s,h}\right| + 1 \implies \left|\Gamma_{s,h}\right| \geq \left\lfloor \frac{x_s}{U^{max}} \right\rfloor$. Summing up this for all task-sets $\Gamma_{s,h}$ of all cores, along with the new task τ_i to be placed, which all together constitute a partitioning of Γ, we obtain an evident contradiction to Eq. (9). □

In case of a particularly "nasty" task-set with only a heavyweight task with utilization U^{max} much higher than all other tasks, bringing down significantly the maximum number of tasks that could be hosted according to Proposition 2, we can identify an equally simple but more convenient condition to verify, peeking at the second maximum utilization U^{max2}:

Proposition 3. *A set of n tasks $\Gamma = \{1, \ldots, n\}$, ordered from the maximum to the minimum nominal utilization, is schedulable by the technique presented in Section 5 on a platform having a multi-core island s with m_s cores if:*

$$n \leq 1 + \left\lfloor \frac{x_s - U^{max}}{U^{max2}} \right\rfloor + (m_s - 1)\left\lfloor \frac{x_s}{U^{max2}} \right\rfloor. \tag{10}$$

where U^{max} and U^{max2} are the maximum and second-maximum nominal utilizations among all the tasks in Γ, and x_s is the speed of the core at the maximum island OPP k_s.

Proof. This is easily obtained again by contradiction, putting ourselves into a scenario where we have a task $\tau_i \neq \tau_1$ whose utilization U_i does not fit onto any core. This time, however, we can put ourselves into a worst-case scenario where we have one of the cores, say $\tilde{h}$, hosting the heavyweight utilization U_1, and all other cores hosting tasks with a maximum utilization of U^{max2}. Therefore, for the core $\tilde{h}$ we have: $x_s < U_{s,\tilde{h}} + U_i \leq U_1 + \left(\left|\Gamma_{s,\tilde{h}}\right| - 1\right) U_2 + U_2 \equiv U_1 + \left|\Gamma_{s,\tilde{h}}\right| U_2 \implies \frac{x_s - U_1}{U_2} < \left|\Gamma_{s,\tilde{h}}\right| \implies \left|\Gamma_{s,\tilde{h}}\right| \geq \left\lfloor \frac{x_s - U_1}{U_2} \right\rfloor + 1$. For any other core $h \neq \tilde{h}$, we have $\left|\Gamma_{s,h}\right| \geq \left\lfloor \frac{x_s}{U_2} \right\rfloor$. Putting tasks of all cores together, alongside the new task being placed τ_i, we conclude that the overall number of tasks must be

$$|\Gamma| = \left|\Gamma_{s,\tilde{h}}\right| + (m_s - 1)\left|\Gamma_{s,h}\right| + 1 \geq \left\lfloor \frac{x_s - U_1}{U_2} \right\rfloor + (m_s - 1)\left\lfloor \frac{x_s}{U_2} \right\rfloor + 2,$$

which is in contradiction with Eq. (10).

The case $\tau_i = \tau_1$ can be handled in a very similar way. □

Generalizing, we can design an admission test peeking at the 2nd, 3rd, …, *k*th maximum utilization, but the test looses its simplicity and its complexity grows in a combinatorial way. A discussion of these tests for SMP platforms can be found in Mascitti et al. (2020).

We focus now on the general problem of admitting a set of real-time tasks Γ into a big.LITTLE platform making use of the scheduling strategy presented in Section 5. First, we need to distinguish among possible *heavyweight* tasks Γ_H with a too high utilization that would not fit on any LITTLE core, from the other *lightweight* tasks Γ_L:

$$\Gamma_H \triangleq \{\tau_i \in \Gamma \mid U_i > x_L\}\,;\, \Gamma_L \triangleq \{\tau_i \in \Gamma \mid U_i \leq x_L\}\,. \tag{11}$$

In our target scenarios, we do not expect to see many heavyweight tasks on the platform. However, whenever they are present, we assume to be able to spread them out evenly on the big cores, for simplicity in the analysis that follows. If heavyweight tasks can arrive dynamically, this means we should be ready to migrate one or more lightweight tasks, to ensure this property. However, we omit here further details for the sake of brevity.

Theorem 1. *Given a big.LITTLE platform with m_L and m_B LITTLE and big cores respectively, and a set of real-time tasks $\Gamma = \Gamma_H \cup \Gamma_L$ partitioned as heavyweight and lightweight ones as from Eq. (11), with heavyweight tasks spread evenly on big cores and lightweight ones placed according to the technique proposed in Section 5, then we always manage to schedule them if the following conditions hold true:*

$$\begin{cases} |\Gamma_H| \leq m_B \left\lfloor \frac{1}{U_B^{max}} \right\rfloor \vee |\Gamma_H| \leq 1 + \left\lfloor \frac{1 - U_B^{max}}{U_B^{max2}} \right\rfloor + (m_B - 1)\left\lfloor \frac{1}{U_B^{max2}} \right\rfloor \\ |\Gamma_L| \leq m_L \left\lfloor \frac{x_L}{U_L^{max}} \right\rfloor + \left\lfloor \frac{1 - h \times U_H^{max}}{U_L^{max}} \right\rfloor (m_B - k) + \left\lfloor \frac{1 - (h+1)U_H^{max}}{U_L^{max}} \right\rfloor k, \end{cases} \tag{12}$$

where $h \triangleq \left\lfloor \frac{|\Gamma_H|}{m_B} \right\rfloor$ and $k \triangleq |\Gamma_H| \mod m_B \equiv |\Gamma_H| - \left\lfloor \frac{|\Gamma_H|}{m_B} \right\rfloor m_B$.

