Multi-Criteria Optimization of Real-Time DAGs on Heterogeneous Platforms under P-EDF

TOMMASO CUCINOTTA, Scuola Superiore Sant'Anna, Italy
ALEXANDRE AMORY, Scuola Superiore Sant'Anna, Italy
GABRIELE ARA, Scuola Superiore Sant'Anna, Italy
FRANCESCO PALADINO, Scuola Superiore Sant'Anna, Italy
MARCO DI NATALE, Scuola Superiore Sant'Anna, Italy

This paper tackles the problem of optimal placement of complex real-time embedded applications on heterogeneous platforms. Applications are composed of directed acyclic graphs of tasks, with each DAG having a minimum inter-arrival period for its activation requests, and an end-to-end deadline within which all of the computations need to terminate since each activation. The platforms of interest are heterogeneous power-aware multi-core platforms with DVFS capabilities, including big.LITTLE Arm architectures, and platforms with GPU or FPGA hardware accelerators with Dynamic Partial Reconfiguration capabilities. Tasks can be deployed on CPUs using partitioned EDF-based scheduling. Additionally, some of the tasks may have an alternate implementation available for one of the accelerators on the target platform, which are assumed to serve requests in non-preemptive FIFO order. The system can be optimized by: minimizing power consumption, respecting precise timing constraints; maximizing the applications' slack, respecting given power consumption constraints; or even a combination of these, in a multi-objective formulation.

We propose an off-line optimization of the mentioned problem based on mixed-integer quadratic constraint programming (MIQCP). The optimization provides the DVFS configuration of all the CPUs (or accelerators) capable of frequency switching and the placement to be followed by each task in the DAGs, including the software-vs-hardware implementation choice for tasks that can be hardware-accelerated. For relatively big problems, we developed heuristic solvers capable of providing suboptimal solutions in a significantly reduced time compared to the MIQCP strategy, thus widening the applicability of the proposed framework.

We validate the approach by running a set of randomly generated DAGs on Linux under SCHED_DEADLINE, deployed onto two real boards, one with Arm big.LITTLE architecture, the other with FPGA acceleration, verifying that the experimental runs meet the theoretical expectations in terms of timing and power optimization goals.

CCS Concepts: • **Computer systems organization** → **Embedded software**; **Real-time systems**; • **Software and its engineering** → **Real-time systems software**; *Real-time schedulability*; *Operating systems*; **Power management**; **Scheduling**; • **Hardware** → **Power estimation and optimization**; **Hardware accelerators**; • **Theory of computation** → **Mixed discrete-continuous optimization**.

Additional Key Words and Phrases: end-to-end timing requirements, energy efficiency, DVFS, software-to-hardware mapping, mixed-integer quadratic constraint programming, heuristic optimization, Linux kernel, SCHED_DEADLINE

Authors' addresses: Tommaso Cucinotta, Scuola Superiore Sant'Anna, Pisa, Italy, tommaso.cucinotta@santannapisa.it; Alexandre Amory, Scuola Superiore Sant'Anna, Pisa, Italy, alexandre.amory@santannapisa.it; Gabriele Ara, Scuola Superiore Sant'Anna, Pisa, Italy, gabriele.ara@santannapisa.it; Francesco Paladino, Scuola Superiore Sant'Anna, Pisa, Italy, francesco.paladino@santannapisa.it; Marco Di Natale, Scuola Superiore Sant'Anna, Pisa, Italy, marco.dinatale@santannapisa.it.

ACM Reference Format:
Tommaso Cucinotta, Alexandre Amory, Gabriele Ara, Francesco Paladino, and Marco Di Natale. 2023. Multi-Criteria Optimization of Real-Time DAGs on Heterogeneous Platforms under P-EDF. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2023), 35 pages. https://doi.org/10.1145/3592609

1 INTRODUCTION

Modern embedded real-time systems are becoming increasingly complex and demanding, not only in computing power, but also in terms of energy efficiency. Indeed, it is commonplace to find computing systems embedded into evolved real-time and mobile computing scenarios. These include soft real-time scenarios like multimedia processing and virtual and augmented reality, Internet-of-Things (IoT) and smart-cities, as well as hard real-time scenarios like Industry 4.0 and connected factories, modern automotive, railway and aero-space industrial use-cases with self-driving vehicles, among others. In these domains, embedded systems not only have to be equipped with increasingly complex networking capabilities, to be able to connect to Cloud/Edge devices enriching their sensing, control and actuation functionality. They also need to possess higher and higher degrees of complexity, intelligence and autonomy [54, 65], to the point that they need to be developed according to parallel programming paradigms typical of the high-performance computing domain, in order to be able to meet their stringent timing constraints. This is leading embedded software development towards a new “embedded HPC” era [3], where these applications need to exploit the parallel computing capabilities of multi-core architectures, and at the same time leverage on the performance acceleration and power savings coming from highly heterogeneous hardware platforms, including GPU and FPGA hardware acceleration.

In the typical target platforms, many of the processing units may have Dynamic Voltage and Frequency Scaling (DVFS) capabilities and may be configured at different frequencies, either statically or dynamically at run-time. It is common to see one or more islands of multiple cores controlled within the same power/clock domain, but also GPUs with frequency switching capabilities.

When designing applications on these systems, it is becoming increasingly difficult to understand how to leverage optimally the available platform configuration, and particularly the power saving tunables, and how to deploy exactly real-time components, in a way that does not compromise the timing and power consumption requirements of the applications. This is particularly true with complex applications, often made of a number of components interacting in a directed-acyclic-graph (DAG) topology. In such a scenario, end-to-end response times have to be guaranteed, from the time a whole DAG is activated, to the time its final outputs are ready and provided to some external system, where the whole end-to-end DAG activation is expected to repeat with a minimum inter-arrival period depending on the physical characteristics of the environment the system is embedded within.

Additional complexity comes also from the fact that different workloads exhibit different behaviors in terms of performance and power consumption, not only when scheduled on accelerators vs CPUs, but also when scheduled on different types of CPUs on energy-efficient architectures [9, 38]. For example, Arm big.LITTLE[1] and Intel Alder Lake[2] are CPU architectures that exhibit cores optimized for performance and cores optimized for energy-efficiency, with full compatibility at the level of the Instruction Set Architecture (ISA). Therefore, processes can be seamlessly deployed on either core type, and migrated dynamically among them, to optimize performance or energy-efficiency of the computations at run-time.

[1]More information at: https://www.arm.com/technologies/big-little.
[2]More information at: https://www.intel.com/content/www/us/en/products/platforms/details/alder-lake-s.html.

In addition to the classical questions posed by real-time systems designers, such as what configuration of the system may guarantee respecting all end-to-end deadlines, there may be more nasty questions to tackle. For example, it may become cumbersome to figure out whether it is more convenient, from an energy efficiency perspective, to deploy one or more functional blocks in software, as a task running on some CPU, or in hardware, as a kernel executed on a GPU or FPGA accelerator, if available on the platform, notably in presence of multiple such acceleration options that may be fulfilled on a limited set of accelerators (often just one GPU device or FPGA fabric, sometimes with partitioning capabilities). It is also difficult to find a configuration of the system that minimizes the expected power consumption, for example to maximize the life-time for battery-operated devices in mobile scenarios, or to limit heat emissions and the need for ventilation, in limited form factor scenarios or for user-comfort reasons.

Paper Contributions. In this paper, we address two major variants of the above introduced problem: one in which we want to minimize the average power consumption expected on the platform, subject to a set of end-to-end deadline constraints; the other in which we have a constraint on the maximum power consumption budget we can sustain at run-time, but we need to deploy applications with the maximum slack to their end-to-end deadlines, so to maximize robustness of the configuration with respect to unforeseen effects, such as sporadic violations of the estimated execution time bounds. We assume that real-time tasks are scheduled on CPUs using preemptive EDF-based scheduling, while accelerators serve requests non-preemptively and in strict FIFO order. This allows us to propose a formulation of the problem in the form of a mixed-integer quadratic constraint programming (MIQCP) optimization problem, that can be solved using a standard (non-linear) solver.

Differently from many approaches in the real-time scheduling literature where local scheduling deadlines for individual tasks are assigned using an end-to-end deadline splitting strategy, our approach leaves local deadlines as completely free variables that are automatically found by the solver, allowing for potentially better solutions.

The optimum approach is dealt with using the Gurobi commercial solver. This, albeit capable of finding the optimum solution, might exhibit computations prohibitively expensive for large problems. So, we also propose a few heuristic solvers that, despite being unable to find optimum solutions, can be used to find good-enough solutions with more practical optimization times.

The approach is validated by real experimentation on two embedded boards typical of the above-mentioned introduced computing scenarios: an ODROID-XU4 board with big.LITTLE architecture, and a Xilinx UltraScale+ ZCU102 board with FPGA acceleration, where its Dynamic Partial Reconfiguration (DPR) capabilities have been used through the open-source FRED middleware[3].

We focus on soft real-time scenarios where the use of SCHED_DEADLINE under Linux is acceptable. In our experimental setup, we had to apply a small patch to SCHED_DEADLINE in order to deal properly with the timing of FPGA acceleration, as discussed in Section 3. Then, we performed extensive experiments by generating randomly nearly 2.7 thousand DAG sets, comparing the ability of the heuristic solvers to optimize the scenarios with respect to the optimum one, and running a few hundreds of the scenarios on the real boards under Linux. We obtained promising experimental results, showing quite a good match between the timing behavior expected by the optimized scenarios (no deadline misses), and the power consumption measurements performed on the boards.

Moreover, we used the WATERS19 challenge [67] as an industrial use-case to validate further the applicability of our approach.

[3]More information is available at: https://github.com/fred-framework.

Paper Organization. The paper is organised as follows. After reviewing some of the key research works dealing with scheduling and allocation of real-time DAGs in Section 2, we formalize our problem of energy-aware task allocation for heterogeneous platforms in Section 3. Then, our proposed solution strategies are presented in Section 4, including the optimum one based on MIQCP optimization, the classical heuristic based on mixed-integer linear programming (MILP) optimization with pre-fixed end-to-end deadlines, and two heuristics. The proposed approach is validated experimentally in Section 5, where the different proposed solvers are compared in terms of optimization time and achieved energy-efficiency levels. Furthermore, experimental results are presented from the execution of a synthetic benchmark running a set of randomly generated DAGs sets, optimized according to the different strategies, in addition to our experience in applying the proposed methodology to the WATERS19 industrial challenge. Finally, conclusions are drawn in Section 6, followed by a sketch of possible future research directions.

2 RELATED WORK

The research literature on modeling and analysis of real-time systems is vast. The seminal work by Liu and Layland [44] established the foundations for performing a quick and easy schedulability analysis of independent real-time task sets under rate-monotonic or EDF-based scheduling on uniprocessors, using simple utilization-based tests. However, the excess of pessimism intrinsically associated to the well-known concept of worst-case execution time, led to the development of richer task models, trying to capture the iterative or conditional execution of the various code segments within each task. This was often done using a directed acyclic graph (DAG) of computations, as for example in the recurring real-time task model [12], generalizing previous iterative models like the multiframe [48] or others [1].

Since then, the platforms under consideration by real-time application designers evolved [64] copying the technological trends, with increasing attention to systems with multiple processing units in multi-processor or multi-core platforms, or even across a network of distributed nodes [7, 28, 41], possibly with the addition of hardware acceleration, energy efficiency features like DVFS, and complex or specialized memory hierarchies [37]. With reference to multi-core/processor platforms, a great attention was dedicated to investigating the pros and cons of global vs partitioned, or even clustered, scheduling disciplines, with a plethora of both theoretical [31, 34] and experimental [13, 43] works. Additionally, researchers investigated on energy-efficient real-time systems [4, 10, 68], a topic of increasing importance due to the attention being posed on mobile sensing and control systems, mobile robots, and others.

Correspondingly, the formalization techniques evolved to model and analyze increasingly complex real-time applications, composed of multiple tasks to be deployed over the available processing units, so to make a higher and higher use of the parallelism degree available underneath. Then, rich computational structures have been widely used to model complex parallel real-time applications with dependencies among tasks/components, similar to what done in parallel and distributed systems. These include: linear processing chains [17, 29, 63]; applications using thread pools [18] for acceleration of parts of their computations on multi-core platform; synchronous data-flow graphs to be deployed on massively parallel processing platforms [58], possibly accounting for context-switch and preemption overheads [39]; DAG-alike computations [25]. For example, schedulability of parallel tasks has been investigated in [42], adopting the Fork-Join model typical of various constructs of OpenMP [15].

The latter has grown to become an increasingly popular development and run-time environment to build parallel code, to the point that several linear-algebra libraries widely used for real-time control, image and signal processing are based on OpenMP, that provides them with the seamless ability to accelerate computations exploiting multi-core as well as GP-GPU processing. Recently, OpenMP developers make more and more use of `parallel region` and `task` constructs, essentially declaring the computational DAG-alike model and dependencies within the code, leaving to the OpenMP run-time the details on how to orchestrate its execution.

As a consequence, we are seeing a raising number of research works dealing with modeling and analyzing parallel DAGs of computations [55]. Sometimes, the old idea of conditional DAGs is mixed with the more recent one of parallel DAGs, as done in [23, 35], showing how to perform a transformation of a conditional DAG into an equivalent single synchronous DAG of servers, that can be analyzed using traditional techniques. In [69], the problem has been proposed of scheduling conditional DAGs [11] on preemptible accelerators under partitioned preemptive EDF with non-negligible preemption costs, proposing a MILP approach to perform deadline and offset assignment for tasks assigned to a single core, but it has been embedded within a heuristic workflow to optimize the whole system. In the important automotive application domain, the AUTOSAR standard allows for modeling complex real-time applications in terms of DAGs of Runnables, packed into real-time tasks, where a problem that was extensively investigated is the one of optimum priority assignment and mapping among Runnables and OS tasks [24]. A similar methodology is AMALTHEA [20, 57], where DAG-alike dependencies among Runnables can be specified in the form of data dependencies through labels. Analysis of DAG-alike constructs gained recently additional traction in the real-time community in the variant needed for dealing with the ROS2 computational model [19].

In DAG-alike models of computation, a problem that received a fair amount of attention is the one of designing deadline splitting strategies, so to be able to divide an end-to-end application deadline into a set of local deadlines to be used for individual real-time tasks [22, 59]. This problem was tackled also using MILP-based approaches [53].

Additionally, another dimension that has been growing in the real-time systems literature is the one of heterogeneous processing platforms, often being used to investigate on particularly energy-efficient architectures, that may attain high levels of performance, at reduced power consumptions. For example, intelligent heuristics for tuning the DVFS capabilities may result in significant reductions of the energy needed to deploy real-time applications [49, 50, 66], where some authors proposed ILP-based frameworks for optimum configuration of DVFS settings [60] under precise assumptions. Others proposed greedy heuristics [47] to perform dynamic scheduling and placement of real-time tasks based on the instantaneous conditions of DVFS-capable systems, and specifically big.LITTLE platforms. On the other hand, the increasingly popular use of GP-GPUs and FPGA devices brings usually benefits in terms of achieved FLOPS/Watt [40, 72], particularly interesting for machine-learning and DNN scenarios [72].

In this area, some authors investigated on richer DAG-alike task models running on heterogeneous platforms, aimed at capturing the need for tasks to suspend and wait for the delivery of results from accelerators, throughout their execution. For example, this is the case of the event-driven, delay-induced (EDD) task model introduced in [6], where an analysis technique for this kind of models was introduced, when preemptive priority-based scheduling is used for software tasks, and validated through experimental results.

