Precise Scheduling of DAG Tasks with Dynamic Power Management

Ashikahmed Bhuiyan
Department of Computer Science, West Chester University, PA, USA

Mohammad Pivezhandi
Department of Computer Science, Wayne State University, Detroit, MI, USA

Zhishan Guo
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Jing Li
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, USA

Venkata Prashant Modekurthy
Department of Computer Science, University of Nevada, Las Vegas, NV, USA

Abusayeed Saifullah
Department of Computer Science, Wayne State University, Detroit, MI,USA

Abstract

The rigid timing requirement of real-time applications biases the analysis to focus on the worst-case performances. Such a focus cannot provide enough information to optimize the system's typical resource and energy consumption. In this work, we study the real-time scheduling of parallel tasks on a multi-speed heterogeneous platform while minimizing their typical-case CPU energy consumption. Dynamic power management (DPM) policy is integrated to determine the minimum number of cores required for each task while guaranteeing worst-case execution requirements (under all circumstances). A Hungarian Algorithm-based task partitioning technique is proposed for clustered multi-core platforms, where all cores within the same cluster run at the same speed at any time, while different clusters may run at different speeds. To our knowledge, this is the first work aiming to minimize typical-case CPU energy consumption (while ensuring the worst-case timing correctness for all tasks under any execution condition) through DPM for parallel tasks in a clustered platform. We demonstrate the effectiveness of the proposed approach with existing power management techniques using experimental results and simulations. The experimental results conducted on the Intel Xeon 2680 v3 12-core platform show around 7%-30% additional energy savings.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Parallel task, mixed-criticality scheduling, energy minimization, dynamic power management, cluster-based platform

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.8

Funding *Ashikahmed Bhuiyan*: Provost's Research Grant, West Cheser University
Zhishan Guo: NSF CCF-2028481, Startup Grant at NC State University
Jing Li: NSF CNS-1948457
Venkata Prashant Modekurthy: UNLV Troesh Center, CNS-2211640
Abusayeed Saifullah: CNS-2301757, CAREER-2306486, CNS-2306745

1 Introduction

Multi-core processors are increasingly appearing as an enabling platform for embedded systems (e.g., mobile phones, tablets, drones, computerized numerical controls, etc.). The parallel task model can exploit the multi-core platform's capability as they support *intra-task parallelism*, where a task can execute on multiple cores simultaneously. Many computation-intensive systems (e.g., self-driving cars) that demand stringent timing requirements often evolve in the form of parallel tasks. Many recent studies on real-time scheduling and analysis have focused on the directed acyclic graph (DAG) model of parallel tasks [8, 9, 16, 37, 38, 53, 55]. The DAG model is a general workload model for representing intra-task parallelism, where nodes represent threads of execution and edges represent their dependencies. Several real-world applications use the DAG model [29].

Energy efficiency is essential for embedded systems, as they rely on a time-limited energy sources (i.e., batteries, energy harvesting devices). Modern generation processors minimize power consumption through *dynamic voltage and frequency scaling* (DVFS) that adjusts the voltage and frequency at runtime. To date, some works considered the energy-aware real-time scheduling of parallel tasks [10, 11, 54]. These works adopted the federated scheduling [38] or task decomposition framework [55] with the DVFS policy for minimizing system energy consumption in the per-core or per-node (of a DAG task) speed modulation settings. Such speed tuning is inefficient as it increases the hardware cost [31]. Also, there is an ongoing trend of considering the cluster-based platform (e.g., big.LITTLE [48]), which groups processors into multiple islands, each execute at the same speed. Such a cluster-based platform balances energy efficiency and cost [43]. To date, few efforts have been made to study the energy-aware real-time scheduling of parallel tasks in a clustered platform [11, 25].

All these works assumed that hard real-time constraints must be satisfied to guarantee the system's correctness. For a hard real-time task, missing a deadline is considered a system failure and may result in catastrophic consequences. Hence, most schedulability analyses considered that a task could execute up to its worst-case execution time (WCET). WCET-based schedulability test is often very pessimistic [42], and the task execution pattern varies significantly across different job releases, rarely executing up to its WCET [58]. Thus, designing a system that relies primarily on WCET may lead to resource over-provisioning in typical cases [46]. Moreover, modern embedded systems pose strict energy constraints and demand leveraging richer system models to optimize energy consumption under typical-case instead of worst-case. To address this issue, this paper requires system designers to provide at least two execution time estimates for a task: a WCET and a typical-case execution budget (no more than the WCET). The first will give worst-case guarantees, while the latter is used for energy minimization. Although a dual-execution-estimation setting may appear similar to the mixed-criticality (MC) framework [59], this work focuses on a different problem. In an MC setup, a common platform integrates different tasks with varying levels of criticality. A criticality level is assigned to each task with multiple execution time thresholds. Under such settings, existing works studied energy minimizing [12, 13, 34, 47, 60]. However, these works assume that all the tasks execute up to their WCET at the respective criticality levels. Meanwhile, our approach proposes optimizing energy consumption under the typical-case execution time instead of WCET. Our approach also ensures that all tasks must receive full-service guarantees under all circumstances.

The energy-aware scheduling of real-time tasks is challenging due to the complicated dependencies among frequency, energy consumption, and execution time [24]. Existing works focused mostly on the DVFS policy [10, 11, 25, 26, 54] with a significant limitation: it is

not effective in reducing static power consumption, which may elevate to 50% or more of the overall power consumption [33]. Besides, existing energy minimization approaches are applied to WCET and thus will only lead to better power/energy behaviors in the worst case (rare event) instead of the typical/average scenarios. Given both *typical* and *worst-case* execution time estimates, we propose an energy-aware technique that minimizes the *typical* energy consumption while guaranteeing (worst-case) timing correctness for all tasks.

Challenges. Handling the dual execution estimation is challenging for the following reasons. *First*, schedulers are unaware of each task's exact behavior before run-time. Such non-clairvoyance of execution length typically leads to NP-Hard problems. *Second*, all tasks receive full-service assurances under any circumstances. *Third*, some recent works have studied the energy-aware scheduling of the MC task model [12, 13, 34, 47], but thy did not consider intra-task parallelism.

Motivated by these facts, we study the real-time scheduling of DAG tasks in a clustered platform to minimize their CPU energy consumption, one of the significant contributors to the overall system power consumption. The scheduling problem aims to achieve both worst-case real-time guarantees and typical-case energy efficiency. In a clustered platform, all cores in the same cluster execute at the same speed. However, different clusters can operate at different speeds [48]. We adopt the DPM policy to reduce static power consumption. DPM policy reduces static power consumption by utilizing idle intervals. If the idle interval is at least equal to a certain threshold (known as the *break-even time* [18]), the processor is switched to a low-power sleep mode, thus reducing its static power consumption. Our approach finds the minimum but a sufficient number of low-speed cores for each task, leaving many high-speed cores idle for a long duration. If the low-speed cores are insufficient to schedule all the tasks, we assign additional high-speed cores. Therefore, the proposed approach can lead to high energy savings resulting from the power-down of CPU components such as cores, caches, and translation look-aside buffer.

The key objective of the proposed method is to conserve energy during the actual execution of a DAG task when its nodes do not execute until their WCET. Several factors, including pipelines, branch predictors, and caches, impact the WCET estimation of the nodes in the DAG task, thereby impacting the task's makespan estimation. In addition, hardware features often introduce pessimism to the WCET estimation, implying the difference between WCET estimation and actual execution time increases considerably. In such cases, the proposed approach will significantly increase energy savings by allocating resources only when needed. Specifically, we make the following key contributions:

- We utilize DPM to propose an energy-aware federated scheduling strategy of parallel DAG tasks on dual-speed platforms. Given both typical and worst-case execution time estimations, our energy-aware approach determines the required (minimum) number of low-speed processors to minimize the typical CPU energy consumption while guaranteeing worst-case timing correctness for all tasks.
- Under multi-speed cluster-based settings, we propose an energy-efficient task-cluster partitioning technique without violating the schedulability guarantees.
- We perform the experimental study under randomly generated task sets. We report the *schedulability ratio* of our approach, and demonstrate a minimum of 29.23% less power consumption against state-of-the-art approach [39].
- We present onboard experiments conducted on Intel Xeon 2680 v3 multi-core (12-core) platform and report up to 30% energy savings compared to the existing approach.

2 Related Work

There have been works studying the energy-efficient real-time scheduling of sequential tasks in both uni- and multi-processor platforms (few to mention [15,17,19,35,47,50,51]; refer to [4] for a comprehensive survey). However, all these works considered the sequential task model. In contrast, a parallel task can make use of multiple cores simultaneously and complete the same amount of work in a shorter time via exploiting the internal parallelism. Hence, the scheduling strategy and analysis of parallel task is significantly different from the sequential task. The state-of-the-art parallel real-time scheduling primarily emphasizes scheduling analysis and does not account for energy awareness [1,5,8,9,23,55,56].