Proof sketch. The upper condition in Eq. (12) derives from the simple fact that heavyweight tasks can only be placed on big cores, so the results of Propositions 2 and 3 can be directly applied to the big island and the heavyweight tasks alone, observing that $x_B = 1$.

The lower condition is obtained as the sum of three terms, which are explained as follows. The first term accounts for the maximum number of lightweight tasks that can fit on the LITTLE island, again reusing the result in Proposition 2 (the one in Proposition 3 might be used as well, this is omitted for the sake of simplicity). The second and third terms refer to how many lightweight tasks can fit at most into the big island, using the residual space left by possible heavyweight tasks. Here, we need to consider how the heavyweight tasks are spread throughout the big cores. Whenever we have $|\Gamma_H|$ heavyweight tasks, the worst-fit algorithm will cause $\left\lfloor \frac{|\Gamma_H|}{m_B} \right\rfloor \equiv h$ tasks in Γ_H to be present in each big core, however exactly $(|\Gamma_H| \mod m_B) \equiv k$ cores of the big island will host one more heavyweight task. Therefore, a corresponding maximum utilization of heavyweight tasks needs to be subtracted from the maximum possible big cores capacity, $x_B = 1$, to obtain the nominal utilization available for lightweight tasks to be hosted. □

7. Implementation details

BL-CBS has been implemented[3] within RTSim (Palopoli et al., 2002), a portable, open-source event-based simulator written in C++ allowing to simulate the execution timing of real-time tasks running on multi-processor platforms using various schedulers. This section discusses a few implementation notes regarding an efficient and sound implementation of various critical steps of the algorithm introduced in Section 5.

7.1. Push algorithm

First, in the push algorithm in Algorithm 1, we need to find the core with the minimum nominal utilization. Nowadays big.LITTLE architectures present a small number of cores per island (typically 4), so it is easy to scan through the cores in an island to this purpose. However, for bigger islands of possible future platforms, we can use a min-heap to keep track of the lowest-utilization core for each island. With such a solution, we can choose quickly which core a newly arrived task fits into, with a logarithmic complexity in the number of per-island cores.

Second, in order to evaluate what move is the most convenient one from the average power consumption viewpoint, we

[3] Source code is available at https://gitlab.retis.santannapisa.it/a.mascitti/rtsim-efficient-public-grub-pa.

needed to implement the function $P_s(\cdot)$ that is the min-power curve obtained by combining all the "lowest power" segments highlighted in the plots of Fig. 1. From the in-kernel device tree, we can extract the table reporting, for each available island s and OPP j, the corresponding power consumptions when computing $p_{x,j}$ and when idle $p^{idle}_{x,j}$, as introduced in Section 5.1. However, for computing the linear formula in Eq. (5), we need a data structure for retrieving the correct power terms corresponding to the minimum OPP able to host a given total nominal utilization. This is relatively expensive, so this search for the power terms related to a nominal utilization can be accelerated keeping a sorted data structure, for example, a balanced binary tree, allowing us to perform the look-up in $O(\log k_s)$, with k_s number of available OPPs. Alternatively, with a little waste of memory, it is possible to store a sampled version of the same curve using a utilization expressed, for example, in 1024th, creating a table with 1024 entries, so that the same look-up can be performed in constant $O(1)$ time instead.

7.2. Pull algorithm

The choice of which job to pull has an impact on how much frequencies can actually be reduced. In the proposed method, it is possible to choose a task from the big core with maximum utilization if it fits into the destination core, as described in Section 5. This strategy allows for achieving the best frequency reduction of the big island since the core with maximum utilization is the one that constraints the island frequency.

For the just mentioned choice, we need to find the busiest big core. Similarly to the above observation, this is not difficult for very small islands as in nowadays big.LITTLE platforms. However, in case of a big island with a relatively high number of cores, it is always possible to implement a fast look-up by recurring to a max-heap with alteration operations having logarithmic complexity in the number of cores per island.

After having identified the busiest core, we need to find its biggest hosted task that fits into the destination core. If scanning through the real-time ready tasks of the found core needs acceleration, we can recur to a data structure sorted by (nominal) utilization, like a red-black-tree, as used already in SCHED_DEADLINE for implementing the EDF scheduler queue.

7.3. Frequency switch delay/overhead

The assumption of negligible frequency switch time performed in Section 5 can easily be relaxed considering that we can commonly assume that the frequency switch takes a non-negligible but bounded time at most equal to δ, and that during the transition the CPUs in the island keep computing at an undefined frequency that is comprised between the current frequency and the new frequency, till the new frequency is stable.

