Optimizing the Deployment of Real-Time OpenMP Applications for Energy Efficiency

Francesco Paladino[a], Federico Aromolo[b], Luca Abeni[b], Tommaso Cucinotta[b]

[a] *University of California, Berkeley, USA*
[b] *Scuola Superiore Sant'Anna, Pisa, Italy*

Abstract

Designing and deploying real-time computing pipelines efficiently on modern embedded platforms is increasingly challenging due to the growing complexity of hardware architectures, often featuring multi-core processors, frequency scaling capabilities, heterogeneous cores for enhanced power efficiency, and hardware accelerators. OpenMP is a prominent tool for parallelizing applications on multi-core platforms and is gaining increasing adoption in the domain of real-time systems. However, providing sound performance guarantees on the timing behavior of complex parallel computations organized as graph structures on heterogeneous platforms, while achieving optimal or near-optimal energy efficiency, is all but trivial. This paper tackles this problem by proposing a methodology to deploy and analyze both traditional parallel real-time applications and OpenMP parallel applications, modeled as directed acyclic graphs (DAGs) and coexisting on the same heterogeneous platform. Specifically, the approach targets asymmetric multi-core platforms with frequency scaling capabilities, with the aim of minimizing energy consumption while guaranteeing end-to-end latency constraints via schedulability analysis. The proposed approach features an optimal solver based on a mixed-integer quadratic constrained programming formulation, and a computationally efficient heuristic to extract high-quality solutions with reduced solving time. The concept is experimentally validated using randomly generated sets of DAGs, optimized by the two techniques and deployed using an OpenMP-based DAG synthetic benchmark on Linux running on an embedded board. Results demonstrate that the methodology enables energy-efficient deployment of mixed traditional and OpenMP real-time DAG applications while preserving end-to-end latency guarantees.

Keywords: Real-time systems, directed acyclic graphs, OpenMP, schedulability analysis, frequency scaling, energy efficiency.

1. Introduction

Modern embedded platforms are equipped with several powerful processing units (PUs) enabling the execution of complex and heterogeneous applications. Applications developed for multi-core systems take advantage of the high computational power of the hardware platform with multi-threading, so that parts of the code can execute concurrently on multiple cores. The efficient development of parallel applications, however, can be challenging: multiple threads must be carefully orchestrated to enforce data dependencies between tasks, while at the same time leveraging the inherent parallelism of the application and of the underlying computing platform. These challenges are exacerbated by the important requirements on energy efficiency often having critical impact on form-factor design and battery duration of embedded systems[1]. Frameworks and APIs are available to aid developers in the design and development of parallel applications. Most notably, OpenMP [1] is a cross-platform parallel computing API that automates the creation and synchronization of threads at runtime with simple `#pragma` directives in the application code. With the wider availability of heterogeneous computing platforms, OpenMP has gained significant popularity due to its ease of use and compatibility with a wide range of platforms and applications, including real-time and multimedia applications [2, 3, 4, 5, 6].

In these domains, applications developed with OpenMP are commonly modeled using directed acyclic graphs (DAGs) [6, 2, 7]. The DAG model highlights tasks and their dependencies to determine which tasks can execute in parallel on different PUs, enabling the representation of complex dependency topologies through OpenMP task constructs and dependency specifications (i.e., `depend` clauses), beyond the loop-level parallelism typically expressed with work-sharing constructs such as `parallel for`. Real-time applications leveraging OpenMP may also include multiple independent DAGs of computations that share the same set of platform resources, each characterized by distinct end-to-end timing constraints (i.e., a deadline on the response time of each DAG) to guarantee predictable real-time performance. Parts of different DAGs may coexist on the same PU, generating mutual interference due to resource sharing. Moreover, OpenMP DAGs can coexist with other types of DAGs, implemented by leveraging programming constructs for concurrency and synchronization and coordinated according to different policies [8]. This coexistence complicates two fundamental problems: **(i)** analyzing the timing behavior of an

Email addresses: francesco.paladino@berkeley.edu (Francesco Paladino), federico.aromolo@santannapisa.it (Federico Aromolo), luca.abeni@santannapisa.it (Luca Abeni), tommaso.cucinotta@santannapisa.it (Tommaso Cucinotta)

[1] Note that general-purpose and high-performance computing are areas where power awareness/minimization is a growing concern as well, due to the associated monetary/operational costs and to the general attention to green computing and sustainability.

application composed of multiple DAGs, which is essential for verifying the satisfaction of timing constraints, and **(ii)** determining a suitable allocation of the tasks of each DAG to the available PUs to optimize resource usage and system performance while satisfying all timing constraints.

These challenges are further amplified when realistic platform models are considered, including heterogeneous processors and frequency scaling capabilities. Specifically, powerful embedded platforms can consume substantial energy when executing parallel workloads, which may be a limiting factor for battery-powered devices. More generally, excessive energy consumption leads to increased operational costs and thermal management challenges in both embedded systems and data centers. Dynamic voltage and frequency scaling (DVFS) capabilities play a key role in managing the trade-off between speed and energy, allowing cores to operate at lower or higher frequencies, thereby reducing or increasing power consumption, respectively. For example, asymmetric platforms such as Arm big.LITTLE or DynamIQ architectures are characterized by ISA-compatible CPU islands (among which processes can be freely migrated), where high-performance "big" cores execute tasks quickly but consume more power, while low-power "LITTLE" cores cause tasks to take longer to execute. Similar functionality is available on Intel Alder Lake architectures, with P-cores and E-cores [9]. Placing tasks on the LITTLE cores and lowering the operating frequencies of the platform can reduce the power consumption of the application; however, this may compromise its timing requirements expressed as end-to-end deadlines of the DAGs, as tasks will run more slowly. Thus, a central design challenge for applications running on energy-constrained platforms is to determine a suitable deployment strategy that keeps power consumption to a minimum while meeting timing requirements [10, 11].

To the best of our knowledge, there exist no specialized methods providing a comprehensive schedulability analysis for real-time software composed of multiple OpenMP DAGs, particularly when explicitly accounting for energy management capabilities or when considering coexistence with other types of DAG-based applications or heterogeneous tasks sharing the same computational resources. Specifically, existing approaches either focus on single DAGs, homogeneous platforms, or do not account for energy-aware deployment under frequency scaling. To fill this gap, this paper proposes a complete methodology for the design, analysis, and deployment of real-time DAG-based applications, including those developed with OpenMP, on heterogeneous big.LITTLE-like platforms with frequency scaling capabilities. First, a specialized model and schedulability analysis is presented for real-time software including multiple OpenMP DAGs sharing the same PUs. Then, an optimization approach is presented for DAG-based applications leveraging OpenMP, explicitly accounting for the presence of multiple DAGs and energy management capabilities. The objective is to configure the frequency scaling settings of the platform and the placement of tasks to minimize the power consumption of the system while guaranteeing the end-to-end deadlines of DAGs. Two approaches will be presented: **(i)** an optimal strategy based on a mixed-integer quadratic constrained programming (MIQCP) formulation, which integrates schedulability analysis in the form of constraints, and **(ii)** a heuristic solver which is significantly faster than the former, but which yields results of comparable quality. Experimental results are presented to validate the configurations obtained with the proposed approaches using randomly generated workloads, optimized for, and deployed onto, a commercial embedded Linux platform.

The rest of this paper is structured as follows: Section 2 provides an overview of related works, Section 3 introduces the considered system model, Section 4 presents the proposed analysis for DAG-based applications, Section 5 details the proposed optimization algorithms, Section 6 compares the two approaches and validates the obtained configurations on an ODROID-XU4 board, and Section 7 concludes the paper.

2. Related work

Previous works investigated the scheduling of real-time applications with OpenMP. The design of the OpenMP tasking model [12] has been extensively discussed by the OpenMP tasking subcommittee, with the objective of efficiently exploiting the parallelism of applications. Duran et al. [13] compared the different available task scheduling strategies in OpenMP, among which there are *work-first* and *breadth-first*, both in the case of `tied` tasks, i.e., pinned to a thread, and of `untied` tasks. `tied` tasks have been analyzed by Sun et al. [14], who developed a real-time scheduling algorithm and related analysis. The algorithm avoids tying too much workload on the same worker thread to efficiently exploit parallelism. Furthermore, extensions to OpenMP runtime were developed, aiming at improving real-time performance and supporting operation in safety-critical and high-reliability systems. Silvestri et al. [15] designed an extension to the GNU implementation of OpenMP based on a user-space library and a dedicated Linux kernel module, to guarantee work-conservativeness and a better management of priorities. Yu et al. [16] developed a framework for the Clang/LLVM implementation of OpenMP runtime which considerably reduces the runtime overhead of the tasking model by improving the resolution of dependencies between OpenMP tasks. Vargas et al. [3] proposed a lightweight OpenMP runtime for embedded systems, focusing on minimizing runtime and memory overheads while supporting the tasking model. Royuela et al. [4] proposed modifications and requirements for the OpenMP specification and runtimes to guarantee safety properties in safety-critical embedded applications. The application of OpenMP in embedded domains was also further explored by Royuela et al. [5] by supporting the OpenMP tasking model in Ada applications. El Maach et al. [17] proposed leveraging dynamic OpenMP task variants (i.e., alternative task implementations) to enable energy-aware scheduling. Native support for periodic real-time tasks in OpenMP was also investigated [18, 19]. Furthermore, Serrano et al. [20] studied the timing characteristics of the OpenMP 4 tasking model, showing how tied and untied tasks influence execution predictability in real-time systems. An analysis also exists for OpenMP applications running on heterogeneous platforms [21], as OpenMP

allows executing tasks on hardware accelerators like GPUs. However, all of these analysis target software systems including a single DAG corresponding to a single OpenMP application with dedicated access to all available PUs in the platform. In contrast to existing approaches, the analysis proposed in this paper supports a broader scope by targeting an arbitrary number of OpenMP DAGs executing on heterogeneous platforms with frequency scaling capabilities (Section 4).

More broadly, parallel task models in the theory of real-time scheduling capture the scheduling behavior of complex parallel workloads by considering their internal topology in terms of precedence constraints using graph structures, going beyond the traditional sporadic task model for independent tasks by Liu and Layland [22] and graph-based models for uniprocessor systems [23, 24]. The *fork-join* model [25] considers an interleaving of sequential and parallel execution, with a synchronization occurring at the boundary of each segment, thus effectively representing fork-join computing topologies. The *sporadic DAG* task model [26] supports more general parallel structures where the precedence constraints between the subtasks are represented with a DAG topology, in similar fashion to the modeling approach considered in this paper (see Section 3), and has been studied under both global and partitioned scheduling. Notably, later works demonstrated that the sporadic DAG task model can capture the scheduling behavior of common parallel programming models, including OpenMP [2, 27]. The presence of precedence constraints in parallel DAG scheduling significantly increases the complexity of schedulability analysis, due to their impact on workload execution dynamics. Execution delay effects in DAG scheduling have been modeled using self-suspending task models [28, 29, 30, 31].

The *deadline splitting* approach has been proposed in previous work [32, 33, 34] to schedule parallel real-time tasks by dividing the end-to-end application deadline into a set of local deadlines for each task in the precedence chain. All such techniques are compatible with DAG-based applications leveraging specific combinations of scheduling paradigms and topologies. Although they serve as important foundational modeling and analysis methods for general DAGs, they are not directly applicable for the analysis OpenMP DAG tasks, which feature specific tasking and scheduling models. In this work, we show how to integrated this approach with OpenMP.

