Noname manuscript No.
(will be inserted by the editor)

Optimizing Resource Speed for Two-Stage Real-Time Tasks

Alessandra Melani · Renato Mancuso · Daniel Cullina · Marco Caccamo · Lothar Thiele

Received: date / Accepted: date

Abstract Multiple resource co-scheduling algorithms and pipelined execution models are becoming increasingly popular, as they better capture the heterogeneous nature of modern architectures. The problem of scheduling tasks composed of multiple stages tied to different resources goes under the name of "flow-shop scheduling". This problem, studied since the '50s to optimize production plants, is known to be NP-hard in the general case. In this paper, we consider a specific instance of the flow-shop task model that captures the behavior of a two-resource (DMA-CPU) system. In this setting, we study the problem of selecting the optimal operating speed of the two resources with the goal of minimizing power usage while meeting real-time schedulability constraints. In particular, we derive an algorithm that finds the optimal speed of one resource while the speed of the other resource is kept constant. Then, we discuss how to extend the proposed approach to jointly optimize the speed of the two resources. In addition, applications to multiprocessor systems and energy minimization are considered. All the proposed algorithms run in polynomial time, hence they are suitable for online operation even in the presence of variable real-time workload.

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1219064 and CNS-1302563. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

A. Melani
Scuola Superiore Sant'Anna, Pisa, Italy
E-mail: alessandra.melani@sssup.it

R. Mancuso, D. Cullina and M. Caccamo
University of Illinois at Urbana-Champaign, USA
E-mail: {rmancus2, dcullina, mcaccamo}@illinois.edu

L. Thiele
Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
E-mail: thiele@ethz.ch

Keywords Co-scheduling · Schedulability analysis · Flow-shop scheduling · Multi-stage model · Multi-resource model · Power saving · Energy saving · Speed optimization · Real-time systems

Preliminary Publication

This work is an extended version of the DATE 2016 paper on *Speed Optimization for Tasks with Two Resources* [22]. With respect to the conference version, this paper provides a more exhaustive explanation and illustrative examples about the main results. In addition, the following novel contributions are introduced:

- In Section 6.2, we extend our approach to power minimization by allowing two degrees of freedom for speed selection, i.e., we derive an algorithm that *jointly* optimizes the speed of both resources.
- In Section 7.1, we derive a heuristic approach for optimizing resource speed in a partitioned multi-core system with a single DMA channel.
- We discuss in Section 7.2 what are the required steps to extend our technique to minimize energy consumption instead of instantaneous power usage.
- We strengthen the motivation of the adopted task model by formally proving the NP-completeness of the two-stage flow-shop problem with intermediate deadlines. The proof is reported in Section 8.
- In Section 9, we experimentally evaluate the performance of the proposed algorithms, both in the single- and multi-core setting.

1 Introduction

The current trend in embedded systems industry is to exploit the high degree of parallelism offered by modern architectures. In fact, they are rich in computational resources and offer a plethora of specialized components capable of efficiently performing specific sub-tasks. For instance, in scratchpad-based architectures, data engines (DMAs) are first used to load the task to be executed, or the data to be processed, onto the scratchpad. Next, the loaded task is executed on the CPU. Similarly, if we consider hybrid CPU-GPU architectures, the CPU is responsible for device initialization and data preparation, while image processing kernels are dispatched to the GPU to boost performance. In the real-time literature, execution models [24, 36] have been proposed whose tasks are composed by two phases: a memory phase and an execution phase. During the former one, task data are loaded from main memory to cache or scratchpad memory; during the latter phase, the task is executed using the preloaded data. A precedence constraint between memory and execution phase exists because the execution of a task cannot begin before the required data have been loaded. Moreover, while the execution phase is performed on the

CPU, data load can be carried out using a DMA. Thus, two phases of different tasks can be performed in parallel.

In general, the class of scheduling problems for multiple-stage jobs that execute on an ordered sequence of resources takes the name of “flow-shop scheduling”. This class of problems has been largely studied since the early ’50s because it is also relevant to schedule resources and assembly phases in production plants. The problem of selecting the optimal schedule for flow-shop jobs with more than two stages, however, has been proven to be NP-hard in [14]. In this work, we focus on flow-shop tasks characterized by *two stages* and consider a task model where: a) each task stage requires to be executed on a specific type of resource, and b) one resource of each type exists in the system. Given this setup, we study the problem of determining the *minimum speed* at which one or both resources can be operated such that real-time schedulability constraints are met. Specifically, we propose an on-line algorithm that, given a batch of jobs with the same deadline as input, determines the minimum (optimal) speed at which to operate the two resources subject to deadline constraints. The proposed algorithm runs in polynomial time and is thus suitable for online operation in open systems, where real-time workload changes at run-time and the system needs to adapt its scheduling policy. In order to solve the described problem and without loss of generality, we instantiate our model on a DMA-CPU scenario. First, we derive our results assuming fixed DMA speed and variable CPU speed; then, we show how the same approach can be entirely reused for an equivalent system where the speed of the first resource is varied instead; next, we show a possible approach to optimize power usage considering the speed variation of both resources; finally, we propose a (non-optimal) extension targeting partitioned multiprocessor systems. In the following, speed is quantified in terms of variable clock period of the computing resource. As clarified in Section 5, this allows us to reason on piecewise linear functions, so that it is easier for the reader to follow the proposed results.

In a nutshell, this work introduces a novel and efficient (polynomial-time) algorithm that derives the optimal speed of resources, either memory, CPU or both, when single-rate periodic tasks that run across two stages of single-unit resources are considered. The selected speed allows optimizing power usage while ensuring that schedulability constraints are met. Moreover, in this work, we provide additional insights about the hardness of two-stage flow-shop problem in the more general setting with intermediate deadlines. Additionally, applications to multiprocessor systems and energy consumption minimization are discussed. Finally, we perform a quantitative evaluation of the proposed approach in comparison to simple heuristics.

Organization of the paper. The remainder of this paper is organized as follows. In Section 2, we review related work. In Section 3, we present the adopted system model and assumptions, while Section 4 establishes the necessary background. Next, we describe the proposed algorithm and its complexity in Sections 5 and 6. Additional extensions targeting multiprocessor systems and energy consumption are discussed in Section 7. Section 8 contains hard-

ness considerations for the more general scheduling problem with intermediate deadlines. In Section 9 we evaluate the performance of our approach. Finally, Section 10 concludes the paper and outlines the future work.

2 Related Work

The flow-shop problem has been extensively studied by the combinatorial optimization community, especially in the context of production scheduling. In 1954, Johnson proposed an optimal solution to the flow-shop problem in the case of a two-stage production facility and a collection of independent jobs to be processed in sequence on the two resources [20]. More recently, the flow-shop problem has been shown to be strongly NP-hard if jobs consist of more than two stages [14], or if two or more resources are available for each stage [15]. To overcome such limitations, several heuristic solutions [8] and polynomial-time approximation schemes (PTAS) [27] have been proposed.

Lately, along with the advent of modern embedded systems, the real-time scheduling community renewed the interest towards multi-stage execution models. In the embedded and high-performance domain, co-scheduling algorithms are increasingly used to bound the memory interference due to concurrent accesses to shared memory by different cores [26]. An attempt in this direction has been pursued by Pellizzoni et al. [24], who introduced the PRedictable Execution Model (PREM). This scheduling framework models a task as comprised of two distinct phases: a memory phase, where the task context is loaded into local memory, and an execution phase where the task executes with no memory contention. Schedulability analyses for PREM tasks have been proposed in [36, 1, 2, 23].