In this case, in our calculations in Section 5 and Section 6, it is sufficient to inflate properly each task utilization $U_i = C_i/T_i$. Indeed, even considering the scenario when the task is immediately the earliest deadline one and it is readily scheduled on the CPU, then it will run for a maximum time δ at the minimum speed x^* between the two under consideration, then it will run at the correct final CPU speed for the rest of the time. This means that the correct scaled WCET to use for the task is not $\tilde{C}_{i,s,j} = C_i/x_{s,j}$, but rather $\delta/x^* + (C_i - \delta)/x_{s,j}$ (assuming $C_i > \delta$ for simplicity). The full discussion of the impact of this relationship on the results previously presented is omitted for the sake of brevity.

8. Simulation results

8.1. Compared schedulers

We validated the BL-CBS algorithm described in Section 5 and measured its performance and energy consumption through the class *EnergyMRTKernel* in RTSim with respect to: (i) a variant of BL-CBS based on a first-fit placement strategy, called EDF-FF in what follows, implemented in the same *EnergyMRTKernel* class; (ii) a similar variant based on best-fit, called EDF-BF below; (iii) G-EDF, as available through the class *MRTKernel_Linux5_3_11* in RTSim, simulating the behaviour of the mainline Linux kernel a few years back, running SCHED_DEADLINE CBS reservations without GRUB nor power awareness features; (iv) and *MRTKernel_Linux5_3_11_GRUB_PA* in RTSim, implementing the energy-related behaviour of GRUB-PA, i.e. decreasing the frequency to the minimum required to sustain the utilization of every CPU, and reflecting (part of) the behaviour of the implementation of GRUB-PA in the mainline Linux running SCHED_DEADLINE CBS reservations. In fact, a complete implementation of GRUB-PA would add the bandwidth reclaiming mechanism, which would not benefit our simulated scenarios since task overruns are not considered.

Concerning our implementation of EDF-FF, this policy needs an ordering of the cores, so we consider the LITTLE cores preceding the big ones. When a task τ_i with nominal utilization U_i arrives (i.e., a job begins), it is dispatched to the first core h where it fits in the mentioned ordering, raising its OPP if needed, i.e.: $h = \min\{k \mid U_i \leq x_{s_k} - V_{s_k,k}\}$ (s_k denotes the island of core k). Note that, if $U_i > x_L$, then only a big core can be chosen. Whenever a core h is left idle, the first ready task fitting in h from the last non-idle core $k > h$ where there is one such task, is pulled onto h, if any exists.

In our implementation of EDF-BF, when a task τ_i with nominal utilization U_i arrives (i.e., a job begins), it is placed on the core h of the same island s where it was located with minimum residual capacity where it fits, raising the island OPP if needed: $h \in s$ s.t. $x_s - V_{s,h} = \min_{k \in s}\{x_s - V_{s,k} \mid U_i \leq x_s - V_{s,k}\}$.
However, if the current island has no core satisfying said condition, then the other island is tried. Under EDF-BF, pulls are disabled, i.e., when a job completes its execution or its virtual time expires, leaving its core idle, no pull from the other cores is performed.

8.2. Comparison results

The hardware energy consumption model is the one of the ODROID-XU3 board, which uses the Samsung Exynos 5422 SoC. This is an ARM big.LITTLE architecture with four Cortex-A15 and four Cortex-A7 cores. The model has been taken from Balsini et al. (2016), where it has been implemented in RTSim.

We performed a number of experiments with a total nominal utilization in the range [1.6; 5.6] with spacing 0.4, for the reservations. This corresponds to an average per-core nominal utilization in the range [0.2; 0.7] with spacing 0.05. Tasksets contain a total of 24 tasks in each task set, and they have been generated with a modified version of the Taskgen program[4] by Emberson et al. (2010), in which: (i) in order to represent a more realistic and dynamic environment, each task (nominal) WCET has been sized so as to be randomly distributed between 0.6 and 0.9 times the budget of the CBS reservation wrapping it; (ii) the taskgen algorithm is used to generate budgets and periods where periods are randomly generated in the range [1; 100] ms with a relatively

[4] Original version available at: http://retis.sssup.it/waters2010/tools.php.

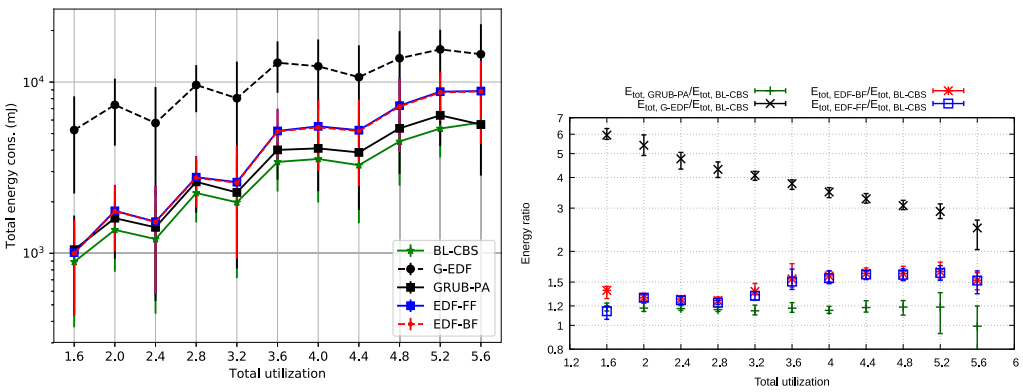


Fig. 3. Left: total energy consumption for different total utilizations, considering 10 experiments for each utilization. Right: energy consumption ratio between (i) G-EDF, BL-CBS; (ii) GRUB-PA and BL-CBS; (iii) EDF-BF and BL-CBS; and (iv) EDF-FF and BL-CBS for different utilizations. *Y* axes are in logarithmic scale.