Several works focused on optimizing energy efficiency in the scheduling of real-time tasks, with specific efforts targeting multiprocessor and distributed platforms featuring frequency scaling and complex memory hierarchies [35, 36, 37, 38]. Frequency scaling capabilities were leveraged to minimize the energy requirements of real-time applications with fine-tuning strategies [39, 40]. Specifically, optimization strategies based on integer linear programming [41], MIQCP [8], or approximate algorithms based on greedy heuristics [10] were leveraged to determine optimal frequency scaling configurations. However, none of these works address the combined issue of optimizing the energy efficiency of the overall system by jointly managing the placement of OpenMP DAGs on heterogeneous platforms and configuring the available frequency scaling capabilities.

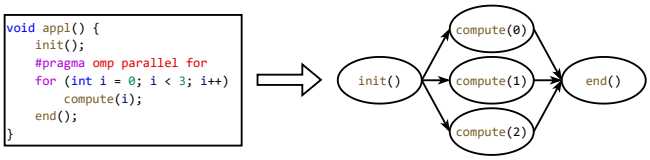


Figure 1: Example of modeling of an OpenMP application leveraging the `parallel for` construct as a DAG with a fork-join structure.

3. System model

This work considers a system comprising a set of DAG applications, optionally leveraging the OpenMP API to facilitate parallelization of the program, scheduled on a heterogeneous computing platform with DVFS capabilities. In this section, we characterize and model the scheduling of OpenMP DAGs and generic DAG-based applications, as well as the characteristics of the reference hardware platform.

3.1. Modeling OpenMP scheduling

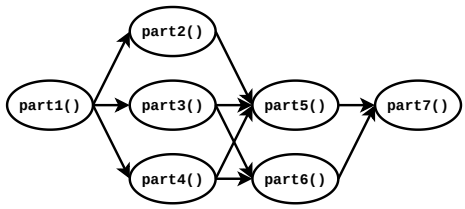
In programs leveraging OpenMP, ad hoc `#pragma` directives processed by the compiler allow for specifying which code segments can be parallelized for faster computation. Then, the OpenMP runtime automatically manages a pool of threads, scheduled by the underlying operating system, that execute the parallel code segments, dealing also with the needed synchronization, transparently to the programmer. OpenMP was originally created to automatically parallelize the iterations of `for` loops with the directive `#pragma omp parallel for`, notably in the very common case where each iteration can be handled independently from the others, and just simple reduction operations are needed to aggregate the partial results computed independently in the iterations, to produce the final result. Recent versions added more and more features and options, including the support for explicit declaration of units of work called *OpenMP tasks* that can be executed in parallel, and the dependencies among them, to ensure the same results that would be obtained with a sequential execution of the program. This is supported with the `#pragma omp task` construct, and its option `depend`.

Applications developed with `parallel for` or with `task` and `depend` can be modeled as DAGs [6, 2]. Figure 1 shows a simple example of parallel application developed with OpenMP and its corresponding DAG model. In the example, after the execution of the `init` function, the `#pragma` directive instructs the OpenMP environment to create three OpenMP tasks to parallelize the three iterations of the `for` loop. The three instances of the `compute` function can execute in parallel, each with a different parameter. The main task waits for the completion of all three before executing the `end` function. The resulting topology is that of a fork-join graph.

Figure 2a shows an example of OpenMP application using the `task` construct with the related `depend` option to express a more general task dependency graph. Specifically, the program starts with a single thread (the *master* thread) that encounters a sequence of `task` constructs. When the `task` construct is encountered in the master thread, a new task instance is created.

```
1  #pragma omp parallel
2  {
3  #pragma omp master
4  {
5    #pragma omp task depend(out:x1)
6    { part1(); }
7    #pragma omp task depend(in:x1, out:x2)
8    { part2(); }
9    #pragma omp task depend(in:x1, out:x3)
10   { part3(); }
11   #pragma omp task depend(in:x1, out:x4)
12   { part4(); }
13   #pragma omp task depend(in:x2,x3,x4, out:x5)
14   { part5(); }
15   #pragma omp task depend(in:x3,x4, out:x6)
16   { part6(); }
17   #pragma omp task depend(in:x5,x6, out:x7)
18   { part7(); }
19 }}
```

(a)



(b)

Figure 2: Example of OpenMP program leveraging the `task` construct and its option `depend` to define dependencies (a) and corresponding DAG model (b).

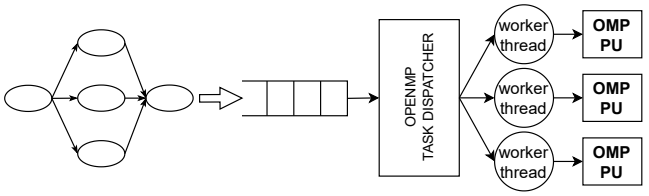


Figure 3: Schematic view of the scheduling of OpenMP DAGs performed by the runtime.

The task instance can be assigned for execution to any available OpenMP thread as soon as all its input dependencies (specified with `in` in the `depend` clause) are satisfied. A dependency on a variable is considered satisfied when all tasks with an output dependency on that variable (i.e., a variable defined as `out` in the `depend` clause of that task) have completed. The corresponding DAG model is shown in Figure 2b.

OpenMP tasks are executed by the OpenMP runtime. The stages of OpenMP scheduling are depicted in Figure 3. When an OpenMP task instance included in the DAG is ready for execution due to its precedence constraints being satisfied, it is inserted into a ready queue. The ready queue is managed by a dispatcher unit, which assigns task instances to a set of *worker threads*, each managing OpenMP execution on a specific processing unit.

The dispatch policy is not standard and depends on the specific OpenMP implementation. The `libgomp` [42] runtime within the GCC compiler uses a shared *queue* where all ready tasks are pushed. When an OpenMP worker thread becomes idle, it pops a task from this queue and executes it (each OpenMP process uses a separate thread-pool and shared tasks queue). LLVM with its front-end C compiler Clang provides instead an OpenMP runtime [43] where each worker thread has its own local queue of ready tasks to execute and, whenever a thread drains its local queue, it steals a task from the others with a *work-stealing* policy.

This paper considers the following set of assumptions relating to the structure of OpenMP programs and properties of OpenMP tasks relevant to this paper.

1. OpenMP applications are assumed to have a static topology (in terms of OpenMP tasks and their dependencies) described by a DAG known beforehand.
2. OpenMP tasks do not invoke any blocking APIs; hence, they *run to completion*.
3. Despite OpenMP supporting hardware accelerators like GPUs, the model does not exploit such possibility, as the hardware platform targeted by this paper provides only CPUs.
4. When tasks are declared, no priority is explicitly assigned to them; therefore, all tasks have the same default priority. Tasks are dispatched to the worker threads following the `libgomp` runtime (part of the GCC compiler): once an OpenMP worker thread becomes idle, the OpenMP task dispatcher assigns the next task from the queue (in FIFO order) to that worker.

3.2. Software model

The software in the scope of this paper consists of n_T tasks $\Gamma = \{\tau_i\}_{i=1}^{n_T}$. The tasks in Γ are partitioned into n_G applications modeled as independent periodic directed acyclic graphs (DAGs) and collected in the set $G \triangleq \left\{G_j\right\}_{j=1}^{n_G}$. Periodic DAGs are suitable to model real-time applications, such as multimedia or control applications, where data is processed by several tasks in a producer-consumer topology, and an output of the whole process is expected within an end-to-end deadline. DAGs also originate from parallelism transparent to the programmer when developing programs with OpenMP, where portions of the code are declared as parallel and the OpenMP runtime automatically handles the implementation of the parallelism during the execution. Therefore, DAGs in G are in turn partitioned into two subsets: those that model OpenMP parallel applications belong to the part G^{omp}, while those that are regular DAG-based applications to G^{reg}. Formally, $G^{omp} \cup G^{reg} = G, G^{omp} \cap G^{reg} = \emptyset$.

Each task τ_i is characterized by:

- a reference estimated execution time bound (EETB) C_i, being an upper-bound to its execution time when placed on a reference processing unit operating at a reference frequency ϕ_{ref}, chosen to be the highest for that unit;
- a non-scalable part of the EETB C_i^{ns} constituting the time spent during cache misses, that is not affected by the operating frequency and the capacity of the processing unit;

- a *release time* $r_{i,k}$ of the k-th instance of τ_i and the corresponding finishing time $f_{i,k}$.

Every DAG is a tuple $G_j = (\Gamma_j, \mathcal{E}_j, T_j, D_j)$, where:

- Γ_j is the set of tasks of G_j, with $\{\Gamma_j\}$ being a partition of the tasks in Γ;
- $\mathcal{E}_j$ is a set of pairs of tasks in Γ_j that model their dependencies as directed edges: $(\tau_1, \tau_2) \in \mathcal{E}_j$ implies that each activation of τ_2 can only start once the same activation of τ_1 completed; $\mathcal{E}_j$ is a fully-connected acyclic topology with one starting task τ_{s_j} and one ending task τ_{e_j};
- T_j is the minimum interarrival time between activations of the starting task τ_{s_j}:

$$\forall k,\ r_{s_j,k+1} \geq r_{s_j,k} + T_j; \tag{1}$$

- D_j is the end-to-end relative deadline of the DAG, which constrains the time between the release time of τ_{s_j} and the finishing time of τ_{e_j}:

$$\forall k,\ f_{e_j,k} \leq r_{s_j,k} + D_j. \tag{2}$$

The k-th activation (or *job*) of a task $\tau_i \in \Gamma_j$ is constrained by the DAG topology, namely it can only execute after the k-th activations of *all* its predecessors, as described by $\mathcal{E}_j$, have completed:

$$r_{i,k} = \max_{\tilde{i} \in \Gamma_j | (\tilde{i},i) \in \mathcal{E}_j} \left\{ f_{\tilde{i},k} \right\} \quad \forall i \in \Gamma_j. \tag{3}$$

In this work, DAGs are assumed to have end-to-end deadlines lower than their periods:

$$D_j \leq T_j,\ \forall G_j \in G. \tag{4}$$

In addition to the above, each starting task τ_{s_j} of a DAG G_j can release a new job *only* when the previous activation of G_j has completed its execution. Therefore, DAG instances cannot overlap; within a DAG period T_j, only one instance of G_j can be running.

We deal with the end-to-end deadlines of real-time DAGs by adopting a *deadline splitting* paradigm [32]: each task τ_i in a DAG G_j is assigned an *intermediate relative deadline* d_i, which is computed by our configuration and placement optimization strategy. However, the duration of the DAG tasks depend on the placement decisions, as the underlying CPUs have different capacity and frequency-switching abilities, influencing deadlines and critical paths.

3.3. Platform model

The hardware platform on which the applications run provides n_p processing units (PUs) in the set $P = \{\psi_1, ..., \psi_{n_p}\}$. P is partitioned into n_s disjoint subsets $I = \{I_1, ..., I_{n_s}\}$ called *islands*. The function $\rho(\tau) : \Gamma \rightarrow I$ maps tasks to islands: $\rho(\tau_i) = I_s$ means that τ_i executes on a PU of island I_s. The function $\chi(\tau) : \Gamma \rightarrow P$ specifies on which PU the task runs. The PUs of each island I_s are in turn partitioned into two disjoint subsets: I_s^{omp} which can execute *solely* OpenMP DAGs in G^{omp}, and I_s^{reg} on which *only* regular DAGs in G^{reg} can run. Each island supports frequency scaling and can run at n_f different frequencies, called operating performance points (OPPs), selected in the discrete set $\Phi_s = \{\phi_{s,1}, ..., \phi_{s,n_f}\}$. The real number *capacity* $\xi_s \in [0, 1]$ characterizes each island I_s, and is interpreted as its maximum speed of computation: an island with capacity 0.5 is two times slower than one with capacity 1. When an island I_s is configured at frequency $\phi_{s,m}$, each of its PUs contributes to the power consumption of the island in two ways: when busy processing, the PU is associated with a power consumption value $\mathcal{P}^B_{s,m}$; when idle, the PU consumes $\mathcal{P}^I_{s,m}$.