While the main objective of PREM is to implement the load-execute task model on COTS cache-based architectures, this model represents the commonly adopted one for scratchpad-based architectures. A number of works that focus on schedulability analysis of real-time tasks on scratchpad-based systems employ a two-stage, two-resource task model [13, 34, 35]. These works are concerned with the arrangement of scratchpad memory in space and load-/unload operations in time. While we inherently share similarities in the task model adopted, to the best of our knowledge none of the existing works considers the problem of deriving the optimal speed of the two resources while satisfying real-time constraints. By restricting our setting to the case where an optimal schedule can be computed, we take a first step in this direction by deriving the minimum operating speed for the computing or memory resource (or a combination of the two speeds) that satisfies schedulability constraints.

The design and implementation of a scratchpad-centric OS was proposed in [29]. In this work, a three-stage (load, execution, unload) task model is used. The load and unload phases are performed using a single DMA engine, while execution phases are performed in parallel on two application cores. During the load phase, the DMA engine copies the task code/data image from main memory to local memory (scratchpad), so that contention-free execution can

be performed. Although a schedulability analysis is provided, two main differences substantially set this work apart. First, in [29] the goal is to achieve time-deterministic task execution on commercial multi-core platforms, rather than power/energy optimization. Second, in this work we discuss scheduling strategies that specifically target two-stage tasks. Conversely, in [29] partitioned Rate Monotonic and TDMA for load/unload operations are used.

The analysis of multi-stage tasks has some similarities with the literature about DAG task analysis [28, 7, 5]. Two main differences, however, set this work apart. First, we consider only one precedence constraint per job: execution phase cannot start before memory phase has been completed. Second, the execution of each job phase is tied to a specific resource type. In addition, we consider task parameters that variate as speed of resources is adjusted to achieve power efficiency. Some works about DAG-task scheduling have also added further optimization dimensions, and thus can be considered as resource co-scheduling problems. For instance, the work in [33] proposes a ILP-based technique and an heuristic to optimize scheduling of DAG-tasks while considering the inter-processor communication overhead and the memory footprint of message buffering. In [19], a non-linear programming formulation is proposed to jointly optimize temperature and performance for applications with loops deployed on hybrid memory architectures (containing both on-chip cache and scratchpad memory).

3 System Model

While our results can be applied to generic two-resource flow-shop tasks, we instantiate our problem on traditional computing platforms, considering DMA-CPU tasks (e.g., PREM tasks). Thus, we express tasks as composed of a memory phase (M-phase), followed by a computation phase (C-phase). More formally, we consider a set $\mathcal{T}$ of n periodic real-time tasks $\tau_1, \ldots, \tau_n$. We assume that the two resources can operate with any value of clock period[1] in the range $T_{ck} \in [1, +\infty)$. Each task τ_i is defined by a worst-case memory-access time M_i and a worst-case computation time C_i, relative to the initial configuration where the clock period T_{ck} is equal to 1, i.e., is the minimum possible. Hence, if the objective is to optimize the speed of the first (resp., second) resource, the memory-access time M_i (resp., computation time C_i) of each task is linearly scaled[2] as $M_i^t = M_i \cdot t$ (resp., $C_i^t = C_i \cdot t$) for any value of $T_{ck} = t \geq 1$, while the speed of the other resource is kept constant

[1] Reasoning in terms of clock period T_{ck} describes well the performance of CPUs; however, it is more appropriate to reason in terms of bandwidth when describing the performance of memory subsystems. Since a dualism exists between the two concepts, we will adopt the notation T_{ck} when referring to either resource type.

[2] Computation is performed over data that has been preloaded into local memory, while memory operations do not involve computation. Thus computation time scales linearly with clock speed as long as CPU speed and local memory are tied to the same clock. Similarly, performance of memory-only operations scales linearly with the configured transfer bandwidth.

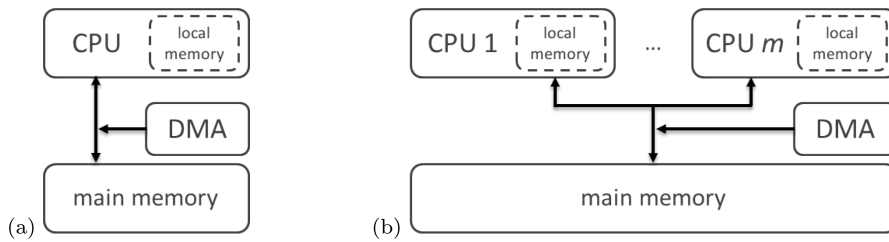


Fig. 1: Considered platform model for single-core (a) and multi-core (b) case.

in the considered platform. The underlying assumption is that the speed of each resource can be varied continuously. Albeit this is not true in general, it is worth noticing that increasing attention is given to advanced power scaling features in all modern architectures, from embedded platforms to data-centers. Since it is known that operating frequency has a directly proportional impact on power usage [25], modern platforms are endowed with Dynamic Voltage and Frequency Scaling (DVFS) units that allow live adjustments of the operating frequencies at a fine granularity. For example, the Nvidia Tegra K1 SoC[3] is a hybrid CPU-GPU architecture designed for embedded applications that allows for nine levels of frequency scaling on its low-power CPU cores, twenty levels for its high-power CPUs and fifteen levels for its GPU. Similarly, the Intel i7 4770K[4], designed for workstation machines, provides sixteen frequency scaling levels.

Additionally, we assume that the power usage of both resources is defined by their *power profile*. In particular, $P_M(s)$ denotes the *memory access power profile*, i.e., the function that associates the power used while accessing memory to any value of clock speed $T_{ck}^{-1} = 1/t = s$. Analogously, $P_C(s)$ denotes the *computation power profile*, i.e., the function that expresses the power used while performing computation for any value of clock speed.

The following assumptions are made on the power profiles:

- $P_M(s)$ is a monotonically increasing linear function, with slope $\alpha > 0$. This trend is in line with typically available mechanisms to perform in-hardware bandwidth control of DMA engines. In fact, as explained later, DMA speed is controlled by configuring the number of idle cycles between the transfer of two memory blocks. This leads to a linear correlation between selected speed and power level;
- $P_C(s)$ is a monotonically increasing convex function, with the intuitive physical interpretation that, given a fixed energy budget, the amount of computation that can be performed can only decrease as the CPU speed increases. This is a known trend commonly assumed in the literature [4].

The platform model considered in this paper is depicted in Figure 1. In the single-core case (Figure 1a), the execution phase of each task is carried

[3] See http://www.nvidia.com/object/tegra-k1-processor.html
[4] See http://ark.intel.com/products/75123/Intel-Core-i7-4770K

out on the CPU. The CPU has a local memory where the task code/data is located after the memory phase is completed. As a given task executes on the CPU, a memory phase for a different task can be performed in parallel using the DMA engine. For the sake of this work, we assume that the local memory is large enough in size to accommodate the payload of all the simultaneously released jobs. In the multi-core case (Figure 1b), $m > 1$ CPUs use a shared interconnect to access a single block of main memory. Each CPU embeds a private local memory where application tasks can be loaded using a DMA engine. Even if multiple CPUs are present, a single DMA engine is shared by all the CPUs. Note that when more than one DMA is available, it is possible to reason on the system by defining clusters of CPUs that share a single DMA engine. Hence the model in Figure 1b can be re-used.