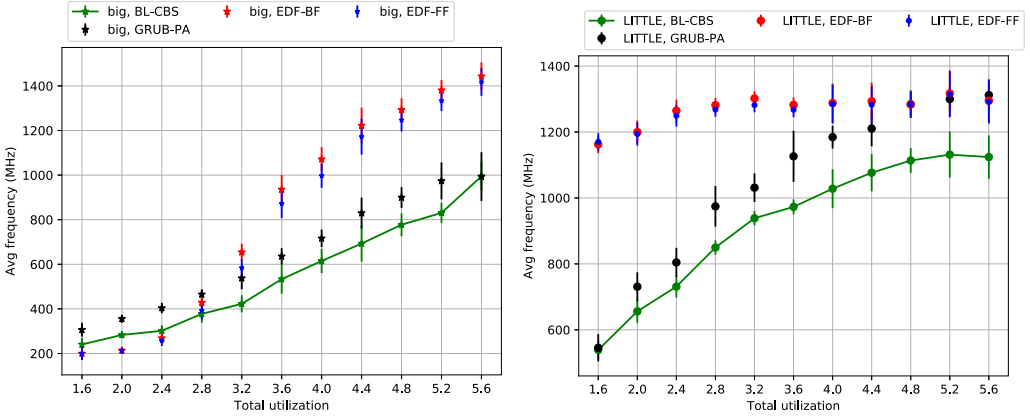


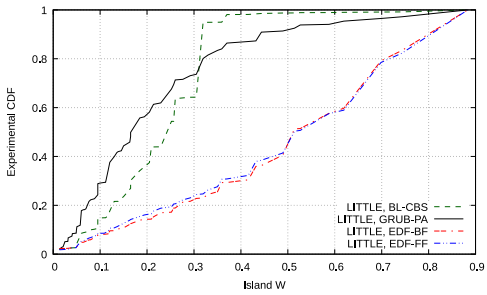
Fig. 4. Average frequency for different total utilizations for BL-CBS, EDF-BF, EDF-FF and GRUB-PA, for the big island (left) and the LITTLE island (right), considering 10 experiments for each utilization.

rough granularity of 0.5 ms, in order to keep under reasonable limits the generated hyperperiod.

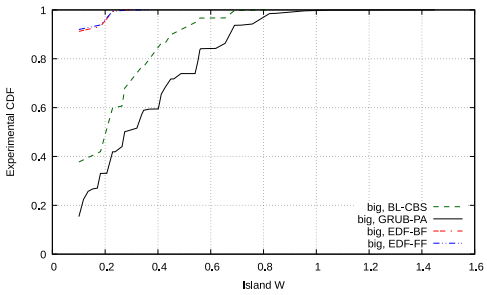
Each run has been repeated 10 times with different seed values for the taskset generator, running each taskset with either BL-CBS, EDF-BF, EDF-FF, GRUB-PA or G-EDF, then statistics have been computed on the resulting values of the various metrics of interest throughout the different runs.

Fig. 3 (left) depicts the obtained overall energy consumption throughout the runs (on the *Y* axis, in logarithmic scale) for each total nominal utilization (on the X axis, divided by 8), and both for the proposed technique, GRUB-PA, EDF-BF, EDF-FF and the G-EDF scheduler (different curves in the plot). Each reported point and its associated vertical bar represents the average energy consumption obtained for the 10 runs and its corresponding standard deviation. The obtained energy consumption with BL-CBS is consistently lower than the one obtained with G-EDF, EDF-BF, EDF-FF and GRUB-PA. This was expected with G-EDF, as BL-CBS on average keeps the frequencies of the islands on lower values than G-EDF, which constraints the two islands to their maximum frequencies. Also, BL-CBS consumes less than GRUB-PA for each total utilization since at each job arrival BL-CBS chooses the core with the minimum power increase, while GRUB-PA picks the core with the latest deadline task, without taking into account the consequent energy consumption. Moreover, GRUB-PA consumes less than G-EDF since the latter keeps the highest cores frequencies. Finally, BL-CBS consumes less than EDF-FF since the latter tends to load the first cores in the ordering as much as possible before using a subsequent empty one, causing high loads on the first core of an island, forcing its OPP to increase in presence of idle cores on the same island. Similarly, BL-CBS consumes less than EDF-BF since the latter does not spread the tasks on the cores of each island, and thus tasks tend to be dispatched on fewer cores of each island, which reduces the chances to reduce the island OPP due to job completions. Generally speaking, as the total utilization grows, the power consumption of all the considered algorithms tends to grow, as the islands are kept for longer times at higher frequencies. Overall, EDF-BF, EDF-FF and G-EDF consume more than BL-CBS and GRUB-PA. BL-CBS achieves 15% of energy saving with respect to GRUB-PA, on average across all the performed experiments.