To date, a few works studied the energy-aware scheduling of parallel tasks. Li et al. [40] studied a non-recurrent task model with a fixed number of parallel threads. Paolillo et al. [52] studied the energy-aware scheduling of the gang task model. Zhu et al. [61,62] proposed a slack stealing based scheduling approach considering the inter-dependent sequential tasks. Energy-aware scheduling of DAG tasks was proposed by Bhuiyan et al. [10] and Guo et al. [26]. Both of them have considered a simplified model (i.e., the number of cores cannot be pre-fixed). They have considered table-driven scheduling. Hence, the entire schedule until the hyper-period needed to be created in advance. Some recent efforts have been made to study the energy-efficient scheduling of parallel (and sequential) tasks in a clustered platform [11,20,25,41,45]. However, our work's focus differs from existing ones. We study the energy-efficient scheduling of parallel tasks on a clustered platform while minimizing their typical-case CPU energy consumption while guaranteeing worst-case execution requirements.

Meanwhile, extensive research has investigated the real-time scheduling of the MC task model considering both the sequential and parallel task model (e.g., [1,5,7,12–14,22,27,39]). Regarding the task model, the work in [1] is most near to us. However, our paper's contributions differ significantly from [1] regarding problem statement, challenges, solution techniques, and evaluation. For MC DAG tasks, [1] did not consider energy-aware scheduling. In contrast, our paper adopts the DPM policy to minimize power consumption. Incorporating the DPM policy into the existing analysis is not trivial because (i) DPM policy utilizes the processor's idle slot that is unknown apriori; (ii) We have considered the clustered platform. Hence, we cannot turn off some processors in a cluster while others are running.

3 System Model and Background Concepts

In this work, we consider a set of sporadic parallel DAG tasks denoted by $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each DAG contains a set of nodes, where a node represents some sequential computation. The precedence constraint between two nodes is represented by a directed edge. A node can only execute if all of its predecessors have finished execution. For each task $\tau_i \in \tau$, we consider the following two parameters: (i) *total work*, which is the sum of the number of clock cycles (i.e., total computation work) performed by all nodes in τ_i; and (2) the *critical-path length*, which is defined as the number of clock cycles of the longest directed path of the DAG (i.e., the path with the largest computation work).

Each task $\tau_i \in \tau$ is characterized by a 5-tuple $(C_i^N, C_i^O, L_i^N, L_i^O, T_i)$. Here, C_i^N denotes the typical-case execution time of the task (referred to as LO-criticality execution time[1]), while C_i^O denotes the overload execution time (referred to as HI-criticality execution time).

[1] Although the terms LO- and HI-criticality may lead to the assumption that we are considering tasks with different criticality levels [59]. In this work, the term criticality is used to distinguish whether a job exceeds its typical workload and whether a job's execution modes are still exhibiting the typical (or overload) workload estimates.

Similarly, $L_i^N (L_i^O)$ denotes the typical (overload) critical-path length estimates, referred to as LO(HI)-criticality critical-path length estimates. Note that the typical workload estimates (i.e., C_i^N and L_i^N) are obtained using a less pessimistic yet practical tool and are expected to occur during regular operations. In contrast, a more pessimistic tool (by considering all possible scenarios, including the worst-case ones) is used to obtain C_i^O and L_i^O. Thus, the overload workload estimates may exceed the typical ones in several orders of magnitudes. However, executing up to its overload workload is rare for a task. We assume that under any condition, a task's total work and critical-path length will never exceed C_i^O and L_i^O, respectively. The period of task τ_i is denoted by T_i. We assume that each task τ_i has an *implicit* deadline, i.e., the relative deadline D_i is equal to T_i. For τ_i to be scheduled during an overload scenario, the condition $T_i \geq L_i^O$ must be satisfied, where L_i^O denotes the minimum time required to complete task τ_i even when an infinite number of cores are available. The typical (overload) utilization of task τ_i is defined as the ratio of its typical (overload) total work and its period. Let, $u_i^N = \frac{C_i^N}{T_i}$ and $u_i^O = \frac{C_i^O}{T_i}$ denote the typical and overload utilization, respectively.

▶ **Example 1.** Consider two DAG tasks τ_1 and τ_2, where $\tau_1 = (12, 24, 4.08, 8.16, 21.6)$ and $\tau_2 = (12, 36, 3.36, 12.24, 33.12)$. The typical total work of τ_1 is 12 and the overload total work is 24. The typical and overload critical-path lengths of τ_1 are 4.08 and 8.16, respectively. Because τ_1 has a period of 21.6, its typical utilization $u_1^N = \frac{12}{21.6} = 0.56$ and its overload utilization $u_1^O = \frac{24}{21.6} = 1.11$. Similarly, the typical utilization u_2^N and overload utilization u_2^O of task u_2^N are $u_2^N = \frac{12}{33.12} = 0.36$ and $u_2^O = \frac{36}{33.12} = 1.087$, respectively.

System behavior. In this work, we consider a task's two different execution (typical and overload) requirement. At runtime, the scheduler does not know the exact behavior of each job of a task, e.g., the exact workload or critical-path length of the job. Thus, it is expected that the job of a task starts execution using its typical execution budget. If a job's total work or the critical-path length exceed this task's typical total work or critical-path length, this job execute according to its HI-criticality execution budget and critical-path length estimate. Note that the precise timing of when a task may require the use of its overload execution budget is unpredictable. However, once a job (of any task) finishes executing its overload execution budget, the next job (of the same task) starts running according to its typical execution requirement.

Power/Energy model. In this work, we consider the following power model to represent the CPU power consumption by a processor [10, 26, 34, 47, 50, 51]. Let s denotes the main frequency (speed) of a processor, and the power consumption $P(s)$ can be expressed as:

$$P(s) = P_{sta} + P_{dyn}(s) = \beta + \alpha s^{\gamma} \tag{1}$$

Here, P_{sta} and $P_{dyn}(s)$ denote the static/leakage consumption and dynamic power consumption, respectively. If a processor is not entirely turned off, P_{sta} (represented as β) is introduced in the system due to leakage current, and the frequency-dependent switching activities introduce $P_{dyn}(s)$ (represented as αs^{γ}). For modeling the dynamic power consumption $P_{dyn}(s)$, α depends on the effective switching capacitance and $\alpha > 0$ [50]; γ is a fixed parameter determined by the hardware and it ranges between [2,3]. Pagani et al. [50] have shown that this model is highly realistic by comparing the power consumption (estimated by this model) with the actual power consumption results from [32]. Table 1 illustrates the

Table 1 Comparison of the CPU power consumption P_{equ} considering the power model in Eq (1) and experimental data P_{exp} from [32]. The *error rate* is defined as $100 \times (P_{\text{equ}} - P_{\text{exp}})/P_{\text{exp}}$.

Frequency	P_{exp}	P_{equ}	Error rate
0.24 GHz	0.54 W	0.52 W	-3.7%
0.46 GHz	0.70 W	0.67 W	-4.28%
0.68 GHz	1.04 W	1.05 W	0.96%
0.84 GHz	1.5 W	1.54 W	2.67%
0.92 GHz	1.96 W	1.88 W	-4.07%
1.02 GHz	2.30 W	2.35 W	2.17%

detailed comparison. Given a fixed amount of workload C executing on a speed-s processor, we can calculate the total energy consumption $E(C, s)$ as the integral of power throughout C/s, where $E(C, s) = (\beta + \alpha s^{\gamma}) \times \frac{C}{s} = \frac{\beta C}{s} + \alpha C s^{\gamma - 1}$.

We aim to reduce static energy consumption by employing the DPM approach, which utilizes the processor's idle time. If the idle interval in a processor is greater or equal to a certain threshold (i.e., break-even time [18]), the processor enters a low-power sleep mode. In this work, we try to allocate low-speed cores to all tasks, leaving the high-speed cores idle. The high-speed cores are used when some jobs enter HI-criticality mode due to exceeding their typical workload estimates. In this case, additional resources (i.e., some high-speed cores) are needed to complete the overload workload. Because tasks rarely exceed their typical workload estimates, the high-speed cores typically idle for a long duration. Thus, these idle cores can enter the low-power sleep mode, reducing static energy consumption.

Platform model. We are examining a homogeneous multi-core architecture called the Intel Xeon 2680 V3, where each core can have a designated clock frequency and a set of customized fine-tuned cores. In contrast to the initially clustered platforms such as the Odroid XU4 ARM's big.LITTLE architecture [48], which forces a fixed number of cores in each cluster to be synchronized at the same speed. This feature of the Xeon Processor facilitates the identification of the appropriate number of cores in each cluster. It gives a final general solution where clusters are initialized with a constant speed that can be fixed to a corresponding speed optimized through DAG Task features. In this case study, we halve the platforms cores referring to ξ clusters (each with n cores), and all cores in the same cluster execute at the same speed. However, different clusters can operate at different speeds. The maximum speed is s^L (s^H) for the LO(HI)speed clusters, where $s^L \leq s^H$. In a clustered architecture, we assume that any core can be put to sleep (i.e., processor clock turned off) and only an entire cluster can be put to deep sleep (processor and L2 clock turned off) [3].

Virtual Deadline. The concept of the virtual deadline was first proposed in [6], considering MC scheduler. High-criticality task is assigned a virtual deadline which is less than the actual deadline (and thus a higher priority). This ensures the HI-criticality jobs get sufficient slack for their overload workload to complete after a mode switch.