While frequency scaling is an effective way to scale performance and power usage on CPUs, DMA engines are regulated using bandwidth control (BWC) features instead. BWC features allow specifying the number of intermediate idle states between every block of transferred bytes in a DMA operation. Since both the transfer block size and the number of idle states are a configurable parameter, BWC features provide a fine granularity of performance scaling. Note that since speed and power control on memory phases is enforced through DMA's BWC features, no frequency scaling is performed or required on the system memory or bus. Table 1 reports a non-exhaustive list of commercially available platforms that are compliant with the model considered in Figure 1. For instance, the Freescale MPC5777M SoC[5] can be configured to perform DMA operations with up to eight idle states between two memory transactions, and with block sizes ranging from 1 byte to 16 bytes. Similarly, the Freescale P4080[6] features eleven levels of DMA throttling with four configurable transfer block sizes.

Features	**MPC5777M**	**MPC5746M**	**TMS320C66x**
Local memories	✓	✓	✓
DMA engines	✓	✓	✓
Nr. of cores	3	3	8

Table 1: Suitable Commercial Multicore COTS platforms.

We assume that all tasks share the same relative deadline D, which is constrained to be smaller than or equal to their period T. Therefore, starting from an arbitrary time r when all tasks release their first job, subsequent job releases of all tasks will happen at times $r + kT$, being k any positive integer. Each job released by τ_i first executes its M-phase on a data processor, and then executes its C-phase on a CPU. Thus, M-phases (C-phases) of a given

[5] See http://cache.freescale.com/files/32bit/doc/fact_sheet/MPC5777MFS.pdf
[6] See http://cache.freescale.com/files/32bit/doc/prod_brief/P4080PB.pdf

task τ_i can progress in parallel with C-phases (M-phases) of a different task τ_j.

Since in our model tasks are synchronously released and have the same deadline D, preemption does not provide any schedulability advantage. Thereby, our results do not use preemption. Also, modeling resources as non-preemptive allows capturing the realistic behavior of some resources. For instance, DMA operations can be aborted/canceled, but it is often not safe to assume that a certain amount of data has been successfully transferred.

For a given schedule of jobs in a period, the *makespan* is defined as the time between the release of the jobs and the completion of the C-phase of the last job in the schedule. In this setting, the schedulability problem (i.e., verify whether deadlines are met) is equivalent to the problem of makespan minimization, for which an optimal solution that runs in polynomial time exists [20]. More specifically, since one job of each task is released at multiples of T, it is enough to compute the (optimal) makespan of such a collection of jobs and check it against the global relative deadline $D \leq T$ to verify the schedulability of a given task-set. Since the execution pattern repeats identically in each period, we can restrict our analysis to consider only the first instance of each task, denoting such a collection of jobs as $J_1, \ldots, J_n$. Without loss of generality, we assume all such jobs to be released at time 0 and to have deadline at time D.

We remark that, when more general task models are considered, the problem loses some of the desirable properties it has in the case of two resources and two-stage tasks. We previously mentioned that the makespan minimization problem becomes NP-hard when each task consists of more than two phases [14] or when two or more resources are available for each stage [15]. Moreover, if tasks do not share the same period/deadline, the schedulability problem is no longer equivalent to that of makespan minimization. In particular, we prove in Section 8 that the flow-shop scheduling problem with intermediate deadlines is indeed NP-complete.

Nonetheless, despite the negative results on the tractability of the problem in the generic case, significant performance gains can be achieved by devising novel co-scheduling policies able to exploit the potential parallelism and energy saving offered by modern embedded architectures.

This work addresses the broader problem of speed selection for the class of multi-stage execution models. We envision that future research can extend this work to encompass task models with different rates. Note that in this work we discuss how our solution can be applied, albeit not optimally, to partitioned multiprocessor systems (Section 7.1). This allows a straightforward extension to platforms featuring multiple DMA engines, each used by a set of processors. In this case, it is possible to reason in terms of isolated clusters. Hence, the assumption on a single system-wise task rate can be lifted, as long as tasks allocated to the same cluster are tied to the same period.

4 Background

In this section, we provide the necessary background for the reader to understand the analogies of our scheduling problem with the well-known "flow-shop" problem.

4.1 Johnson's algorithm

Johnson's algorithm [20] provides an optimal solution to schedule a collection of same-deadline, time-synchronized two-resource tasks.

The steps of Johnson's algorithm for constructing an optimal schedule are the following:

1. Partition the jobs into two sets S_1 and S_2. S_1 contains the jobs having $M_i < C_i$, S_2 contains the jobs with $M_i > C_i$. The jobs with $M_i = C_i$ may be put in either set;
2. Jobs in S_1 are sorted in ascending order of M_i, while jobs in S_2 are sorted in descending order of C_i;
3. The final ordering is obtained by concatenating the two sequences as $S = [S_1; S_2]$.

The cost of sorting the two sets dominates over other operations, hence the time complexity of Johnson's algorithm is $O(n\ log(n))$.

The following theorem defines the relative ordering between pairs of jobs in an optimal schedule derived by Johnson's algorithm.

Theorem 4.1 *(from [20]) Given a collection $J_1, \ldots, J_n$ of two-stage flow-shop jobs, J_i precedes J_j in an optimal schedule if*

$$\min(M_i, C_j) < \min(M_j, C_i). \tag{1}$$

In case of equality, either ordering is optimal, provided it is consistent with all the definite relations between the other jobs.

In the rest of the paper, we denote as $\sigma^t = \{\sigma_1^t, \ldots, \sigma_n^t\}$ a generic schedule, while we indicate as $\hat{\sigma}^t = \{\hat{\sigma}_1^t, \ldots, \hat{\sigma}_n^t\}$ the *optimal schedule*, i.e., the permutation of jobs that determines the minimum makespan, given a particular value of clock period $T_{ck} = t$.

4.2 Computing the optimal makespan

The *makespan* μ^t of task-set $\mathcal{T}$ corresponds to the time that elapses between the activation of any instance of $\mathcal{T}$ and the completion of the last job in a generic schedule σ^t. Note that when the optimal schedule $\hat{\sigma}^t$ is considered, μ^t corresponds to the *minimum* makespan for the task-set $\mathcal{T}$. In the remainder of this paper, we will say that a job J_i *contributes* to the makespan of schedule σ^t with its M-phase (resp., C-phase) to indicate that the term M_i (resp., $t \cdot C_i$)

belongs to the longest path in the usage of the two resources that determines the value of μ^t, and therefore is accounted for when computing the makespan.

In order to describe how to compute μ^t, we first introduce the notion of *crossover job*, which applies to any generic schedule (not necessarily optimal). Intuitively, the crossover job J^* determines the actual makespan, because it identifies the *critical path* in the usage of the two resources. Specifically, the crossover job is such that jobs preceding it in the schedule contribute to μ^t with their memory-access time, while subsequent jobs contribute to it with their computation time; instead, J^* is the only job that contributes to μ^t with both its execution phases. The following theorem formalizes this notion.

Theorem 4.2 *Given a job ordering $\sigma^t = \{\sigma^t_1, \ldots, \sigma^t_n\}$, each job J_i in the sequence contributes to the makespan with either the term C_i or M_i (but not both), except for a single job J^* that contributes with both M^* and C^*. The latter job is called* crossover job*.*

Proof Since memory phases of different jobs do not have precedence constraints, there are no gaps in the usage of the first resource, which is occupied for $M = \sum_{i=1}^{n} M_i$ time units. Conversely, gaps are possible in the usage of the second resource whenever there exist two jobs J_s and J_e, with $s < e$, such that

(i.) $\sum_{i=s+1}^{e} M_i > \sum_{i=s}^{e-1} C_i$;
(ii.) job J_e starts executing its C-phase as soon as its M-phase has terminated.

While Condition (i.) identifies the relation among execution times to produce a gap in the usage of the second resource, Condition (ii.) delimits the gap extension by selecting as job J_e the first job that initiates a sequence of contiguous computations after the gap.