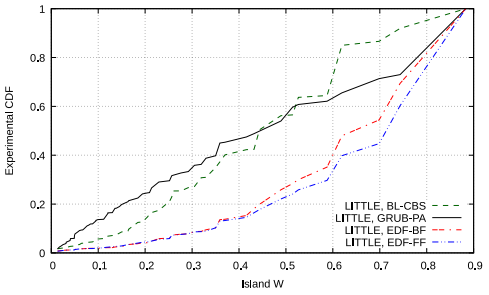
While Fig. 3 (left) takes into account experiments with different hyperperiods (which can be very different among the experiments), Fig. 3 (right) is hyperperiod-independent and shows the ratio (on the *Y* axis) of the overall energy consumption between both (i) GRUB-PA and BL-CBS; (ii) EDF-BF and BL-CBS; (iii) EDF-FF and BL-CBS and (iv) G-EDF and BL-CBS (different curves) over 10 experiments for each total nominal utilization (on the *X* axis, divided by 8). Also, the vertical bars represent the minimum and the maximum ratios found among the experiments. While the ratio between GRUB-PA and BL-CBS is quite stable throughout the *X* axis, the ratio between G-EDF and BL-CBS is more marked and decreases with the utilizations. In case of the highest total nominal utilization $U = 5.6$, we found only 2 experiments where BL-CBS consumes up to 20% more than GRUB-PA. As expected, the ratio between BL-CBS and GRUB-PA is the minimum, and it is similar to the one between BL-CBS and EDF-BF and between BL-CBS and EDF-FF, while the gap between BL-CBS and G-EDF is remarkable. Also, the ratio between BL-CBS and EDF-BF and BL-CBS and EDF-FF is very close because both fill fewer cores



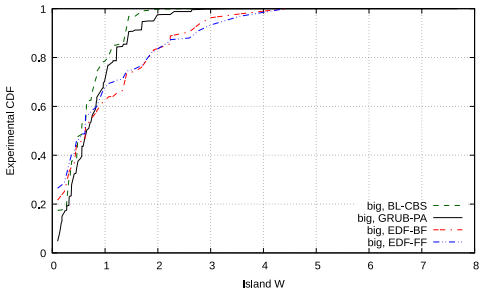
(a) LITTLE island with a total utilization of 2.



(b) big island with a total utilization of 2.



(c) LITTLE island with a total utilization of 4.



(d) big island with a total utilization of 4.

Fig. 5. Comparison of power consumption over time for different utilizations (plots refer to just 1 run out of the 10 performed for each configuration).

firstly, before filling up the others, so their frequency and energy consumption profiles are similar.

Fig. 4 reports time-statistics on the frequency that BL-CBS, EDF-BF, EDF-FF and GRUB-PA (different curves) have set throughout the performed simulations (on the Y axis), on the big (left figure) and LITTLE islands (right figure), for each total nominal utilization (on the X axis). Each reported point and its associated vertical bar represents the average frequency obtained for the 10 runs and its corresponding standard deviation for each algorithm and island. As evident, with BL-CBS, the LITTLE island has higher average frequency than the corresponding big one for each total utilization because, in general, our technique is aware of the difference in power consumption between the two islands, while GRUB-PA is unaware of the said difference and does not distinguish explicitly among big and LITTLE cores. Also, BL-CBS performs an opportunistic pull operation whenever the active utilization of a terminating job on a LITTLE core expires and would leave the core idle, favouring the usage of the LITTLE island (see Algorithm 2). GRUB-PA has the same trend, where the LITTLE island has higher average frequency than the corresponding big one for each total utilization. Moreover, for each island and total utilization, BL-CBS keeps the average frequency lower than GRUB-PA. In general, BL-CBS uses lower frequencies than both EDF-BF and EDF-FF. In fact, BL-CBS tends to spread the jobs on the cores of each island and thus the frequencies are lower, while EDF-BF and EDF-FF both prefer to use cores that are already loaded, which increases the frequencies of the islands.

Fig. 5 reports a visual comparison between the instantaneous power consumption (on the Y axis) obtained throughout one simulation run when using the GRUB-PA scheduler, EDF-BF, EDF-FF, and the one resulting from the application of the technique proposed in this paper, highlighting the power consumption of the LITTLE vs big islands (left vs right subplots), for two different total utilizations (top vs bottom subplots). Figs. 5(b) and 5(d) take into account only the big island and highlight how our algorithm saves power throughout the whole simulations and keeps the power consumptions always lower than GRUB-PA. In fact, GRUB-PA does not reason on the impact on the energy consumption when placing a task and only picks the core with latest deadline, without even performing an EDF-based admission test and setting the maximum frequency if the cores of an island have utilization higher than the island maximum speed, which instead we consider. As for the LITTLE island in Figs. 5(a) and 5(c), the power consumptions of BL-CBS is generally lower than the one of GRUB-PA. EDF-BF and EDF-FF show higher consumptions over the considered simulations compared to BL-CBS and GRUB-PA, with the exception of the big island for total utilization $U =$