Federated Scheduling Algorithm. In real-time systems, multiprocessor algorithms are implemented considering either the *global* or *partitioned* approach. In the partitioned approach, a task to processor mapping is performed before run-time for each task. During run-time, no job migration is allowed, and all the jobs generated by a task execute only on its mapped processor. Considering the parallel DAG task models, Li et al. [37] proposed

Table 2 Major notations used throughout the paper.

	Symbol	Description
Workload	τ_i	The i^{th} task
	$C_i^N(C_i^O)$	typical (overload) execution budget of task τ_i
	$L_i^N(L_i^O)$	typical (overload) critical-path length estimates of task τ_i
	$u_i^N(u_i^O)$	typical (overload) utilization of τ_i
	$T_i(D_i)$	period (relative deadline) of task τ_i
	D_i'	virtual deadline of task τ_i
Platform	$\mathcal{K}_i$	i^{th} cluster
	M_i	number of sub-cluster in i^{th} cluster
	$\mathcal{K}_m^n$	n^{th} sub-cluster inside m^{th} cluster
	$s^L(s^H)$	execution speed of the low (high) speed cores
	S_k	Speed of k^{th} cluster
	$m_i^L(m_i^H)$	number of low(high) speed cores allocated to task τ_i
	$E_i^{\mathcal{K}_m^n}$	energy consumption by a sub-cluster $\mathcal{K}_m^n$ when executing a task τ_i

the *federated scheduling* approach, which is considered a reasonable extension of partitioned scheduling for parallel tasks. A federated scheduling algorithm classifies a task as a heavy task if its utilization is greater than or equal to 1; otherwise, the task is classified as a light task. Each heavy task receives a set of cores dedicated to this task. All the remaining cores (i.e., the cores left after each heavy task receives its portion) are given to all light tasks. A multiprocessor scheduling algorithm (e.g., partitioned earliest deadline first [44] or rate monotonic schedulers [2]) is used to schedule all these light tasks sequentially. In contrast to heavy tasks, light tasks can share cores.

Given a task set τ, a federated scheduler works as follows. The task set τ is divided into disjoint sets, i.e., τ_{heavy} and τ_{light}. Here, τ_{heavy} contains all the heavy tasks (utilization is at least 1), and τ_{light} contains all the light tasks (utilization is less than 1). A heavy task $\tau_i \in \tau_{heavy}$, receives m_i cores. Here, $m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil$ (refer to [37]), where, C_i is the WCET of τ_i, L_i is the critical path length, and $D_i(= T_i)$ the deadline. Then, the remaining m_{light} cores, where $m_{light} = m - \sum_{\tau_i \in \tau_{heavy}} m_i$, can be used by all the light tasks. The light tasks are forced to execute sequentially and scheduled by a multiprocessor scheduling algorithm. After a valid task to core allocation, runtime scheduling is performed as follows:

- For a high-utilization task $\tau_i \in \tau_{heavy}$, any greedy or work-conserving parallel scheduler is used to schedule τ_i on m_i cores.
- Any multiprocessor scheduling algorithm is used to schedule all light tasks on the remaining m_{light} cores if the algorithm's schedulability test is passed.

4 Energy-aware Federated Scheduling for the Dual-Speed Platform

This section discusses our energy-aware federated scheduling strategy for platforms with dual-speed cores. Considering that the platform supports cores with two speeds, i.e., low- and high-speeds, we aim to minimize the CPU energy consumption under typical scenarios while guaranteeing that all tasks receive enough execution budget even under overload scenarios. Our approach relies on the DPM approach to reduce energy consumption and tries to allocate only the low-speed cores to all tasks, leaving the high-speed cores idle in most cases, leading to reduced dynamic power consumption. Besides, all these high-speed idle cores can enter the low-power sleep mode, which further minimizes the static energy consumption.

Towards this goal, Subsection 4.1 determines the required minimum number of low-speed cores m_i^L for each task $\tau_i \in \tau$ to complete the overload workload. If there are not enough low-speed cores to serve all the tasks under overload scenarios, we compensate for this shortage by assigning additional high-speed cores to some of the tasks when their jobs exceed typical workloads; on the other hand, the numbers of low-speed cores allocated to these tasks are reduced under typical scenarios. Because tasks rarely exhibit overload behavior, the high-speed cores typically are not used, which is beneficial for reducing energy consumption. Subsection 4.2 presents a greedy approach (Algorithm 1) that checks if there are enough low-speed cores to serve all the tasks under overload scenarios. If not, Algorithm 1 allocates additional high-speed cores under overload scenarios to some tasks that provide the maximum relative core saving ratio under typical scenarios. Note that, in this approach, we assume that the processor speeds are given.

4.1 Determining the Number of Cores for Each Task

In this subsection, we determine the number of cores required for each task $\tau_i \in \tau$ to meet its deadline. Our analysis considers the following assumption.

Assumption. This section assumes that the platform consists of two types of cores, i.e., low-speed cores and high-speed cores. The execution speed of any task τ_i on the low-speed and high-speed cores are respectively denoted by s^L and s^H.[2] The execution speed of a core denotes the (minimum) amount of computation that can be completed per time unit. This section assumes that the total workload and the processor speed have a linear relationship [13]. Hence, for any task τ_i, we can translate the total workload to the execution time on s-speed cores as C_i^χ/s, where $\chi = \{N, O\}$. Similarly, for any task τ_i, we can translate the critical path to the execution time on s-speed cores as L_i^χ/s. Now, we classify a task $\tau_i \in \tau$ into the following three categories:

Category 0: $C_i^O \leq s^L D_i$. Task τ_i in this category with WCET C_i^O/s^L and deadline D_i is a light (or low utilization) task even on low-speed cores. We enforce these tasks to execute sequentially and use any traditional multiprocessor scheduling (e.g., partitioned EDF) approach to schedule them on the low-speed cores.

Category 1: $C_i^O > s^L D_i$ and $C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$. For tasks in this category, we allocate m_i^L low-speed cores for both LO- and HI–criticality modes. According to Lemma 1 in [1], any task τ_i (where τ_i belongs to Category 1) has a maximum makespan of $(C_i^O - L_i^O)/m + L_i^O$, where m is the number of unit-speed cores allocated to τ_i. Therefore, to meet deadline D_i on cores with speed s^L, the lower bound on m_i^L can be calculated as follows:

$$\frac{\frac{C_i^O}{s^L} - \frac{L_i^O}{s^L}}{m_i^L} + \frac{L_i^O}{s^L} \leq D_i \Longrightarrow \frac{C_i^O - L_i^O}{s^L \times m_i^L} \leq \frac{s^L \times D_i - L_i^O}{s^L}$$
$$\Longrightarrow \frac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \leq m_i^L$$

Hence, we allocate m_i^L low-speed cores to τ_i belonging to Category 1, where $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \right\rceil$. Here, we assume that $m_i^L \geq 0$, meaning that the critical-path length of each DAG on the low-speed cores is shorter than its corresponding deadline, and it is

[2] This section assumes that all the low (or high) speed cores execute at the same speed, which is independent of the executing tasks. Hence, for any task $\tau_i, \tau_j \in \tau$, their execution speeds are the same, if they are allocated in the same cluster. Such a restriction appears commonly in existing systems [48], where all processors within the same cluster/island execute at the same speed during run time.

feasible to assign each task τ_i to a low-speed core. One could relax this constraint and include tasks that are assigned to high-speed cores; however, the energy-saving techniques proposed in this paper are no longer applicable in this situation. In this work, we focus on minimizing energy consumption while ensuring that all tasks meet their deadlines using low-speed cores. Hence, handling any infeasible task to schedule on low-speed cores falls beyond the scope of this paper.

Category 2: $C_i^O/(s^L \times D_i) > 1$ and $C_i^N/s^L \leq C_i^O/s^H - L_i^O/s^H$. For any task τ_i that belongs to Category 2, we try to allocate $\left\lceil \frac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \right\rceil$ low-speed cores to it in both modes, if there is sufficient number of low-speed cores available. Otherwise, we allocate fewer low-speed cores and set a virtual deadline D_i' for the LO-criticality mode. The virtual deadline D_i' is set as $D_i' = \frac{C_i^N}{m_i^L \times s^L}$, so that task τ_i can finish its nominal workload by D_i' if no core idles. In summary, we set m_i^L as follows:

$$m_i^L = \begin{cases} \left\lceil \dfrac{C_i^O - L_i^O}{s^L \times D_i - L_i^O} \right\rceil ; \text{If available LO-speed cores} \\ \text{are sufficient.} \\ \left\lceil \dfrac{C_i^N/s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})} \right\rceil ; \qquad \text{Otherwise.} \end{cases} \tag{2}$$

Here, m_i^H and s^H denote the number of high-speed cores (allocated to task τ_i) and their speed, respectively, for the case where not enough low-speed cores available. See Theorem 1 for the derivation of m_i^L.