To deal with heterogeneous processing on big.LITTLE-like architectures, we use a different OpenMP runtime deployed on each island I_s, where it spawns a thread-pool of n_s^{WT} worker threads to execute the OpenMP tasks placed on that island; each spawned worker thread is statically pinned to a PU, therefore $n_s^{WT} \leq |I_s|$. $\{n_s^{WT}\}$ are configuration parameters determined by our optimization methodology. PUs having a worker thread pinned to them can *exclusively* execute OpenMP tasks. When a task satisfies all its dependencies, it is ready to be executed, so the OpenMP runtime dispatches it to worker threads. As mentioned in Section 3.1, the dispatch policy follows the GCC dispatch model.

Regular DAGs in G^{reg} are scheduled according to the Earliest Deadline First (EDF) scheduling policy on the PU to which they are assigned, meaning that jobs with smaller absolute deadline are considered as having higher priority, where the absolute deadline of the k-th job of a task τ_i in a DAG G_j is computed as $r_{i,k} + d_i$.

The execution time of a task τ_i follows the model of [44] and is an expression of: the reference EETB of τ_i made of C_i and C_i^{ns}; the capacity ξ_s and the OPP $\phi_{s,m}$ of the island where it runs. Formally:

$$C_{i,s,m} = C_i^{ns} + \frac{C_i - C_i^{ns}}{\xi_s \phi_{s,m}} \phi_{s,ref}. \tag{5}$$

Alternatively, our approach also works if, instead of using Equation (5) to determine the scaled execution times, the values come from profiling the tasks on the platform at every frequency, or from the application of worst-case execution time (WCET) estimation techniques. In the case where the $C_{i,s,m}$ values are the WCETs, then our approach provides hard real-time guarantees. Notice that assuming known estimations for the execution time bounds simplifies the analysis (allowing us to treat $C_{i,s,m}$ as a scalar value). Still, the results presented in this paper can be extended to situations where such bounds are unknown by using probabilistic bound estimations such as the pWCET [45, 46]. We denote by $C_{\Gamma_j} = \sum_{\tau_i \in \Gamma_j} C_i$ and $U_{\Gamma_j} = C_{\Gamma_j}/T_j$ the cumulative reference EETB and the cumulative utilization of the tasks Γ_j in a DAG G_j, respectively.

4. Schedulability analysis

This section presents the proposed analysis for OpenMP DAGs, which builds upon an existing analysis for regular DAGs,

presented in [8]. Both analyses are integrated into our optimization framework (described later in Section 5.1) to support both OpenMP DAGs and regular DAGs.

4.1. Analysis of regular DAG tasks

In the following, we report fundamental analytical concepts relating to the analysis of DAGs, followed by a summary of the schedulability analysis from [8]. Some of these concepts will be later reused and adapted to the case of OpenMP DAGs.

The DAG topology defined in Section 3 restricts the parallelism of the execution of tasks. In fact, as formalized by Equation (3), a task can only start when all its predecessors have completed their execution. Therefore, not all tasks in Γ_j can run at the same time. This topological consideration paves the way for schedulability tests that do not take into account *all* tasks in the DAG, but only those that can possibly run concurrently. The schedulability tests described in the following build upon the definition of *reachability* and *unreachability*.

Definition 1. *Two tasks $\tau_a, \tau_b \in \Gamma_j$, $\tau_a \neq \tau_b$ are* reachable*, i.e., $\tau_a \sim \tau_b$, if there exists a path in G_j connecting them. Formally:*

$$\begin{aligned} \forall \tau_a, \tau_b \in \Gamma_j,\ \tau_a \neq \tau_b\ , \tau_a \sim \tau_b \iff \\ \exists t \in \mathbb{N},\ \tau_{i_1}, \tau_{i_2}, ..., \tau_{i_t} \in \Gamma_j \mid \\ (\tau_{i_1} = \tau_a \wedge \tau_{i_t} = \tau_b) \vee (\tau_{i_1} = \tau_b \wedge \tau_{i_t} = \tau_a), \\ (\tau_{i_l}, \tau_{i_l+1}) \in \mathcal{E}_j\ \forall l = 1, ..., t-1. \end{aligned} \quad (6)$$

Reachable tasks can never be executed concurrently, because there is a dependency relation between them that prevents it. For identifying parallelism in the execution, the opposite concept of *unreachability* is required.

Definition 2. *Two tasks $\tau_a, \tau_b \in \Gamma_j$ are* unreachable*, i.e., $\tau_a \nsim \tau_b$, if no path exists in G_j connecting them.*

In these definitions, it is handy that a task is not reachable from itself, i.e., $\tau_a \nsim \tau_a$. It is important to identify all sets of unreachable tasks that can run in parallel. Given a DAG G_j, V_j is defined as the set of unreachable subsets of tasks belonging to G_j:

$$V_j \triangleq \left\{ S \subset \Gamma_j \mid \forall \tau_a, \tau_b \in S,\ \tau_a \nsim \tau_b \right\}. \quad (7)$$

We define W_j as the subset of V_j with the biggest ("maximal") subsets of unreachable tasks.

Definition 3. *Given a DAG G_j, W_j is the set containing the maximal unreachable task subsets of Γ_j:*

$$W_j \triangleq \left\{ S \in V_j \mid \nexists \tilde{S} \in V_j \, with \, S \subsetneq \tilde{S} \right\}. \quad (8)$$

The set W_j is essentially the set of all possible combinations of tasks in G_j that can happen to be active and running concurrently at any given time instant. In the example DAG of Figure 4, the maximal unreachable task subsets for DAG G_2 are:

$$W_2 = \{\{\tau_6\}, \{\tau_7, \tau_8\}, \{\tau_7, \tau_{10}\}, \{\tau_8, \tau_9\}, \{\tau_9, \tau_{10}\}, \{\tau_{11}\}\}. \quad (9)$$

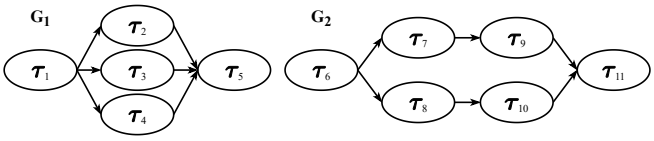


Figure 4: Example of reference application made of two DAGs, G_1 and G_2.

Identifying which tasks can run concurrently in a DAG at any given time instant is key to derive an effective schedulability analysis and to determine how much processing power of PUs the DAG demands and how many tasks populate the scheduling queues of the OpenMP runtime.

In the baseline analysis, compatible with regular DAGs only, tasks in the DAGs are scheduled using an earliest deadline first (EDF) scheduler based on the intermediate deadline d_i computed by the optimizer for each task. Individual tasks in DAGs are not activated independently, nor do they have an individual period, as they follow the activation pattern (the period) and dependencies of the whole DAG. Also, note that each task $\tau_i \in G_j$ has a deadline d_i that cannot exceed the DAG deadline D_j, which is lower than the DAG period T_j. The analysis takes into account the concept of constrained parallelism, as specified in Definition 3. Each maximal unreachable task subset $S \in W_j$ is one of all possible combinations of tasks of a DAG G_j that can be active and running at any given time instant. By construction, the tasks in S are the *only* tasks that can be running in that combination; otherwise, S would not be maximal. Therefore, the sets in W_j do not interfere with each other, and the schedulability can be assessed by considering one set at a time.

Focusing on a single DAG G_j, only one subset of tasks $S \in W_j$ can be active at any given time instant. There might be tasks of other DAGs executing on the PU ψ_p: for each DAG, only one unreachable task subset can be active. Hence, we should theoretically compute all possible combinations of unreachable task subsets running on ψ_p and separately evaluate the schedulability of each of these. However, focusing on the utilization, if we consider the combination of subsets that corresponds to the maximum total utilization on ψ_p and this is deemed schedulable, then all other combinations with lower utilization are automatically schedulable.

The utilization of a generic subset S' on a PU ψ_p of an island I_s^{reg} running at frequency $\phi_{s,m}$ is defined as the sum of the utilization of all its tasks:

$$u_{j,p,m,S'} = \sum_{\substack{\tau_k \in S' \\ \chi(\tau_k) = \psi_p}} \frac{C_{k,s,m}}{d_k}. \quad (10)$$

The maximum contribution of G_j to the total utilization on ψ_p is the maximum over all its subsets:

$$u_{j,p,m}^{max} = \max_{S' \in W_j} u_{j,p,m,S'}. \quad (11)$$

Then, the combination of subsets with the maximum utilization on ψ_p is when we consider, for each DAG, its subset having the maximum utilization as computed by Equation (11). Hence, we apply the utilization-based test from [47] to the sum of the

maximum utilizations of all DAGs having tasks on ψ_p:

$$\sum_{G_j \in G^{reg}} u^{max}_{j,p,m} \leq U_{max}, \tag{12}$$

where $U_{max} \leq 1$ is a configurable parameter that defines how much bandwidth is left to non-real-time activities like OS tasks.

The schedulability of a regular DAG G_j is guaranteed when the individual task deadlines d_i for each $\tau_i \in G_j$, computed to satisfy the end-to-end DAG deadline, are such that Equation (12) holds.

4.2. Analysis of OpenMP tasks

In the following, we derive a method to bound the waiting time of OpenMP tasks in an OpenMP DAG in G^{omp}, which can be leveraged to evaluate their schedulability based on intermediate deadlines. We assume an OpenMP runtime assigning the tasks in the DAGs in G^{omp} to worker threads pulling them out of a shared queue, as with the GCC implementation. This model captures common OpenMP runtimes, and is also applicable to other parallel runtime systems based on thread-pool execution and a shared queue when respecting the assumptions in Section 3.1.

Essential to determine the schedulability of an OpenMP DAG is the derivation of an upper bound of the time spent by each single task waiting into the shared queue before it starts executing. The worst-case scenario for the waiting time of a task $\tau_i \in \Gamma_j$, with $G_j \in G^{omp}$, is when all tasks that can possibly run concurrently with τ_i have been added to the shared queue right before it.