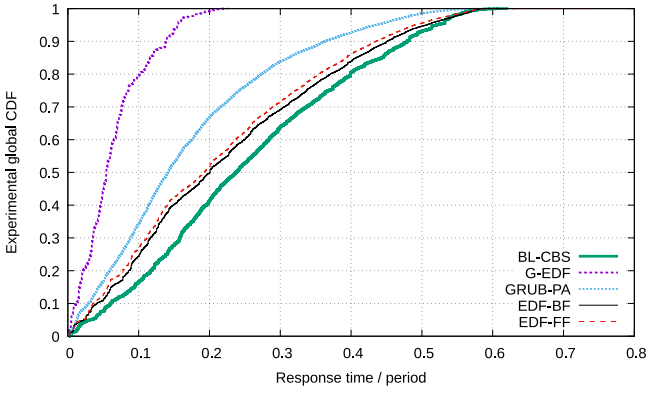


Fig. 6. Experimental global cumulative distribution function (CDF) for 10 experiments with total cores utilization 0.25. Jobs take more to complete with BL-CBS than with G-EDF, EDF-BF, EDF-FF and GRUB-PA, but all of them terminate before their deadlines.

0.25. Notice that the power consumptions of Fig. 5 also take into account the power consumptions when the cores are idle on the X axis for both islands.

The power savings just discussed, obtained with CPUs at lower frequencies on average, impact on the response time experienced by the real-time tasks. Fig. 6 reports the obtained experimental CDF of the response times relative to the periods ($\frac{f_{i,j}-r_{i,j}}{T_i}$ for all instances of all tasks), for BL-CBS, GRUB-PA, EDF-BF, EDF-FF, and G-EDF schedulers (different curves). Jobs response times are higher with BL-CBS than with GRUB-PA, EDF-BF, EDF-FF, and G-EDF, where island frequencies are just kept at the maximum. Since BL-CBS picks the core with the minimum power increase, while GRUB-PA simply picks the one with latest deadline, cores tend to be less loaded and frequencies lower in average than with GRUB-PA, which increases the jobs response times of BL-CBS. Also, BL-CBS tends to spread the jobs on the cores of each island, while EDF-BF and EDF-FF prefer the cores already occupied, which increases the frequencies and lowers the jobs response time. Jobs take less to terminate with GRUB-PA than with EDF-BF and EDF-FF, since GRUB-PA uses higher frequencies for total utilization $U = 2.0$ (which the CDF refers to), especially on the big island as in Fig. 4.

Further experiments have been performed also with randomly-generated tasksets with 16 and 32 tasks and, for each configuration, the same total nominal utilization range. Results show a behaviour similar in terms of energy saving to the experiments presented above and depicted in Fig. 3, i.e., G-EDF has much higher energy consumptions than the other three algorithms, while BL-CBS consumes consistently less than GRUB-PA, EDF-BF, and -EDF-FF for each total nominal utilization.

In our experimentation, all tasksets respecting the theoretical condition of Eq. (12) did not experience any deadline miss during their execution. Furthermore, the simulated tasksets included also many ones that did not respect said condition, which is clearly pessimistic, as its result is heavily affected by the biggest-utilization task in the set. Particularly, among the tasksets with utilization from 0.25 onwards, we had many of them not respecting the theoretical schedulability condition, yet they did not experience any deadline miss throughout the run, as expected due to the pessimism of the theoretical analysis. The only exception has been for the highest total utilization we considered ($U = 5.6$), where we found a 0.53% of jobs having missed their deadline at the end of the simulation (for a taskset not respecting Eq. (12)).

Finally,[5] our implementation of the proposed algorithm, as provided in the RTSIM simulator, has been measured to take on average 0.8 us (and a maximum of 16.43 us, when executing a routine to report that a job cannot be dispatched onto any core because the system is overloaded) when running on an Intel i7-8700 at 4.6 GHz and 16 GB RAM.

9. Conclusions and future work

In this paper, we have presented and simulated big-LITTLE Constant Bandwidth Server (BL-CBS), an adaptive partitioning approach to schedule energy-efficiently real-time tasks. This algorithm has been mainly designed for ARM big.LITTLE, exploiting the underlying hardware architecture features to provide both real-time guarantees and energy saving. It has been shown that BL-CBS allows for guaranteeing timing constraints of real-time task sets that satisfy certain theoretical conditions. Simulations, based on experimental measurements made on the ODROID-XU3 board, show that the algorithm is actually promising, allowing 15% of energy saving in average with respect to the state of the art GRUB-PA.

Concerning possible lines of future work on the topic, we plan to increase the chances of migrations between the two islands and, possibly, to balance the load of the cores in even more situations, which opens many possibilities to decrease frequencies further. Moreover, the pessimism of our admission test can be reduced so that its usability can be expanded to a broader set of scenarios. Moreover, we will consider more complex task sets, with inter-tasks relationships (DAGs) and workload types. We also plan to investigate on how to modify the proposed mechanism in order to properly consider possible deep-idle states of the CPU. Finally, we plan to realize BL-CBS within the current SCHED_DEADLINE codebase in the Linux kernel, to perform further experimentation and validation using real application workloads on Linux/Android.