As a visual example, consider the schedule in Figure 2(a), consisting of three jobs J_1, J_2, J_3. In this case, a gap in the usage of the second resource exists between the execution of C_2 and C_3. Conditions (i.) and (ii.) are verified by $J_s = J_2$ and $J_e = J_3$. In fact, J_3 starts executing its C-phase as soon as its M-phase has terminated, delimiting the gap in the usage of the second resource.

Let us now consider the maximum possible value of s and e such that Conditions (i.) and (ii.) above are satisfied. From the way s and e are selected, it follows that no gaps in the usage of the second resource are possible after the execution of C_e for job J_e. Hence, all jobs J_i with $i > e$ only contribute to the makespan with the term C_i. Conversely, all jobs J_i with $i < e$ contribute to the makespan with the term M_i, since a gap in the usage of the second resource exists between the execution of C_{e-1} and C_e, and the memory phases of all jobs are executed with no gaps. Finally, the execution of J_e contributes to the makespan with both M_e and C_e. Hence, J_e is unique and corresponds to the crossover job J^*. □

In the example of Figure 2(a), J_3 is the crossover job, since $s = 2$ and $e = 3$ are the maximum job indexes satisfying Conditions (i.) and (ii.). Indeed, J_3 contributes with both M_3 and C_3 to the makespan, while preceding jobs only contribute with their M-phase.

We rely on the result in Theorem 4.2 to calculate the makespan μ^t of a schedule σ^t.

Theorem 4.3 *Given a job ordering $\sigma^t = \{\sigma_1^t, \ldots, \sigma_n^t\}$, the corresponding value of makespan μ^t is given by:*

$$\mu^t = \max_{i=1}^{n} \left(\sum_{j=1}^{i} M_j + t \cdot \sum_{j=i}^{n} C_j \right), \quad (2)$$

and the value of i that maximizes the expression corresponds to the index of the crossover job J^* in σ^t.

Proof As previously stated, each job J_i except J^* contributes with either C_i or M_i to the makespan. Equation (2) captures the latter consideration by maximizing the sum of first- and second-resource-only contribution around a pivot job. The pivot job is accounted for both contributions. When the crossover job J^* is considered as a pivot, it follows that the first-resource-only (second-resource-only) contribution on the left (right) of J^* is greater than the second-resource-only (first-resource-only) contribution. Hence, the theorem follows. □

Note also that Equation (2) can be implemented efficiently, that is, to run in linear time in the size of the task-set.

5 Optimal resource speed selection

In this section, we present our algorithm to derive the minimum resource speed that guarantees the schedulability of a given task-set. In the rest of the paper, we will denote as $F_C(t)$ (resp., $F_M(t)$) the function that associates the value of the optimal makespan to any value of clock period $T_{ck} = t$ when variations in the speed of the second (resp., first) resource are considered. For ease of understanding, we will instantiate the problem in the case of a fixed DMA speed and a variable CPU speed, and then show how to reuse the same approach when considering variations in the speed of the first resource.

For any fixed value of T_{ck}, Johnson's algorithm (see Section 4.1) can be used to find the job ordering that corresponds to the minimum makespan. However, as the clock period is scaled, the value of the optimal makespan increases, due to the scaling factor applied to computation times. Additionally, depending on the scheduling decisions imposed by Equation (1), jobs can be possibly rearranged in a different order. As an example, consider a task-set composed of three tasks $\tau_1 = (M_1, C_1) = (4, 4)$, $\tau_2 = (3, 2)$ and $\tau_3 = (5, 1)$, with $T = D = 20$. Initially, when $T_{ck} = 1$, Johnson's algorithm orders the

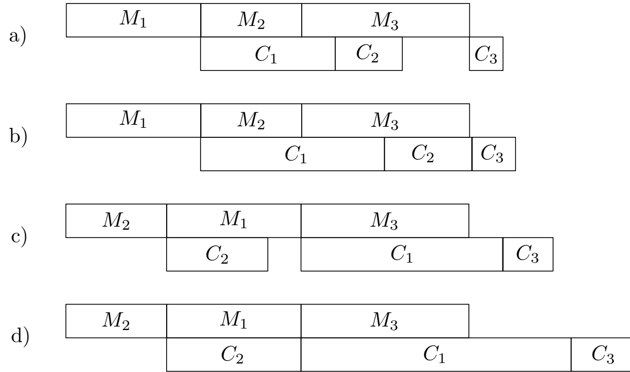


Fig. 2: Example of a task-set composed of three tasks with parameters $\tau_1 = (4, 4)$, $\tau_2 = (3, 2)$, $\tau_3 = (5, 1)$. The four insets illustrate the optimal schedule when a) $T_{ck} = 1$; b) $T_{ck} = 1.33$; c) $T_{ck} = 1.5$; d) $T_{ck} = 2$.

jobs as depicted in Figure 2(a), with J_3 being the crossover job. The optimal makespan, equal to 13, can be found by Equation (2), where the maximum is achieved for $i = 3$. As the clock period is scaled, the optimal makespan linearly increases, due to the inflation of the C-phase of J_3. However, when the value of C_1+C_2 reaches that of M_2+M_3, i.e., when $T_{ck} = (M_2+M_3)/(C_1+C_2) = 1.33$, J_1 becomes the crossover job, as shown in Figure 2(b), and the makespan increases at a higher rate. Then, as soon as the computation time of τ_2 reaches the value of its memory-access time (i.e., when $T_{ck} = M_2/C_2 = 1.5$), Johnson's algorithm imposes a job reordering (see Figure 2(c)) that reduces the makespan growth rate. Finally, the rate of the optimal makespan will increase again as soon as C_2 reaches the value of M_1, i.e., when $T_{ck} = M_1/C_2 = 2$, because from this point there is no gap in the processor usage, and all three jobs will contribute to the makespan with their computation times (see Figure 2(d)).

Figure 3 illustrates the function $F_C(t)$ for the example above. We can immediately observe that such a function: (a) is monotonically increasing; (b) is piecewise linear; and (c) the points where its slope changes (i.e., $T_{ck} = \{1.33, 1.5, 2\}$) are exactly those described in Figure 2. The intersection point between this function and the horizontal line corresponding to the relative deadline gives the minimum processor speed (i.e., the maximum clock period) that optimizes power usage while ensuring the schedulability of the considered task-set. Note that the clock speed that optimizes power usage does not necessarily guarantee the minimum energy consumption of the system, and vice versa. In Section 7.2, we will discuss how the proposed approach needs to be modified when the optimization objective is energy consumption instead of power usage.

In the rest of the paper, we will denote as *changing points* those values of T_{ck} where the slope of $F_C(t)$ changes. As evident from the example of Figure 2, changing points may be of two types, according to the following definitions.

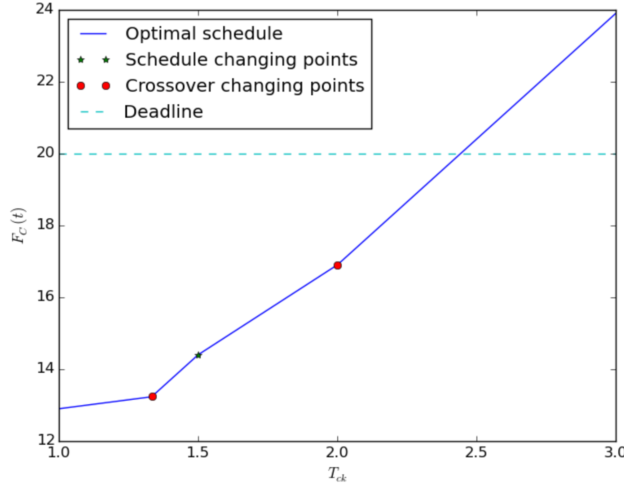


Fig. 3: Example of the function $F_C(t)$ associating the clock period T_{ck} to the value of the optimal makespan for the task-set in Figure 2. The horizontal line corresponds to the relative deadline $D = 20$. The portion of the function lying below the line identifies the values of T_{ck} that make the task-set schedulable.