For task τ_i with fewer low-speed cores available and assigned to it, we assign it m_i^H high-speed cores and consider the actual deadline D_i during the HI-criticality mode. In order to meet the deadline, the computing power in the high-criticality mode must be equal to or greater than the computing power in the low-criticality mode, meaning that, meaning that, so $m_i^H \times s^H \geq m_i^L \times s^L$. Additionally, task τ_i needs to finish the remaining work and critical-path length within $D_i - D_i'$ time units. The worst case scenario happens when there is no progress on the critical-path length L_i^O, which leaves a remaining work of $C_i^O - m_i^L \times D_i' \times s^L$. This is the worst-case scenario because more processor times are idling due to the critical-path length given the relation between m_i^H and m_i^L, where $m_i^H \geq m_i^L \times s^L/s^H$. Therefore, to meet the deadline, m_i^H can be calculated as follows:

$$\begin{aligned} m_i^H &= \max\left\{ \frac{m_i^L \times s^L}{s^H}, \left\lceil \frac{\frac{(C_i^O - m_i^L \times D_i' \times s^L - L_i^O)}{s^H}}{(D_i - D_i') - \frac{L_i^O}{s^H}} \right\rceil \right\} \\ &= \max\left\{ \frac{m_i^L \times s^L}{s^H}, \left\lceil \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{(D_i - D_i') \times s^H - L_i^O} \right\rceil \right\} \end{aligned}$$

Refer to [1] for the formal proof.

▶ **Theorem 1.** *If there is not enough low-speed cores for task τ_i, m_i^L can be reduced to $m_i^L = \left\lceil \frac{C_i^N/s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})} \right\rceil$ by assigning m_i^H high-speed cores to τ_i during the HI-criticality mode, where $m_i^H \geq \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{(D_i - D_i') \times s^H - L_i^O}$.*

Proof. For any task τ_i, when there are not enough low-speed cores, we assign additional high-speed cores to τ_i to update m_i^L. Since $D_i' = \frac{C_i^N}{m_i^L \times s^L}$, we have

$$
\begin{aligned}
& m_i^H \geq \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{(D_i - D_i') \times s^H - L_i^O} \\
\Longrightarrow & (D_i - D_i') \times s^H - L_i^O \geq \frac{C_i^O - m_i^L \times D_i' \times s^L - L_i^O}{m_i^H} \\
\Longrightarrow & D_i \times s^H - \frac{C_i^N \times s^H}{m_i^L \times s^L} - L_i^O \geq \frac{C_i^O - C_i^N - L_i^O}{m_i^H} \\
\Longrightarrow & m_i^L \geq \frac{C_i^N \times s^H}{(D_i \times s^H - L_i^O - \frac{C_i^O - C_i^N - L_i^O}{m_i^H}) \times s^L} = \frac{C_i^N / s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})}
\end{aligned}
$$

which holds since $m_i^L = \left\lceil \frac{C_i^N / s^L}{(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{m_i^H \times s^H})} \right\rceil$. ◀

4.2 When Low-Speed Cores Are Not Sufficient

Section 4.1 determines the number of low-speed cores (m_i^L) required for each task, $\tau_i \in \tau$, to meet its deadline. If there are a finite number of low-speed cores, some tasks may not receive enough low-speed cores. In this section, we present Algorithm 1 that first tries to allocate the desired number (m_i^L) of low-speed cores to each task τ_i. If there are not enough low-speed cores, Algorithm 1 assigns additional high-speed cores to compensate for this shortage. In this approach, we assume that the processor speeds are given.

Algorithm 1 starts by checking whether a task τ_i is a Category-0 task (i.e., $C_i^O/(s^L \times D_i) \leq 1$). If yes, we use traditional multiprocessor scheduling (e.g., partitioned EDF) for sequential tasks with WCET C_i^O/s^L and deadline D_i on the minimum number of low-speed cores (Lines 3-4). If a task τ_i belongs to Category-1, i.e., $C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$, we allocate $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$ low-speed cores and meet its deadline D_i (Lines 5-6). Let, $\tilde{\tau}$ denotes the set of unscheduled tasks, which is updated continuously (Line 4 and Line 6). After allocating the required number of low-speed cores to all Category-0 and Category-1 tasks, we calculate the remaining low-speed cores, $\tilde{m}^L$ (Line 9).

For any task $\tau_i \in \tilde{\tau}$, we set m_i^L as $\left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$ and m_i^H (i.e., number of high-speed cores allocated to τ_i) to 0 (Lines 9-11). If all the tasks in $\tilde{\tau}$ receive the required number of low-speed cores, then the algorithm terminates (Lines 12-13). Else, we update the required number of high-speed cores allocated to task τ_i and denote it as m_i^H (Line 16), and also update m_i^L (Line 17). We calculate the relative core saving ratio (Line 15) and pick the task (say τ_i) that has the highest relative core saving ratio (Line 16). If there are sufficient cores for τ_i, we remove τ_i from $\tilde{\tau}$, and update $\tilde{m}^L$ (Line 18). If $\tilde{\tau}$ is empty, then the algorithm terminates successfully (Line 20). Else, repeat the same process to reduce the number of low-speed cores allocated to a task τ_i (Line 22). At any point, if cores are unavailable for task τ_i, then the task set is not schedulable and the algorithm returns failure. Upon successful completion, this algorithm greedily reduces the number of allocated low-speed cores by assigning (available) additional high-speed cores. If there are total K tasks in τ, the time complexity to calculate core allocation is $O(K)$.

▶ **Example 2.** Let us consider a platform with four low-speed cores of speed 0.75, and four high-speed cores of speed 1.0, two DAG tasks τ_1 and τ_2, where $\tau_1 = (12, 24, 4.08, 8.16, 20)$ and $\tau_2 = (12, 36, 3.36, 12.24, 33.12)$. Here, the low-speed and high-speed are normalized w.r.t. to

Algorithm 1 *greedyAlloc(τ).*

1 **Input:** The set of DAG tasks τ.
2 **Output:** Allocation of low(high)-speed cores to each task.
3 Unscheduled tasks, $\tilde{\tau} = \tau$; $\tilde{m}^L$ = *Available* LO*-speed cores*
4 **for** *(each* $\tau_i \in \tau$*)* **do**
5 **if** *(*$C_i^O/(s^L \times D_i) \leq 1$*)* **then**
6 use traditional multiprocessor scheduling for sequential tasks; $\tilde{\tau} = \tilde{\tau} - \tau_i$;
7 **else if** *(*$C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$*)* **then**
8 $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$; $\tilde{\tau} = \tau - \tau_i$;
9 $\tilde{m}^L = \tilde{m}^L - \sum_{\tau_i \in (\tau - \tilde{\tau})} m_i^L$, and $\forall_i m_i^H = 0$;
10 **for** *(each* $\tau_i \in \tilde{\tau}$*)* **do**
11 $m_i^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil$, and $\tilde{m}^L = \tilde{m}^L - m_i^L$;
12 **if** $\tilde{m}^L \geq 0$ **then**
13 $\forall \tau_i \in \tau$ allocate m_i^L and m_i^H cores and *RETURN SUCCESS*;
14 **else**
15 **for** *(each* $\tau_i \in \tilde{\tau}$*)* **do**
16 $m_i^H == 0 ? m_i^H = \lceil m_i^L \times s^L / s^H \rceil : (m_i^H = m_i^H + 1)$;
17 Update m_i^L as $\bar{m}_i^L$, where $\bar{m}_i^L = \left\lceil (C_i^N/s^L)/(D_i - \frac{L_i^O}{s^H} - \frac{C_i^O - C_i^N - L_i^O}{\bar{m}_i^H \times s^H}) \right\rceil$;
  $coreSaving_i = (m_i^L - \bar{m}_i^L)/(\bar{m}_i^H - m_i^H)$; ▷ *Relative core saving ratio of* τ_i
18 $maxCoreSaving = \forall_{\tau_i \in \tilde{\tau}} max(coreSaving_i)$;
19 **if** *(There are enough cores for* τ_i *)* **then**
20 $\tilde{\tau} = \tilde{\tau} - \tau_i$ and $\tilde{m}^L = \tilde{m}^L + (m_i^L - \bar{m}_i^L)$ ▷ *Update* $\tilde{\tau}$ *and* $\tilde{m}^L$
21 **if** $\tilde{\tau} = NIL$ **then**
22 *RETURN SUCCESS*;
23 **else**
24 Go to Line-9;
25 **else**
26 *RETURN FAILURE*;

maximum speed supported by this platform. Here, τ_1 is a category-1 task ($\frac{12}{0.75} > \frac{24}{1.0} - \frac{8.16}{1.0}$), and τ_2 is a category-2 task ($\frac{12}{0.75} < \frac{36}{1.0} - \frac{12.24}{1.0}$). For task τ_1, we calculate the required number of low-speed cores m_1^L as $m_1^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil = \left\lceil \frac{24 - 8.16}{20 \times 0.75 - 8.16} \right\rceil = 3$ (Line 9). Task τ_2 belongs to $\tilde{\tau}$, and we calculate the required number of low-speed cores m_2^L as $m_2^L = \left\lceil \frac{C_i^O - L_i^O}{D_i \times s^L - L_i^O} \right\rceil = \left\lceil \frac{36 - 12.24}{33.12 \times 0.75 - 12.24} \right\rceil = 2$ (Line 15). As there are not enough low-speed cores available for τ_2, two additional high-speed cores are allocated to τ_2 (Line 21). Now, τ_2 is removed from $\tilde{\tau}$ (Line 27). Both τ_1 and τ_2 receive the required number of cores and hence the algorithm terminates (Line 29).