To this purpose, we need the following preliminary result:

Lemma 1. *Given n tasks with EETB C_i, served non-preemptively in FIFO order by a pool of n^{WT} worker threads deployed on CPUs within the same island and arriving into the shared queue all at once when the pool is idle, with C_i ordered by non-increasing values, i.e., $C_{i+1} \leq C_i$, then the worst-case waiting time b_{n+1} for an additional $(n+1)^{th}$ task arriving at the system right after the arrival of the preceding n tasks satisfies:*

$$b_{n+1} \leq \sum_{i=1}^{k} C_i, \;\; with \;\; k = \left\lfloor \frac{n}{n^{WT}} \right\rfloor. \tag{13}$$

Proof. When the new task arrives, two situations are possible: either the tasks have been dispatched equally in number to the working threads, or there has been some imbalance. In the former case, n is necessarily a multiple of n^{WT}, and each thread has overall picked up exactly k tasks. Therefore, regardless of which working thread $\tilde{j}$ extracts the new task for execution, the condition is trivially satisfied: in the worst case, $\tilde{j}$ serves the k tasks with highest EETB, which is exactly Equation (13). For the imbalanced case, we distinguish between **(i)** a scenario in which a subset of worker threads have picked up exactly $k+1$ tasks, and the others have picked exactly k tasks; and **(ii)** the other scenarios where the queues are imbalanced. In case **(i)**, let $\{f_j\}$ denote the time instants at which the worker threads finish executing their currently allocated tasks. In that case, the scheduling policy picks the worker thread $\tilde{j}$ with the earliest finishing time $f_{\tilde{j}} = \min_{j \in \{1,\dots n^{WT}\}} \{f_j\}$, where the finishing time for each queue is equal to the sum of the $\{C_i\}$ values of the tasks served by that queue. If said worker thread is one that served k tasks, then the condition is satisfied. If, on the other hand, it is one having served $k+1$ tasks, this happens because its $f_{\tilde{j}}$ is earlier than the finishing time of all other worker threads, including one worker thread j that is handling k tasks. Therefore, $b_{n+1} = f_{\tilde{j}} \leq f_j$. Since f_j is upper-bounded by the sum of the greatest k $\{C_i\}$ values, the condition is verified: $b_{n+1} = f_{\tilde{j}} \leq f_j \leq \sum_{i=1}^{k} C_i$. In case **(ii)**, we have at least one worker thread with at least $k+2$ tasks. However, given that the overall number of tasks is n, there must be another worker thread having at least one less task, either from $k+1$ down to k, or from k down to $k-1$. If the policy selects $\tilde{j}$ as a worker thread with $k+2$ tasks, then it must be that its finishing time $f_{\tilde{j}}$ is smaller than the finishing times of all other worker threads, including necessarily at least one worker thread j with k or even with $k-1$ tasks. Therefore: $W = f_{\tilde{j}} \leq f_j$, with f_j containing at most k tasks, which again satisfies the condition. □

Following the same rationale as the schedulability theory derived in Section 4.1 for regular DAGs, not all tasks of G_j can run in parallel with τ_i. Using the definition of unreachable task subsets W_j for G_j (Definition 3), it is possible to derive the task subsets that contain τ_i, thus identifying the tasks in G_j that can possibly run concurrently with τ_i. The definition is more general, as it applies to a generic DAG G_j regardless of whether it is OpenMP or regular.

Definition 4. *Given a DAG G_j and a task $\tau_i \in \Gamma_j$, $G_j \in G$, Λ_i is the set containing the largest unreachable subsets of Γ_j containing τ_i, where the tasks of each subset S can be executed concurrently. Formally:*

$$\Lambda_i \triangleq \left\{ S \in W_j \mid \tau_i \in S \right\}. \tag{14}$$

Λ_i then contains subsets S where each identifies a possible scenario of parallelism for task τ_i due to tasks of the same DAG G_j. However, as explained in Section 3.3, the OpenMP worker threads of an island I^{omp}_s serve all OpenMP tasks (even from different DAGs) placed on that island. Hence, τ_i may see tasks from other OpenMP DAGs queued before it. Since DAGs are independent in our model, all possible combinations of unreachable task subsets of all other DAGs need to be evaluated to determine the worst-case waiting time of task τ_i.

Let us first define the *product operator* $\cdot$ between two sets of unreachable task subsets.

Definition 5. *The* product operator $\cdot$ *between two sets of unreachable task subsets W_a and W_b is defined as:*

$$W_a \cdot W_b \triangleq \{S_a \cup S_b \mid S_a \in W_a, S_b \in W_b\}. \tag{15}$$

We denote by $\prod_{k=1}^{n} W_k = W_1 \cdot W_2 \cdot \ldots \cdot W_n$ the product with n task subsets W_k. Now we can define the set of all possible combinations of unreachable task subsets.

Definition 6. *Given an OpenMP DAG $G_j \in G^{omp}$ and a task $\tau_i \in \Gamma_j$, the set Θ_i is defined as the set of all possible combinations of unreachable task subsets contained in Λ_i and in all W_k, $G_k \in G^{omp}$, $k \neq j$. Formally:*

$$\Theta_i = \Lambda_i \cdot \prod_{\substack{G_k \in G^{omp} \\ G_k \neq G_j}} W_k. \tag{16}$$

In summary, each subset $\mathcal{Z}_v \in \Theta_i$ is the union of an unreachable task subset for each OpenMP DAG. Therefore, $\mathcal{Z}_v$ contains the OpenMP tasks in Γ that can run in parallel with τ_i, and Θ_i collects all possible cross-DAG parallel scenarios for τ_i. Considering the example of Figure 4, the set of all parallel scenarios for τ_2 of DAG G_1 is:

$$\Theta_2 = \{\{\tau_2, \tau_3, \tau_4, \tau_6\}, \{\tau_2, \tau_3, \tau_4, \tau_7, \tau_8\}, \{\tau_2, \tau_3, \tau_4, \tau_7, \tau_{10}\}, \{\tau_2, \tau_3, \tau_4, \tau_8, \tau_9\}, \{\tau_2, \tau_3, \tau_4, \tau_9, \tau_{10}\}, \{\tau_2, \tau_3, \tau_4, \tau_{11}\}\}.$$

Recalling the OpenMP runtime model in Section 3.1, the OpenMP worker threads of an island part I_s^{omp} serve all tasks mapped on I_s^{omp}. Essential to determine the waiting time bound for a task τ_i mapped on I_s^{omp} in the OpenMP queues is filtering out from each parallel scenario $\mathcal{Z}_v \in \Theta_i$ the tasks that are not running on I_s^{omp}, thus obtaining $\mathcal{Z}'_v \in \Theta_i^s$:

$$\Theta_i^s \triangleq \{\mathcal{Z}'_v = \{\tau_k \in \mathcal{Z}_v \mid \rho(\tau_k) = I_s^{omp}\} \mid \mathcal{Z}_v \in \Theta_i\}. \tag{17}$$

As mentioned, we assume that OpenMP tasks placed on an island part I_s^{omp} are dispatched to the worker threads through a shared queue of ready tasks. Using the above result in Lemma 1, consider a scenario for τ_i where it is scheduled in parallel with, say, the v-th subset $\mathcal{Z}'_v \in \Theta_i^s$. Then, Lemma 1 allows us to derive the worst-case waiting time for τ_i before execution on the island $I_s = \rho(\tau_i)$ having n_s^{WT} OpenMP worker threads:

$$n_{v,s}^{queue} = \left\lfloor \frac{|\mathcal{Z}'_v| - 1}{n_s^{WT}} \right\rfloor. \tag{18}$$

Because we aim to evaluate the worst-case waiting time in the queue for τ_i, the most unfortunate case for it is when the *longest* $n_{v,s}^{queue}$ tasks in $\mathcal{Z}'_v$ are already queued, that is, the $n_{v,s}^{queue}$ tasks with the largest EETB in $\mathcal{Z}'_v$.

Given the above-introduced notation, we are now in the position to state the following:

Lemma 2. *Given a task $\tau_i \in \Gamma_j$, with $G_j \in G^{omp}$, placed on the island part I_s^{omp}, that is, $\rho(\tau_i) = I_s$, operating at frequency $\phi_{s,m}$, and the v-th unreachable tasks subset $\mathcal{Z}'_v \in \Theta_i^s$, consider the set $\mathcal{Z}'^{des}_v$ containing the tasks in $\mathcal{Z}'_v$ except τ_i, that is, $\mathcal{Z}'^{des}_v = \mathcal{Z}'_v \setminus \tau_i$, and assume that the tasks in $\mathcal{Z}'^{des}_v$ are sorted by decreasing EETB. An upper bound on the worst-case waiting time for τ_i in the queue in the scenario $\mathcal{Z}'_v$ is given by*

$$b_{i,s,m,v} = \sum_{\{\tau_k \in \mathcal{Z}'^{des}_v \mid k \leq n_{v,s}^{queue}\}} C_{k,s,m}. \tag{19}$$

The worst-case bound for τ_i is computed as the maximum over all sets $\mathcal{Z}'^{des}_v$ transformed from $\mathcal{Z}'_v \in \Theta_i^s$:

$$b_{i,s,m} = \max_{\mathcal{Z}'_v \in \Theta_i^s} b_{i,s,m,v}. \tag{20}$$

Hence, a feasible deadline assignment for an OpenMP task τ_i guaranteeing its schedulability must satisfy:

$$d_i \geq b_{i,s,m} + C_{i,s,m}. \tag{21}$$

5. Optimization of task placement and frequency scaling configuration

This section describes an optimization approach for the placement of time-triggered DAG-based applications, both regular and developed with OpenMP, on DVFS-capable hardware platforms having heterogeneous-capacity islands. This approach pursues the objective of minimizing the power consumption following two strategies: Section 5.1 derives a MIQCP formulation of the optimal design problem, whereas Section 5.2 builds a heuristic placement algorithm that returns suboptimal solutions in a short amount of time. Both algorithms make the following configuration decisions:

- determination of the number of OpenMP worker threads n_s^{WT} for each island $I_s \in I$. Since n_s^{WT} is equal to the number of PUs dedicated to OpenMP DAGs I_s^{omp}, this is equivalent to partitioning the PUs into I_s^{omp} and I_s^{reg};
- placement of each task of regular DAGs on an island I_s, and specifically on a PU in I_s^{reg};
- placement of each OpenMP DAG on an island I_s; differently from regular DAGs, tasks of the same OpenMP DAG must be placed on the same island. No specific assignment of tasks to a PU is made, as the OpenMP runtime will dynamically select a worker thread (thus a PU) for each task when they're pulled out of the shared queue;
- configuration of the OPP of each island I_s.

Additionally, each task $\tau_i \in \Gamma_j$ is assigned an appropriate intermediate deadline d_i, which helps in meeting the DAG end-to-end deadline D_j.

The optimization algorithms are based on the timing analysis presented in Section 4, which is needed to determine whether DAGs meet their end-to-end deadline or not. The approach is validated in Section 6.4 with experiments on an embedded Linux board.

5.1. MIQCP formulation

In the following, we describe the proposed MIQCP formulation by enumerating the considered variables, objective function, and constraints.

Placement variables. The placement of tasks on PUs modeled by the function $\chi(\tau_i)$ is encoded as a set of binary variables $\{x_{i,p}\}$, with $x_{i,p} = 1$ when $\chi(\tau_i) = \psi_p$ and 0 otherwise. Similarly, the function $\rho(\tau_i)$ representing the placement of tasks

onto islands is translated as a set of binary variables $\{m_{i,s}\}$, with $m_{i,s} = 1$ when $\rho(\tau_i) = I_s$ and 0 otherwise. The partition of the PUs of each island I_s into I_s^{omp} and I_s^{reg} is encoded as a set of binary variables $\{a_{s,p}\}$, where $a_{s,p} = 1$ indicates that $\psi_p \in I_s^{omp}$ and thus ψ_p executes OpenMP tasks only, whereas $a_{s,p} = 0$ indicates that $\psi_p \in I_s^{reg}$ and that it executes regular tasks only. The frequency configuration of each island is expressed with a set of binary variables $\{y_{s,m}\}$. When $y_{s,m} = 1$, the island I_s operates at frequency $\phi_{s,m}$. To simplify the writing of problem constraints, another set of binary variables $\{z_{i,p,m}\}$ is introduced, which encode the logical AND operation $x_{i,p} \wedge y_{s,m}$, with $\psi_p \in I_s$ (which are expressed as linear constraints).