Definition 5.1 (Schedule changing points) *A* schedule changing point *is a value of clock period $\tilde{t}$ of the form M_i/C_i, for some $i \in [1,\ldots,n]$, such that* $\lim_{t\to\tilde{t}^-} F'_C(t) > \lim_{t\to\tilde{t}^+} F'_C(t)$ *(i.e., the slope of $F_C(t)$* decreases *in correspondence of $t = \tilde{t}$).*

Definition 5.2 (Crossover changing points) *A* crossover changing point *is a value of clock period $\hat{t}$ of the form*[7]

$$\frac{M_{\hat{\sigma}^{\hat{t}}_{i+1}} + \ldots + M_{\hat{\sigma}^{\hat{t}}_{i+k+1}}}{C_{\hat{\sigma}^{\hat{t}}_{i}} + \ldots + C_{\hat{\sigma}^{\hat{t}}_{i+k}}},$$

for some $i \in [1,\ldots,n]$ and some $k \in [0,\ldots,n-i]$, such that $\lim_{t\to\hat{t}^-} F'_C(t) < \lim_{t\to\hat{t}^+} F'_C(t)$ *(i.e., the slope of $F_C(t)$* increases *in correspondence of $t = \hat{t}$).*

Intuitively, schedule changing points correspond to values of clock period at which a job reordering occurs according to Johnson's algorithm. The slope of $F_C(t)$ decreases after a schedule changing point because, if a reordering takes place in the optimal schedule, then the minimum makespan in the new configuration must be strictly dominated by the previous one.

The next lemma shows that a job may change its position in an optimal schedule only when its computation time becomes equal to its memory-access time.

[7] We recall that $\hat{\sigma}^t_1,\ldots,\hat{\sigma}^t_n$ is the permutation of jobs corresponding to the minimum makespan when $T_{ck} = t$.

Lemma 5.1 *For any pair of jobs J_i and J_j, such that J_i precedes J_j in an optimal schedule at $T_{ck} = t' \geq 1$, a job swapping may occur in the interval $T_{ck} \in (t', +\infty)$ only at $T_{ck} = M_j/C_j$, provided that $M_j/C_j > t'$.*

Proof According to Theorem 4.1 and the corresponding algorithm described in Section 4.1, the optimal schedule does not change as long as sets S_1 and S_2 do not change. A job reordering is only possible if some job passes from set S_2 to S_1, which can only happen for values of t such that $M_i = t \cdot C_i$. □

On the other hand, crossover changing points correspond to clock periods at which the crossover job changes. In other words, when some of the gaps in the processor usage are filled, a larger number of jobs could start contributing to the makespan with their computation times. To better clarify the difference between the two sets of changing points, consider again the example in Figure 2. Here, 1.5 is a schedule changing point, while 1.33 and 2 are crossover changing points. Note also that when $T_{ck} \in [1, 1.33)$, the slope of $F_C(t)$ is given by C_3, as J_3 is the crossover job, while when $T_{ck} \in [1.33, 1.5)$, $F_C(t)$ starts increasing with a larger slope ($\sum_{i=1}^{3} C_i$), since now J_1 has become the crossover job.

The next lemmas justify the relation between each type of changing point and its effect on the slope of $F_C(t)$.

Lemma 5.2 *In correspondence of a schedule changing point, the slope of $F_C(t)$ can only* decrease.

Proof Each schedule changing point determines a job reordering. By contradiction, assume that after any schedule changing point the makespan starts increasing at a higher rate (i.e., more jobs start contributing to $F_C(t)$) until the subsequent changing point is reached. This would contradict the optimality of $F_C(t)$, because, by keeping the previous job ordering, the makespan would increase at a smaller rate. It then follows that a schedule changing point can only determine a slope decrease of $F_C(t)$. □

Lemma 5.3 *In correspondence of a crossover changing point, the slope of $F_C(t)$ can only* increase.

Proof The lemma trivially follows by observing that a gap in the processor usage is closed in correspondence of each crossover changing point, but the job ordering remains the same. This means that when a crossover changing point is reached, the index of the crossover job moves to the left in the current optimal schedule. By Theorem 4.2, it follows that more jobs start contributing to the makespan with their computation time, hence the slope of $F_C(t)$ can only increase. □

The two lemmas above can be applied to the example of Figure 3 to visually identify schedule and crossover changing points.

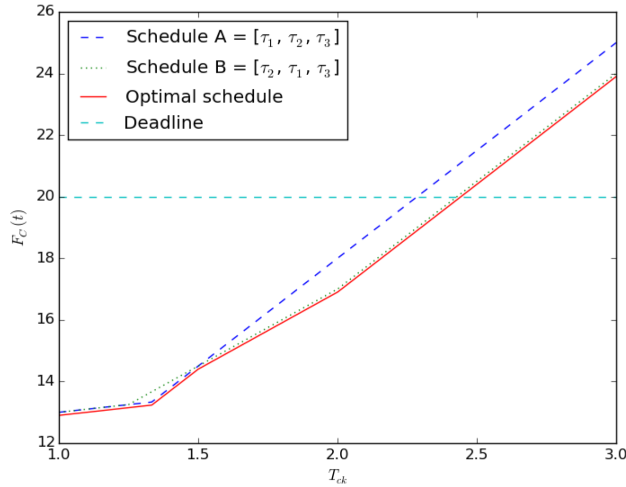


Fig. 4: Example of different functions $F_C(t)$ associated to fixed scheduling decisions as a function of the clock period T_{ck}. The considered task-set is the same as in Figure 2. The horizontal line corresponds to the relative deadline $D = 20$.

5.1 Finding changing points

We now describe how, for a given collection of jobs, the changing points of $F_C(t)$ can be computed.

Schedule changing points In order to better understand the occurrence of schedule changing points, consider Figure 4. The figure uses the same task-set considered for Figure 3 and depicts the makespan as a function of T_{ck} under different and fixed scheduling decisions. As can be seen, the scheduling decision that is optimal for a low value of resource frequency, e.g. $T_{ck} = 1$, may not be optimal for larger values of T_{ck}, due to different patterns in the usage of the second resource. The figure shows that, for values of T_{ck} between 1 and about 1.25, the schedule $A = \{\tau_1, \tau_2, \tau_3\}$ behaves exactly as the schedule $B = \{\tau_2, \tau_1, \tau_3\}$. In this case, the makespan grows linearly with the length of C_3 because it corresponds to the situation depicted in Figure 2(a). For value of T_{ck} beyond 1.25 and less than 1.33, A is the optimal schedule (Figure 2(b)). Next, for values of T_{ck} greater then 1.5, schedule B is the optimal schedule (Figure 2(c) and 2(d)). Hence, a schedule changing point occurs whenever a change in the operating frequency of the second resource determines a change in the optimal schedule. In the example depicted in Figure 4, only one schedule changing point exists at $T_{ck} = 1.5$.