5 Energy-aware Federated Scheduling for Multi-Speed Clustered Platform

We now describe how to allocate the low (or high) speed cores to each task. We extend the analysis presented in Sec. 4.1 to fit a multi-speed clustered platform, where different clusters offer different speeds. Hence, the energy consumption by different clusters (while

executing the same task) may vary significantly. For such a multi-speed clustered platform, we propose a task to cluster Hungarian assignment algorithm [36] to minimize the CPU energy consumption while satisfying the real-time schedulability guarantee.

5.1 Task to Cluster Assignment Approach

In this subsection, we discuss our approach to allocate all Category-1 (tasks with $C_i^N/s^L > C_i^O/s^H - L_i^O/s^H$) and Category-2 (tasks with $C_i^N/s^L < C_i^O/s^H - L_i^O/s^H$) DAG tasks into clusters, such that CPU energy consumption is reduced without violating the real-time guarantee. We assume that a task can not be allocated to multiple clusters.

Let, $\mathcal{Z}$ denotes the total number of available low-speed clusters, where each cluster is denoted as $\{\mathcal{K}_1, \mathcal{K}_2, \cdots \mathcal{K}_\mathcal{Z}\}$. We assume that each of these processors in the K^{th} cluster (where $1 \leq K \leq \mathcal{Z}$) execute at speed S_k. This assumption is motivated by Theorem 4 in [26], which asserted that executing a task with a consistent speed reduces the energy consumption significantly. In Section 4.1, we have shown the steps to determine the minimum number of processors to τ_i such that τ_i finishes execution within its deadline. That analysis assumes only two speed settings, i.e., low and high speed. Note that, different cluster offers different speed settings. Hence, for the same task, the required number of exclusively allocated processors may vary in different clusters. From now on, we assume these exclusively allocated processors form a *sub-cluster* inside the cluster. Let us assume that we know the number of available sub-clusters inside each cluster and each sub-cluster's size [3]. Let us denote a sub-cluster as $\mathcal{K}_m^n$ which denotes the n^{th} sub-cluster inside m^{th} cluster, and $\mathcal{K}_m^n \in \{\mathcal{K}_1^1, \mathcal{K}_1^2, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_\mathcal{Z}^1, \cdots, \mathcal{K}_\mathcal{Z}^y\}$.

Algorithm 2 *createTable*($\tau_{heavy}, \mathcal{K}$).

```
1  Input: The set of heavy DAGs τ_heavy and the set of sub-clusters
     K = {K_1^1, ··· , K_1^x, ··· , K_Z^1, ··· , K_Z^y}.
2  Output: A table 𝔼 storing the energy consumption value.
3  𝔼[size(τ_heavy)][size(K)];  /*Store energy consumption*/;
4  for x = 1 to size(τ_heavy) do
5      for y = 1 to size(K) do
6          if the considered sub-cluster satisfy the minimum allocation requirement then
7              Calculate E_x^y;      /* Energy consumed by task x in sub-cluster y */
8              𝔼[x][y] = E_x^y;
9          else
10             𝔼[x][y] = ∞; /* Set to a very large value */;
```

Let a task τ_i is allocated to the K^{th} cluster, and it needs m_i^K cores. We conclude that, there exists an $\mathcal{K}_K^n \in \mathcal{K}_K$ such that m_i^K fits to $\mathcal{K}_K^n$. Now, we calculate the energy consumed by task τ_i (when executing in a sub-cluster $\mathcal{K}_K^n$), which is denoted as $E_i^{\mathcal{K}_K^n}$. We repeat this step for all task $\tau_i \in \tau$, and for all sub-cluster $\in \{\mathcal{K}_1^1, \mathcal{K}_1^2, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_\mathcal{Z}^1, \cdots, \mathcal{K}_\mathcal{Z}^y\}$. Refer to Section 3 for details regarding energy consumption.

Algorithm 2 starts by creating a table $\mathbb{E}$, which stores the energy consumption by a DAG task when allocated to a sub-cluster (Line 3). Then, it traverses each heavy DAG task τ_i and each sub-cluster $\in \{\mathcal{K}_1^1, \mathcal{K}_1^2, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_\mathcal{Z}^1, \cdots, \mathcal{K}_\mathcal{Z}^y\}$ (Lines 4-5). Then it checks whether

[3] The number and size of sub-clusters (inside any $\mathcal{K}_K$) depend on which task is allocated to $\mathcal{K}_K$. We handle the task-cluster allocation in Section 5.1.

the DAG task can be allocated to a sub-cluster (Line 7), i.e., the number of cores available in this sub-cluster satisfies the minimum processor required (refer to Subsection 4.1 and 4.2) for this DAG task. If it satisfies the constraints, we store the energy consumed by this DAG task (when allocated to this sub-cluster) at Table $\mathbb{E}$ (Line 8). Else, we put an arbitrarily large value to $\mathbb{E}$ (Line 10). We do this to ensure that the scheduler will never assign a DAG task τ_i to any sub-cluster that does not have enough cores to execute τ_i.

Determining the number of sub-cluster and number of cores inside each sub-cluster. So far, we have discussed how to create the energy consumption table and find the task to sub-cluster allocation using the information presented in this table. However, we did not mention the total number of available sub-cluster inside each cluster and the number of cores in each sub-cluster. Recall that the cluster speed influences the minimum number of cores required (i.e., the sub-cluster size) for any task τ_i. As we do not know the task-cluster mapping, we are unaware of the available sub-clusters inside any cluster. As a preliminary approach, we divide any cluster $\mathcal{K}_K$ into M_K sub-cluster, where:

$$M_K = \begin{cases} \left\lfloor \frac{M}{m^K} \right\rfloor + 1, & \text{if } M(mod\ m_K) \neq 0 \\ \frac{M}{m^K}, & \text{Otherwise} \end{cases} \tag{3}$$

Here, M is the number of cores inside any cluster $\mathcal{K}_K \in \mathcal{K}$. We calculate m^K as $m^K = max\{m_i^K\}$, for all tasks $\tau_i \in \tau$. Here, m_i^K is calculated using the analysis provided in Subsection 4.1. Each of these M_K sub-clusters contains m^K cores, if $M(mod\ m^K) = 0$. Else, the first $M_K - 1$ sub-clusters contain m^K cores, and the remaining sub-cluster contains $M - (M_K - 1) \times m^K$ cores. Note that partitioning a cluster with respect to the task that needs the *maximum* number of cores (in this cluster) may seem pessimistic. This is because any other task that is also allocated in $\mathcal{K}_K$ may not need m^K cores. To tackle this pessimism, we will update the sub-cluster number and their size (in Algorithm 3) until all the tasks are scheduled, or the algorithm returns failure.

Task to Cluster Assignment. Now we know the energy consumption at all possible combinations of the DAG task to the sub-cluster mapping. We use this information to determine the processor allocation that provides the minimum energy consumption. At each sub-cluster, we assign a task that is not allocated to any other sub-cluster previously – we can pick a single entry from each row and column in the energy consumption table. The pseudo-code for this approach is presented in Algorithm 3.

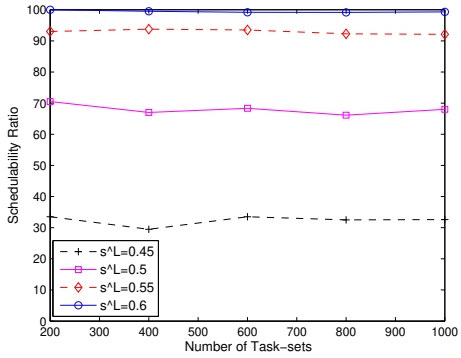
We determine the optimum assignment that minimizes the total energy consumption using the *Hungarian algorithm* [36] (Line 10). The algorithm takes the energy consumption table as input and returns an ordered collection of a task to sub-cluster allocation. The allocation provides the lowest combined energy consumption. The Hungarian algorithm has two significant advantages: it produces an optimal solution if the elements are non-negative (as in our case), and it has a polynomial complexity (i.e., affordable even for a large number of tasks). Note that the Hungarian algorithm works only when the input is an $N \times N$ square matrix. In our case, the energy consumption table may not be square in size. Hence, Algorithm 3 adds extra dummy rows in the table (to make it square in size) if the number of tasks is less than the number of sub-clusters, and fills them with arbitrary large values (Lines 6–9). Recall that we partition a cluster with respect to the task that needs the maximum number of cores (in this cluster), and it minimizes the number of sub-clusters in each cluster. Hence, some tasks may not get any sub-cluster, while some sub-clusters may

Algorithm 3 *taskToClusterAllocation*($\tau_{heavy}, \mathcal{K}$).