Some authors proposed middleware solutions [61, 62] to deal with the complexity of configuring schedulers for real-time applications on heterogeneous platforms, exposing higher-level or more abstract and easy-to-use interfaces to application developers.

Interesting approaches based on MIQCP-based optimization have been proposed in [70, 71], where the end-to-end execution time (makespan) has been minimized, on platforms with multi-core CPUs, GPU and FPGA devices, where the non-linearity of the formulation was due to the presence of communication latency terms. The above mentioned research works cover a wide range of topics that are at the core of the concepts exposed in the present paper, where we tackle the problem of: optimum configuration of DVFS-capable heterogeneous platforms, with a focus on the increasingly popular Arm big.LITTLE architectures and hardware acceleration via GP-GPU or FPGA fabrics; task

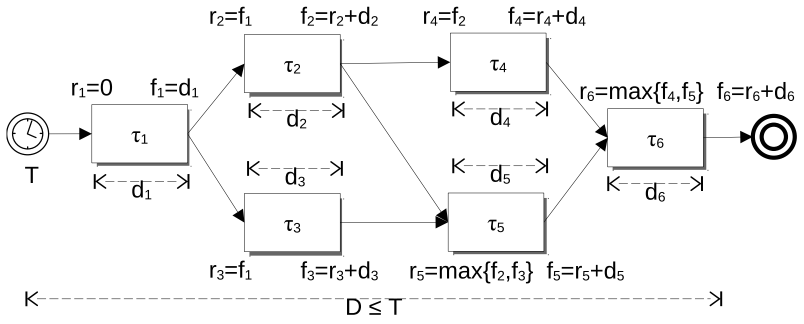

Fig. 1. Visualization of a sample DAG application with 6 tasks, highlighting the intermediate deadlines $\{d_i\}$ and their relationship with the worst-case release and finishing times $\{r_i\}$ and $\{f_i\}$ relative to the DAG activation time.

allocation and placement and acceleration/variant selection; optimum scheduling parameters selection under EDF-based scheduling, using SCHED_DEADLINE on Linux, with a necessary modification to deal with FPGA acceleration. This problem is formalized as a mixed-integer quadratic constraint programming (MIQCP) optimization problem, then solved by a standard solver (we used Gurobi). Alternative faster heuristic solvers are also proposed to deal with large problems. The approach by extensive experimentation with two real heterogeneous boards, one based on Arm big.LITTLE, the other one with FPGA acceleration.

To the best of our knowledge, most of the prior literature discussed above has still a great focus on fixed-priority real-time scheduling, as popular in the automotive domain for example, or it is often based on standard deadline splitting techniques, to apply in the end a purely MILP-based approach, or focuses on heterogeneous platforms considering solely hardware acceleration using GP-GPUs or FPGAs, but it neglects the DVFS capabilities of the CPU, or does not deal with Arm big.LITTLE or similarly heterogeneous architectures, that are anyway so popular in the mobile computing domain.

3 SYSTEM MODEL

This section presents the model upon which the approach described in the paper was developed. The set of applications under consideration is composed of a set of n_T tasks $\Gamma = \{\tau_i\}_{i=1}^{n_T}$. Each task τ_i is characterized by:

- a reference estimated execution time bound (EETB) C_i, constituting a reasonable upper-bound to the execution time when the task is deployed onto a reference processing unit running at a reference frequency ϕ_{ref}, typically the maximum frequency among the set of the available ones for a processing unit; this will also be called *unscaled* EETB in what follows;
- a non-scalable part of its EETB C_i^{ns}, relevant to account for cache misses, which does not change with the frequency and computing capacity of the execution unit;
- a *release time* of the k-th instance $r_{i,k}$ and its *finishing time* $f_{i,k}$.

The considered tasks are partitioned into n_G independent applications, constituting the set $G \triangleq \left\{G_j\right\}_{j=1}^{n_G}$. The interactions and dependencies among tasks in each application are modelled as a real-time directed acyclic graph (DAG) of computations, as exemplified in Figure 1. More precisely, each DAG is modelled as a tuple $G_j = (\Gamma_j, \mathcal{E}_j, T_j, D_j)$, where:

- Γ_j is a partition of the whole taskset Γ, i.e., $\cup_{j=1}^{n_G}\Gamma_j = \Gamma$, and $\forall i \neq j, \Gamma_i \cap \Gamma_j = \emptyset$;
- $\mathcal{E}_j \subset \Gamma_j \times \Gamma_j$ is a set of pairs of tasks within Γ_j, with $(\tau_a, \tau_b) \in \mathcal{E}_j$ modeling a directed edge from τ_a to τ_b; each $\mathcal{E}_j$ is assumed to be a fully connected and cycle-free topology, with exactly one *starting task* τ_{s_j} $(\forall \tau_i \in \Gamma_j,\ (i, s_j) \notin \mathcal{E}_j)$ and one *ending task* τ_{e_j} $(\forall \tau_i \in \Gamma_j,\ (e_j, i) \notin \mathcal{E}_j)$;
- T_j is the minimum DAG activation period, defining the minimum time between two subsequent activations of the starting task $\tau_{s_j} \in \Gamma_j$, i.e.:
$$\forall k,\ r_{s_j,k+1} \geq r_{s_j,k} + T_j; \tag{1}$$
- D_j is the end-to-end relative deadline between the k^{th} activation of the DAG and the corresponding finishing time of its ending task τ_{e_j}; more precisely, the end-to-end timing constraint for each DAG G_j is stated as:
$$\forall k,\ f_{e_j,k} \leq r_{s_j,k} + D_j. \tag{2}$$

Sample application scenarios like the one just introduced, with a periodic activation of DAG-alike data-flow processing topologies, are commonly seen in real-time audio/video multimedia processing [8, 26, 27], as well as real-time control and data processing pipelines [51]. In the following, for each task τ_i we will also use the symbols r_i and f_i to refer to its latest possible activation and finishing times, respectively, relative to the activation of the DAG. Also, we assume that:

$$D_j \leq T_j,\ \forall G_j \in G. \tag{3}$$

This implies that, in any configuration of the system in which end-to-end deadlines are respected, we have the guarantee that all tasks of an activation of the DAG will have completed, before the next activation of the same DAG occurs. In other words, in each DAG activation, each task cannot overlap with any activation of the same or other tasks from the same DAG pertaining to any of the previous activations of the DAG. This will be important to formalize the schedulability condition for the system. Moreover, for ease of exposition, in the following we implicitly assume that $D_j = T_j,\ \forall G_j \in G$.

Concerning the hardware part, the modelled platform is provided with a set of n_P processing units (PUs) $P = \{\psi_p\}_{p=1}^{n_P}$. PUs may be *CPU cores* or *accelerators*, like FPGA slots or GPU devices. This set is partitioned into n_S disjoint subsets $\{I_s\}_{s=1,\ldots,n_S}$ called *islands*. Each island is capable of switching among a discrete set Φ_s of operating performance points (OPPs), corresponding to different frequencies $\phi_{s,m}$. Also, an island is associated with a processing-unit *capacity* ξ_s defining its maximum speed of computation (i.e., an island with a 0.5 capacity exhibits a $2x$ slow-down in tasks execution compared to an island with capacity 1.0). Finally, each island I_s is characterized by a power consumption $\mathcal{P}_{s,m}^{B}$ when one of its PUs is busy processing at frequency $\phi_{s,m}$, and a power consumption $\mathcal{P}_{s,m}^{I}$ when idle (this still depends on the configured frequency m).

When a task τ_i is deployed onto any processing unit within an island s with capacity ξ_s, configured with an OPP corresponding to a frequency $\phi_{s,m}$, its scaled EETB $C_{i,s,m}$ becomes:

$$C_{i,s,m} = C_i^{ns} + \frac{C_i - C_i^{ns}}{\xi_s \phi_{s,m}} \phi_{s,ref} \tag{4}$$

However, the approach we propose below is totally agnostic in terms of how the $C_{i,s,m}$ values are actually obtained, for each task τ_i. In a certain system design workflow, these can be obtained by profiling the tasks at a reference frequency and then Equation (4) can be applied, or individual $C_{i,s,m}$ values can be profiled for each island and power mode (or by interpolation from data referring to a subset of the available frequencies), or one can apply worst-case execution time (WCET) estimation techniques [45, 46]. Note that, if for a given system we were able to compute WCETs for the

Table 1. Summary of the major notational elements adopted throughout the paper.

Symbol	Description
$\Gamma = \{\tau_1, \ldots, \tau_{n_T}\}$	set of all n_T tasks in the system
$\tau_i \in \Gamma$	i^{th} task
C_i, C_i^{ns}	EETB of τ_i and its non-scalable part
r_i, f_i	release and finishing times of τ_i, relative to the DAG arrival time
$G_j \in G$	j^{th} DAG
$\Gamma_j \subset \Gamma$	set of tasks belonging to j^{th} DAG
$\mathcal{E}_j \subset \Gamma_j \times \Gamma_j$	set of directed edges belonging to j^{th} DAG
$j(i)$	DAG the i^{th} task belongs to
T_j	minimum inter-arrival period for j^{th} DAG
D_j	end-to-end deadline for j^{th} DAG
$\tau_{s_j} \in \Gamma_j$	start-task of j^{th} DAG
$\tau_{e_j} \in \Gamma_j$	ending-task of j^{th} DAG
$P = \{\psi_1, \ldots, \psi_{n_P}\}$	set of all n_P processing units
$\psi_p \in P$	p^{th} processing unit
$I_1, \ldots, I_{n_S} \subset P$	partitioning of P into n_S islands
$s(p)$	island the p^{th} PU belongs to
ξ_s	capacity of processing units of the s^{th} island
Φ_s	set of possible power modes for PUs in island I_s
$C_{i,s,m}$	scaled EETB of τ_i if deployed within I_s under power mode m
$\mathcal{P}^I_{s,m}$	power consumption of s^{th} island in power mode $m \in \Phi_s$, with all PUs idle
$\mathcal{P}^B_{s,m}$	power consumption of s^{th} island in power mode $m \in \Phi_s$, when busy computing with a single CPU-hog task on one PU, and the other PUs idle

tasks when deployed at the various frequencies, then these values could be used as EETBs in the theoretical model and accompanying MIQCP optimization technique we introduce in this section. This would result in deterministic hard real-time guarantees for the deployed real-time DAGs. However, in this paper we prefer to focus on soft real-time scenarios, given the use of a general-purpose operating system like Linux, deployed onto general-purpose platforms, and the fact that we perform an empirical estimation of the EETBs in our experimental validation (see Section 5).

Table 1 summarizes the main notational elements used throughout the paper.

4 PROPOSED APPROACH

We consider the problem of energy-efficient deployment of a number of real-time DAGs on a heterogeneous processing platform, with the software/hardware system model just detailed in Section 3.

4.1 Reference deployment architecture

In this paper, we assume a reference software architecture making use of the SCHED_DEADLINE [43] scheduler for Linux, thus we focus on soft real-time application scenarios where the use of Linux is acceptable. These are often characterized by a certain difficulty in carrying out precise EETB estimations, and in managing to have a very accurate model of the activities interfering with applications, due to the complexity of the operating system, its kernel and often the general-purpose platforms underneath. Therefore, desirable properties within the system are the ones of *temporal isolation* among the various independent applications, and *robustness* with respect to possible unmodelled interferences.

The former refers to ensuring that possible violations of the timing behavior (e.g., execution times sporadically bigger than the EETBs known at design and analysis time) within tasks belonging to a DAG would impact only the ability to meet deadlines of that specific application, without any consequence on other applications that are behaving according to their specification. This is guaranteed for example by schedulers based on *resource reservations* [56] just like SCHED_DEADLINE, that is based on the well-known Constant Bandwidth Server (CBS) [2], providing temporal isolation with a preemptive EDF policy. The latter problem refers to unforeseen interferences on real-time applications due to unmodelled activities, or overheads in the system, e.g., interferences by device drivers. This is tackled by maximizing the slack available to applications till their end-to-end deadline. Moreover, we make sure each CPU has a minimum spare capacity, also useful for possible background activities, in addition to the modelled real-time workload.

For what concerns the scheduling strategy of hardware accelerators, instead, we assume that accelerated tasks are managed with a FIFO non-preemptive scheduler. This is reasonable for GPUs on embedded platforms that may accept one kernel to be computed at a time, or FPGA devices with DPR capabilities that, even when partitioned into multiple slots, provide the ability to reprogram each slot dynamically, serving acceleration requests submitted to them in FIFO order. This is the case of the FRED open-source middleware [14] for FPGA acceleration, for example, which is what is used in our experiments shown in Section 5.

The mentioned optimization problem has been tackled recurring to three different strategies:

- an optimal approach based on an MIQCP formulation, where task mapping, intermediate deadlines and power configurations are all variables; this is solved with a commercial MIQCP solver;
- a sub-optimal approach constituting a sort of baseline benchmark in the field, where end-to-end deadline splitting is applied first, to assign intermediate deadlines to the tasks, then an optimum deployment is found using a MILP formulation; this is solved with a commercial MILP solver;
- two heuristic algorithms designed to not suffer of the scalability issues of the above two approaches, and that we will shortly release with an open-source license.

4.2 MIQCP problem formalization

The MIQCP formulation aims to place the tasks of the application DAGs onto the available processing units (either CPU cores or accelerators), to assign them intermediate deadlines $\{d_i\}_{\tau_i \in \Gamma}$ and also to set the operating frequency of each processing unit.

Several variables are needed to setup the MIQCP formulation. The placement of tasks on processing units is expressed with a set $\{x_{i,p}\}$ of boolean variables, where $x_{i,p}$ is equal to 1 if task τ_i is mapped on processing unit ψ_p and 0 otherwise. Similarly, the operating frequency of each island is indicated by a set $\{y_{s,m}\}$ of booleans, where $y_{s,m}$ equals 1 if island s is configured at frequency $\phi_{s,m}$ and 0 otherwise. With the purpose of simplifying the formulation, the additional boolean variable $z_{i,p,m}$ was introduced, whose value is 1 when task τ_i is mapped on processing unit ψ_p running at frequency $\phi_{s,m}$. Essentially, $z_{i,p,m}$ is the result of the logical AND operation between $x_{i,p}$ and $y_{s,m}$. As widely known, these can be encoded as linear constraints.

Optimization objective. The overall objective of the problem is to minimize the energy consumption of the application deployed onto the platform, while guaranteeing to meet the DAG end-to-end deadlines. In alternative scenarios, the system is given an overall average power budget, and the optimization objective is changed to maximizing the robustness of deployed applications in terms of timing requirements. Specifically, the robustness is quantified as the distance of the finishing time of each DAG instance from its deadline, relative to the deadline itself, i.e., the *relative slack*. The two concepts can also be combined, in an overall multi-objective optimization that minimizes the power consumption first, then it maximizes the relative slack.