Optimization objective. The objective of the optimization is to minimize the average power consumption of the DAGs deployed on the hardware platform. The mapping of tasks to islands affects power consumption, as islands have diverse energy-efficiency characteristics. Adopting in the optimizer the power model discussed in [44] results in the following objective function:

$$\min \mathcal{P}^{tot} = \sum_{p \in P} \sum_{m \in \Phi_{s(p)}} \left(y_{s(p),m} \mathcal{P}^{I}_{s(p),m} + \Delta\mathcal{P}_{s(p),m} \sum_{\tau_i \in \Gamma} \frac{z_{i,p,m} \tilde{C}_{i,s(p),m}}{T_{j(i)}} \right), \quad (22)$$

$$\Delta\mathcal{P}_{s,m} = \mathcal{P}^{B}_{s,m} - \mathcal{P}^{I}_{s,m}, \quad (23)$$

where $s(p)$ identifies the index s of the island I_s to which ψ_p, indexed by p, belongs, $j(i)$ the index j of the DAG G_j τ_i belongs to, and $\tilde{C}_{i,s(p),m}$ is the average execution time of τ_i when deployed on ψ_p at frequency $\phi_{p,m}$. Equation (22) computes the average power consumed by the platform by summing the idle power $\mathcal{P}^{I}_{s(p),m}$, which varies with the configured frequency of each island, and the contribution to the busy power of the application. The latter term depends on the average workload over time due to all tasks placed on the PU. The variables $\{z_{i,p,m}\}$ select the tasks that are effectively placed on ψ_p at frequency $\phi_{s,m}$. In Equation (22), one can also use the EETBs $C_{i,s(p),m}$ in place of the $\tilde{C}_{i,s(p),m}$, obtaining a conservative estimate of the average power consumption of the platform.

Constraints. The DAG topology imposes constraints on the release and finishing times $r_{i,k}$ and $f_{i,k}$ of tasks, as stated in Equation (3). In the MIQCP formulation, we remove the dependency on the DAG instance k by reasoning with the latest possible finishing times f_i relative to any DAG activation. Recalling that any individual task τ_i must complete within its individual deadline d_i, the topological constraints of Equation (3) can be expressed with the latest finishing time f_i only:

$$f_i = \max_{\tilde{i} | (\tilde{i},i) \in \mathcal{E}_j} \{f_{\tilde{i}}\} + d_i \quad \forall i \in \Gamma_j. \quad (24)$$

In no DAG instance k the finishing time of τ_i can be greater than f_i, because all tasks complete within their d_i. The end-to-end DAG latency guarantee is enforced by the following constraint on the last task:

$$f_{e_j} \leq D_j. \quad (25)$$

Tasks of the same OpenMP DAG need to be deployed on the same island I_s:

$$m_{i,s} = m_{k,s} \quad \forall \tau_i, \tau_k \in \Gamma_j,\ G_j \in G^{omp}. \quad (26)$$

A crucial part of the problem for meeting end-to-end DAG deadlines is the schedulability of tasks on PUs. When dealing with regular DAGs, the schedulability analysis presented in Section 4.1 is applied by making Equations (11) and (12) compliant with the variables of the problem formulation:

$$u^{max}_{j,p,m} = \max_{S \in W_j} \sum_{\tau_i \in S} \left(\frac{1}{d_i} \sum_{\phi_m \in \Phi_{s(p)}} z_{i,p,m} C_{i,s(p),m} \right), \quad \sum_{G_j \in G^{reg}} u^{max}_{j,p,m} \leq U_{max}, \quad (27)$$

where $z_{i,p,m}$ selects the current placement of τ_i and the frequency configuration of I_s. The above constraint is quadratic, because the d_i variable is at the denominator.

The decisive parameter for the scheduling of OpenMP DAGs is the number of worker threads n_s^{WT} for an island I_s. As explained in Section 3.3, each worker is statically pinned to a PU, so the determination of n_s^{WT} is equivalent to determining its part I_s^{omp}. The former number can be encoded with a set of real variables defined as follows:

$$n_s^{WT} = \sum_{\psi_p \in I_s} a_{s,p}. \quad (28)$$

The computation of the waiting time bound $b_{i,s,m}$ of Equation (20) for an OpenMP task τ_i involves the search over every subset $\mathcal{Z}'_v \in \Theta_i^s$ of the longest tasks that in the worst case are enqueued before τ_i in the runtime queues. In the MIQCP formulation, $\mathcal{Z}'_v$, which contains the tasks in $\mathcal{Z}_v$ mapped on I_s^{omp}, is computed on-the-fly from $\mathcal{Z}_v$ with the use of $\{m_{i,s}\}$ variables. The selection of the longest tasks in each subset happens via a set of binary variables $\{\alpha_{i,k,s,v}\}$, with $\alpha_{i,k,s,v} = 1$ meaning that task τ_k is one of the longest tasks of $\mathcal{Z}_v$ that must be considered in the waiting time bound for τ_i on I_s^{omp}. Note that $\alpha_{i,k,s,v} = 1$ assumes that τ_i and τ_k are placed on the same island I_s, otherwise τ_k can never happen to be in the same queue with τ_i and $\alpha_{i,k,s,v} = 0$. Equation (18) states the number of tasks concurring in the bound, which then constrains the sum of the $\{\alpha_{i,k,s,v}\}$ variables set to 1 for each subset v:

$$\sum_{\substack{\tau_k \in \mathcal{Z}_v \\ \tau_k \neq \tau_i}} \alpha_{i,k,v,s} = \left\lfloor \frac{\sum_{\tau_k \in \mathcal{Z}_v} m_{k,s} - 1}{n_s^{WT}} \right\rfloor. \quad (29)$$

The sum of $m_{k,s}$ variables counts the tasks of $\mathcal{Z}_v$ that are mapped on I_s^{omp}, which is equivalent to $|\mathcal{Z}'_v|$. Note that the denominator of the fraction is the variable n_s^{WT}, hence the constraint is quadratic. The linearization of the floor operator is performed using standard techniques.

The $\{\alpha_{i,k,s,v}\}$ variables are needed to find the longest tasks in $\mathcal{Z}_v$. The bound to the deadline d_i of Equation (21) is encoded by the following constraint:

$$d_i \geq C_{i,s,m} + \max_{\{\alpha_{i,k,v,s}\}} \sum_{\substack{\tau_k \in \mathcal{Z}_v \\ \tau_k \neq \tau_i}} \alpha_{i,k,v,s} C_{k,s,m} \quad \forall \mathcal{Z}_v \in \Theta_i. \quad (30)$$

With the sum of $\{\alpha_{i,k,s,v}\}$ locked by Equation (28), the max operator finds the $n_{v,s}^{queue}$ longest tasks in $\mathcal{Z}_v$. Writing the constraint above $\forall \mathcal{Z}_v \in \Theta_i$ is equivalent to finding the maximum over all subsets of Θ_i^s, as Equation (20). The max operator is encoded with standard techniques.

5.2. Heuristic solver

The optimal approach presented in Section 5.1, like most of the approaches based on integer linear programming, does not scale well with large-size placement problems, and the solver may take several hours to obtain the optimal solution. As an alternative to the MIQCP formulation, this section describes a heuristic algorithm for the power-aware placement problem, which determines suboptimal placement solutions in significantly less time. The heuristic placement strategy is detailed in Algorithm 1.

The procedure takes as input parameters the set of islands I in the system, the OPPs in Φ, the DAGs G^{reg} and G^{omp}, and the tasks Γ, and it returns whether the provided system is feasible or not. The algorithm starts by sorting the tasks by descending EETB and the DAGs by descending cumulative utilization U_Γ (Lines 2-3). Then, the function proceeds with an initialization phase (Lines 4-6), where each island is set to its maximum frequency, that is, the most advantageous condition for the schedulability. All PUs are initially configured to schedule OpenMP tasks only, as the algorithm maps OpenMP DAGs first. The function then attempts to place OpenMP DAGs one at a time, from the lowest-capacity island onwards, so as to prioritize low-power islands and minimize the overall energy consumption (Lines 7-12).

The schedulability of an OpenMP DAG is checked with the isOmpDagSchedulable routine (described in Algorithm 2), which implements the analysis presented in Section 4.2. For all tasks in the OpenMP DAG, the function sets the respective deadline to its worst-case traversal time. The worst-case traversal time requires computing the waiting time bound of Lemma 2 (Line 9 of Algorithm 2) by analyzing all possible parallel scenarios in which the given task can be found based on type of cores the DAG is assigned to and the corresponding frequency settings (Line 5). After computing all task deadlines, the algorithm ensures that all deadlines are compatible with the DAG end-to-end deadline (Line 11). Specifically, the procedure checkDagDeadline verifies that the critical path of the DAG, expressed in terms of intermediate deadlines, is smaller than or equal to the end-to-end deadline.

Based on this schedulability test, if no island makes the DAG schedulable in Algorithm 1, then the placement problem is deemed unfeasible. Otherwise, after mapping all OpenMP DAGs, the placement procedure proceeds by partitioning the island into I_s^{omp} and I_s^{reg} by reducing the number of OpenMP worker threads to the minimum that preserves the schedulability of OpenMP DAGs, so as to leave available as many PUs as possible for the placement of regular DAGs (Lines 13-18). The areOmpDagsSchedulable procedure invokes isOmpDagSchedulable (Algorithm 2) on all OpenMP DAGs to check their schedulability.

Algorithm 1 Heuristic frequency scaling configuration and placement optimization procedure.

```
1: function placement(I, Φ, G^reg, G^omp, Γ)
2:     sortDescendingEETB(Γ)
3:     sortDescendingUtil(G)
4:     for I_s ∈ I do
5:         setIslandFrequency(I_s, φ_{s,1}) ▷ Φ_s sorted by descending frequency (φ_{s,1} is max)
6:         I_s^omp ← I_s                    ▷ All PUs for OpenMP DAGs
7:     for G_j ∈ G^omp do                   ▷ Mapping OpenMP DAGs
8:         for I_s^omp ∈ I sorted by ascending capacity ξ_s do
9:             if isOmpDagSchedulable(G_j, I_s^omp) then
10:                placeDag(G_j, I_s^omp)
11:                break                    ▷ Moving to DAG G_{j+1}
12:        if G_j not placed then return UNFEASIBLE
13:    for I_s^omp ∈ I do ▷ Reducing the number of OpenMP worker threads
14:        for ψ_p ∈ I_s^omp do
15:            I_s^omp ← I_s^omp \ {ψ_p}
16:            if ¬areOmpDagsSchedulable(G^omp, I_s^omp) then
17:                I_s^omp ← I_s^omp ∪ {ψ_p}
18:                break ▷ Cannot lower OpenMP worker threads anymore on I_s^omp
19:    assignIndividualDeadlines(G^reg) ▷ Using deadline splitting with nominal EETBs
20:    for τ_i ∈ Γ_j | G_j ∈ G^reg do       ▷ Mapping regular DAGs
21:        for I_s^reg ∈ I sorted by descending capacity do
22:            ψ_p ← findPuWorstFit(I_s^reg, τ_i) ▷ Finding the PU with lowest utilization ≤ U_max
23:            if ψ_p not null then
24:                placeTask(τ_i, ψ_p)
25:                break                    ▷ Moving to task τ_{i+1}
26:            assignIndividualDeadlines(G_j) ▷ Reassigning deadlines when moving to I_{s+1}^reg
27:        if τ_i not placed then return UNFEASIBLE
28:    for τ_i ∈ Γ_j | G_j ∈ G^reg do ▷ Trying to move tasks to lower-capacity islands
29:        I_s^reg ← getMappedIsland(τ_i)
30:        for I_k^reg ∈ getLowerCapacityIslands(I_s^reg) do
31:            assignIndividualDeadlines(G_j)
32:            ψ_p ← findPuWorstFit(I_k^reg, τ_i)
33:            if ψ_p not null then
34:                placeTask(τ_i, ψ_p)
35:    if ¬areRegDagsSchedulable(G^reg) then
36:        return UNFEASIBLE
37:    for I_s ∈ I do                       ▷ Lowering frequencies until unschedulable
38:        for φ_{s,m} ∈ Φ_s \ {φ_{s,1}} do ▷ Φ_s sorted by descending frequency
39:            setIslandFrequency(I_s, φ_{s,m})
40:            if ¬areAllDagsSchedulable(G) then
41:                setIslandFrequency(I_s, φ_{s,m−1})
42:                break
43:    return FEASIBLE
```

Algorithm 2 Checking schedulability of an OpenMP DAG.