To compute the list of schedule changing points $\mathcal{P}_s$, we first define a list $\mathcal{CP}_s$ of *candidate schedule changing points*:

$$\mathcal{CP}_s = \{M_i/C_i \mid M_i > C_i, i = 1, \dots, n\}. \tag{3}$$

The list of candidates $\mathcal{CP}_s$ may be larger than $\mathcal{P}_s$ because not necessarily a job reordering takes place when the computation time of a job J_i reaches the value of M_i. In fact, it may happen that the precedence relations imposed by Equation (1) remain unchanged, meaning that J_i is already in its "right position" with the current ordering. In this case, M_i/C_i does not represent a schedule changing point.

The list $\mathcal{P}_s$ can be identified starting from $\mathcal{CP}_s$ as described in Algorithm 1. The algorithm takes as input the task-set $\mathcal{T}$ and returns two pieces of information. First, it provides the list of schedule changing points $\mathcal{P}_s$ ordered according to their occurrence as the clock period is scaled in $(0^+, +\infty)$. Second, the algorithm generates a list of schedules S. Each element of S corresponds to the schedule that minimizes the makespan for all values t of clock period in the interval $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$. In other words, it always holds that $S_i = \hat{\sigma}^t$ for $\mathcal{P}_{s,i} \leq t < \mathcal{P}_{s,i+1}$.

Algorithm 1 Computation of the lists $\mathcal{P}_s$ and S

1: **procedure** SchedulePoints($\mathcal{T}$)
2: $B \leftarrow$ DSort($\mathcal{T}, key = C_i$)
3: $E \leftarrow$ ASort($\mathcal{T}, key = M_i$)
4: $SW \leftarrow$ ASort($\mathcal{T}, key = M_i/C_i$)
5: $L \leftarrow$ Array(size $= n$, value $= null$)
6: $S \leftarrow B$; $\mathcal{P}_s \leftarrow \{0\}$; $f \leftarrow -1$
7: **for** $j = 1$ **to** n **do**
8: $r \leftarrow SW_j.M/SW_j.C$
9: $k \leftarrow$ IndexOf(SW_j, E)
10: $L_k \leftarrow SW_j$
11: $B \leftarrow$ Remove(SW_j, B)
12: $L' \leftarrow$ Filter(L, value $= null$)
13: $\sigma_{curr} \leftarrow$ Concat(L', B)
14: **if** $\sigma_{curr} \neq$ Last(S) **then**
15: $\mathcal{P}_s \leftarrow$ Append($r, \mathcal{P}_s$)
16: $S \leftarrow$ Append(σ_{curr}, S)
17: **if** $f = -1$ **and** $r \geq 1$ **then**
18: $f \leftarrow r$
19: **end if**
20: **end if**
21: **end for**
22: $\{\mathcal{P}_s, S\} \leftarrow$ Reinit($\mathcal{P}_s, S, f$)
23: **return** $\{\mathcal{P}_s, S\}$
24: **end procedure**

Algorithm 1 first constructs the beginning and ending optimal schedules for values of T_{ck} ranging in $(0^+, +\infty)$. According to Equation (1), when the computation time C_i of each job is shorter than its corresponding memory-access time M_i, the optimal schedule is obtained by sorting the jobs by C_i

in descending order. Thus, this schedule is the initial one for $T_{ck} \approx 0^+$, and is calculated as B at line 2, and also stored as first element of S at line 6. Similarly, for $T_{ck} \approx +\infty$, the jobs are sorted in ascending order of M_i. This sequence is calculated and stored into E at line 3. The key idea to find the schedule changing points is to observe that each candidate is associated to a single job, and, by Johnson's rule, if a schedule changing point occurs at $T_{ck} = t$, only the associated job will swap position from $\hat{\sigma}^{t^-}$ to $\hat{\sigma}^{t}$. Intuitively, this is because when a schedule changing point for job J_i is reached, the result of the comparison in Equation (1) may change, thereby determining a new position of J_i within the schedule, as depicted in Figures 2(b) and 2(c). Hence, by sorting the candidate changing points in ascending order, we can build a list of possibly swapping jobs SW (line 4).

It also follows from Johnson's rule that any job that has passed its own schedule changing point (i.e., whose C-phase has become longer than its M-phase) will appear in the schedule before any job that has not passed it. In fact, consider a job J_i that has passed its schedule changing point. A second job J_j can only appear before J_i in the schedule $\hat{\sigma}^t$ if $M_j < M_i$. However, if $C_j < M_j$, then τ_j will be computation-dominated and always scheduled after J_i. Therefore, the *for* loop at lines 7-21 scans all the jobs in the schedule distinguishing between jobs that have passed their schedule changing point and jobs that have not. It is then enough to sort the former class in ascending order of M_i and the latter class in descending order of C_i. The concatenation of the two sets will represent the optimal schedule at $T_{ck} = t$. More in detail, at line 8, r stores the jth job in the list SW of candidate changing points. Also, the array L, initialized at line 5, progressively stores the jobs that have passed their schedule changing point. To prevent reordering at every step, jobs are positioned in L at the same index they have in the final sequence E and they are removed from B (lines 9-11). The filtering on L at line 12 is needed to remove placeholder *null* objects and construct a valid candidate schedule. Maintaining empty slots (*null*) for all the jobs that have not reached their respective schedule changing point allows preserving the relative ordering and thus avoiding additional sorting operations.

At line 13, the current schedule σ_{curr} is updated as the concatenation of the filtered array L' and the remaining elements in B. If the schedule σ_{curr} obtained at line 13 is different from the previously generated one, i.e., $\textsc{Last}(S)$, it follows that r indeed represents a schedule changing point. This check is performed at line 14. If the check is passed, the lists $\mathcal{P}_s$ and S are updated by appending r to $\mathcal{P}_s$ and σ_{curr} to S. (lines 15 and 16). Finally, since we are only interested in the changing points within $[1, +\infty)$, all points below 1 should be discarded. Therefore, the check at line 17 verifies whether the first changing point greater or equal than 1 is encountered. If so, f is updated with the current value of r (line 18). At line 22, when the full list of schedule changing points has been constructed, the function $\textsc{Reinit}$ filters the points, based on the value previously stored in f. In particular, the function $\textsc{Reinit}$ should set 1 as first element of $\mathcal{P}_s$ (if not already present in the list) and update S

accordingly (i.e., possibly eliminating from S the jobs whose schedule changing point is smaller than 1).

Finally, the algorithm returns as output the two lists at line 23.

Crossover changing points Since a job reordering occurs only in correspondence of schedule changing points, the optimal schedule never changes between pairs of adjacent schedule changing points. Then, to characterize $F_C(t)$, it is necessary to find the crossover changing points falling between each pair of adjacent elements in $\mathcal{P}_s$, i.e., in any interval of the form $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$[8]. The challenging task is then to predict in which order the gaps in the processor usage are filled as the clock period is scaled. Indeed, the number of crossover changing points strictly depends on the *order* in which the gaps are filled. If they are filled in order, starting from the last job in the schedule, distinct crossover changing points are generated, because the slope of $F_C(t)$ increases when each of the gaps is filled. If they are filled *out of order*, a crossover changing point may be generated only when the first of the considered jobs becomes the crossover job.

We provide some intuition of the problem by means of a simple example. Consider a task-set composed of three tasks, with the following parameters: $\tau_1 = (M_1, C_1) = (2, 3)$, $\tau_2 = (4, 6)$ and $\tau_3 = (7, 8)$. The job ordering that minimizes the makespan with $T_{ck} = 1$ is illustrated in Figure 5(a). In the initial configuration, J_3 is the crossover job, hence it is the only one that contributes to the makespan rate with its computation time. However, when T_{ck} reaches the value $M_3/C_2 = 1.17$ (Figure 6(b)), J_2 becomes the crossover job, hence 1.17 is a crossover changing point of $F_C(t)$. Finally, when T_{ck} becomes equal to $M_2/C_1 = 1.33$ (Figure 5(c)), the slope of $F_C(t)$ is further increased, because also J_1 starts affecting the makespan increase rate, becoming the crossover job. Therefore, also 1.33 can be classified as a crossover changing point of $F_C(t)$. In this scenario, two crossover changing points are found because the gap in the processor usage relative to C_2 is filled *before* the one relative to C_1.