The placement of the tasks in the $\{x_{i,p}\}$ variables and the choice of the operating frequencies of the islands in the $\{y_{s,m}\}$ ones, affect the average power consumption of the applications deployed on the platform, which the solver attempts to minimize. Precisely, following the power model discussed in [5], based on extensive experimental power consumption data gathered for a set of diverse workloads on various embedded platforms, the objective of the optimization is formulated as:

$$\min \sum_{p \in P} \sum_{m \in \phi_{s(p)}} \left(y_{s(p),m} \mathcal{P}^{I}_{s(p),m} + \Delta\mathcal{P}_{s(p),m} \sum_{\tau_i \in \Gamma} \frac{z_{i,p,m} C_{i,s(p),m}}{T_{j(i)}} \right) \tag{5}$$

$$\Delta\mathcal{P}_{s,m} = \mathcal{P}^{B}_{s,m} - \mathcal{P}^{I}_{s,m} \tag{6}$$

being $s(p)$ the island the PU p belongs to, and $j(i)$ the DAG of task τ_i. In Equation (5), we sum up contributions due to each of the PUs, highlighted in the parenthesis, where the left term of each contribution gives us the idle power to be used, depending on the chosen optimized operating modes $\{y_{s,m}\}$. The right term, instead, gives us the delta $\Delta\mathcal{P}_{s,m}$ to be added to said idle power to obtain the busy power of the PU, but accounted only for a fraction of time equal to the average workload deployed on the PU. This fraction is the sum of the utilization of all tasks chosen to be deployed on the PU, according to the $\{z_{i,p,m}\}$ variables. In said formula, using EETBs leads actually to an estimate of the maximum power consumed on the platform. However, in our optimizer we can also use average execution times instead of EETBs in Equation (5), if we need to refer more precisely to the expected average power consumption in the calculations.

The alternate objective focused on a power-aware maximization of the slack, can be formalized as the maximization of the minimum relative slack over all the the DAGs:

$$\max \min_{G_j \in G} \frac{D_j - f_{e_j}}{D_j} \tag{7}$$

where f_{e_j} is the finishing time of the ending task of the DAG τ_{e_j}, thus representing the finishing time of the DAG itself. In order to search for a power-aware configuration of the system, the formulation can be provided with a power budget B dedicated to all the applications deployed on the platform. To this purpose, Equation (5) can be turned into a constraint, forcing the solver to find a configuration satisfying the assigned power budget:

$$\sum_{p \in P} \sum_{m \in \phi_{s(p)}} \left(y_{s(p),m} \mathcal{P}^{I}_{s(p),m} + \Delta\mathcal{P}_{s(p),m} \sum_{\tau_i \in \Gamma} \frac{z_{i,p,m} C_{i,s(p),m}}{T_{j(i)}} \right) \leq B \tag{8}$$

As mentioned, the two objectives in Equations (5) and (7) can be combined, with the meaning that, among all the solutions that would keep the average power consumption at its minimum level, we would prefer solutions maximizing the relative slack of applications (in the combined version, the power budget constraint would not be needed).

Topology, placement and CPU schedulability constraints. The additional constraints of the formulation originate from the requirements imposed by the topology of the application, the restrictions of mapping the software entities on the hardware elements and the hardware itself. The former ones are related with the dependencies among the tasks described in the DAGs of the application. Specifically, the topology under consideration mandates that every task in the topology is activated only when *all* its direct predecessors finish their computations. This activation rule affects the finishing time f_i of each task τ_i, and the problem formulation bounds their value with a set of constraints. For the starting task of the DAG τ_{s_j}, the finishing time trivially corresponds to its intermediate deadline d_{s_j}, because it is required to terminate within that time value in order to be schedulable, whilst for all the others the following constraint holds:

$$f_i = \max_{\tilde{i} \mid (\tilde{i},i) \in \mathcal{E}_j} \left\{ f_{\tilde{i}} \right\} + d_i \quad \forall i \in \Gamma_j \tag{9}$$

Finally, for the ending task τ_{e_j} of the DAG an additional bound was included in the formulation to guarantee that the overall end-to-end DAG deadline D_j is never exceeded:

$$f_{e_j} \le D_j \tag{10}$$

When placing a task onto a PU, not only we have to ensure that the DAG end-to-end deadlines are always met, but it is also important that for every PU, all the tasks mapped to it are *schedulable*. Basically, each task shall meet its intermediate deadline appropriately assigned by the solver.

In our reference deployment architecture, the scheduler varies depending on the type of PU under analysis, as specified in Section 4.1. On CPU cores, tasks are scheduled with EDF, meaning that the total utilization of all the tasks running on a given PU shall not exceed 1, otherwise the taskset is unschedulable. However, given the topology of the applications under consideration, not all the tasks mapped to a core can be activated and run concurrently. Indeed, the tasks within a DAG are not independent but bound with dependency relationships described by the DAG itself and their activations constrained by Equation (9). Hence, testing the sum of the utilizations of all tasks mapped to a core would result in an overly pessimistic schedulability condition. A tighter schedulability test is built considering the key concept of *(un)reachability*: two tasks τ_a and τ_b of the same DAG are reachable, i.e., $\tau_a \sim \tau_b$, if there exists a path of the DAG connecting them, regardless of its direction. In other words, τ_a and τ_b are *dependent* and will never be activated concurrently. For the schedulability, what matters is the *unreachability*, i.e. the negation of the reachability: if $\tau_a \nsim \tau_b$, then τ_a and τ_b are *independent* and are allowed to be executed concurrently. Considering a DAG j, the set of unreachable task subsets V_j can be defined as:

$$V_j \triangleq \left\{ S \subset \Gamma_j \mid \forall \tau_a, \tau_b \in S,\ \tau_a \nsim \tau_b \right\} \tag{11}$$

Consequently, the set containing the biggest unreachable task subsets W_j is:

$$W_j \triangleq \left\{ S \in V_j \mid \nexists \tilde{S} \in V_j \; with\, S \subsetneq \tilde{S} \right\} \tag{12}$$

Essentially, given a DAG j, the corresponding set W_j contains the biggest subsets of tasks that can run concurrently. For example, for the DAG in Figure 1, we have $W_j = \{\{\tau_1\}, \{\tau_2, \tau_3\}, \{\tau_4, \tau_3\}, \{\tau_4, \tau_5\}, \{\tau_6\}\}$. Since W_j contains all the biggest subsets of tasks in Γ_j that can be active at any given time, the worst-case schedulability scenario for a core needs to consider, for each G_j, only the maximum utilization $u_{j,p}$ among those associated to said subsets:

$$u_{j,p}^{max} = \max_{S \in W_j} \sum_{i \in S} \left(\frac{1}{d_i} \sum_{m \in \Phi_{s(p)}} z_{i,p,m} C_{i,s(p),m} \right) \tag{13}$$

Equation (13) depends on the specific PU p so that to compute the maximum utilization of DAG j on that PU. As a matter of fact, the $z_{i,p,m}$ variable distinguishes the tasks that are currently mapped to p (along with its OPP).

All the DAGs of the application can potentially contribute to the total utilization of p. Therefore, $u_{j,p}^{max}$ is computed for all the DAGs and the values are summed up, thus obtaining the worst-case utilization of the core. Finally, the EDF schedulability test is applied to this overall contribution

$$\sum_{j=1}^{n_G} u_{j,p}^{max} \leq U^{max}, \tag{14}$$

with a U^{max} that theoretically can be set to 1, but it needs to be set at 0.95 or lower for any practical reason. This parameter can be used to leave a minimum of spare unallocated capacity on each CPU in the system. This is useful to counter the effect of possible unforeseen interferences in EETB estimates (useful when used in combination with the SCHED_FLAG_RECLAIM option of SCHED_DEADLINE, enabling reclaiming of unused bandwidth), and/or to allow non real-time activities to have sufficient time to execute on the platform. Note that the constraints in Equation (13) are quadratic because of the variables $\{d_i\}$ at the denominator, leading to a MIQCP formulation.

Accelerators. The tasks assigned to accelerator PUs are scheduled in a FIFO non-preemptive fashion. Hence, when a task is activated, it may experience a *waiting time* before being served. In the worst-case scenario, when a task is activated, all the other tasks mapped to that PU are already queued waiting to be scheduled. Therefore, the task cannot run until all the others terminate. However, similarly to the formerly analyzed schedulability on CPU cores, not all the tasks of the system can run concurrently.

The d_i symbol, for an accelerator task, is not a deadline, but it represents its worst-case traversal delay, inclusive of the execution time needed on the accelerator, plus the worst-case waiting time. This allows us to extend seamlessly the timing constraints formulations in Equations (9) and (10) to the cases of tasks mapped both to CPUs and to accelerators. In order to define an upper bound to the waiting time in the FIFO queue for a task τ_i belonging to a DAG j, two contributions have to be taken into account: the waiting time caused by tasks in Γ_j that are unrelated to τ_i, and the one produced by tasks within all other DAGs in the system. Concerning the former, the set of tasks in Γ_j that are unrelated with τ_i can be defined as:

$$\Lambda_{i,j} \triangleq \left\{ S \in W_j | \tau_i \in S \right\} \tag{15}$$

In other words, $\Lambda_{i,j}$ contains the unrelated task sets of the DAG W_j which include τ_i, thus identifying the tasks that can run in parallel with τ_i. Given a set $S \in \Lambda_{i,j}$, when τ_i is activated, in the worst case all the tasks except τ_i itself are queued for execution on the accelerator p, so τ_i is required to wait for their completion before it can run. Considering the entire set $\Lambda_{i,j}$, the worst waiting time for τ_i is given by the maximum over all the sets of the sum of the execution time of the tasks in the set:

$$w_{i,p}^{own} = \max_{S \in \Lambda_{i,j(i)}} \sum_{k \in S, k \neq i} \sum_{m \in \Phi_{s(p)}} z_{k,p,m} C_{k,s(p),m} \tag{16}$$

The second contribution to the waiting time originates from the tasks belonging to all the other DAGs of the application. Similarly to the former contribution, the worst case waiting time for τ_i can be expressed as:

$$w_{i,p}^{oth} = \sum_{h=1, h \neq j(i)}^{n_G} \max_{S \in W_h} \sum_{k \in S} \sum_{m \in \Phi_{s(p)}} z_{k,p,m} C_{k,s(p),m} \tag{17}$$

Ultimately, the worst-case waiting time for τ_i for being scheduled on accelerator p is the sum of the two formerly defined contributions:

$$w_{i,p} = w_{i,p}^{own} + w_{i,p}^{oth}. \tag{18}$$

Consequently, the lower bound to the processing delay d_i of τ_i within the MIQCP formulation corresponds to:

$$x_{i,p} = 1 \implies d_i \geq \sum_{m \in \Phi_{s(p)}} z_{i,p,m} C_{i,s(p),m} + w_{i,p} \tag{19}$$

with the meaning that the solver shall assign task τ_i a deadline being *at least* greater than its execution time and worst-case waiting time to ensure its schedulability. Equation (19) is written in the form of *indicator constraint*, i.e., a constraint that is only active when its boolean *control variable* ($x_{i,p}$) evaluates to true. This is needed because the lower bound to the deadline of task τ_i shall be considered only when it is mapped to p being an accelerator PU.

The MIQCP formulation enables modelling of accelerators, like FPGA slots with dynamic partial reconfiguration (DPR) capabilities, that exhibit a *reconfiguration delay*, i.e., an amount of time to wait after the end of a task before scheduling a new one. When this delay is of the same order of magnitude of the execution time of the tasks to be scheduled, or even greater, it shall be considered in the mapping problem for realistic results. Given the nature of this delay, it only occurs whenever at least 2 tasks are mapped to that accelerator PU, otherwise there is no need for reconfiguration at runtime. When an accelerator p is assigned a reconfiguration delay δ_p, all the tasks that are allowed to run on it may experience a bigger execution time. To account for this difference, the new variable $h_{i,p,m}$ is introduced.

$$h_{i,p,m} = z_{i,p,m} C_{i,s(p),m} + r_{i,p,m} \delta_p \tag{20}$$

Essentially, $h_{i,p,m}$ represents the execution time of task τ_i when mapped on PU p operating at OPP m. As a matter of fact, the first term of Equation (20) is the nominal execution time of that task given its current mapping. The second term constitutes the reconfiguration delay, instead. Summing this delay is conditional to the boolean variable $r_{i,p,m}$, which evaluates to true only when the task is currently mapped to the accelerator p and there is at least 1 other task currently assigned to it. Hence, it is sufficient to replace the product $z_{i,p,m} C_{i,s(p),m}$ with $h_{i,p,m}$ in Equations (5), (16), (17) and (19) to take into account the reconfiguration delay only when required.

FPGA acceleration and DPR on Linux. In the experimentation performed in Section 5.2, we used FPGA acceleration through the open-source FRED framework for Linux [14]. This middleware allows for keeping the bitstreams of the accelerators pre-loaded in the main memory of the host. Whenever an acceleration request is done to the FRED daemon, this checks whether the requested IP is already available on the FPGA, and it is capable of reconfiguring on-the-fly one of the available FPGA slots with the requested IP. This kind of interaction between the application code (the FRED client) and the FRED daemon using the above introduced computational model for real-time tasks, implies the need for assigning to the FRED daemon a scheduling deadline, and considering it as a further delay to be added to the execution of FRED calls. However, given that the FRED daemon essentially implements a device driver in user-space, and it just routes data back and forth between the client and the memory buffers mapped onto the hardware device, its execution time also happens to be quite small, in the order of a few tens of microseconds per FRED call.

Therefore, we just raised the scheduling priority of the daemon above the SCHED_DEADLINE scheduling class, by applying a very simple patch to the Linux kernel that relocates SCHED_DEADLINE from above to below the POSIX real-time scheduling class. This allowed us to run the FRED daemon at RT priority, with the assurance that, whenever a call was made to it, it could preempt for a very short time any SCHED_DEADLINE task from our deployed workload, to drive the interaction with the FPGA slots. This is pretty much what an in-kernel device driver would do, and the execution time that is "stolen" to the execution of SCHED_DEADLINE tasks can simply be accounted for in the worst-case execution times of the tasks (that need to be slightly inflated).

4.3 Schedulability Guarantees

The correctness of the MIQCP formulation can be proved, so as to guarantee that whenever the solver finds a feasible system configuration for the given problem, all the tasks (and thus all the DAGs) are schedulable.

Assumptions. The assumptions to build the schedulability proof originate from the model for which the analysis was provided:

(1) the dependency relationships among tasks described by the DAGs shall be guaranteed, i.e., each task in a DAG is activated only when *all* its predecessors have completed their execution;
(2) the EETB of each task of each DAG shall not be violated; hence, the bound to the execution time is safe and no overruns can happen;
(3) each DAG j shall be activated no earlier than its specified minimum inter-arrival time T_j;
(4) the system is configured according to any feasible solution, thus satisfying all constraints, of the MIQCP problem as formulated in Section 4.2.

Theorem 1. *Every task $\tau_i \in \Gamma$, when mapped to the CPU core $\psi_{\bar{c}} \in P$ whose island $\bar{s}$ is running at frequency $\phi_{\bar{s},\bar{m}}$, configuration given by the optimal solution of the related MIQCP formulation, completes within its intermediate deadline d_i, assigned by the solver, from its local activation.*

Proof. Because of assumption 1, not all the tasks contained in Γ can be activated concurrently. As explained in Section 4.2, at any given time of the execution of the application, for each DAG j only a subset S of its tasks in Γ_j can be potentially running concurrently, because there are no precedence relationships among them. This subset S belongs to the set W_j defined by Eq. (12), which enumerates all the subsets of the tasks in Γ_j that can be executed concurrently.

Consequently, the maximum utilization a DAG j can exhibit at any given time on the CPU core $\psi_{\bar{c}}$ is $u^{max}_{j,\bar{c}}$ defined in Eq. (13), computed as the maximum utilization of the tasks over the sets in W_j. This means that $u^{max}_{j,\bar{c}}$ is the bound on the bandwitdh required by the DAG that can never be exceeded throughout the execution of the application.