1: **function** isOmpDagSchedulable(G_j, I_s^{omp})
2: $n_s^{WT} \leftarrow |I_s^{omp}|$ ▷ *Number of OpenMP worker threads (see Sec. 3.3)*
3: **for all** $\tau_i \in \Gamma_j$ **do** ▷ *Computing the deadline of each task in G_j*
4: $d_i \leftarrow 0$ ▷ *The deadline of task τ_i is its maximum traversal time*
5: **for all** $\mathcal{Z}'_v \in \Theta_i^s$ **do** ▷ *Considering all parallel scenarios for τ_i (see Eq. 17)*
6: $\mathcal{Z}'^{des}_v \leftarrow \mathcal{Z}'_v \setminus \{\tau_i\}$ ▷ *Considering the interfering tasks (see Lemma 2)*
7: $n_{v,s}^{queue} \leftarrow \left\lfloor \frac{|\mathcal{Z}'_v| - 1}{n_s^{WT}} \right\rfloor$ ▷ *Worst number of tasks queued before τ_i (see Eq. 18)*
8: Let $\tau_{(1)}, \ldots, \tau_{(|\mathcal{Z}'^{des}_v|)}$ be tasks in $\mathcal{Z}'^{des}_v$ sorted by descending C
9: $b_{i,s,m,v} \leftarrow \sum_{h=1}^{n_{v,s}^{queue}} C_{h,s,m}$ ▷ *Waiting time for τ_i in the current scenario (see Lemma 2)*
10: $d_i \leftarrow \max(d_i,\ b_{i,s,m,v} + C_{i,s,m})$ ▷ *Finding the maximum traversal time for τ_i over all parallel scenarios*
11: **if** ¬checkDagDeadline(G_j) **then**
12: **return** UNSCHEDULABLE
13: **return** SCHEDULABLE

Algorithm 3 Deadline splitting procedure.

1: **function** splitDeadline($\tau_{src}, \tau_{dst}, D$)
2: $P \leftarrow$ getCriticalPath(τ_{src}, τ_{dst})
3: $L \leftarrow$ getPathLength(P)
4: **for** $\tau_i \in P$ **do**
5: $d_i^{new} \leftarrow \frac{D}{L} \times C_i$
6: **if** $d_i = 0 \vee d_i^{new} < d_i$ **then** ▷ *Update d_i only if it is tighter than the previous value*
7: $d_i \leftarrow d_i^{new}$
8: **for** $\tau_k \in$ getSuccessors(τ_{src}) **do**
9: splitDeadline($\tau_k, \tau_{dst}, D - d_{src}$)

Algorithm 4 Checking schedulability of all regular DAGs

1: **function** areRegDagsSchedulable(G^{reg})
2: **for** $G_j \in G^{reg}$ **do**
3: **if** ¬checkDagDeadline(G_j) **then** ▷ *Checking end-to-end deadline*
4: **return** UNSCHEDULABLE
5: **for** $I_s \in I$ **do**
6: **for** $\psi_p \in I_s^{reg}$ **do** ▷ *Checking utilization on each PU (see Eq. (12))*
7: $U_p \leftarrow 0$
8: **for** $G_j \in G^{reg}$ **do**
9: $u_{j,p,m}^{\max} \leftarrow \max_{S' \in W_j} \sum_{\substack{\tau_k \in S', \\ \chi(\tau_k) = \psi_p}} \frac{C_{k,s,m}}{d_k}$ ▷ *m is the OPP index $\phi_{s,m}$*
10: $U_p \leftarrow U_p + u_{j,p,m}^{\max}$
11: **if** $U_p > U_{max}$ **then**
12: **return** UNSCHEDULABLE
13: **return** SCHEDULABLE

The algorithm then moves on with the placement of tasks in regular DAGs. The schedulability test of regular DAGs on PUs requires the utilization of the single tasks; therefore, at Line 19, the deadline splitting function is called to determine the individual deadline of tasks on the basis of their reference EETB, as no placement has been made yet. The algorithm follows state-of-the-art techniques [32] where the end-to-end DAG deadline is distributed to single tasks proportionally to their EETB. The core function of the deadline splitting procedure is detailed in Algorithm 3. Given a DAG G_j, the splitting function finds the critical path between τ_{s_j} and τ_{e_j} in terms of EETBs, that is, the path connecting τ_{s_j} and τ_{e_j} having the highest sum of the EETBs of the traversed tasks, and proportionally assigns an individual deadline to its tasks. Then, it proceeds recursively on the critical paths starting from all successors of τ_{s_j} and ending on τ_{e_j}, considering an overall deadline reduced by the deadline assigned to τ_{s_j}.

The placement of regular DAGs in Algorithm 1 happens from Line 20 to Line 27. Starting from the highest-capacity island, in order to facilitate the schedulability, the algorithm calls the findPuWorstFit function at Line 22, which finds within the current island the PU with the lowest current utilization (hence the name *worst-fit*, with reference to the terminology of bin-packing heuristics). This is done to distribute the placed tasks over the available cores in a balanced fashion, so to increase the chance of lowering the island frequency later in the final frequency assignment step (Lines 37-42)[2]. Note that regular DAGs are placed using only cores left unused by OpenMP DAGs, that are placed earlier [3]. To ensure schedulability, the capacity of the returned PU must be below U_{max} after placing the current τ_i on that PU. If the worst-fit algorithm cannot find any feasible PU, then the function proceeds with the next lower-capacity island, thus acting in a similar fashion to a worst-fit bin-packing heuristic on the available PU capacities. Before testing the next island, the deadline splitting is performed again to properly account for the fact that the tasks that will be possibly placed on the next island will have higher EETBs due to Equation (5) (Line 26). Therefore, their individual deadline is relaxed accordingly.

The placement algorithm then tries to move as many tasks as possible from high-capacity islands to low-capacity ones (Lines 28-34), with the aim of minimizing the overall energy consumption. The function getMappedIsland(τ_i) returns the island I_s where τ_i is currently placed with the help of the $\rho(\tau_i)$ function. After reassigning the individual deadlines due to the possible placement on a lower-capacity island, findPuWorstFit is called again to find a suitable PU for the current task τ_i on the new island and, if that is the case, the task τ_i is reassigned to that island.

On Line 35, the algorithm verifies the schedulability of the placed regular DAGs to ensure that the obtained placement is

[2] As opposed to that, both first-fit and best-fit heuristics would have the drawback to load one or a few CPUs much more than others in the island, impairing the possibility to lower its frequency later.

[3] If we placed regular DAGs first, then the worst-fit heuristic would end up using as many cores as possible, leaving no cores available for placing OpenMP DAGs later

feasible. The procedure for doing so (areRegDagsSchedulable) is described in Algorithm 4. Fundamentally, the function performs the PU utilization test of Eq. 12 on each PU after identifying which tasks are placed on the PU (Line 9 of Algorithm 4).

Finally, on Lines 37-42, the placement function attempts to reduce the OPP of all islands to the minimum that retains schedulability, as lower frequencies correspond to lower power. The schedulability is checked with areAllDagsSchedulable, which simply invokes areOmpDagsSchedulable on all OpenMP DAGs and areRegDagsSchedulable on all regular DAGs. At this point, our heuristic has all tasks of all regular DAGs placed on specific cores with pre-computed intermediate deadlines, and all OpenMP DAGs placed on dedicated OpenMP cores. Under these conditions, given the monotonically decreasing behavior of tasks EETBs with respect to the frequency of their assigned cores (Eq.(5)), each island has a minimum frequency below which it is impossible to guarantee schedulability of the DAGs. Using any set of frequencies above such minima ensures schedulability as well. Therefore, our heuristic at Lines 37-42 finds this set of minimum frequencies (in the worst-case, these end up being the maximum allowed frequencies for all islands). For platforms with a high number of frequencies, it would be advisable to use a binary search to make this part of the algorithm more efficient.

6. Experimental evaluation

This section presents the results of an experimental evaluation and validation of the two proposed optimization approaches with randomly generated DAGs deployed on an ODROID-XU4 embedded board. The implementation is publicly available online [48].

6.1. Generation of random DAGs

The number n_G of DAGs to be generated for each task set Γ is fixed to $n_G = 3$. For each task set Γ, its first DAG G_1 is always flagged as an OpenMP application (i.e., it is inserted in G^{omp}), while the others are flagged as such with a probability of 0.2. The remaining DAGs are flagged as regular DAG-based applications (i.e., they are inserted in G^{reg}).

The topology of the DAGs G_j in each task set Γ is randomly generated with the generation strategy in [49]. Topology generation for each DAG starts by first producing a series-parallel graph with multiple levels of nested parallel branches. This series-parallel graph is obtained starting from an initial graph composed of two interconnected nodes (i.e., the source node and the sink node of the DAG) and then by recursively expanding each non-sink node into a set of additional parallel subgraphs with a given probability, until a recursion limit is reached. The generation parameters n_{rec}, p_{par}, and n_{par} respectively select the maximum recursion depth, the probability with which a non-sink node is expanded into a parallel subgraph by forking it into a number of parallel successor nodes, and the maximum level of parallelism resulting from each node expansion; specifically, the number of branches into which a node is expanded is selected from the discrete uniform distribution $[2, n_{par}]$. The resulting series-parallel graph is then transformed into a DAG by randomly adding edges between pairs of nodes with a probability p_{add}, without introducing cycles.

Given this strategy to generate the DAG topologies, the random generation procedure for the scheduling parameters in each DAG G_j is as follows [29, 50]. Given a parameter U_Γ representing the desired overall system utilization for an application Γ, the UUniFast algorithm [51] is applied to generate the cumulative utilization U_{Γ_j} for the tasks Γ_j of each DAG G_j in Γ, such that $\sum_{\Gamma_j \in \Gamma} U_{\Gamma_j} = U_\Gamma$. For each DAG G_j, the minimum inter-arrival time T_j of the tasks in Γ_j is selected from a discrete log-uniform distribution in the range $[T_{min}, T_{max}]$, whereas the deadline of G_j is set to $D_j = T_j$ (implicit deadlines). The cumulative EETB C_{Γ_j} of G_j is then set to $C_j = T_j \cdot U_{\Gamma_j}$, and the reference EETB C_i of each task $\tau_i \in \Gamma_j$ is generated by applying the UUniFast algorithm to partition the available cumulative EETB C_{Γ_j} among the tasks in Γ_j, such that $\sum_{\tau_i \in \Gamma_j} C_i = C_{\Gamma_j}$. Specifically, UUniFast is used to uniformly select $|\Gamma_j|$ real values $\hat{c}_i \in [0, 1]$ with the constraint that $\sum_{\tau_i \in \Gamma_j} \hat{c}_i = 1$; then, the EETB C_i for each node $\tau_i \in \Gamma_j$ is set to $C_i = C_{\Gamma_j} \cdot \hat{c}_i$. The generation procedure for each task τ_i is repeated in case either **(i)** $C_i > D_j$ holds for some node $\tau_i \in \Gamma_j$; or **(ii)** the length of the critical path of G_j in terms of traversed EETBs of the corresponding tasks Γ_j exceeds the deadline D_j.