However, if we change the computation time of τ_2 to be 4 instead of 6, as in Figure 6(a), we observe that when T_{ck} reaches $M_2/C_1 = 1.33$ (Figure 6(b)), the gap corresponding to C_1 is filled, but no crossover changing point is generated, because the makespan increase rate is not affected by the aggregation between the two jobs. Indeed, before and after 1.33 only J_3 contributes to increase the makespan rate with its computation time. Finally, when $T_{ck} = (M_2 + M_3)/(C_1 + C_2) = 1.57$, a single crossover changing point is generated, as J_1 becomes the crossover job and all three jobs start contributing to increase the slope of $F_C(t)$ with their computation times, as in Figure 6(c).

This example shows that the number of crossover changing points strictly depends on the *order* in which the gaps in the processor usage are filled. In the example of Figure 5, the two gaps are filled in sequence (*in order*), because $M_3/C_2 > M_2/C_1$. In this case, two distinct changing points are generated, because the slope of $F_C(t)$ increases when each of the two gaps is filled. In the

[8] For the last element of $\mathcal{P}_s$, we consider the interval $[\mathcal{P}_{s,i}, +\infty)$.

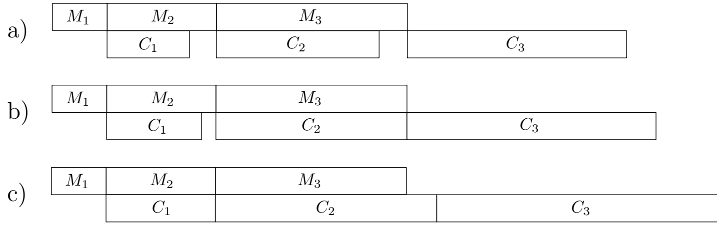


Fig. 5: Example of a task-set composed of three tasks with parameters $\tau_1 = (2, 3)$, $\tau_2 = (4, 6)$, $\tau_3 = (7, 8)$. The three insets illustrate the optimal schedule when a) $T_{ck} = 1$; b) $T_{ck} = 1.17$; c) $T_{ck} = 1.33$.

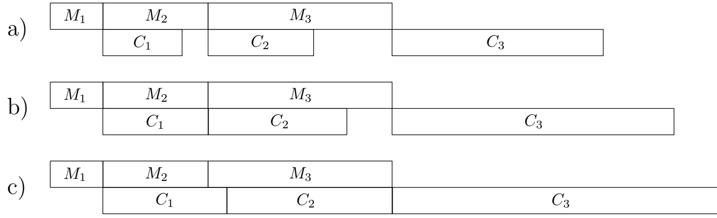


Fig. 6: Example of a task-set composed of three tasks with parameters $\tau_1 = (2, 3)$, $\tau_2 = (4, 4)$, $\tau_3 = (7, 8)$. The three insets illustrate the optimal schedule when a) $T_{ck} = 1$; b) $T_{ck} = 1.33$; c) $T_{ck} = 1.57$.

second case, instead, $M_3/C_2 < M_2/C_1$, implying that the gaps are filled *out of order*: first, the gap relative to C_1 is closed, and then the makespan starts increasing at a higher rate when $C_1 + C_2$ becomes equal to $M_2 + M_3$, i.e., in correspondence of the (unique) changing point at $T_{ck} = 1.57$.

Relying on these observations, Algorithm 2 derives the sublist of crossover changing points falling inside any interval $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$. The algorithm takes as input the initial task-set $\mathcal{T}$, the list $\mathcal{P}_s$, the index therein representing the left endpoint of the interval, and the list of optimal schedules S. It produces in output a structure ρ of crossover changing points, where the field $\rho.ck$ stores their values, while the field $\rho.x$ contains the index of the crossover job in the optimal schedule. As explained in Section 5.3, this is needed to efficiently compute the value of $F_C(t)$ once the list of changing points is known.

At line 2, t stores the value of $\mathcal{P}_{s,i}$, the list ρ of crossover changing point is initialized to the empty set, and $\hat{\sigma}^t$ stores the ith schedule S_i. Then, at line 3, the gaps in the processor usage are computed by the function AdjacentJobs, which initializes the structure Z as follows. The field $Z.jobs$ stores the groups of jobs whose C-phases are executed consecutively in the optimal schedule, with the exception of the last group, which may not give rise to a gap in the processor usage (e.g., C_3 in Figure 2(a)). The field $Z.ck$ stores instead the values of clock period at which the gaps are closed. Each of such values $Z_j.ck$ can be computed as $\frac{M(Z_j.jobs)}{C(Z_j.jobs)}$, where the operators $M(J)$ and $C(J)$ take as

Algorithm 2 Crossover changing points in $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$

```
1: procedure CrossoverPoints(𝒯, 𝒫_s, i, S)
2:     t ← 𝒫_{s,i}; ρ ← ∅; σ̂^t ← S_i
3:     Z ← AdjacentJobs(𝒯, σ̂^t); D ← 0; N ← 0
4:     for j = Size(Z) to 1 do
5:         if j ≠ Size(Z) and t · Z_j.ck > φ.ck then
6:             if i == Size(𝒫_s) or φ.ck < 𝒫_{s,i+1} then
7:                 ρ ← Append(φ, ρ)
8:             end if
9:             D ← 0; N ← 0
10:        end if
11:        D ← D + C(Z_j.jobs)
12:        N ← N + M(Z_j.jobs)
13:        φ.ck = t · (N/D); φ.x ← Z_j.jobs_1
14:    end for
15:    if i == Size(𝒫_s) or φ.ck < 𝒫_{s,i+1} then
16:        ρ ← Append(φ, ρ)
17:    end if
18:    return ρ
19: end procedure
```

input a group J of s adjacent jobs in $\hat{\sigma}^t$ of the form $J = \{\hat{\sigma}^t_k, \ldots, \hat{\sigma}^t_{k+s-1}\}$ and are defined as follows:

$$C(J) = \sum_{h=k}^{k+s-1} C_{\hat{\sigma}^t_h}; \tag{4}$$

$$M(J) = \sum_{h=k}^{k+s-1} M_{\hat{\sigma}^t_{h+1}}. \tag{5}$$

Note that the index shift in Equation (5) (i.e., $\hat{\sigma}^t_{h+1}$ instead of $\hat{\sigma}^t_h$) complies with the notion of crossover changing point given in Definition 5.2.

As an example, the function AdjacentJobs applied to the input task-set in Figure 2(a) returns: $Z.jobs = \{\{1,2\}\}$ and $Z.ck = \{(M_2+M_3)/(C_1+C_2)\} = \{1.33\}$. Applied to the task-set in Figure 5(a), the result is: $Z.jobs = \{\{1\},\{2\}\}$ and $Z.ck = \{M_2/C_1, M_3/C_2\} = \{1.33, 1.17\}$.