Given these considerations, when testing the schedulability of all the tasks mapped to $\psi_{\bar{c}}$ using the Liu and Layland test [44] for EDF, the j-th DAG contributes to the overall utilization for at most $u^{max}_{j,\bar{c}}$. Eq. (14) sums the maximum utilization for all the DAGs in the system, thus computing the overall required bandwidth on core $\psi_{\bar{c}}$. Imposing that the sum is lower than U_{max} guarantees the schedulability on that core. Finally, since Eq. (14) is a constraint of the MIQCP formulation, the configuration given by the solution of the problem is guaranteed to be meeting the schedulability condition for all the tasks in the DAGs mapped to the CPU core $\psi_{\bar{c}}$, thus proving the theorem. □

Theorem 2. *Every task $\tau_i \in \Gamma$, when mapped to the accelerator PU $\psi_{\bar{a}} \in P$ whose island $\bar{s}$ is running at frequency $\phi_{\bar{s},\bar{m}}$, configuration given by the optimal solution of the related MIQCP formulation, completes within its intermediate deadline d_i, assigned by the solver, from its local activation.*

Proof. Being the accelerators modelled as serving requests in FIFO order, guaranteeing that a hardware-accelerated task is schedulable means bounding its delay in the queue before it is executed and checking whether its intermediate deadline tolerates the delay bound. The worst-case delay suffered by τ_i in the queue happens when all the tasks mapped to $\psi_{\bar{a}}$ are enqueued right before its arrival. However, similarly to what stated in the proof of Theorem 1, because of the intra-DAG dependencies, not all the tasks can be activated concurrently. The set $\Lambda_{i,j}$ defined by Eq. (15) identifies all the possible combinations of the tasks in Γ_j that can run concurrently with τ_i. Hence, at any given time of the execution, the tasks of a subset $S \in \Lambda_{i,j}$ are concurrently active and, at worst, those that are mapped to $\psi_{\bar{a}}$ are enqueued for being served when τ_i arrives. The worst-case delay induced on τ_i by the tasks in Γ_j is given by the maximum delay over the sets in $\Lambda_{i,j}$, as computed by Eq. (16). Additionally, other tasks from the other DAGs in the system might be queued waiting for the accelerator. Therefore, Eq. (17) computes the worst-case delay produced on τ_i by them. Finally, the overall worst-case delay suffered by τ_i is the sum of the above explained components, as in Eq. (18). The intermediate deadline d_i of τ_i, interpreted as the bandwidth assigned to the task, in order to be safe and guarantee the schedulability of the task, takes into account not only the EETB of τ_i but also the worst-case waiting time it might experience in the queue of $\psi_{\bar{a}}$, and this is enforced by Eq. (19). Being this a constraint of the MIQCP formulation, the configuration given by the solution of the problem is guaranteed to be meeting the schedulability condition for τ_i on the accelerator $\psi_{\bar{a}}$, thus proving the theorem. □

Theorem 3. *Every task $\tau_i \in \Gamma_j$ terminates within the finishing time f_i as defined in Eq.* (9)*, from the activation of its DAG j.*

Proof. The theorem can be proved by induction referring to Eq. (9), which is a constraint of the MIQCP formulation.

Base case: *the start task of the j-th DAG τ_{s_j} terminates within f_{s_j}.*

From Eq. (9), $f_{s_j} = d_{s_j}$ because the starting node does not have any predecessors by definition. Theorem 1 and 2 guarantee that τ_{s_j} terminates within d_{s_j} from its local activation. Hence, the base case follows.

Induction step: *being $\Xi = \{\tau_{\tilde{i}} \in \Gamma_j \mid (\tilde{i}, i) \in \mathcal{E}_j\}$ the set of predecessor tasks of τ_i, if every $\tau_{\tilde{i}}$ in Ξ terminates within $f_{\tilde{i}}$, then τ_i terminates within f_i.*

The finishing time of τ_i is computed by Eq. (9). The first term is the maximum finishing time of all the predecessors of τ_i. Being all of them guaranteed to complete within their finishing time as per inductive hypothesis, that term is never exceeded during the execution and it is a safe bound for the local activation of τ_i. The second term is the deadline of τ_i, d_i, and Theorem 1 and 2 guarantee that τ_i terminates within d_i after its local activation. Therefore, this latter term is never exceeded during the execution. The sum of the two terms is a safe bound for the termination of τ_i and this proves the induction step. □

Theorem 4. *Every DAG j meets its end-to-end deadline D_j, i.e., its end task τ_{e_j} finishes within D_j.*

Proof. From Eq. (10), the finishing time of the end task f_{e_j} is imposed to be less than or equal to the end-to-end DAG deadline D_j. Being this a constraint of the MIQCP formulation, any optimal configuration that is solution of the problem guarantees the theorem. □

Additionally, we provide the clarification provided by the following Lemma.

Lemma 1. *Two subsequent activations of task $\tau_i \in \Gamma_j$ are distanced by at least the intermediate deadline of the task d_i, assigned by the solution of the MIQCP formulation.*

Proof. We denote by a_i^k the absolute activation time of the k-th instance of task τ_i, and by $t_{j(i)}^k$ the absolute activation time of the k-th instance of DAG j which τ_i belongs to. Because of Theorem 3, the finishing time f_i of task τ_i is a safe bound of the termination of the task itself relative to the activation of the j-th DAG; hence:

$$a_i^k + d_i \leq t_{j(i)}^k + f_i \tag{21}$$

Because of Theorem 4, the completion of τ_i is guaranteed to be within the j-th DAG end-to-end deadline $D_j \leq T_j$, with T_j separating two consecutive activations of the DAG:

$$t_{j(i)}^k + f_i \leq t_{j(i)}^k + D_j \leq t_{j(i)}^k + T_j \leq t_{j(i)}^{k+1} \tag{22}$$

The $k+1$-th activation of τ_i necessarily happens after the $k+1$-th activation of DAG j, by construction of the DAG model itself:

$$t_{j(i)}^{k+1} \leq a_i^{k+1} \tag{23}$$

Combining Eq. (21), (22) and (23):

$$a_i^k + d_i \leq a_i^{k+1} \implies a_i^{k+1} - a_i^k \geq d_i \tag{24}$$

meaning that two consecutive activations of task τ_i are distanced by at least its intermediate deadline d_i assigned by the solver. Therefore, the lemma follows. □

4.4 MILP problem formalization

The MILP formulation is obtained as a simplification of the above introduced MIQCP one, where intermediate deadlines (and their inverse values) are found using a standard end-to-end deadline splitting technique [22] assigning intermediate deadlines proportionally to the execution times of the tasks. The algorithm is the same used in our heuristics, so it will be described below in detail. While doing this step, we don't know yet what DVFS settings will be chosen for the CPU islands where tasks will be deployed, nor do we know the computational capacity of the islands each task will be deployed onto. Therefore, we chose to apply the end-to-end deadline splitting proportionally to the reference execution times provided for the reference island at maximum frequency (the C_i values introduced earlier).

Clearly, the MILP approach cannot reach the optimality level of the MIQCP, that is completely free to choose arbitrary intermediate deadlines satisfying the timing constraints of the DAGs. The extent of this difference will be clarified in the experimentation in Section 5.

4.5 Heuristic solvers

The optimum approach discussed above can become prohibitively expensive in case of large scenarios with many tasks, CPUs, or power modes. In this paper, we propose heuristic solvers for the introduced problem that manage to keep the execution times limited, albeit they cannot provide the same optimality guarantees. The trade-offs among achievable optimality vs solving times are discussed in Section 5.2.

4.5.1 DAG end-to-end deadline splitting. Both heuristic solvers introduced below make use of a classical end-to-end deadline splitting for real-time DAGs: given an end-to-end deadline for the DAG, we aim at finding a set of intermediate deadlines for the individual tasks of the DAG, so that the sum of the deadlines on the critical path from the starting to the ending task of the DAG, does not exceed the assigned end-to-end deadline requirement. A popular and very reasonable assignment is the one that gives more time for the heavier computational activities, thus assigning deadlines proportionally to the execution times of the tasks [22].

In the general problem formulation tackled in this paper, such a deadline splitting can only be performed once we have a tentative assignment of tasks to islands, along with their tentatively assigned power modes, as this lets the scaled EETBs $\{C_{i,s,p}\}$ become known (as an alternative, for the MILP approach mentioned in Section 4.4, we run this algorithm referring to the unscaled EETBs $\{C_i\}$). Note that this algorithm is applied to each DAG independently from one another, and it does not take into consideration any precise task-to-core mapping nor schedulability constraint for the tasks. This will be done in the heuristic algorithms that follow in the subsections below.

The end-to-end deadline splitting is performed according to the following recursive algorithm:

(1) Consider the problem of assigning intermediate deadlines from the starting node n_{src} to the ending node n_{dst} of a DAG, so to respect an end-to-end deadline of D; note that at the first iteration of the algorithm, these nodes are the starting task τ_s and the ending task τ_e of the DAG;
(2) Find the critical path P of nodes with the highest sum of scaled execution times from n_{src} to n_{dst}; let L be its length (sum of scaled execution times in P);
(3) Compute the deadlines to be assigned to each node n_i in P: $d'_i = D \cdot C_i / L$;
(4) If any intermediate node has an already assigned deadline d_i that is lower than the just computed d'_i, then do not modify it, and repeat the computations of the previous step after subtracting C_i from L and d_i from D, for all nodes with already assigned lower deadline;
(5) If n_{src} and n_{dst} have no deadline already assigned, then assign them the computed values; otherwise, assign them the minimum deadline between the computed values and the already assigned ones;
(6) For each node $n_{src'}$ following n_{src}, proceed calling recursively this algorithm, but using $n_{src'}$ and n_{dst} as the new starting and ending nodes, and using an end-to-end deadline D' obtained subtracting the intermediate deadline value just assigned to n_{src} from D: $D' = D - d_{src}$.

The minimum operations in step 4 and the subsequent one above are needed because we might have intermediate nodes that belong to multiple paths, from the starting to the ending task. In such a case, the first time we encounter such a node we assign it a tentative deadline. However, the next time we encounter it, we either keep its previously assigned deadline, if tighter than required by the new path being considered; in such case, we exploit the tighter deadline by trying to relax constraints on the new path being processed. If the new path requires a tighter deadline than previously assigned, instead, then we change the deadline to the new value. As the deadline of the common node is being decreased, this operation cannot break the end-to-end deadline constraint on the previously considered path(s), that was/were satisfied with a looser intermediate deadline.

4.5.2 Top Island First (TIF) Heuristic. Top Island First (TIF) is a heuristic we designed to be fast, loosely inspired by the First-Fit algorithm. As we will show in the experimental evaluation in Section 5.2, for quite large problems TIF is faster compared to the second heuristic, BB-Search, described below, thus it is potentially more useful for big problems.

Inputs:

- A hardware model defined as a list of I_s *islands* sorted by decreasing processing-unit *capacity* ξ_s and the islands operating performance points (OPPs), sorted in decreasing frequency;
- A software model defined as a set of disjoint DAGs G as defined in Section 3.

Initialization:

- Populate a task list T with all the tasks in G sorted in descending order of runtime, assuming all tasks are placed onto the top island at its maximum OPP. This approach creates a common configuration where the tasks can be sorted by their workloads;
- Initialize the islands' OPP to their respective highest value, i.e, all islands are initially assumed to be at their maximum frequency;
- Create a list of sets P_s, with $|I_s|$ initially empty sets, representing the tasks mapped to each island s. The first set P_1 in P_s represents the top capacity island, and the last set $P_{|I_s|}$ represents the lowest capacity island. $P_{s,t}$ denotes a task $\tau \in T$ mapped onto the island s;

- Create a copy of the DAGs G, called Gz, where the runtime of their nodes is zeroed. Despite that, Gz has the same topology as G, and it is used next to derive an initial task placement.

Steps:

(1) For each task τ in T;
(2) Assign τ runtime to its correspondent task in Gz such that, in the first iteration of the loop, only task τ has a non-zero runtime in Gz. Next, assign τ to the first island P_1, and update τ runtime in Gz according to its P_1 capacity;
(3) Compute the intermediate deadlines using the algorithm in Section 4.5.1, and test all the constraints; If all constraints passed, τ stays on P_1, otherwise, τ is tested onto the next islands until τ finds an island that satisfies all the constraints or when all islands were tested. If the latter happens, then the scenario is declared unfeasible;
(4) The last two steps above are repeated until all tasks are placed or the scenario is unfeasible;
(5) For each task τ that is not assigned to the lowest-capacity island already, check whether, moving τ one island down in terms of capacity in the current task placement, would result into breaking any of the constraints; if not, then keep the move, otherwise consider the next task. Repeat this step until no task is moved any further;
(6) Consider the task-to-island mapping $\{P_s\}$ obtained so far (under maximum DVFS settings); now try to reduce the frequency of each island, from the highest-capacity to the lowest-capacity one, updating the tasks scaled execution times $C_{i,s,p}$, updating the end-to-end deadlines, and checking whether any of the constraints is violated. Continue trying to reduce the frequency of all islands.

4.5.3 BB-Search Heuristic. This heuristic searches for a feasible placement of real-time tasks onto processing islands of a heterogeneous computing platform while minimizing its power consumption. This tool was designed to be faster than using general optimization techniques, thus more scalable to larger problem instances.

BB-Search does an exhaustive search of the task-to-island mapping possibilities. Thus, the search space size is $(|I|^{n_T}) \cdot |\Phi|$ where $|I|$ is the number of islands, n_T is the number of tasks for all DAGs, and $|\Phi|$ is the set with all combinations of OPPs. The algorithm is described as follows:

(1) Initialize the lowest power to $+\infty$;
(2) For each potential task partitioning among the islands (there are $|I|^{n_T}$ such assignments):
(3) Set all islands to their top OPP;
(4) Adjust the task runtimes $C_{i,s,p}$ according to the current task-to-island placement and OPP;
(5) Update the total power. If it is $\geq$ than the lowest power, then skip to the next OPP, or task placement;
(6) Update the task intermediate deadlines (with the algorithm in Section 4.5.1) and check if the DAGs end-to-end deadline is violated. If yes, skip to the next task placement;
(7) The tasks mapped onto each island are now placed onto CPUs using a Worst-Fit heuristic, in an attempt to avoid violation of the CPU utilization bound. If any CPU in the island turns out to have utilization $> U^{max}$, skip to the next task placement;
(8) This configuration of task placement and OPP is the best feasible found so far, therefore update the lowest power.
(9) The final step is to decrease the OPP for each island, in order to find the lowest valid OPP for the current task-to-core mapping that satisfies all the constraints.

This algorithm might seem expensive at a glance, because of the exhaustive enumeration of all possible mappings at Step 2. However, for each such assignment, the first check is made based on the estimated power consumption of the resulting configuration, as from Step 5, skipping any further action if we cannot improve with respect to a previously found mapping. Thus, the number of times we descend into the remaining steps is very limited, from a practical standpoint. The experimentation carried out in Section 5 will provide quantitative details in this regard. Note that the OPP set Φ is sorted by decreasing frequency. Thus, when an OPP is declared unfeasible, the next OPPs are no more tried, as it is unlikely that with a lower frequency the placement would be feasible. This observation further reduces the number of searched OPPs for each considered task placement. The algorithm is prone to parallel implementations, since all task placement tests are independent. However, the current implementation is sequential for simplicity. Its performance will be evaluated in Section 5.2.

The main simplifying assumptions for the sake of performance are:

- Relative intermediate deadlines of tasks are assigned using a classical end-to-end deadline splitting approach, where deadlines are proportional to the scaled computing times of the tasks they are attached to (once their placement on islands at given OPPs has been fixed); on the other hand, the optimal approach is not constrained to choosing deadlines according to such a rule, keeping a bigger search space.
- The placement of tasks onto CPUs within an island is done with a worst-fit heuristic. This has the advantage of trying to balance the load across the CPUs within the island.