For all system configurations, the generation parameters were set as follows: $n_{rec} = 2$, $n_{par} = 3$, $p_{par} = 0.6$, $p_{add} = 0.01$, $T_{min} = 100$ ms, and $T_{max} = 1000$ ms.

6.2. Hardware model parameters

For the placement problem, we considered a reference logical platform modeling an Arm big.LITTLE architecture. Specifically, the placements determined by the proposed algorithms are validated in Section 6.4 with an ODROID-XU4 embedded board, which is based on a Samsung Exynos5 Octa ARM big.LITTLE CPU [52]. In detail, the hardware model is equipped with eight cores, evenly clustered into two islands. The *big* island with four Cortex-A15 cores is more powerful in terms of computational power, since it has capacity $\xi_1 = 1$, but is more energy-consuming. The *LITTLE* island with four Cortex A7 cores is instead energy efficient, but has lower capacity: $\xi_2 = 0.44$. Each island has DVFS capabilities. *LITTLE* cores can run at a frequency ranging from 200MHz to 1.4GHz, whereas *big* cores from 200MHz to 2GHz. However, to avoid thermal throttling issues, the maximum frequency of *big* cores has been limited to 1.4GHz. In our model, the frequency can be configured at discrete points in the range [200MHz, 1400MHz], with a step of 100MHz.

The power consumption values $\mathcal{P}^I_{s(p),m}$ and $\mathcal{P}^B_{s(p),m}$ for each island and OPP for use in Equation (22) are configured based on the measurements reported in previous work for the ODROID-XU4 embedded board [8].

6.3. Results of placement optimization

The DAG generator presented in Section 6.1 was used to generate test cases by varying the total utilization U_Γ in the range $[0.5, 3]$ with a step of 0.5. 70 scenarios were generated

for each utilization point, thus obtaining 420 test cases in total. Each test case was optimized by both the MIQCP optimizer of Section 5.1 and the heuristic placement algorithm of Section 5.2. The MIQCP formulation was solved using Gurobi [53], a commercial solver for linear and quadratic programming problems. The optimizers were executed on a desktop computer equipped with the Intel Xeon CPU E5-2640 having 40 logical cores (the physical ones are 20) running at 2.40GHz, and with 128GB of main memory. For the purpose of analysis and optimization, the U_{max} parameter was set to 0.95. This reflects the typical default configuration of Linux-based systems, where, when scheduling real-time processes, 5% of the processor bandwidth (50ms every 1s) is reserved for the execution of non-real-time activities.

We initially performed a small set of experiments with a subset of the generated DAGs to tune the timeout for Gurobi. After several attempts with variable timeout, we set it at 3h: this value allowed to obtain either a feasible solution or the determination of unfeasibility in 95% of the evaluated cases. Such a timeout value enabled a large and exhaustive evaluation of the proposed approaches. A few selected test cases for which Gurobi, at the expiration of the timeout, terminated without a feasible solution, were executed again with a 24h timeout.

Figure 5 shows the number of tests each algorithm managed to optimize (Y axis), per total utilization (X axis). Their ability to solve the optimization problem is comparable. In both cases, there is a substantial degradation of the problem feasibility when $U_\Gamma \geq 2$. The small gap between the MIQCP and the heuristic solutions when $U_\Gamma \in [0.5, 1.5]$ is due to the 3h timeout in Gurobi: in those cases, the timeout expired when the solver had not found any feasible solutions yet. A few cases were optimized again with a 24h timeout to mitigate the gap. With $U_\Gamma \in [2.0, 3.0]$ the heuristic algorithm solved fewer cases: with increasing utilization values, the problem becomes more complex. Since the MIQCP has more freedom in the placement decisions than the heuristic solver, it has more chances to find a solution.

A qualitative comparison between the results of the two optimizers is presented in Figure 6. Each solution is represented as a point whose X-axis value is the power consumption $\mathcal{P}^{tot}$ of that placement, and its ordinate is the runtime of the optimizer to return that solution. The resulting spacial representation denotes the largeness of the runtime of Gurobi, that is orders of magnitude greater than that of the heuristic placement. The latter instead is almost always below 10s, making it suitable for large-scale problems. However, the distribution of the placements found by the MIQCP formulation appears to be more on the left side of the plot, meaning that the optimal solver tends to find lower-power configurations. This is supported by the fact that the MIQCP formulation finds the optimal value, with the downside that it requires more time to execute.

A quantitative evaluation of the placements found by the heuristic solver is showed in Figure 7, which is based on the definition of *cost* of the heuristic expressed as a percentage value and computed as $\frac{\mathcal{P}^{tot}_{heur} - \mathcal{P}^{tot}_{MIQCP}}{\mathcal{P}^{tot}_{MIQCP}} \times 100$. This serves to assess how far the placements found by the heuristic solver are

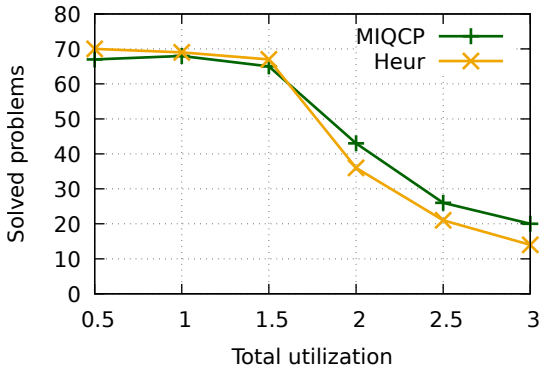


Figure 5: Number of optimized scenarios for the optimal and heuristic placements as a function of the total utilization. The problems the MIQCP could not optimize with $U_\Gamma \in [0.5, 1.5]$ are due to the 3h timeout: when it expired, Gurobi had not found any feasible solutions yet.

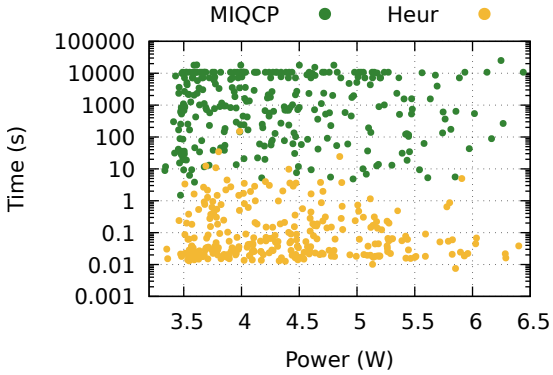


Figure 6: Spacial representation of the obtained placements by the MIQCP optimizer and the heuristic placement algorithm, where the X axis is the power consumption of the placement and the Y axis is the time the optimizer took to obtain that solution.

from the optimal value. The plot shows the cost statistics as a function of the total utilization U_Γ. The average cost in the performed experiments is lower than 4%, which demonstrates that the placements found by the heuristic algorithm are close to the optimal placements found by the MIQCP. In conclusion, the heuristic solver proved to optimize the generated placements in a significantly reduced time compared to the MIQCP-based optimal solver, without sacrificing much the number of solved problems and the quality of the solutions in terms of power consumption.

6.4. Validation on Linux

To verify on a real system the validity of the two optimizers and to show that the presented mathematical model matches an actual OpenMP implementation, a simple test application was written using the C language. Such an application, named rt-dag, implements an arbitrary real-time DAG that is activated periodically. Each node of the DAG consumes a predefined amount of CPU time and can be executed in a POSIX thread (created and handled using the `pthread` library) or an OpenMP task. In the first case, rt-dag creates a POSIX thread per DAG node and schedules it with the Linux SCHED_DEADLINE [54]

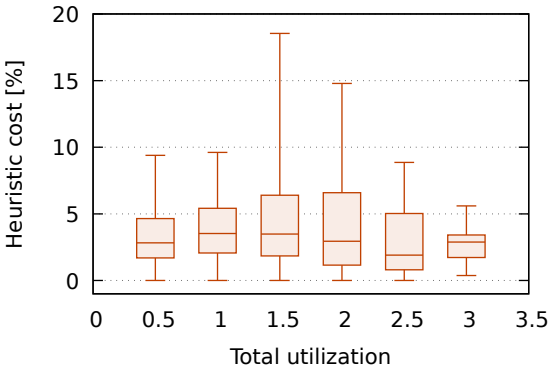


Figure 7: Cost increase over the optimal power (found by the MIQCP optimizer) of the heuristic placement as a function of the total utilization.

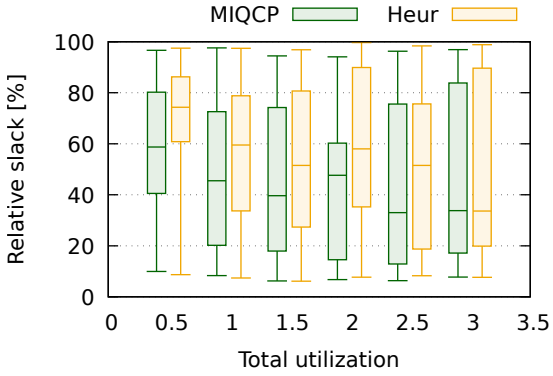


Figure 8: Relative slack (normalized with the deadline) of the DAGs measured in the validation experiments as a function of the total utilization.

scheduler with the runtime and deadline computed by the optimizers. Each thread is blocked on a condition variable. The thread corresponding to the first node of the DAG is periodically woken up by using the `clock_nanosleep()` call, while the other threads are woken up when all their inputs are generated by the corresponding threads. In the second case, the application periodically executes the first node of the DAG, dynamically creating an OpenMP task per node with the `#pragma omp task` directive, using the `depend` option to let the task execute only when all predecessors have finished.

The program reads the description of DAGs from files in the YAML format (one file per DAG) generated by the optimizers. These files describe tasks and dependencies, their scheduling algorithms and placement, and the frequency configuration. The program executes the DAGs using POSIX threads or OpenMP and sets the CPU frequency with some additional shell scripts according to the output of the optimizer. The last node of the DAG measures the response time of the DAG and saves it in an array that is printed at the end of the program's execution (so that outputting the response times does not affect the execution of the DAG).

This software setup was used to test whether the response times of the DAGs met their imposed deadline when the application ran on the ODROID-XU4 board described in Section 6.2, with the DAG deployment and the frequency configuration found by the optimizers. 78 DAG configurations returned by the heuristic solver and 79 obtained with the MIQCP formulation were executed on the board and the generated response times were compared with the DAG deadlines. We considered the response time of the k-th activation of a DAG G_j as the time difference between the finishing time of the last task of the DAG and the start time of its first task. We computed the *slack* of a DAG instance as the time interval between its finishing time and its deadline, and we normalized it with the deadline, i.e., $\frac{D_j - f_{e_{j,k}} + r_{s_{j,k}}}{D_j}$. Every DAG was executed for 200 periods, and the normalized slack of every periodic activation was measured.

The aggregated results are shown in Figure 8 and are grouped by total utilization. All measured slacks are strictly greater than 0, which means that all DAG instances met their deadline during the validation runs, confirming the correctness of the analysis and design. Interestingly, the placements obtained by the MIQCP formulation exhibit a lower slack on average compared to the heuristic placements. This is because the MIQCP optimizer finds the lowest-energy system configuration, which makes the tasks run longer and generally closer to their deadline, than in the case of the configuration found by the heuristic algorithm.