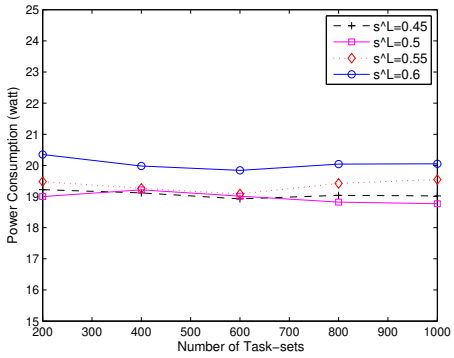
```
1  Input: The set of Category-1 heavy DAGs τ_heavy and the set of sub-clusters
     K = {K_1^1, ··· , K_1^x, ··· , K_Z^1, ··· , K_Z^y}.
2  Output: Processor to task allocation.
3  Set ∞ to 10^6    /* An arbitrary large value */;
4  createTable(τ_heavy, K)    /* The number of sub-clusters (in any cluster) is calculated using
     Equation (3) */
5  j = size(τ_heavy); k = size(K);
6  if j < k then
7      for x = j + 1 to k do
8          for y = 1 to k do
9              E[x][y] = ∞;   /*Dummy row makes E square */
10 Solve E using the Hungarian algorithm [36].
11 for i = 1 to n do
12     if τ_i is allocated in any sub-cluster in K_x^y ∈ K then
13         τ_heavy = τ_heavy − τ_i    /* Update task set */;
14         if (size(K_x^y) − m_i^x) > 0 then
15             /* y^th sub-cluster of K_x has idle cores */;
16             size(K_x^y) = size(K_x^y) − m_i^x;
17 if 0 < size(τ_heavy) < j then
18     Repeat taskToClusterAllocation(τ, K);
19 else if size(τ_heavy) == 0 then
20     return the optimal processor to task allocation and allocate the remaining light DAGs to
         remaining cores of each sub-cluster;
21 else
22     Return FAILURE;
```

remain underutilized. To tackle this issue, we continuously check for the tasks that get an allocation and remove them from the task set τ (Line 13). If any sub-cluster is underutilized (i.e., a task receives more cores than required), we update the sub-cluster size (Lines 14–16). We repeat Algorithm 3 if some tasks are removed from the task set, i.e., some update in the task set takes place, but some tasks are still unassigned to any cluster (Lines 17–18). When all tasks are allocated to some sub-cluster (i.e., $size(\tau)$ becomes 0). Algorithm 3 concludes by returning the task to sub-cluster allocation that results in minimum energy consumption (Lines 19–20). Else, i.e., no task receives any cores as there are not sufficient cores, the algorithm stops and returns failure (Line 22).

Algorithm 3 performs a task to cluster allocation considering the Category-1 DAG tasks and low-speed clusters. We use a slightly modified version of Algorithm 3 to allocate Category-2 tasks to the remaining low-speed cores. When all Category-1 tasks receive the required number of low-speed cores, we use Algorithm 3 again to allocate the remaining low-speed cores to the Category-2 DAG tasks. This time the input to Algorithm 3 is the set of Category-2 DAG tasks and the set of sub-clusters $\mathcal{K} = \{\mathcal{K}_1^1, \cdots, \mathcal{K}_1^x, \cdots, \mathcal{K}_{\mathcal{Z}}^1, \cdots, \mathcal{K}_{\mathcal{Z}}^y\}$ which has some low-speed cores unused. If the available low-speed cores are insufficient to accommodate all the Category-2 tasks, we call Algorithm 3 again with a modified input parameter (i.e., set of Category-2 DAG tasks that do not receive enough low-speed cores and set of sub-clusters containing high-speed cores) to allocate additional high-speed cores.



(a) Schedulability ratio for different s^L values.



(b) Power consumption for different s^L values.

Figure 1 Schedulability ratio and power consumption for different s^L values.

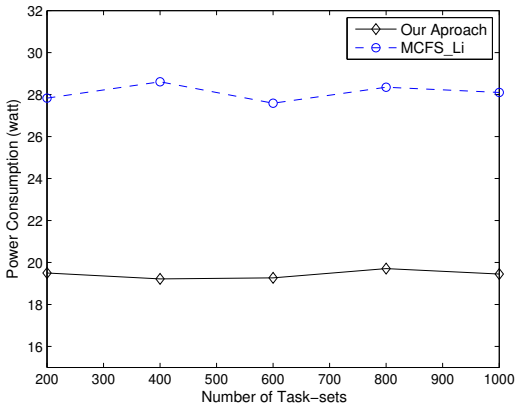


Figure 2 Power consumption comparison between our approach and MCFS_Li [39]. In this simulation, we set the value of s^L to 0.55.

6 Evaluation

This section demonstrates the algorithm's performance through evaluation conducted on a randomly generated task set. We report the *schedulability ratio* and the power consumption of our approach for different speed settings of a low-speed cluster. In this setup, the low-speed is normalized w.r.t. to the maximum speed supported by this platform, ranging from 0.45 to 0.6. We use the following parameters to generate the random task-set:

- ζ: number of task set, ranged between [200-1000].
- ζ_G: number of tasks per task set, ranged between [5-10].
- $D_i := xC_i^O$: relative deadline of a task, $x = [0.9 - 1.0]$.
- $[Z_{down}, Z_{up}]$: the range of the ratio of normal and overload execution budget. We set $1 \leq \frac{Z_{up}}{Z_{down}} \leq 8$.
- s^L: speed of the low-speed cluster, ranging [0.45-0.6].

The reference approach. To date, no work has investigated the same problem studied in this paper, i.e., minimize CPU power consumption for the DAG tasks by adopting the DPM policy in a clustered platform. Hence, we do not have a direct baseline to compare.

We consider a reference approach (for performance comparison) based on the DAG tasks scheduling [39], denoted as *MCFS_Li*. Similar to us, the reference approach has characterized a DAG task using its typical (and overload) execution requirement and the critical path length. The approach in [39] also proposed a core assignment to each DAGs. However, unlike us, [39] did not consider a multi-speed clustered platform. Hence, we assume that all the cores execute at the maximum speed (i.e., s^H) possible for the reference approach.

In Figure 1a, we vary the s^L values for a different size task set and report the schedulability ratio. As expected, the schedulability ratio is directly proportional to s^L, but not strongly correlated with with task-set size. Figure 1b shows the system power consumption for different s^L values over the different sizes of task sets. We see that energy consumption increases with a higher s^L value. In Figure 2, we compare our approach with an existing approach [39], denoted as MCFS_Li. We set the value of s^L to 0.55, and our approach leads to a power-saving of at least 29.23% compared to MCFS_Li (Figure 2). While guaranteeing real-time correctness, our approach utilizes the low-speed core as much as possible (Subsection 4.1 and 4.2), which leads to an energy-saving.

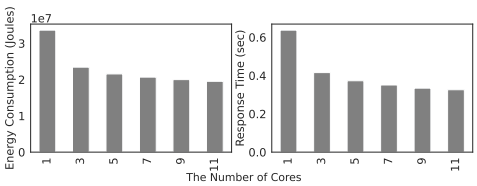
All these experiments in this work involve varying the number of randomly generated task sets. The aim is to investigate whether our proposed method's results are sensitive to changes in the number of tasks. However, our findings show no significant correlation between power consumption, schedulability ratio, and the number of tasks. This observation concludes that the proposed method is robust to different task set sizes.

7 Proof-of-Concept on Real-Time Platform Experiments

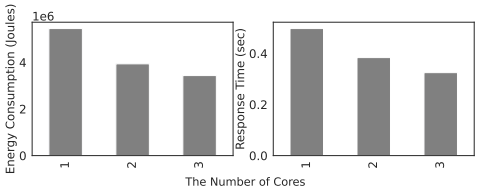
This section evaluates the proposed approach on a 12-core Intel Xeon 2680 v3 platform. The proposed method applies to ARM big.LITTLE architectures and modern Intel processors. However, the choice of the Xeon 2680 Processor stems from its well-studied power and energy consumption behavior [28], the ease of in-kernel status monitoring while having the per-core speed adjustment, per-core sleep, energy monitoring, and tracking the per-core temperature. The energy consumption behavior of modern Intel processors shows a significant deviation from energy models obtained on older Intel platforms due to latencies in changing a core's energy state, uncore frequencies, and out-of-order throttling at lower frequencies [57] – furthermore, documentation for turning off an entire cluster in ARM big.LITTLE architecture is sparse, and ARM does not provide public libraries for energy management.

On the platform, 11 of the 12 cores are isolated from kernel processes, user processes, and interrupts using `isolcpus` option in the kernel bios. Among these eleven cores, six cores represent Low-speed cores, and five cores represent High-speed cores. For a High-speed core, the minimum and maximum frequencies are normalized in the frequency range between 1.2 GHz and 2.5 GHz, respectively, with a minimum transition time of $20\mu s$. The minimum frequency of a Low-speed core is 1.2 GHz while the maximum frequency is a parameter of the evaluation. Additionally, each core can be independently turned off using DPM.

We conducted experiments on an Ubuntu 20.04 operating system, utilizing the default Completely Fair scheduler (CFS) introduced in Linux kernel version 2.6.23 [49]. The CFS scheduler is designed to allocate CPU times fairly among all runnable tasks on the system, making it ideal for our experiments. As a non-real-time scheduler, it provides a fair CPU time allocation among all runnable tasks on the system. We utilized the CFS scheduler by not specifying a scheduler type through the `sudo cset set` command. Our implementation of the proposed task-to-cluster allocation algorithm was written in Python. It periodically executed each benchmark task on the isolated 11 cores using the `cset` command-line option



(a) Correlation between response time and energy consumption of an OpenMP Benchmark Task on Xeon processor.



(b) Correlation between response time and energy consumption of an OpenMP Benchmark Task on Core i7 processor.