[5] It would be very interesting to have an implementation of BL-CBS in a real kernel, such as in the SCHED_DEADLINE scheduler within the Linux kernel.

CRediT authorship contribution statement

Agostino Mascitti: Software, Writing - original draft, Data curation, Investigation. **Tommaso Cucinotta:** Supervision, Conceptualization, Methodology, Writing - original draft. **Mauro Marinoni:** Supervision, Conceptualization, Methodology. **Luca Abeni:** Supervision, Conceptualization, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abeni, L., Buttazzo, G., 1998. Integrating multimedia applications in hard real-time systems. In: Proc. 19th IEEE Real-Time Systems Symposium. pp. 4–13.

Abeni, L., Cucinotta, T., 2020. Adaptive partitioning of real-time tasks on multiple processors. In: Proc. 35th Annual ACM Symposium on Applied Computing. SAC'20, ACM, New York, NY, USA, ISBN: 9781450368667, pp. 572–579.

Andersson, B., Tovar, E., 2006. Multiprocessor scheduling with few preemptions. In: 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. (ISSN 2325-1271) pp. 322–334.

ARM, 2019. ARM Technologies: Dynamiq. https://www.arm.com/why-arm/technologies/dynamiq. (Accessed November 4 2019).

Aydin, H., Melhem, R., Mossé, D., Mejía-Alvarez, P., 2004. Power-aware scheduling for periodic real-time tasks. IEEE Trans. Comput. 53 (5), 584–600.

Balsini, A., Cucinotta, T., Abeni, L., Fernandes, J., Burk, P., Bellasi, P., Rasmussen, M., 2019. Energy-efficient low-latency audio on android. J. Syst. Softw. (ISSN: 0164-1212) 152, 182–195.

Balsini, A., Pannocchi, L., Cucinotta, T., 2016. Modeling and simulation of power consumption and execution times for real-time tasks on embedded heterogeneous architectures. In: Proc. International Workshop on Embedded Operating Systems. Torino, Italy.

Bambagini, M., Bertogna, M., Marinoni, M., Buttazzo, G., 2013. An energy-aware algorithm exploiting limited preemptive scheduling under fixed priorities. In: 2013 8th IEEE International Symposium on Industrial Embedded Systems. pp. 3–12.

Bambagini, M., Marinoni, M., Aydin, H., Buttazzo, G., 2016. Energy-aware scheduling for real-time systems: A survey. ACM Trans. Embed. Comput. Syst. 15 (1), 7.

Baruah, S., Carpenter, J., 2003. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. In: Proc. 15th Euromicro Conference on Real-Time Systems. pp. 195–202.

Burns, A., Davis, R.I., Wang, P., Zhang, F., 2012. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. Real-Time Syst. 48 (1), 3–33.

Casini, D., Biondi, A., Buttazzo, G., 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. Dubrovnik, Croatia.

Cheramy, M., Hladik, P., Deplanche, A., Dube, S., 2014. Simulation of real-time scheduling with various execution time models. In: Proc. 9th IEEE International Symposium on Industrial Embedded Systems. pp. 1–4.

Chwa, H.S., Seo, J., Yoo, H., Lee, J., Shin, I., 2015. Energy and feasibility optimal global scheduling framework on big. LITTLE platforms. In: Proc. Real-Time Scheduling Open Problems Seminar. Lund, Sweden, pp. 1–11.

Colin, A., Kandhalu, A., Rajkumar, R., 2014. Energy-efficient allocation of real-time applications onto heterogeneous processors. In: Proc. 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 1–10.

Emberson, P., Stafford, R., Davis, R.I., 2010. Techniques for the synthesis of multiprocessor tasksets. In: Proc. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems. Brussels, Belgium, pp. 6–11.

Funk, S., 2004. EDF Scheduling on Heterogeneous Multiprocessors (Ph.D. thesis). University of North Carolina at Chapel Hill.

Guo, Z., Bhuiyan, A., Liu, D., Khan, A., Saifullah, A., Guan, N., 2019. Energy-Efficient real-time scheduling of DAGs on clustered multi-core platforms. In: IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 156–168.

Guo, Z., Bhuiyan, A., Saifullah, A., Guan, N., Xiong, H., 2017. Energy-efficient multi-core scheduling for real-time DAG tasks. In: 29th Euromicro Conference on Real-Time Systems. Dubrovnik, Croatia, pp. 156–168.

Imes, C., Hoffmann, H., 2015. Minimizing energy under performance constraints on embedded platforms: Resource allocation heuristics for homogeneous and single-ISA heterogeneous multi-cores. ACM SIGBED Rev. 11 (4), 49–54.

Li, T., Zhang, T., Yu, G., Song, J., Fan, J., 2019. Minimizing temperature and energy of real-time applications with precedence constraints on heterogeneous MPSoC systems. J. Syst. Archit. 98, 79–91.