7. Conclusions and future work

This paper proposed a workflow to deploy OpenMP and general DAG-based applications on Arm big.LITTLE-like heterogeneous platforms with the aim of minimizing energy consumption while meeting end-to-end latency constraints. First, a schedulability analysis for multiple DAG-based applications, optionally supporting OpenMP parallelization, was presented. Then, a pair of optimization methods are presented to determine a suitable allocation of tasks to the cores in the platform: an optimal solution to the problem was formulated as MIQCP, and a heuristic algorithm was developed for fast, good quality solutions. The approach was also validated on an ODROID-XU4 board. In the future, the placement strategy can be extended to configure OpenMP priorities to improve real-time performance. Also, hardware accelerators (e.g., GPUs) can be considered in the placement problem.

References

[1] OpenMP Architecture Review Board, OpenMP Application Programming Interface, version 5.2 (Nov 2021).

[2] R. Vargas, E. Quinones, A. Marongiu, OpenMP and timing predictability: A possible union?, in: Proceedings of the 18th IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE 2015), IEEE, 2015, pp. 617–620.

[3] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, E. Quinones, A lightweight OpenMP4 run-time for embedded systems, in: 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, 2016, pp. 43–49.

[4] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones, X. Martorell, A functional safety OpenMP for critical real-time embedded systems, in: International Workshop on OpenMP, Springer, 2017, pp. 231–245.

[5] S. Royuela, L. M. Pinho, E. Quiñones, Enabling Ada and OpenMP runtimes interoperability through template-based execution, Journal of Systems Architecture 105 (2020) 101702.

[6] J. Sun, N. Guan, F. Li, H. Gao, C. Shi, W. Yi, Real-time scheduling and analysis of OpenMP DAG tasks supporting nested parallelism, IEEE Transactions on Computers 69 (9) (2020) 1335–1348. doi:10.1109/TC.2020.2972385.

[7] A. Balsini, T. Cucinotta, L. Abeni, J. Fernandes, P. Burk, P. Bellasi, M. Rasmussen, Energy-efficient low-latency audio on android, Journal of Systems and Software 152 (2019) 182–195. doi:10.1016/j.jss.2019.03.013.

[8] T. Cucinotta, A. Amory, G. Ara, F. Paladino, M. D. Natale, Multi-criteria optimization of real-time dags on heterogeneous platforms under P-EDF, ACM Transactions on Embedded Computing Systems 23 (1) (2024) 1–35.

[9] Intel, Whitepaper – intel performance hybrid architecture & software optimizations – part one: Introduction to performance hybrid architecture for 12th generation intel core processors.
URL https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-architecture.html

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

[11] E. Quinones, S. Royuela, C. Scordino, P. Gai, L. M. Pinho, L. Nogueira, J. Rollo, T. Cucinotta, A. Biondi, A. Hamann, D. Ziegenbein, H. Saoud, R. Soulat, B. Forsberg, L. Benini, G. Mando, L. Rucher, 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), IEEE, 2020. doi:10.1109/isorc49007.2020.00042.

[12] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang, The design of OpenMP tasks, IEEE Transactions on Parallel and Distributed Systems 20 (3) (2009) 404–418. doi:10.1109/TPDS.2008.105.

[13] A. Duran, J. Corbalán, E. Ayguadé, Evaluation of OpenMP task scheduling strategies, in: Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism, IWOMP'08, Springer-Verlag, Berlin, Heidelberg, 2008, p. 100–110.

[14] J. Sun, N. Guan, Y. Wang, Q. He, W. Yi, Real-time scheduling and analysis of openmp task systems with tied tasks, in: 2017 IEEE Real-Time Systems Symposium (RTSS), 2017, pp. 92–103. doi:10.1109/RTSS.2017.00016.

[15] E. Silvestri, A. Pellegrini, P. D. Sanzo, F. Quaglia, Effective runtime management of tasks and priorities in GNU OpenMP applications, IEEE Transactions on Computers 71 (10) (2022) 2632–2645. doi:10.1109/TC.2021.3139463.

[16] C. Yu, S. Royuela, E. Quiñones, Taskgraph: A low contention OpenMP tasking framework, IEEE Trans. Parallel Distrib. Syst. 34 (8) (2023) 2325–2336. doi:10.1109/TPDS.2023.3284219.

[17] A. Bourramouss el Maach, A. Munera, S. Royuela, Energy-aware performance portability with OpenMP dynamic variants, in: Proceedings of the SC'25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2025, pp. 1183–1195.

[18] B. McDonald, F. Mueller, OpenMP-RT: Native pragma support for real-time tasks and synchronization with LLVM under linux, in: Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, 2024, pp. 119–130.

[19] B. McDonald, F. Mueller, OpenMP-RT: Pragma support for scheduling periodic real-time tasks, in: International Workshop on OpenMP, Springer, 2025, pp. 66–80.

[20] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, E. Quinones, Timing characterization of OpenMP4 tasking model, in: 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), IEEE, 2015, pp. 157–166.

[21] M. Han, N. Guan, J. Sun, Q. He, Q. Deng, W. Liu, Response time bounds for typed DAG parallel tasks on heterogeneous multi-cores, IEEE Transactions on Parallel and Distributed Systems 30 (11) (2019) 2567–2581. doi:10.1109/TPDS.2019.2916696.

[22] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.

[23] A. K. Mok, D. Chen, A multiframe model for real-time tasks, IEEE Transactions on Software Engineering 23 (10) (1997) 635–645.

[24] M. Stigge, P. Ekberg, N. Guan, W. Yi, The digraph real-time task model, in: Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011), IEEE, 2011, pp. 71–80. doi:10.1109/RTAS.2011.15.

[25] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010), IEEE, 2010, pp. 259–268.

[26] A. Saifullah, K. Agrawal, C. Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, in: Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011), 2011, pp. 217–226.

[27] A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quinones, G. Buttazzo, A static scheduling approach to enable safety-critical OpenMP applications, in: Procedings of the 22nd IEEE Asia and South Pacific Design Automation Conference (ASP-DAC 2017), IEEE, 2017, pp. 659–665.

[28] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, G. von der Brüggen, Many suspensions, many problems: A review of self-suspending tasks in real-time systems, Real-Time Systems 55 (1) (2019) 144–207.

[29] F. Aromolo, A. Biondi, G. Nelissen, G. Buttazzo, Event-driven delay-induced tasks: Model, analysis, and applications, in: Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021), IEEE, 2021, pp. 53–65.

[30] F. Aromolo, A. Biondi, G. Nelissen, Response-time analysis for self-suspending tasks under EDF scheduling, in: Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS 2022), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022, pp. 13:1–13:18.

[31] M. Günzel, F. Aromolo, A. Biondi, J.-J. Chen, Requirement-based analysis of self-suspending tasks under EDF, in: 2025 IEEE Real-Time Systems Symposium (RTSS), IEEE, 2025, pp. 528–540.

[32] H. Chetto, M. Silly, T. Bouchentouf, Dynamic scheduling of real-time tasks under precedence constraints, Real-Time Systems 2 (3) (1990) 181–194.

[33] J. M. Rivas, J. J. Gutiérrez, J. C. Palencia, M. G. Harbour, Deadline assignment in EDF schedulers for real-time distributed systems, IEEE transactions on parallel and distributed systems 26 (10) (2014) 2671–2684.

[34] B. Peng, N. Fisher, T. Chantem, 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, 2016, pp. 13–22.

[35] H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Determining optimal processor speeds for periodic real-time tasks with different power characteristics, in: Proceedings 13th Euromicro Conference on Real-Time Systems, 2001, pp. 225–232. doi:10.1109/ECRTS.2001.934038.

[36] M. Bambagini, M. Marinoni, H. Aydin, G. Buttazzo, Energy-aware scheduling for real-time systems: A survey, ACM Transactions on Embedded Computing Systems (TECS) 15 (1) (2016) 1–34.

[37] Y. Yu, V. K. Prasanna, Power-aware resource allocation for independent tasks in heterogeneous real-time systems, in: Ninth International Conference on Parallel and Distributed Systems, 2002. Proceedings., IEEE, 2002, pp. 341–348.

[38] T. A. AlEnawy, H. Aydin, Energy-aware task allocation for rate monotonic scheduling, in: 11th IEEE Real Time and Embedded Technology and Applications Symposium, IEEE, 2005, pp. 213–223.

[39] S. Moulik, Z. Das, R. Devaraj, S. Chakraborty, SEAMERS: A semi-partitioned energy-aware scheduler for heterogeneous multicore real-time systems, Journal of Systems Architecture 114 (2021) 101953.

[40] G. Wang, W. Li, X. Hei, Energy-aware real-time scheduling on heterogeneous multi-processor, in: 2015 49th Annual Conference on Information Sciences and Systems (CISS), IEEE, 2015, pp. 1–7.

[41] J. Roeder, B. Rouxel, S. Altmeyer, C. Grelck, Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021, pp. 501–510.

[42] GNU Project, GNU libgomp.
URL https://gcc.gnu.org/onlinedocs/libgomp/

[43] LLVM/OpenMP, LLVM/OpenMP 21.0.0git documentation.
URL https://openmp.llvm.org/

[44] G. Ara, T. Cucinotta, A. Mascitti, Simulating Execution Time and Power Consumption of Real-Time Tasks on Embedded Platforms, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, SAC

'22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 491–500. doi:10.1145/3477314.3507030.

[45] R. I. Davis, A. Burns, D. Griffin, On the meaning of pWCET distributions and their use in schedulability analysis, in: In Proceedings Real-Time Scheduling Open Problems Seminar at ECRTS, 2017.

[46] S. Bozhko, F. Marković, G. von der Brüggen, B. B. Brandenburg, What really is pWCET? a rigorous axiomatic proposal, in: 2023 IEEE Real-Time Systems Symposium (RTSS), 2023, pp. 13–26. doi:10.1109/RTSS59052.2023.00012.

[47] S. Baruah, A. Mok, L. Rosier, Preemptively scheduling hard-real-time sporadic tasks on one processor, in: [1990] Proceedings 11th Real-Time Systems Symposium, 1990, pp. 182–190. doi:10.1109/REAL.1990.128746.

[48] F. Paladino, F. Aromolo, L. Abeni, T. Cucinotta, Optimizing the deployment of real-time openmp applications for energy efficiency (artifact), https://github.com/lucabe72/rtdag (2026).

[49] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, G. C. Buttazzo, Response-time analysis of conditional DAG tasks in multiprocessor systems, in: Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015), IEEE, 2015, pp. 211–221. doi:10.1109/ECRTS.2015.26.

[50] F. Aromolo, G. Nelissen, A. Biondi, Replication-based scheduling of parallel real-time tasks, in: Proceedings of the 35th Euromicro Conference on Real-Time Systems (ECRTS 2023), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023, p. 18:1–18:23.

[51] E. Bini, G. C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Systems 30 (1-2) (2005) 129–154.

[52] ODROID Wiki, ODROID-XU4 [ODROID Wiki].
URL https://wiki.odroid.com/odroid-xu4/odroid-xu4

[53] Gurobi Optimization, LLC, Gurobi Optimizer Reference Manual.
URL https://docs.gurobi.com/

[54] J. Lelli, D. Faggioli, T. Cucinotta, G. Lipari, An experimental comparison of different real-time schedulers on multicore systems, Journal of Systems and Software 85 (10) (2012) 2405–2416, automated Software Evolution. doi:10.1016/j.jss.2012.05.048.