Figure 3 Change in energy consumption dependency with response time.

with a nice default value. In addition, our program modified kernel parameters, turned off appropriate clusters and monitored power consumption information from the allocated cluster. We used several average-case power management governors from the Linux Kernel to ensure a fair comparison, including schedutil, performance, powersave, conservative, and ondemand as baselines. Hyper-threading was turned off on all cores by adjusting the `scaling_max_freq` parameter in the Linux kernel. We fixed the frequency of each core using the `cpufrequtils` tool. Finally, we turned off Turbo mode to avoid unwanted frequency adjustments in each core.

The Barcelona OpenMP tasksuit (BOTS) [21] benchmark is used to evaluate the energy consumption of the proposed approach. Each of the 43 tasks within the BOTS benchmark follows the DAG task model discussed in Section 3. Nodes within a benchmark task are created using a task directive, while edges between nodes are generated using depend or taskwait directives. Nodes within a DAG task are scheduled using a greedy algorithm, as proposed in [37]. While intel p-state and c-state configurations transfer resource and power control to the hardware, we turned off this configuration for our application. We used advanced configuration and power interface (`ACPI`), which gives software access to touch voltage and frequency for speed adjustment and provides the baselines and `userspace` configuration. The `ACPI-Freq` is portable to other platforms.

To extract performance results, the `perf` Linux profiler and a hardware energy counter tool called reduced average power limit (`RAPL`) are used to monitor energy consumption during the execution of each DAG task [30]. Since the Linux profiler does not provide tool with the capability to profile multiple DAGs simultaneously, this evaluation focuses on evaluating single DAG tasks. This simplification is due to the limitation on profiling ring buffers and a socket hardware profiling register.

The paper proposes an approach for energy minimization while ensuring the schedulability of DAG tasks. Although other factors may affect the energy consumption in DAG benchmarks, including context switches, branch misses, and the number of instructions, these effects are out of the scope of this research. The correlation between response time and energy consumption is shown in Fig. 3, as we used the different numbers of cores. The results on Intel Xeon 2680 and core i7 show as we increased the number of cores, the energy consumption decreased on average on all the targeted benchmarks due to increasing the level of parallelism. When allocating only one core, the time to process increases, and the number of inactive cores adds much more waste on energy consumption.

This paper aims to determine the schedulability condition at runtime. To achieve this, we estimate DAG tasks' workload execution and critical path when executed at normal speed and allocate the number of cores accordingly. We assume the critical path can be determined when all cores are assigned to a DAG benchmark. Due to the absence of execution time

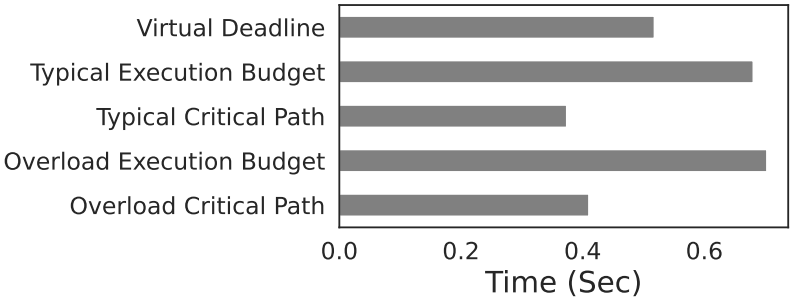


Figure 4 The response time of one DAG makespan is based on the critical path definition, workload execution time, and the virtual deadline.

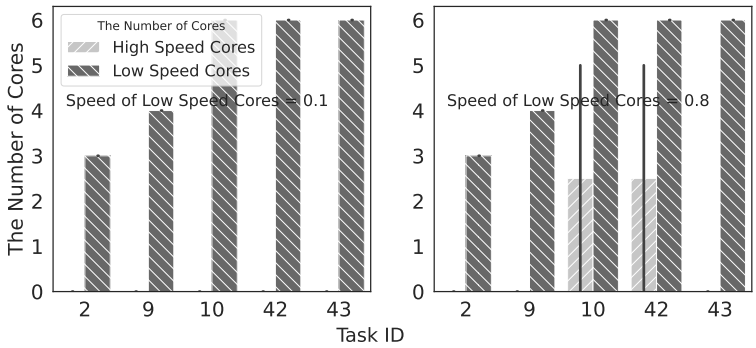


Figure 5 The light dash color refers to an increase in the number of high-speed cores due to the satisfaction of Category 2 in resource allocation.

tools for DAG tasks, the execution budget is obtained by running the DAG on a single core. From 20 corresponding iterations, the average value represents the typical condition, and the maximum value represents the overload condition. The results of this configuration for one task are shown in Fig. 4, and we obtain deadline constraints using Graham's bound. The virtual deadline is estimated using the following expression $VD < L + (C - L)/m$.

This paper defines different categories and allocates high-speed and low-speed cores based on core speeds and virtual deadlines. As explained in Section 4, there are three categories, and Category 2 involves meeting the condition $C_i^N/s^L < C_i^O/s^H - L_i^O/s^H$. This means that we may need to increase the speed of low-speed cores, decrease the speed of high-speed cores, or add high-speed cores to meet the required deadline. In theory, speed refers to the amount of computation in a given unit of time, while in experiments, it refers to the frequency of operation that determines the amount of computation that can occur in a given unit of time. Experiments utilize DVFS-enabled processors to control frequency within a predetermined range of frequencies. While the speed can theoretically range from 0 to 1, in practice, it is limited to the frequency range supported by the platform, as shown in Figure 5.

Based on Figure 5, the DAG tasks are assigned two low-speed and one high-speed core set. Six cores were assigned to low speed and five to high speed. In the experiments conducted for this paper, multiple clusters were used to test different scenarios where the number of low-speed and high-speed cores exceeded the schedulability requirements. Figure 5 shows an example where task 10 is allocated six low-speed cores with a speed of 0.1 and high-speed cores with a speed of 0.7 while maintaining schedulability. Similarly, this schedulability

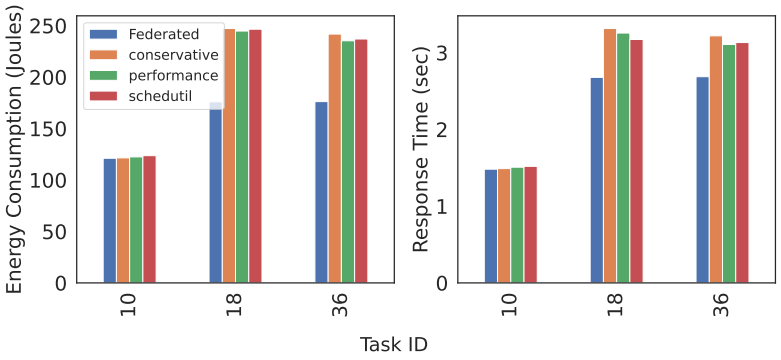
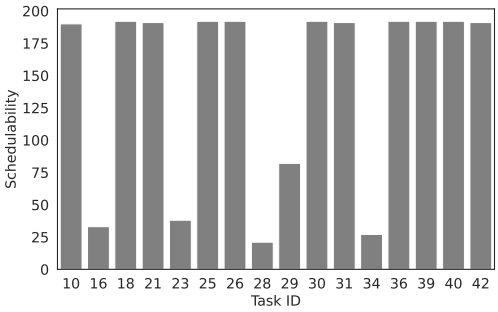
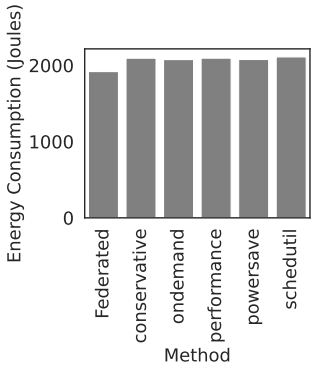


Figure 6 The federated energy-aware scheduling algorithm gives better results in terms of energy consumption and execution time compared to all the available Linux governors. Here task 18 is 30% better energy-efficient compared with the Linux governors.



(a) The schedulability of each DAG task for all the elements of clusters and sub-clusters in Algorithms 3. Having at least one schedulable task result means full schedulability in our results.



(b) Aggregated energy consumption of all the studied tasks versus the available linux governors shows 5% improvement over executing all the tasks.

Figure 7 Evaluation on schedulability and aggregated energy consumption.

can be achieved by increasing the number of high-speed cores or by increasing both the number and speed. To test energy efficiency under these conditions, we developed a clustering algorithm. Thus, cluster allocation is needed to get the optimal spot in energy minimization. We changed the speed of low-speed and high-speed cores to test the proposed method. We would fix the speed of high-speed cores to 0.7 of maximum speed in the Intel Xeon processor and observe the extra allocation of high-speed cores to heavy tasks in Fig. 5 as the speed of low-speed cores increases.

After setting the high and low-speed cores for each task and running the tasks in the directed acyclic graph (DAG) to create an energy matrix, we can accurately estimate the required low and high-speed cores. The results obtained with the proposed Federated scheduler were compared to state-of-the-art Linux governors, and it was found that the energy consumption was 30% less than the best result obtained from the governors, while the overall results were not worse than state-of-the-art. The advantage of this algorithm is that it takes schedulability conditions into account, unlike the Linux governors, which cannot do so.