5 EXPERIMENTAL RESULTS

This section describes the various experiments that were carried out to validate the approach presented in the paper and and evaluate its effectiveness. First, the basic assumption made on the execution time scaling model is validated in Section 5.1. Then, the MIQCP, BB-Search and TIF solving strategies are compared in Section 5.2. The soundness of the proposed mathematical formulation is verified by simulation in Section 5.4, then Section 5.5 reports on the effectiveness of the platform configurations found by the various solving strategies, that are experimentally compared applying them to randomly generated synthetic workload scenarios, deployed on a big.LITTLE board and on a FPGA-accelerated Xilinx board. Finally, Section 5.5.4 shows how the secondary optimization goal of slack maximization can be conveniently leveraged to obtain more robust configurations.

5.1 DVFS Scaling Model

Our first set of experimental results aims at validating the execution-time scaling model as a function of the CPU frequency, assumed in our optimization framework and formalized in Equation (4). To this purpose, we considered a number of simple micro-benchmark applications, i.e., `encrypt` and `decrypt` (using 3DES), `hash` (using SHA256) and compression with `gzip` (using various compression levels), processing a fixed amount of input data. These have been run on an ODROID-XU4 board, with big.LITTLE architecture. Each application has been run on both big and LITTLE cores, and for all of the available CPU frequencies. The data-set has been gathered using PARTProf, and embedded within the PARTSim open-source real-time systems simulator [5].

We performed a least-squares fit of the obtained experimental execution-time data, with the theoretical model in Equation (4). Focusing on results obtained for the LITTLE island, for example, we found out that, for some workloads, like `encrypt`, `decrypt` and `gzip-1`, the model fits with basically $C_{ns} = 0$, so the execution time scales perfectly well with the frequency. On the other hand, for `hash`, the model fits with $C_{ns} = 0.028\,C$, i.e., it has a residual 2.8% of the execution time that does not scale with the frequency, whilst for `gzip-1` and `gzip-9` this percentage is much higher,

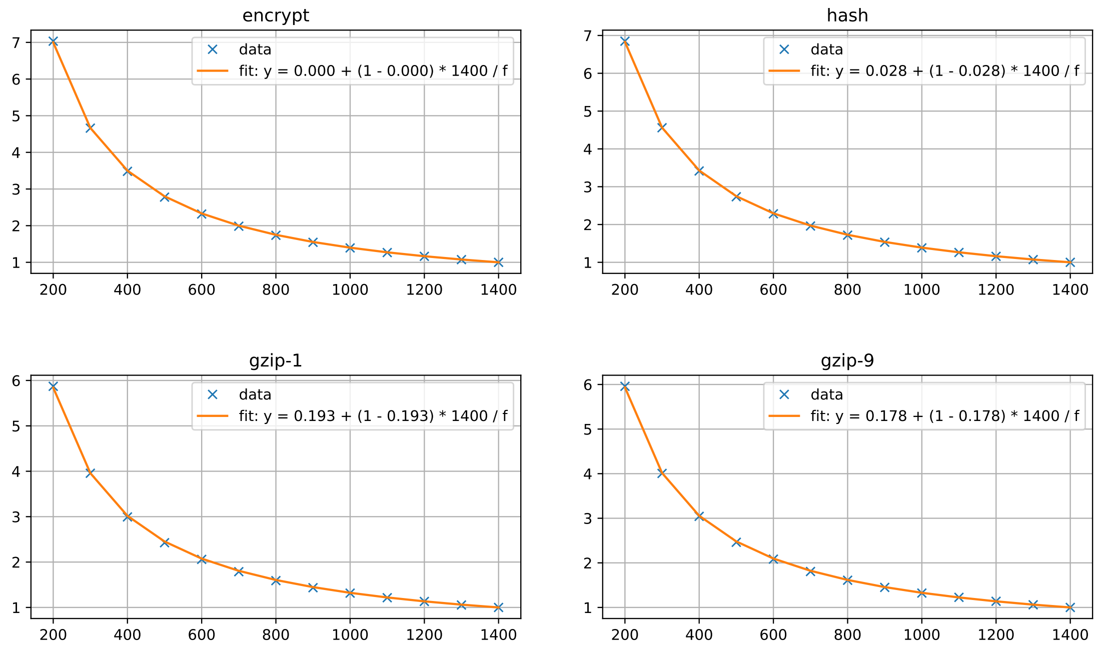

Fig. 2. Execution times (on the Y axis) obtained running the microbenchmarks at various frequencies (on the X axis), normalized to the value obtained at the highest frequency.

below 20%. These results are summarized in Figure 2, comparing the experimentally gathered data with the data coming from the theoretical model in Equation (4) after fitting.

In this micro-benchmarking process, we detected that any frequency of the big island above 1.4 GHz was unstable due to overheating of the CPU, despite the cooling fan was being kept active at maximum speed at all times. So, these frequencies have not been used in the experiments that follow.

5.2 Comparison among optimization strategies

In order to compare the effectiveness and the scalability of the proposed platform optimization techniques in Section 4, we performed a benchmarking campaign applying the four techniques to a number of randomly generated real-time DAG sets, optimized for an ODROID-XU4 platform.

The DAGs were generated according to a random DAG generation algorithm [30], adapted and configured to generate:

- end-to-end DAG deadlines in the range between 10 ms and 100 ms, in a granularity of 10 ms;
- unscaled task execution times in the range between 0.5 and 2.5 per DAG;
- minimal and maximal number of DAG layers (number of hops between the starting and ending tasks), ranging from 3 to 6;
- maximum parallelism per layer from 3 to 8 tasks per layer;
- connection probability among the tasks from 20% to 40%.

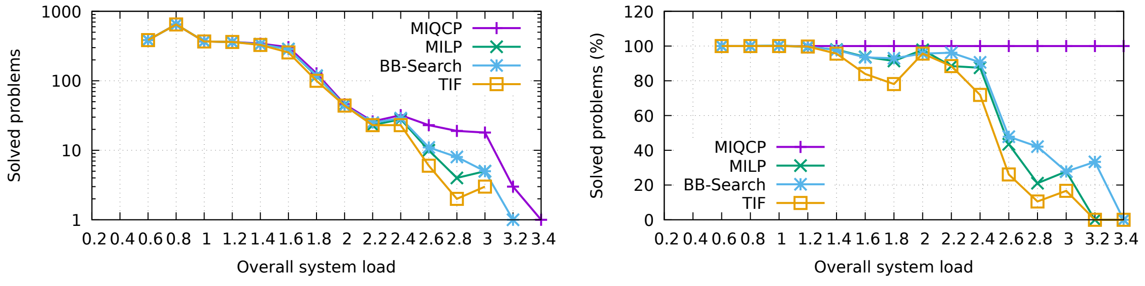

Fig. 3. Left: number of problems (on the Y axis, in logscale) for which various solving strategies (in various curves) found a solution, for various ranges of the overall system unscaled load (on the X axis, in bins of 0.2). Right: same information, in percentage with respect to the problems solved by MIQCP.

Overall, we generated 2678 DAG sets, out of which 200 were deployed on the board for experimental validation. The remaining DAGs have been simulated using the PARTSim power-aware real-time systems simulator [5], as explained in Section 5.4.

Each considered DAG set has been submitted to the four solving strategies (MIQCP-based, MILP-based, BB-Search and TIF), while in addition to the objective function value and the optimum placement output by the solver, we also gathered its solving time. A timeout of 1h was initially set to perform the optimizations, then the longest lasting ones have been rerun with a timeout of 24h.

Figure 3 reports on the left the number of problems for which the considered solving strategies managed to find a solution (on the Y axis, note the log scale), and on the right the percentage compared to the number of problems solved by the optimum MIQCP-based strategy. This information is reported in bins for the overall unscaled load of 0.2 in size. Note that the considered ODROID-XU4 board has a capacity of 1.0 for each big core, and a capacity of 0.24 for each little core, so the maximum load it could theoretically host is 4.96. However, all optimizations have been run under a realistic configuration, so to allow us deploying a part of the optimized configurations to the real board. Therefore, a maximum cap of $U^{max} = 0.95$ utilization per-core has been configured in the solver (see Equation (14)), leading to a maximum system load of 4.7 that could be accommodated on the board in these optimizations.

As evident, all heuristic solvers start exhibiting serious shortcomings for overall platform loads going beyond roughly half of the platform capacity, where the MIQCP-based approach outperforms all of the others (including the MILP-based approach that can be found in various other papers), with its ability to set task deadlines arbitrarily, while others are constrained to deadlines proportional to the task load. Interestingly, the BB-Search heuristic is capable of solving more problems in the high-load range $[2.0, 3.0]$ than MILP, thanks to the fact that the deadline splitting algorithm is applied only after having fixed a tentative assignment of tasks to islands (and their DVFS settings). Note that the focus of this paper is NOT on highlighting what is the percentage of DAG sets admitted and found feasible by our proposed technique, but on the performance and energy efficiency achievable with platform configurations obtained using our proposed framework. Therefore, the majority of our randomly generated DAG sets have been designed around an "average" overall (unscaled) load for the platform, in order to avoid areas where we might easily find DAGs that would not be found as schedulable by some of the proposed solvers.

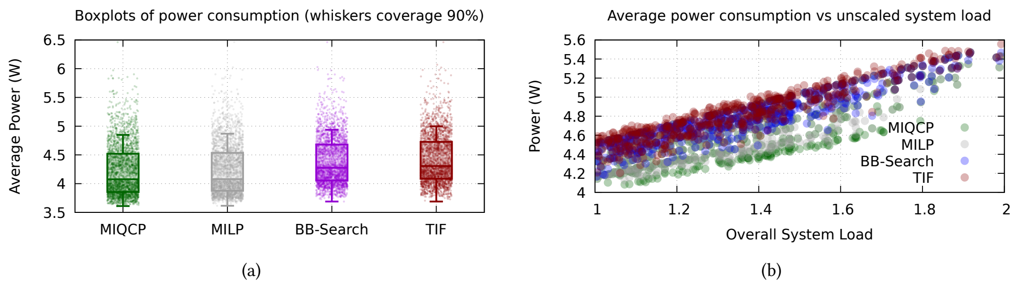

Fig. 4. (a) Boxplots of optimized average power, alongside their visual distribution, across all considered problems, under various solving strategies (different curves). (b) Detail of the optimized power as a function of the unscaled system load (X axis).

In order to compare the optimality and usability of the various solvers, in the following we show results that have been pre-filtered only on problems where all of them provided a solution. Figure 4.(a) reports boxplots of the distribution of the optimized average power value obtained across all of the considered problems, with each of the considered optimization strategies. As expected, strategies based on optimizers, i.e., MIQCP and MILP, outperform the simpler heuristics, with their median and inter-quartile power values visually lower than what obtained by BB-Search and TIF. The latter heuristics perform similarly, but BB-Search can be observed to obtain generally lower power values compared to TIF, given its extended search design.

Figure 4.(b) details further the obtained data, by reporting the average power obtained by the various solvers, at varying overall unscaled system load (on the X axis). The plot is zoomed in the load range $[1.0, 2.0]$, where we have the majority of the problems solved by all solvers. Generally, the optimized average power consumption output by all solvers grows linearly with the system load, as expected. The plot highlights that the just discussed pros/cons among the various solving strategies, in terms of achievable power consumption optimality, are roughly equally distributed throughout the overall system load range. Indeed, we can visually distinguish "stripes" of points in the plot corresponding to MIQCP, MILP, BB-Search and TIF, from the bottom towards the top of the plot.

However, a more complete picture of the situation is obtained when considering also the solving time needed by the various solvers, so to provide an idea of the obtained benefits (objective function value) vs the cost at design time (solving time). To this purpose, Figure 5 (left) reports a scatter plot of the average power consumption (on the X axis) and the solving time (on the Y axis), obtained for each scenario (data point) when optimized using either one of the four solving strategies (different point series). The optimum MIQCP solver has a points cloud visually positioned in the topmost-leftmost area of the plot, corresponding to lower (better) optimized power values, but also higher solving times (note the logarithmic scale on the time axis). The runs that hit the 3600s timeout are clearly visible as the points accumulating in a sort of horizontal segment in the top area of the plot (the points above the 3600s time limit correspond to a further re-run of these very long optimizations with a 24h timeout, as detailed in the next subsection). Points related to the MILP solver are visually positioned in the leftmost but vertically centered area of the plot, still preserving a good optimality level of the found solutions, at reduced solving times compared to MIQCP, but generally higher solving times compared to BB-Search. The latter presents an interesting density of points at the bottom of the plot, at quite low solving times, with achieved optimality of the solutions spread on a wider range, compared to the MIQCP and MILP approaches. Finally, TIF achieves much faster execution times, below one second, for big problems requiring

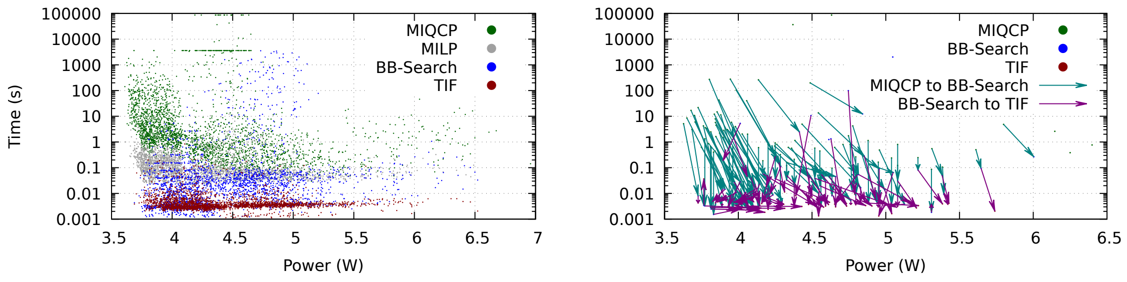

Fig. 5. Left: average solving time (Y axis) obtained by the various optimizers (different data series) vs the achieved optimized average power (X axis). Right: visualization with arrows connecting points obtained by various solvers for the same problems (random subset of 100 problems).

minutes or hours of solving times with MIQCP or MILP. One can also note that only few scenarios of MIQCP and fewer from BB-Search took more than 1000 s. The other optimizers (TIF and MILP) took way less than 1000 s.

Figure 5 (right) details a sample of 100 randomly chosen problems, for which the arrows connect points corresponding to the same problems, optimized with the MIQCP, BB-Search and TIF optimizers. The arrows from MIQCP to BB-Search have generally a "south-east" orientation, highlighting that BB-Search produces solutions with worse objective function value (higher values on the X axis), at reduced execution times (lower values on the Y axis). However, some vertical arrows can also be appreciated, where BB-Search manages to achieve the optimum power value, still at much reduced solving times (note the logarithmic scale on the Y axis). Similarly, the arrows from BB-Search to TIF points are generally directed to the right, highlighting that TIF tends to find worse solutions, but for big problems the solving time of TIF is much lower than both BB-Search and MIQCP. However, sometimes TIF may sporadically find better solutions than BB-Search, as they are different heuristics, and there is no guarantee that one of them is always better than the other.

5.3 Scalability Evaluation

We considered 29 DAG sets that timed out running the MIQCP optimizer (all of them are dual DAG scenarios), and re-ran them with an extended timeout of 24 hours. Ten of these DAGs sets could find the optimal value while 19 still timed out, visible in Figure 5 (left) as the points that accumulate at 86400s, at the very top of the plot. From these 19 scenarios that timed out, three scenarios got no improvement, meaning they presented the same partial solution presented when the timeout was 1 hour. On the other hand, 16 scenarios got a partial solution of some minor improvement, on average, 0.0089 W of power reduction. From the ten scenarios which found an optimal solution, five of them resulted in no power reduction, meaning that the partial solution found after 1h was, in fact, the optimal solution. The other five scenarios got an average power reduction of 0.0096 W.