Liu, D., Spasic, J., Chen, G., Stefanov, T., 2015. Energy-efficient mapping of real-time streaming applications on cluster heterogeneous MPSoCs. In: 13th IEEE Symposium on Embedded Systems for Real-Time Multimedia. ESTIMedia. pp. 1–10.

Liu, D., Spasic, J., Wang, P., Stefanov, T., 2016. Energy-efficient scheduling of real-time tasks on heterogeneous multicores using task splitting. In: IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 149–158.

Mascitti, A., Cucinotta, T., Abeni, L., 2020. Heuristic partitioning of real-time tasks on multi-processors. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing. ISORC. pp. 36–42. http://dx.doi.org/10.1109/ISORC49007.2020.00015.

Mascitti, A., Cucinotta, T., Marinoni, M., 2020. An adaptive, utilization-based approach to schedule real-time tasks for ARM big.LITTLE architectures. In: Proc. International Workshop on Embedded Operating Systems, vol. 17. ACM, New York, NY, USA, pp. 18–23,

Moulik, S., Devaraj, R., Sarkar, A., 2019. HEALERS: A heterogeneous energy-aware low-overhead real-time scheduler. IET Comput. Digit. Tech. 13 (6), 470–480.

Nogues, E., Pelcat, M., Menard, D., Mercat, A., 2016. Energy efficient scheduling of real time signal processing applications through combined DVFS and DPM. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, pp. 622–626.

Palopoli, L., Lipari, G., Abeni, L., Di Natale, M., Ancilotti, P., Conticelli, F., 2001. A tool for simulation and fast prototyping of embedded control systems. In: Proc. 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems. Snow Bird, Utah, USA, pp. 73–81.

Palopoli, L., Lipari, G., Lamastra, G., Abeni, L., Bolognini, G., Ancilotti, P., 2002. An object-oriented tool for simulating distributed real-time control systems. Softw. - Pract. Exp. 32 (9), 907–932.

Perret, Q., 2018. Energy aware scheduling. https://lwn.net/Articles/760647.

Pillai, A.S., Isha, T.B., 2013. ERTSim: An embedded real-time task simulator for scheduling. In: 2013 IEEE International Conference on Computational Intelligence and Computing Research. Chennai, India, pp. 1–4.

Pillai, P., Shin, K.G., 2001. Real-Time dynamic voltage scaling for low-power embedded operating systems. In: Proc. 18th ACM Symposium on Operating Systems Principles. vol. 35. Banff, Canada, pp. 89–102.

Qin, Y., Zeng, G., Kurachi, R., Li, Y., Matsubara, Y., Takada, H., 2019a. Energy-efficient intra-task DVFS scheduling using linear programming formulation. IEEE Access 7, 30536–30547.

Qin, Y., Zeng, G., Kurachi, R., Matsubara, Y., Takada, H., 2019b. Execution-variance-aware task allocation for energy minimization on the big. LITTLE architecture. Sustain. Comput. Inform. Syst. 22, 155–166.

Saewong, S., Rajkumar, R., 2003. Practical voltage-scaling for fixed-priority RT-Systems. In: Proc. 9th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 106–114.

Scordino, C., Abeni, L., Lelli, J., 2018. Energy-aware real-time scheduling in the Linux Kernel. In: Proc. 33rd Annual ACM Symposium on Applied Computing. Pau, France, pp. 601–608.

Scordino, C., Abeni, L., Lelli, J., 2019. Real-time and energy efficiency in Linux: Theory and practice. ACM SIGAPP Appl. Comput. Rev. (ISSN: 1559-6915) 18 (4), 18–30.

Scordino, C., Lipari, G., 2004. Using resource reservation techniques for power-aware scheduling. In: Proc. 4th ACM International Conference on Embedded Software. EMSOFT '04, ACM, New York, NY, USA, ISBN: 1581138601, pp. 16–25.

Scordino, C., Lipari, G., 2006. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. IEEE Trans. Comput. 55 (12), 1509–1522.

Thakare, G.S., Deshmukh, P.R., 2017. EERTSS: An energy efficient real-time task scheduling simulator. In: 2017 International Conference on Computing, Communication, Control and Automation. pp. 1–4.

Thammawichai, M., Kerrigan, E.C., 2018. Energy-efficient real-time scheduling for two-type heterogeneous multiprocessors. Real-Time Syst. 54 (1), 132–165.

Zahaf, H., Lipari, G., Bertogna, M., Boulet, P., 2019. The parallel multi-mode digraph task model for energy-aware real-time heterogeneous multi-core systems. IEEE Trans. Comput. 68 (10), 1511–1524.

Zhu, Y., Mueller, F., 2004. Feedback EDF scheduling exploiting dynamic voltage scaling. In: 10th IEEE Real-Time and Embedded Technology and Applications Symposium. Toronto, Canada, pp. 84–93.

Zhu, Y., Mueller, F., 2007. Exploiting synchronous and asynchronous DVS for feedback EDF scheduling on an embedded platform. ACM Trans. Embed. Comput. Syst. (ISSN: 1539-9087) 7 (1), 3:1–3:26.