The task to cluster allocation happens according to Algorithm 3, i.e., an energy value would be assigned to each table element according to the defined number of low(high)-speed cores and core's speed. We will allocate the low-speed cores based on the specified task categories. We would evaluate the conditions above requirements to fill the table. The element in the search table showing the optimal energy value would get the configuration needed for each task. The schedulability results in Fig. 7a for the table with 192 elements show we have at least one element in the search table, which follows the virtual deadline and schedulability conditions which means 100% schedulability all the time. While in Fig 6, we see some times 30% improvement in energy consumption on some tasks, the results show similar values in most of the evaluated tasks regarding all the times schedulability is met. The proposed algorithm represents results in Fig. 7b indicate an approximately 7% improvement in energy consumption when aggregating over all the tasks.

8 Conclusion and Future Work

The traditional workload model for real-time embedded systems focuses on worst-case behaviors to provide worst-case guarantees. However, modern embedded systems possess more energy constraints and require a richer system model to optimize energy consumption under typical scenarios instead of worst-case scenarios. In this work, we propose the energy-aware scheduling framework for DAGs in a clustered platform to minimize the typical-case energy while guaranteeing worst-case temporal correctness. Specifically, we propose determining the minimum number of low-speed cores required to schedule each DAG task under a dual-speed platform. If there are not enough low-speed cores, our algorithm assigns additional high-speed cores to a DAG, providing maximum energy-saving benefits. For multi-speed platform, we further propose a task to cluster partitioning approach to reduce the typical energy consumption without violating the worst-case real-time scheduling guarantee. We evaluate our algorithm via extensive simulations on randomly generated task set and report the energy consumption and the schedulability ratio. We also have implemented our algorithm on an Intel Xeon 2680 v3 platform and report that our approach reduces energy consumption by up to 30% w.r.t. the compared baseline.

Our current workload characterization assumes two thresholds: a typical execution length and a WCET. It would be interesting to study the situation when more detailed information can be provided, e.g., in the form of multiple thresholds, even with probability information. We also plan to investigate the impact of other components, e.g., cache misses, context switches, bus accesses, I/O usage, on the total power consumption. In the future, we plan to extend the evaluation to ARM big.LITTLE architecture and 12^{th} generation Intel Core i7 clustered mobile platform with four High-speed and eight Low-speed cores.

References

1 Kunal Agrawal, Sanjoy Baruah, Pontus Ekberg, and Jing Li. Optimal scheduling of measurement-based parallel real-time tasks. *Real-Time Systems*, pages 1–7, 2020.

2 Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 33–40. IEEE, 2003.

3 Arm a15 technical reference manual, 2013. `https://developer.arm.com/documentation/ddi0438/i/functional-description/power-management/dynamic-power-management?lang=en`.

4 Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):7, 2016.

5 Sanjoy Baruah. The federated scheduling of systems of mixed-criticality sporadic DAG tasks. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 227–236. IEEE, 2016.

6 Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo DAngelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154. IEEE, 2012.

7 Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM (JACM)*, 62(2):1–33, 2015.

8 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *ECRTS*, 2015.

9 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS*. IEEE, 2012.

10 Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient real-time scheduling of dag tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(5):84, 2018.

11 Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111, 2020.

12 Ashikahmed Bhuiyan, Federico Reghenzani, William Fornaciari, and Zhishan Guo. Optimizing energy in non-preemptive mixed-criticality scheduling by exploiting probabilistic information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3906–3917, 2020.

13 Ashikahmed Bhuiyan, Sai Sruti, Zhishan Guo, and Kecheng Yang. Precise scheduling of mixed-criticality tasks by varying processor speed. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 123–132, 2019.

14 Ashikahmed Bhuiyan, Kecheng Yang, Samsil Arefin, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Mixed-criticality multicore scheduling of real-time gang task systems. In *RTSS*. IEEE, 2019.

15 Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):31, 2009.

16 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *ECRTS*, 2013.

17 Gang Chen, Kai Huang, and Alois Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):111, 2014.

18 Hui Cheng and Steve Goddard. Online energy-aware I/O device scheduling for hard real-time systems. In *Proceedings of the conference on Design, automation and test in Europe:*. European Design and Automation Association, 2006.

19 Alexei Colin, Arvind Kandhalu, and Ragunathan Rajkumar. Energy-efficient allocation of real-time applications onto heterogeneous processors. In *RTCSA*, 2014.

20 Alberto Corpas, Luis Costero, Guillermo Botella, Francisco D Igual, Carlos García, and Manuel Rodríguez. Acceleration and energy consumption optimization in cascading classifiers for face detection on low-cost arm big. little asymmetric architectures. *International Journal of Circuit Theory and Applications*, 46(9):1756–1776, 2018.

21 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing*, pages 124–131. IEEE, 2009.

22 Pontus Ekberg and Wang Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-time systems*, 50(1):48–86, 2014.

23 David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272. IEEE, 2013.

24 Ana Guasque, Patricia Balbastre, Alfons Crespo, and Gerhard Fohler. Energy characterization of real-time partitioned systems. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 118–124. IEEE, 2018.

25 Zhishan Guo, Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, and Nan Guan. Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 156–168. IEEE, 2019.

26 Zhishan Guo, Ashikahmed Bhuiyan, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient multi-core scheduling for real-time DAG tasks. In *LIPIcs-Leibniz International Proceedings in Informatics*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

27 Zhishan Guo, Luca Santinelli, and Kecheng Yang. EDF schedulability analysis on mixed-criticality systems with permitted failure probability. In *RTCSA*. IEEE, 2015.

28 Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE, 2015.

29 Tarek Hagras and Jan Janecek. A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments. In *Parallel Processing Workshops*. IEEE, 2003.

30 Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

31 Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISLPED*. IEEE, 2007.

32 Jason Howard, Saurabh Dighe, Sriram R Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and DVFS for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, 2011.

33 Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C Buttazzo. Applying real-time interface and calculus for dynamic power management in hard real-time systems. *Real-Time Systems*, 47(2):163–193, 2011.

34 Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, and Lothar Thiele. Energy efficient DVFS scheduling for mixed-criticality systems. In *EMSOFT*. IEEE, 2014.

35 Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86. IEEE, 2015.

36 Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

37 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global EDF for parallel tasks. In *ECRTS*. IEEE, 2013.

38 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *ECRTS*. IEEE, 2014.

39 Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-time systems*, 53(5):760–811, 2017.

40 Keqin Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *The Journal of Supercomputing*, 2012.

41 Xinmei Li, Lei Mo, Angeliki Kritikakou, and Olivier Sentieys. Approximation-aware task deployment on heterogeneous multi-core platforms with dvfs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

42 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, 1995.

43 Di Liu, Jelena Spasic, Gang Chen, and Todor Stefanov. Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs. In *ESTIMedia*. IEEE, 2015.

44 José María López, José Luis Díaz, and Daniel F García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

45 Agostino Mascitti and Tommaso Cucinotta. Dynamic partitioned scheduling of real-time dag tasks on arm big. little architectures. In *29th International Conference on Real-Time Networks and Systems*, pages 1–11, 2021.

46 Alexander Maxiaguine, Simon Kunzli, and Lothar Thiele. Workload characterization model for tasks with variable execution demand. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1040–1045. IEEE, 2004.

47 Sujay Narayana, Pengcheng Huang, Georgia Giannopoulou, Lothar Thiele, and R Venkatesha Prasad. Exploring energy saving for mixed-criticality systems on multi-cores. In *RTAS*. IEEE, 2016.

48 Odroid xu-3, 2017. `http://www.hardkernel.com/`.

49 Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

50 Santiago Pagani and Jian-Jia Chen. Energy efficient task partitioning based on the single frequency approximation scheme. In *RTSS*. IEEE, 2013.

51 Santiago Pagani and Jian-Jia Chen. Energy efficiency analysis for the single frequency approximation (SFA) scheme. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):158, 2014.

52 Antonio Paolillo, Joël Goossens, Pradeep M Hettiarachchi, and Nathan Fisher. Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies. In *RTCSA*. IEEE, 2014.

53 Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *RTNS*. ACM, 2013.

54 Abusayeed Saifullah, Sezana Fahmida, Venkata P Modekurthy, Nathan Fisher, and Zhishan Guo. CPU energy-aware parallel real-time scheduling. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

55 Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

56 Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 2013.

57 Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy efficiency features of the intel skylake-sp processor and their impact on performance. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 399–406. IEEE, 2019.

58 Youngsoo Shin and Kiyoung Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings 1999 Design Automation Conference*, pages 134–139. IEEE, 1999.

59 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*. IEEE, 2007.

60 Kecheng Yang, Ashikahmed Bhuiyan, and Zhishan Guo. F2vd: Fluid rates to virtual deadlines for precise mixed-criticality scheduling on a varying-speed processor. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

61 Dakai Zhu, Nevine AbouGhazaleh, Daniel Mossé, and Rami Melhem. Power aware scheduling for and/or graphs in multiprocessor real-time systems. In *ICPP*. IEEE, 2002.

62 Dakai Zhu, Daniel Mosse, and Rami Melhem. Power-aware scheduling for and/or graphs in real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):849–864, 2004.