All instances of TIF ran in less than one second of CPU time. Its longest execution was 0.83 seconds. The longest BB-Search execution time was 58 minutes, but it has an exponential computational complexity and it will eventually reach execution times longer than 1 hour with more than 32 tasks.

In conclusion, depending on the size of the problem under consideration, one can use the MIQCP approach having the guarantee to find an optimum solution, or, for problems that timeout after the maximum acceptable waiting time for the designer, one may use one of the other heuristics, that are much faster and often bring a result that may be "good enough", albeit non-necessarily optimum.

5.4 Validation by simulation

PARTSim[4], our recent extension [5] of the well-known open-source RTSim [16, 52], is a non-functional power-aware real-time systems simulator capable of simulating the schedule of tasks within a platform under various scheduling policies, along with the associated estimated power consumption. The simulator supports single-core and multi-core platforms, including systems with DVFS capabilities, with either partitioned or global fixed-priority and EDF-based schedulers. For DVFS-capable systems, the simulator can estimate both the power consumption and the thermal profile of the platform during the simulation. This is achieved trough platform description files that can be automatically generated using its companion profiling tool, PARTProf [5]. This tool can extrapolate a realistic model for the power and timing behavior of real-time tasks when executed under DVFS on heterogeneous architectures by profiling the execution of a set of representative tasks on the target embedded platform. The models generated through this analysis of application profiles are used to drive the behavior of the tasks simulated by PARTSim, so that the simulator can closely match the specified platform.

The simulator task model includes the ability to specify a set of tasks with specified EETBs and periods (or minimum inter-arrival times). For each task, the simulator allows the specification of its body as a sequence of instructions, typically consisting of computational activities or synchronization operations among multiple tasks. This set of instructions allows the modeling of critical sections protected by blocking mutual exclusion semaphores and the specification of dependencies and precedence constraints among the tasks. The simulator also provides implementations for several scheduling servers, which can contain one or multiple real-time tasks, including sporadic or constant bandwidth servers.

We used PARTSim to virtually deploy the scenarios optimized for our target ODROID-XU4 platform. The platform description provided for the simulator by the PARTProf tool provides PARTSim with the very same model used by each of our solvers to estimate the timing and power behavior of the systems being optimized. Combining this model with the static configuration provided by each of our solvers, we can accurately simulate the behavior of one or multiple DAGs as if executing on the target platform under the desired DVFS settings and task placement.

In our simulations, the source tasks of each DAG are activated periodically, while all other tasks in each DAG are modeled as a sequence of three actions: 1) wait for all inputs to be provided by all of the predecessors; 2) perform some CPU-intensive computation for a specified amount of time; 3) push outputs of the computations towards the subsequent tasks in the DAG topology.

Starting from a subset of 1700 DAG sets among those described in Section 5.2 (each comprising one or multiple parallel independent DAGs), we obtained 5100 system configurations and task mappings for our ODROID-XU4 target platform using our three proposed solvers: MIQCP, BB-Search and TIF. We ran all of these solved scenarios in the PARTSim simulator. After each run, we checked the simulated end-to-end response times of the deployed DAG sets and the average power consumption calculated for the simulation duration. As expected, no simulated scenario exhibited deadline misses, and the simulated power consumption values matched the theoretical expectations, since both the simulator and the solvers leverage the same power consumption and timing models. Nonetheless, simulation of the end-to-end time schedules obtained in RTSim with the CBS under partitioned EDF, confirms that the proposed optimization approach in Section 4, in terms of schedulability constraints, is sound.

[4]More information is available at: https://github.com/gabrieleara/PARTSim.

5.5 Experimental validaton on Linux

5.5.1 Multi-DAG Scenarios Implementation on Linux. After validating by simulation the optimized scenarios produced by our proposed solvers, we deployed a subset of them on the real ODROID-XU4 platform referenced in our models. To do so, we first developed an application that emulates the behavior of a real application comprising one or multiple real-time DAGs running on Linux. This application, which we will refer to as RTDAG, reads the system configuration provided by one of our three solvers and starts several parallel threads of execution, each corresponding to one of the tasks in the original DAG specification.

Similarly to our simulated tasks implementation, the source task of each DAG is activated periodically while the others execute according to the same three steps described in the previous section. In this case, the implementation of each task performs a series of operations on a set of matrices, a representative workload of a typical real-time image processing pipeline. The operations performed by each task are carefully calibrated offline to result in the expected execution times when running on the most powerful (big) core at the highest available frequency[5]. For more information on the calibration of task execution times, see Section 5.5.2. Thanks to our accurate time scaling model, if the tasks execute for the correct amount in that system configuration we expect them to behave according to that model when run at lower frequencies or on less powerful cores.

The implementation of each task reflects a typical LET pattern, in which each task limits operations on shared memory only at the beginning/end of its activation and performs mostly CPU-intensive operations in between. Each task that is not the DAG originator takes as input a set of matrices and produces another set of matrices as output for its successors. The synchronization operations between predecessors/successors in each DAG are implemented using condition variables. Only when all the predecessors of a task have completed their execution, the task is notified, thus it is unblocked and can proceed forward for its own computations.

Using this application, we selected a representative subset of about 200 task sets out of the 2700 randomly generated ones described above and picked the corresponding scenario provided by each of our three solvers, resulting in about 600 scenarios executed on the target ODROID-XU4 platform overall. For each of these scenarios, RTDAG configures the platform's DVFS state to reflect the one chosen by the solver and pins each of the tasks to the corresponding core. Each task is executed using the SCHED_DEADLINE Linux scheduler using the runtime and deadline calculated by the solver in the offline phase.

The execution of each run consists of a series of steps, some executed on the target platform and some on an external computer (to avoid interfering with the execution of the real-time tasks):

(1) Before starting each instance of RTDAG, we start monitoring the power consumed by the ODROID-XU4 board using an external power monitor connected to its power supply; this step is done on the external computer;
(2) we then kill all Linux services and applications that are non-essential to the execution of RTDAG using `systemctl`; we keep only essential services and the SSH server running on the board to synchronize the two machines involved in the experiment;
(3) we then start RTDAG; RTDAG reads the configuration provided as input, configures the ODROID-XU4 DVFS configuration to match the specified one, and starts all the real-time tasks corresponding to each task in the DAGs set;

[5]We exclude from our considerations all frequencies which may lead the system to thermal throttling.

(4) after running each DAG for the number of hyperperiods so to have a run of about 30 seconds, RTDAG terminates, and we can collect information regarding end-to-end response times and an average of the power consumption measured by the external power monitor during the execution of RTDAG.

5.5.2 Execution Time Calibration. As mentioned in the previous section, the implementation of a real-time task provided by RTDAG is synthetic and as such it does not perform any meaningful manipulation of the data received in input by each task. In reality, each task performs a static set of algebraic operations involving floating-point matrices, which we consider a "*tick*". This *tick* is considered the minimum amount of execution that a task can run for. For longer execution times, we can repeat the execution of a *tick* multiple times, approximating any execution time with an integer number of *ticks*. Selecting a very limited number of matrix operations as part of a *tick* is fundamental, so that the granularity of the exeuction times emulated by the synthetic tasks does not pose a concern.

Given a DAG generated according to the process described at the beginning of Section 5.2, RTDAG assigns to each synthetic task a number of *ticks* to run depending on its declared EETB. Since the EETB declared in each DAG specification is specified as the expected execution time of the task when running on the most powerful (big) core at the highest frequency, we measured the average execution time t_{tick} needed by a *tick* in those conditions, then calculated the number of *ticks* a task τ_i has to run for, in order to exhibit a target execution time of C_i, as equal to $\left\lfloor \frac{C_i}{t_{tick}} \right\rfloor$. This considers that $C_i^{ns} = 0$ for our synthetic workload, as in the basic *tick* computations we used matrices of limited size. Since the number of *ticks* assigned to each task is computed statically, independently of the frequency or CPU type on which the task will be deployed, the amount of work that a task will do at each activation is always the same, emulating the behavior of a real non-synthetic application.

RTDAG provides the option to precisely calibrate the value of t_{tick} when deployed on a new platform, by repeatedly executing the workload of a single *tick* and outputting a number of statistics on the measured execution times.

However, any estimate on the execution time of a real-time task (synthetic or not) running on Linux under SCHED_DEADLINE can be influenced by several factors, including: interfering activities generally known as "OS noise" [32], e.g., due to the execution of IRQs or non-preemptibility sections of the kernel [33]; other tasks that may be scheduled by a scheduler that has a higher priority than SCHED_DEADLINE[6]; cache-level interference due to other tasks (even lower-priority ones), running on the same or different cores causing an unforeseen volume of evicted L2/LLC CPU cache lines[7]. All of these factors may impact the execution times of tasks, that might sporadically happen to be longer than experienced in previous profiling campaigns.

In general, high-level models for task EETB should consider all of these factors when specifying a value, so that the indicated EETB is effectively what its name suggests, an upper-bound to the execution time of a task (when executed at the correct frequency), and no task may ever exceed it, no matter the situation. In our case, the EETB is specified by the DAG generation tool, so we have to reason in the opposite direction when selecting the number of *ticks* of execution for each task. If we used the simple formula indicated above using the average measured time t_{tick}, then a synthetic task within RTDAG would result in having its *average* execution time approximately equal to the specified C_i value, i.e., roughly half of the times we would expect the task to have a longer duration. Thus, to let C_i represent an upper-bound to a task execution time, including the possible sources of interference mentioned above, RTDAG applies a correction

[6]Typically SCHED_DEADLINE is the scheduler with the highest priority among the Linux schedulers, which means that no task scheduler by a different scheduler than SCHED_DEADLINE can ever preempt a task running under SCHED_DEADLINE. However, this behavior can be easily changed via a simple patch to the Linux kernel, as shown in Section 5.5.6.
[7]Cache-colouring techniques may be conveniently leveraged to mitigate these issues.

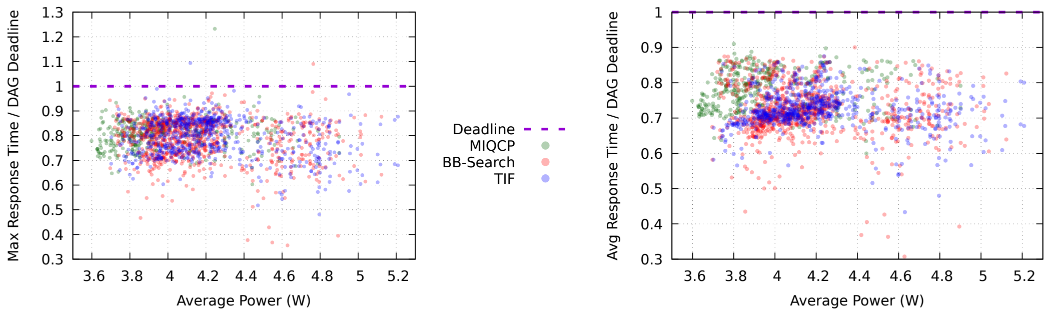

Fig. 6. Maximum (left) and average(right) relative response times (Y axis) vs average power consumption (X axis).

factor R in the formula for computing the target execution time: $R \cdot \left\lfloor \frac{C_i}{t_{tick}} \right\rfloor$, where we verified experimentally that setting $R = 0.95$ was sufficient to avoid actual executions longer than the intended C_i values, at any frequency or core type for our reference platforms, the ODROID-XU4 and the Xilinx UltraScale+ ZCU102. In particular, the latter of these two suffers from the interference of a special task, as addressed in Section 5.5.6, which is mitigated using this parameter.

5.5.3 Experimental Results on ODROID-XU4. In Figure 6, we report the relative DAG response times (end-to-end response-time of each DAG divided by its deadline, on the Y axis) versus the measured power consumption (on the X axis), for the scenarios we ran, optimized according to the considered optimizers (different point series). The left and right plots report the maximum and average relative response times, respectively, on the Y axis. As evident, the MIQCP solver succesfully manages to deploy scenarios generally at a lower average power consumption than the other two approaches. Clearly, this is done by managing to keep the DVFS of the islands at lower-power frequency settings. This unavoidably results in DAG response times that are slightly higher and closer to the DAG end-to-end deadlines (the dashed line at 1.0 on the Y axis in the plot).

We have to observe that, in a very few scenarios, just 3 out of the 600 we ran, we observed deadline misses in optimally configured scenarios, which were not observed in the same scenarios optimized using the heuristics nor in simulation.

From the same runs, we visualized the frequencies chosen by the three optimization strategies in the histogram plots of Figure 7, where we can see the histograms of the frequencies chosen for the big (right subplot) and LITTLE (left subplot) islands by the 3 optimization strategies. We can observe that the TIF and BB-Search heuristics often pick higher values of the frequencies for the two islands. On the other hand, the MIQCP solver is more capable of exploiting all of the available frequencies, especially the middle and low ones, making very little use of the maximum frequencies.

5.5.4 Maximizing robustness under power budget constraints. An experiment was carried out to evaluate the MIQCP formulation optimizing the relative slack, described in Section 4.2, to improve the robustness of an application. The selected test bench was an application composed of 2 randomly generated independent DAGs, each made of 6 tasks, with end-to-end deadlines 40ms and 20ms respectively, intended to be executed on the ODROID-XU4 hardware platform. A first optimization run was performed, minimizing only the overall power consumption, thus obtaining the system configuration exhibiting the minimum possible power for that application, i.e., 4.4077 W. With an in-depth analysis of the solution, it came to light that the finishing time of both DAGs coincided with the assigned deadline, hence they

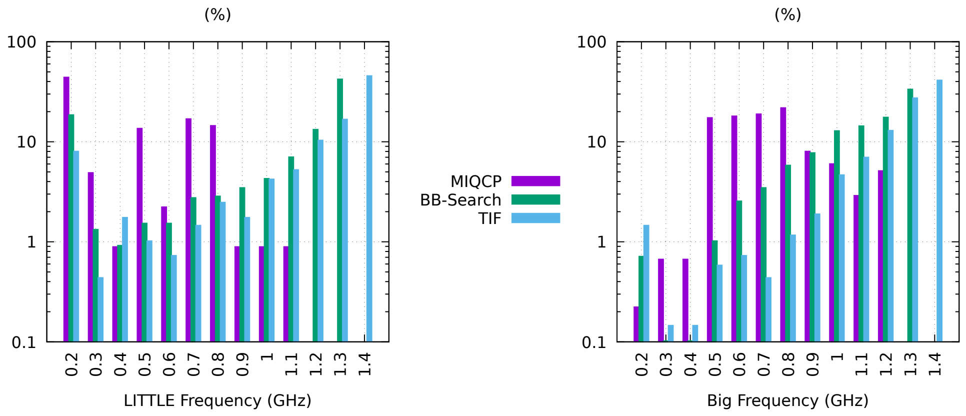

Fig. 7. Chosen frequencies for the two islands

lacked any safety margin for coping with possibly underestimated EETBs. Another optimization run followed, utilizing the formulation that maximizes the minimum relative slack of the application, and setting as power budget the minimum power value for that application returned by the first optimization run. The solver provided a solution with the same hardware configuration as the one with the minimum power, since the formulation imposed not to exceed the optimal power budget. However, task-to-PU mapping was slightly different and allowed to obtain a minimum relative slack of 4.5%, resulting from the DAG finishing times being 38.2ms and 19.1ms respectively. Hence, the slack optimization proved to improve the robustness of the application while still retaining the minimum possible power consumption. The reason being that the two optimization objectives can be conveniently combined, so that once the minimum-power configuration has been identified, there is chance that the solver is able to obtain a non-zero robustness margin with a slight modification of the task mapping.

We also performed a wider experimentation, considering, among the set of the 200 scenarios deployed on the ODROID-XU4 discussed in Section 5.5), the 19 DAG sets with the highest maximum slack, including all those exhibiting sporadic deadline misses. These scenarios, that had already been optimized only for power consumption using MIQCP, were re-optimized, still using the MIQCP solver, adding also the secondary objective of maximization of the minimum relative slack. The optimized scenarios for both objective variants have been run 10 times each. Figure 8 reports the obtained minimum and maximum values (vertical segments) of the maximum relative slack obtained in each of the 10 runs for each of the considered scenarios, under power-only (red segments) and power-plus-slack (green segments) optimization. Visually, it is evident that the scenarios optimized also for slack resulted in a general lower (or sometimes equal) statistics of the obtained relative slack. Additionally, the few scenarios that originally missed their deadlines under power-only optimization, turned out to stay safely below the deadline limit, in the re-optimized configuration maximizing slack (at an equal average power consumption).

5.5.5 WATERS19 industrial case-study. This section summarizes our experience in trying to apply our optimization approach to a real industrial use-case. To this purpose, we considered the WATERS19 challenge, proposing a use-case including an Advanced Driver-Assistance System (ADAS) application running on an NVIDIA Jetson TX2 board, with

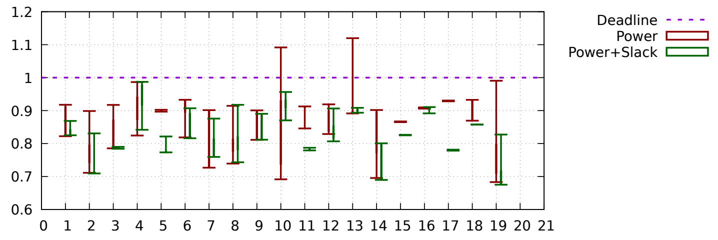

Fig. 8. Maximum response times relative to DAG periods (Y axis) for various DAGs (X axis), with and without slack optimization (green and red lines, respectively), in addition to power optimization.

heterogeneous CPUs (ARM Denver and A57), and a GPU accelerator [21, 67]. Given the use of a GPU-accelerated Linux-based platform, the use-case is a viable one to be tackled with our proposed approach.

The software model consists of nine main tasks, where four of them (SFM, Localization, Lane Detection, and Detection) can run either on CPUs or on the GPU. The remaining tasks can only run on CPUs. Nonetheless, there are differences to the software/hardware model and requirements that need to be addressed before using it with the proposed MIQCP optimizer. Here is how we adapted those differences for our proposed models.

First, WATERS19 models the software as a DAG of periodically activated tasks, whilst ours models it as periodically activated DAGs of tasks with data-triggered activation. Also, WATERS19 constraints are for fixed-priority scheduling, using task period and EETB as main scheduling data. Thus, WATERS19 does not provide an end-to-end deadline for the DAG, required to apply our model. Our approach is to assign the lowest end-to-end deadline supported by each of the three platforms where (detailed next) the WATER19 use-case is evaluated.

Second, the WATERS19 original platform is GPU-based. While our approach is flexible enough also to model GPU-based hardware platforms, as seen in Section 5.5, we have not done the performance and power characterization on the NVIDIA Jetson TX2 board. Thus, we must adapt WATERS19 performance data, captured on the NVIDIA board, for our currently supported platforms. Recall that the WATERS19 source code is unavailable, so it is impossible to accurately get the performance/power characterization on other platforms. Our approach assumes that its original CPU performance data is compatible with ODROID-XU4 CPUs, which is a supported board. It means that if the original WATERS19 says that task A has an EETB of 10ms on the NVIDIA board, we assume that this task also has a similar EETB (after some fixed scaling) on the ODROID-XU4 board.

Taking into account these considerations to adapt WATERS19 to our use-case, we proceed with its evaluation on three different hardware models corresponding to: ZCU102 FPGA board, ODROID-XU4, and a hypothetical platform. The hypothetical platform is used to demonstrate the flexibility of the proposed approach in modeling new, even non-existing, platforms as long as coherent performance and power data are available. It consists of the CPU islands of ODROID-XU4 board, adding the FPGA part of the ZCU102 board to enable hardware acceleration. This hypothetical platform makes sense because the CPU island of ODROID-XU4 has more modern CPU IPs compared to the ones on the ZCU102 board, with better performance, power management with more discrete frequencies, and heterogeneous CPUs (aka big-Little) accessible from the operating system. On the other hand, it lacks acceleration capability such as FPGA or a fully programmable GPU [8].

[8]ODROID-XU4 does have a Mali GPU, but with very limited resources and programmability

For the ZCU102 model, the accelerated tasks (SFM, Localization, Lane Detection, and Detection) are enabled for running both on CPUs and on the FPGA. All the other tasks are CPU only. We are assuming that the original EETB data of tasks running on CPUs are scaled up by 1.166 for the ZCU102 model because the maximum frequency of ODROID-xu4 is 1.4 GHz while for ZCU102 CPUs it is 1.2 GHz (1.4/1.2 = 1.166). For the ODROID-XU4, all tasks are CPU only due to the lack of resources for acceleration. We assume that the reported EETB values have been characterized for the ODROID-XU4 CPUs of the big islands, running at their highest frequency. Thus, in this case, no EETB scaling is required. Finally, for the hypothetical platform with ODROID-XU4 CPUs and ZCU102 FPGA, the accelerated tasks are enabled for running both on the CPUs and on the FPGA, while the other tasks are CPU only.

The accelerated tasks do not have an associated IP core to be run on the FPGA. So, we had to assign a reconfiguration time and power consumption creating fictitious IPs representing each task. We have assigned the same reconfiguration time (22.6 ms) and power (1.14 and 1.19 W of idle and busy power, respectively) required by the *mat128* IP core, reported in Section 5.5.6, which does 128x128 matrix multiplication of 64-bit integers and uses about 50% of the FPGA resources.

After optimizing the scenario on the ZCU102 model, the MIQCP solver reported a feasible solution with an end-to-end deadline of at least 71 ms. The CPUs are running at their maximum frequencies, and only task Localization was accelerated. The reason is that, by choosing only one task for acceleration, the FPGA reconfiguration time is zero. Moreover, the task Localization was chosen because it is clearly the bottleneck in the WATERS19 challenge, with 244.8 ms of CPU EETB against 14.5 ms of its accelerated version. All other tasks have a CPU EETB about ten times shorter than the Localization task.

Optimizing the scenario on the ODROID-XU4 model, which has only CPUs, we could only find a feasible solution if the end-to-end deadline is 350 ms due to a much longer execution time when running the tasks on CPU only. On the other hand, the hypothetical platform found the placement with a deadline of 61 ms, using only the big island at its maximum frequency (the LITTLE island was not used). Similar to the solution found for the ZCU102 model, only the task Localization was accelerated for the same reasons mentioned before.

5.5.6 Implementing FPGA-Accelerated Scenarios on Linux. To validate the results obtained when optimizing DAG sets which contain tasks that can be potentially accelerated, we added to our RTDAG tool support for the FRED FPGA acceleration framework [14] described in Section 4. Leveraging this framework, one or multiple tasks in each scenario can be indicated to be accelerated on FPGA, dynamically reconfiguring the platform if necessary. Since the ODROID-XU4 lacks an FPGA accelerator, we selected a Xilinx UltraScale+ ZCU102 board with FPGA acceleration as our target platform for these experiments.

These tasks are implemented similarly to the their pure-software counterparts, in that they each wait for the termination of all their predecessors before waking up, starting a computation intensive section, and signaling all their successors for completion. The key difference of the hardware tasks is that they implement the computation intensive part by performing an accelerated call via the FRED client library, which signals the FRED daemon that it has to start the execution of a task on the FPGA. As mentioned in Section 4, due to the daemon-client architecture of our selected acceleration framework, we had to raise the scheduling priority of the FRED daemon above the SCHED_DEADLINE scheduling class. We used this approach also for the software part of the hardware-accelerated tasks part of RTDAG; this way, whenever a pure software task wakes up the thread that's supposed to be hardware-accelerated, there is the smallest delay possible between the request of starting the FPGA-accelerated task and the actual request to the FRED daemon to run the requested task on the FPGA.

Since tasks scheduled using SCHED_DEADLINE may be preempted whenever the FRED daemon is selected for execution, effectively the FRED daemon can be considered an interrupt which may fire and interfere with SCHED_DEADLINE tasks. For this reason, it can be addressed in the same way we address IRQ interference in Section 5.5.2. We performed some targeted experiments to measure the interference that any invocation of an accelerated task via the FRED daemon may impose on another task to be no greater than 0.9 ms at the lowest frequency available for the platform (300 MHz). This interference is bound to reduce at higher frequencies, up to no more than 0.4 ms at the highest CPU frequency (1.2 GHz). It should be noted however that any long software tasks may be subject to this kind of interference multiple times, if multiple hardware tasks are activated during its execution and the FRED daemon executes on the same core. This kind of interference can of course be eliminated if the FRED daemon is statically bound to a core that is never used by any software task for execution. If deemed necessary, our optimization framework may be configured to reserve a core for this and/or other activities on the system, albeit this would adversely impact the optimality of the found solutions, so it was not used in our experiments.

Finally, after making sure that the executions were actually working, the results were obtained with FRED daemon compiled in quiet mode, not generating any report. Similarly, the Linux kernel were also configured in low verbosity mode.

Experimental Results on UltraScale+. Here, we report about some experimentation performed with FPGA hardware acceleration, performed using Linux running on a Xilinx UltraScale+ ZCU102 board. Out of the solvers described in Section 4, only our MIQCP formulation (and the MILP derived from it) supports the specification of FPGA or GPU accelerators as part of the target platform specification, while our heuristic solvers only support big.LITTLE scenarios. For this reason, our BB-Search and TIF heuristic solvers are not considered in this section.

We solved 50 random DAG scenarios where two parallel independent tasks can be executed either on the CPUs or on the FPGA. The reference hardware tasks are a 64x64 and a 128x128 64-bit matrix multiplication, called *mat64* and *mat128* IPs cores. These two IPs are configured into a single dynamically reconfigurable slot. This means that the size of their partial bitstreams is determined by the size of the biggest IP, which takes, in this setup, about 51% of the total FPGA resources. This also means that the partial reconfiguration process takes the same time for reconfiguring both IPs, in this case 22.6 ms. Both *mat64* and *mat128* IPs present considerable speedups compared to their pure-software counterpart implementations, respectively 2.12 and 7.87 times faster in FPGA when selected individually.

We have performed experiments with 50 random DAG scenarios with two hardware tasks. Most of these scenarios model single-DAG applications, but there are also some more complex systems made of two independent DAGs. Each scenario is run hundreds of times such that the sum of their execution times reaches tens of seconds, meaning that each scenario runs roughly from 300 to 600 times. In total, we performed 23609 DAG runs. Note that, although there are two hardware tasks, the optimizer might use only one. Most of the scenarios in fact use only one IP because, by doing so, it avoids the dynamic reconfiguration time of 22.8 ms. The 50 scenarios were optimized using our MIQCP formulation and ran on the Xilinx board. 375 deadlines misses were reported out of 23609 DAG runs. All misses were from the same scenario. So, we re-optimized this failing scenario using the MIQCP slack mode, where we add the power budget (obtained by the first optimization in normal MIQCP mode). In this mode, the optimizer maximizes the minimum slack among all tasks, creating a more robust solution. Finally, we ran it again on the ZCU102 platform, where no deadline misses were observed in all the 23609 DAG executions tested in the paper.

6 CONCLUSIONS AND FUTURE WORK

In this paper, a method for modeling and optimizing the platform configuration and the placement and acceleration of real-time DAG tasks onto heterogeneous platforms has been tackled. We used an MIQCP-based formalization of the approach that allows for using an optimum solver for relatively small problems, whereas for large ones we proposed heuristics achieving different trade-offs between optimality and execution times of the solver. The approach was validated via simulation, to have a validation of the theoretical optimization framework, then experimentally on two real boards, one with a big.LITTLE architecture, the other one with FPGA acceleration. The presented results reveal a good consistency of the gathered timing behavior and power consumption data with our theoretical expectations obtained with our optimization framework.

In the future, this work could be extended along various directions. First, acceleration of tasks onto GP-GPUs needs to be properly integrated into the model. This does not seem immediate due to the presence of several run-time threads, for example when trying to use OpenCL on the ODROID-XU4. Second, during the realization of this work, we bumped into some limitations of the FRED architecture we used for FPGA acceleration exploiting DPR capabilities: first, its server-based architecture adds context-switch overheads that might be worked around by re-engineering the FRED run-time so to avoid the presence of the FRED daemon in the critical per-acceleration-request path; second, its current static design for plugging acceleration IPs may be tweaked to improve the power consumption of the managed slots while idle. Moreover, in case multiple consecutive tasks are mapped by the optimizer to FPGA accelerators, the current FRED architecture enforces communication among them in software via the FRED daemon, but this interaction might be much faster by re-engineering FRED internals drawing from data-flow principles [36], for example.

Additionally, the presented MIQCP-based optimization approach is the only one that implements at the moment the complete set of features described in Section 4, including the possibility to optimize for power only, or for slack only under a budget constraint, or for both, or even downgrading the problem to a MILP with pre-assigned intermediate deadlines, and the possibility to consider platforms with hardware accelerators. Its only drawback is the one to require a commercial solver, when using the full MIQCP approach. On the other hand, the presented BB-Search and TIF heuristics do not support such a wide set of features, so it would be useful to extend them adding also these further capabilities. This will be especially useful since we plan to release all the software described in this paper as open-source that can be conveniently leveraged by other researchers for further future research on the topic[9].

ACKNOWLEDGMENTS

This work has been supported by the EU H2020 project AMPERE under the grant agreement no. 871669.

REFERENCES

[1] T.F. Abdelzaher and Chenyang Lu. 2001. Schedulability analysis and utilization bounds for highly scalable real-time services. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium.* 15–25. https://doi.org/10.1109/RTTAS.2001.929862

[2] Luca Abeni and Giorgio Buttazzo. 1998. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium.* Madrid, Spain.

[3] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. 2020. Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT).* 1–4. https://doi.org/10.1109/DFT50435.2020.9250750

[4] T.A. AlEnawy and H. Aydin. 2005. Energy-aware task allocation for rate monotonic scheduling. In *11th IEEE Real Time and Embedded Technology and Applications Symposium.* 213–223. https://doi.org/10.1109/RTAS.2005.20

[9]See: https://retis.santannapisa.it/~tommaso/papers/acmtecs23.php.

[5] Gabriele Ara, Tommaso Cucinotta, and Agostino Mascitti. 2022. Simulating Execution Time and Power Consumption of Real-Time Tasks on Embedded Platforms. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22)*. Association for Computing Machinery, New York, NY, USA, 491–500. https://doi.org/10.1145/3477314.3507030

[6] Federico Aromolo, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. 2021. Event-Driven Delay-Induced Tasks: Model, Analysis, and Applications. In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 53–65. https://doi.org/10.1109/RTAS52030.2021.00013

[7] Ekain Azketa, Juan P. Uribe, Marga Marcos, Luis Almeida, and J. Javier Gutierrez. 2011. Permutational Genetic Algorithm for the Optimized Assignment of Priorities to Tasks and Messages in Distributed Real-Time Systems. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. 958–965. https://doi.org/10.1109/TrustCom.2011.132

[8] Alessio Balsini, Tommaso Cucinotta, Luca Abeni, Joel Fernandes, Phil Burk, Patrick Bellasi, and Morten Rasmussen. 2019. Energy-efficient low-latency audio on android. *Journal of Systems and Software* 152 (2019), 182–195. https://doi.org/10.1016/j.jss.2019.03.013

[9] Alessio Balsini, Luigi Pannocchi, and Tommaso Cucinotta. 2019. Modeling and Simulation of Power Consumption and Execution Times for Real-Time Tasks on Embedded Heterogeneous Architectures. *SIGBED Rev.* 16, 3 (nov 2019), 51–56. https://doi.org/10.1145/3373400.3373408

[10] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-Aware Scheduling for Real-Time Systems: A Survey. *ACM Trans. Embed. Comput. Syst.* 15, 1, Article 7 (jan 2016), 34 pages. https://doi.org/10.1145/2808231

[11] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. 2015. The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *27th Euromicro Conference on Real-Time Systems*. 222–231. https://doi.org/10.1109/ECRTS.2015.27

[12] Sanjoy K. Baruah. 2003. Dynamic- and Static-Priority Scheduling of Recurring Real-Time Tasks. *Real-Time Syst.* 24, 1 (jan 2003), 93–128. https://doi.org/10.1023/A:1021711220939

[13] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. 2010. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers. In *31st IEEE Real-Time Systems Symposium*. 14–24. https://doi.org/10.1109/RTSS.2010.23

[14] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. 2016. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 1–12.

[15] OpenMP Architecture Review Board. 2021. OpenMP Application Programming Interface, version 5.2.

[16] Antonino Casile, Giorgio Buttazzo, Gerardo Lamastra, and Giuseppe Lipari. 1998. Simulation and Tracing of Hybrid Task Sets on Distributed Systems. In *Proceedings of the IEEE Conference on Real Time Computing Systems and Applications*.

[17] Daniel Casini and Alessandro Biondi. 2022. Placement of Chains of Real-Time Tasks on Heterogeneous Platforms under EDF Scheduling. In *25th Euromicro Conference on Digital System Design (DSD)*. Maspalomas, Gran Canaria, Spain.

[18] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. 2019. Analyzing Parallel Real-Time Tasks Implemented with Thread Pools. In *Proceedings of the 56th Annual Design Automation Conference* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 92, 6 pages. https://doi.org/10.1145/3316781.3317771

[19] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. 2019. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:23. https://doi.org/10.4230/LIPIcs.ECRTS.2019.6

[20] Daniel Casini, Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale. 2022. Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration. *Journal of Systems Architecture* 124 (2022), 102416. https://doi.org/10.1016/j.sysarc.2022.102416

[21] Daniel Casini, Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale. 2022. Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration. *Journal of Systems Architecture* 124 (2022), 102416.

[22] H. Chetto, M. Silly, and T. Bouchentouf. 1990. Dynamic Scheduling of Real-Time Tasks under Precedence Constraints. *Real-Time Syst.* 2, 3 (sep 1990), 181–194. https://doi.org/10.1007/BF00365326

[23] Youngeun Cho, Dongmin Shin, Jaeseung Park, and Chang-Gun Lee. 2021. Conditionally Optimal Parallelization of Real-Time DAG Tasks for Global EDF. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. 188–200. https://doi.org/10.1109/RTSS52674.2021.00027

[24] Alessandro Cimatti, Sara Corfini, Luca Cristoforetti, Marco Di Natale, Alberto Griggio, Stefano Puri, and Stefano Tonetta. 2022. A Comprehensive Framework for the Analysis of Automotive Systems. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 379–389. https://doi.org/10.1145/3550355.3552408

[25] Sébastien Collette, Liliana Cucu, and Joël Goossens. 2008. Integrating Job Parallelism in Real-Time Scheduling Theory. *Inf. Process. Lett.* 106, 5 (may 2008), 180–187. https://doi.org/10.1016/j.ipl.2007.11.014

[26] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. 2012. Adaptive real-time scheduling for legacy multimedia applications. *ACM Transactions on Embedded Computing Systems* 11, 4 (Dec. 2012), 1–23. https://doi.org/10.1145/2362336.2362353

[27] Tommaso Cucinotta, Dario Faggioli, and Giacomo Bagnoli. [n. d.]. Low-Latency Audio on Linux by Means of Real-Time Scheduling. In *Proceedings of the Linux Audio Conference (LAC)*.

[28] Tommaso Cucinotta, Antonio Mancina, Gaetano F. Anastasi, Giuseppe Lipari, Leonardo Mangeruca, Roberto Checcozzo, and Fulvio Rusin'a. 2009. A Real-time Service-Oriented Architecture for Industrial Automation. *IEEE Transactions on Industrial Informatics* (2009).

[29] Tommaso Cucinotta and Luigi Palopoli. 2009. QoS Control for Pipelines of Tasks using Multiple Resources. *IEEE Transactions on Computers* (2009).

[30] Xiaotian Dai. 2022. *dag-gen-rnd: A randomized multi-DAG task generator for scheduling and allocation research.* https://doi.org/10.5281/zenodo.6334205

[31] Robert I. Davis and Alan Burns. 2011. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.* 43, 4, Article 35 (oct 2011), 44 pages. https://doi.org/10.1145/1978802.1978814

[32] Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. 2022. Operating System Noise in the Linux Kernel. *IEEE Trans. Comput.* (2022), 1–12. https://doi.org/10.1109/tc.2022.3187351

[33] Daniel B. de Oliveira, Rômulo S. de Oliveira, and Tommaso Cucinotta. 2020. A thread synchronization model for the PREEMPT_RT Linux kernel. *Journal of Systems Architecture* 107 (2020), 101729. https://doi.org/10.1016/j.sysarc.2020.101729

[34] U.M.C. Devi and J.H. Anderson. 2005. Tardiness bounds under global EDF scheduling on a multiprocessor. In *26th IEEE International Real-Time Systems Symposium (RTSS).* https://doi.org/10.1109/RTSS.2005.39

[35] José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. 2015. A Multi-DAG Model for Real-Time Parallel Applications with Conditional Execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15).* Association for Computing Machinery, New York, NY, USA, 1925–1932. https://doi.org/10.1145/2695664.2695808

[36] Roberto Giorgi and Marco Procaccini. 2019. Bridging a Data-Flow Execution Model to a Lightweight Programming Model. In *2019 International Conference on High Performance Computing & Simulation (HPCS).* 165–168. https://doi.org/10.1109/HPCS48598.2019.9188183

[37] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.* 48, 2, Article 32 (nov 2015), 36 pages. https://doi.org/10.1145/2830555

[38] Simon Holmbacka and Jörg Keller. 2017. Workload Type-Aware Scheduling on big.LITTLE Platforms. In *Algorithms and Architectures for Parallel Processing*, Shadi Ibrahim, Kim-Kwang Raymond Choo, Zheng Yan, and Witold Pedrycz (Eds.). Springer International Publishing, Cham, 3–17.

[39] Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. 2021. The HPC-DAG Task Model for Heterogeneous Real-Time Systems. *IEEE Transactions on Computers* 70, 10 (Oct 2021), 1747–1761. https://doi.org/10.1109/TC.2020.3023169

[40] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX Annual Technical Conference (USENIX ATC).* USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/timegraph-gpu-scheduling-real-time-multi-tasking-environments

[41] Tei-Wei Kuo, Jian-Jia Chen, Yuan-Hao Chang, and Pi-Cheng Hsiu. 2018. Real-Time Computing and the Evolution of Embedded System Designs. In *IEEE Real-Time Systems Symposium (RTSS).* 1–12. https://doi.org/10.1109/RTSS.2018.00011

[42] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. 2010. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *31st IEEE Real-Time Systems Symposium.* 259–268. https://doi.org/10.1109/RTSS.2010.42

[43] Juri Lelli, Dario Faggioli, Tommaso Cucinotta, and Giuseppe Lipari. 2012. An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software* 85, 10 (2012), 2405–2416. https://doi.org/10.1016/j.jss.2012.05.048 Automated Software Evolution.

[44] C. L. Liu and J. Layland. 1973. Scheduling alghorithms for multiprogramming in a hard real-time environment. *J. ACM* 20, 1 (1973).

[45] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.* 52, 3, Article 56 (jun 2019), 38 pages. https://doi.org/10.1145/3323212

[46] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. 2015. WCET(m) Estimation in Multi-core Systems Using Single Core Equivalence. In *2015 27th Euromicro Conference on Real-Time Systems.* 174–183. https://doi.org/10.1109/ECRTS.2015.23

[47] Agostino Mascitti, Tommaso Cucinotta, and Luca Abeni. 2020. Heuristic partitioning of real-time tasks on multi-processors. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC).* 36–42. https://doi.org/10.1109/ISORC49007.2020.00015

[48] A.K. Mok and D. Chen. 1997. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering* 23, 10 (Oct 1997), 635–645. https://doi.org/10.1109/32.637146

[49] Sanjay Moulik, Rishabh Chaudhary, and Zinea Das. 2020. HEARS: A heterogeneous energy-aware real-time scheduler. *Microprocessors and Microsystems* 72 (2020), 102939. https://doi.org/10.1016/j.micpro.2019.102939

[50] Sanjay Moulik, Zinea Das, Rajesh Devaraj, and Shounak Chakraborty. 2021. SEAMERS: A Semi-partitioned Energy-Aware scheduler for heterogeneous MulticorE Real-time Systems. *Journal of Systems Architecture* 114 (2021), 101953. https://doi.org/10.1016/j.sysarc.2020.101953

[51] Andrew Nelson, Orlando Moreira, Anca Molnos, Sander Stuijk, Ba Thang Nguyen, and Kees Goossens. 2011. Power Minimisation for Real-Time Dataflow Applications. In *14th Euromicro Conference on Digital System Design.* 117–124. https://doi.org/10.1109/DSD.2011.19

[52] Luigi Palopoli, Giuseppe Lipari, Luca Abeni, Marco Di Natale, Paolo Ancilotti, and Fabio Conticelli. 2001. A Tool for Simulation and Fast Prototyping of Embedded Control Systems. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems* (Snow Bird, Utah, USA) *(OM '01).* Association for Computing Machinery, New York, NY, USA, 73–81. https://doi.org/10.1145/384198.384209

[53] Bo Peng, Nathan Fisher, and Thidapat Chantem. 2016. MILP-Based Deadline Assignment for End-to-End Flows in Distributed Real-Time Systems. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems* (Brest, France) *(RTNS '16).* Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2997465.2997498

[54] Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Qui nones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. 2015. P-SOCRATES: A parallel software framework for time-critical many-core systems. *Microprocessors and Microsystems* 39, 8 (2015), 1190–1203. https://doi.org/10.1016/j.micpro.2015.06.004

[55] Eduardo Quiñones, Sara Royuela, Claudio Scordino, Paolo Gai, Luis Miguel Pinho, Luis Nogueira, Jan Rollo, Tommaso Cucinotta, Alessandro Biondi, Arne Hamann, Dirk Ziegenbein, Hadi Saoud, Romain Soulat, BjÃ¶rn Forsberg, Luca Benini, Gianluca Mando, and Luigi Rucher. 2020. The AMPERE Project: : A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization. In

2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC). 201–206. https://doi.org/10.1109/ISORC49007.2020.00042

[56] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. 1997. Resource kernels: a resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking 1998*, Kevin Jeffay, Dilip D. Kandlur, and Timothy Roscoe (Eds.), Vol. 3310. International Society for Optics and Photonics, SPIE, 150 – 164. https://doi.org/10.1117/12.298417

[57] Falk Rehm, Dakshina Dasari, Arne Hamann, Michael Pressler, Dirk Ziegenbein, Joerg Seitter, Ignacio Sa nudo, Nicola Capodieci, Paolo Burgio, and Marko Bertogna. 2021. Performance modeling of heterogeneous HW platforms. *Microprocessors and Microsystems* 87 (2021), 104336. https://doi.org/10.1016/j.micpro.2021.104336

[58] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. 2016. Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems* (Brest, France) *(RTNS '16)*. Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/2997465.2997472

[59] Juan M Rivas, J Javier Gutiérrez, J Carlos Palencia, and Michael González Harbour. 2014. Deadline assignment in EDF schedulers for real-time distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 26, 10 (2014), 2671–2684.

[60] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2021. Energy-Aware Scheduling of Multi-Version Tasks on Heterogeneous Real-Time Systems. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 501–510. https://doi.org/10.1145/3412841.3441930

[61] Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2021. YASMIN: A Real-Time Middleware for COTS Heterogeneous Platforms. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3464298.3493402

[62] Gabriele Serra, Gabriele Ara, Pietro Fara, and Tommaso Cucinotta. 2021. ReTiF: A declarative real-time scheduling framework for POSIX systems. *Journal of Systems Architecture* 118 (2021), 102210. https://doi.org/10.1016/j.sysarc.2021.102210

[63] Nicola Serreli, Giuseppe Lipari, and Enrico Bini. 2010. The Distributed Deadline Synchronization Protocol for real-time systems scheduled by EDF. In *IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. 1–8. https://doi.org/10.1109/ETFA.2010.5641357

[64] John A. Stankovic. 1996. Strategic Directions in Real-Time and Embedded Systems. *ACM Comput. Surv.* 28, 4 (dec 1996), 751–763. https://doi.org/10.1145/242223.242291

[65] Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. 2015. OpenMP and timing predictability: A possible union?. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 617–620. https://doi.org/10.7873/DATE.2015.0778

[66] Gang Wang, Wenming Li, and Xiali Hei. 2015. Energy-aware real-time scheduling on Heterogeneous Multi-Processor. In *49th Annual Conference on Information Sciences and Systems (CISS)*. 1–7. https://doi.org/10.1109/CISS.2015.7086884

[67] Falk Wurst, Dakshina Dasari, Arne Hamann, Dirk Ziegenbein, Ignacio Sanudo, Nicola Capodieci, Marko Bertogna, and Paolo Burgio. 2019. System performance modelling of heterogeneous hw platforms: An automated driving case study. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 365–372.

[68] Yang Yu and V.K. Prasanna. 2002. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *Ninth International Conference on Parallel and Distributed Systems*. 341–348. https://doi.org/10.1109/ICPADS.2002.1183422

[69] Houssam-Eddine ZAHAF, Giuseppe Lipari, Smail Niar, and Abou El Hassan Benyamina. 2020. Preemption-Aware Allocation, Deadline Assignment for Conditional DAGs on Partitioned EDF. In *IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1–10. https://doi.org/10.1109/RTCSA50079.2020.9203643

[70] Lilia Zaourar, Massinissa Ait Aba, David Briand, and Jean-Marc Philippe. 2017. Modeling of Applications and Hardware to Explore Task Mapping and Scheduling Strategies on a Heterogeneous Micro-Server System. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 65–76. https://doi.org/10.1109/IPDPSW.2017.123

[71] Lilia Zaourar, Massinissa Ait Aba, David Briand, and Jean-Marc Philippe. 2018. Task management on fully heterogeneous micro-server system: Modeling and resolution strategies. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4798. https://doi.org/10.1002/cpe.4798 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4798 e4798 cpe.4798.

[72] Husheng Zhou, Soroush Bateni, and Cong Liu. 2018. S^3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 190–201. https://doi.org/10.1109/RTAS.2018.00028