The variable N (resp., D) is used to compute the numerator (resp., denominator) of the partially computed changing point, stored in $\varphi.ck$, while $\varphi.x$ keeps track of the current index of the crossover job. In the *for* loop at lines 4-14, the structure Z of adjacent jobs is walked backward. If the currently examined group Z_j is not the last one, and its value of clock period is greater than $\varphi.ck$ (line 5), it means that the gap corresponding to Z_j will be filled *later* than the one relative to φ (i.e., *in order*), and the two groups of jobs will give rise to two distinct crossover changing points. Hence, φ is appended to ρ, provided that the check at line 6 is passed. This check ensures that the newly computed changing point does not exceed the right endpoint of the interval $\mathcal{P}_{s,i+1}$. The check is trivially passed if the last schedule changing point is considered ($i ==$ Size$(\mathcal{P}_s)$, line 6), because the right endpoint of the

interval is $+\infty$. In any case, at line 9, the temporary variables can be reset to start creating a new group of jobs.

If the check at line 5 fails, it means that the processor gap corresponding to Z_j will be filled *before* the one relative to φ, hence the two groups of jobs will generate a single changing point. At lines 11-13, the intermediate values of D and N are updated using the operators $C(J)$ and $M(J)$, and the new value of $\varphi.ck$ is computed. The ratio N/D is scaled by t to account for the inflation of computation times occurred in the interval $[1, \mathcal{P}_{s,i})$, as the crossover changing points in each interval are initially computed with respect to $\mathcal{P}_{s,i}$. Also, $\varphi.x$ is updated with the new index of the crossover job, given by the first element of $Z_j.jobs$. After the *for* loop, the last crossover changing point is appended to ρ (subject to the same check performed at line 6), which is finally returned as output.

5.2 Complete algorithm

We now describe the complete algorithm to derive the list $\mathcal{P}$ of changing points of $F_C(t)$. First, the list $\mathcal{P}_s$ is computed by Algorithm 1, then Algorithm 2 is iteratively invoked to derive the crossover changing points in each interval $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$.

Algorithm 3 Computation of the list $\mathcal{P}$

```
1: procedure ChangingPoints(𝒯)
2:     {𝒫_s, S} ← SchedulePoints(𝒯); 𝒫 ← 𝒫_{s,1}
3:     𝒫 ← Append(CrossoverPoints(𝒯, 𝒫_s, 1, S), 𝒫)
4:     for i = 2 to Size(𝒫_s) do
5:         𝒫 ← Append(𝒫_{s,i}, 𝒫)
6:         𝒯' ← ScaleJobs(𝒯, 𝒫_{s,i})
7:         𝒫 ← Append(CrossoverPoints(𝒯', 𝒫_s, i, S), 𝒫)
8:     end for
9:     return 𝒫
10: end procedure
```

The pseudo-code is shown in Algorithm 3. At line 2, Algorithm 1 is invoked to compute the list of schedule changing points $\mathcal{P}_s$, and $\mathcal{P}$ is initialized with its first value (we recall that by construction $\mathcal{P}_{s,1} = 1$). Then, at line 3, Algorithm 2 is initially invoked with $i = 1$, and then the *for* loop at lines 4-8 iterates on the subsequent schedule changing points. At each iteration, the i-th point in $\mathcal{P}_s$ is appended to $\mathcal{P}$, and a new task-set $\mathcal{T}'$ is obtained by scaling the original computation time of each task by the value of $\mathcal{P}_{s,i}$, to account for the inflation occurred in the previous interval (line 6). Then, the crossover changing points in $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$ are computed and appended to $\mathcal{P}$, which is finally returned as output.

Since $F_C(t)$ is a piecewise linear function, it can be completely specified by computing its value in correspondence of all changing points, and determining

the slope of the last piece. As the optimal job ordering is known for all values of T_{ck} (it only changes in correspondence of schedule changing points), Equation (2) can be applied to find the value of $F_C(t)$ for each changing point. The slope of the last piece is simply given by $\sum_{i=1}^{n} C_i$, because, in the final schedule, the computation times of all jobs contribute to the makespan (e.g., see Figure 2(d)).

The value of clock period for which $F_C(t) = D$ (i.e., the intersection point between the function $F_C(t)$ and the horizontal line corresponding to the relative deadline of the task-set) gives the minimum processor speed (i.e., the maximum clock period) that optimizes power usage while ensuring the schedulability of the considered task-set. In the special case of discrete processing frequencies, the obtained value of clock period needs to be rounded down to the closest possible clock period among the available ones, that is, the immediately higher processing frequency should be chosen as the optimal one.

5.3 Complexity

We now derive bounds on the maximum number of changing points of $F_C(t)$ (equivalently, $F_M(t)$).

Lemma 5.4 *$F_C(t)$ contains at most $n-1$ schedule changing points.*

Proof The number of schedule changing points is maximized when the maximum number of swaps between jobs is necessary to reach the final schedule, where M-phases are sorted in ascending order. Such a worst-case configuration corresponds to the one where: (i) for any job J_i, $C_i < M_i$, i.e., in the initial optimal schedule, all jobs are computation-dominated (hence sorted by decreasing C_i), and (ii) M-phases are also sorted in descending order. In this case, it is immediate to see that $n-1$ moves are necessary to reorder the jobs as in the final schedule, proving the lemma.

Lemma 5.5 *Between any two adjacent schedule changing points of $F_C(t)$, there are at most $n-1$ crossover changing points.*

Proof The worst-case scenario that maximizes the number of crossover changing points in any interval $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$ is given by the situation in which there are $n-1$ gaps in the processor usage that are closed in order. The last job is excluded (leading to a bound of $n-1$ instead of n) because, as previously observed, it may not give rise to a gap in the processor usage.

By combining the two lemmas above, it follows that the number of changing points is quadratic in the number of jobs. A bound on the time complexity of the complete algorithm is then given by $O(n^2)$. Indeed, the complexity of Algorithm 1, which finds the list $\mathcal{P}_s$, is $O(n^2)$, because the *for* loop at lines 7-21 iterates n times, and at each iteration the cost of filtering the array L at line 12, also linear in the number of tasks, dominates over the other operations. The cost of finding the crossover changing points is also quadratic, because

Algorithm 3 invokes $n-1$ times Algorithm 2, which in turn has a linear cost, since the for loop at lines 4-14 iterates n times and each iteration has a constant complexity. Note that the operations at lines 11 and 12 do not increase the complexity of the algorithm, because the operators $C(J)$ and $M(J)$ are applied to disjoint sets that have an aggregate cardinality $n-1$.

Note also that since Algorithm 2 keeps track of the index of the crossover job for each crossover changing point, it is then sufficient to compute the value of the optimal makespan μ^t only in correspondence of the schedule changing points (which can be done in $O(n^2)$), and then update its value for each crossover point (based on the knowledge of the crossover job), which requires $O(n^2)$ overall. In this way, the complexity remains quadratic in the task-set size.

While our algorithm derives the function $F_C(t)$ (or, equivalently, $F_M(t)$) analytically, a naïve approach would be to perform a binary search on the clock period domain, trying to find the optimal value of T_{ck} that guarantees the schedulability. Such an approach would require to select a quantization step and to run Johnson's algorithm at each point. Beside having a high computational cost, this solution could imply some technical difficulties, mainly due to the non-convexity of the functions. Also, this method would only be able to identify the optimal solution up to the size of the quantization step.

A final remark concerns the applicability of our method also when considering systems having a small number $k << n$ of speeds. In this case, the computation of the schedule changing points would require $O(n\ log(n))$ for the sorting at line 4 of Algorithm 1, which dominates the cost of the filtering at line 12, given by $O(nk)$, as it should be performed only k times. The computation of crossover changing points should be performed only once, i.e., for the interval $[\mathcal{P}_{s,i}, \mathcal{P}_{s,i+1})$ that delimits the optimum. Hence, the complexity would be comparable to running Johnson's algorithm k times, but without any dependence on the number of available speeds.

6 Multiple Resource Optimization

In this section, we demonstrate how the proposed algorithm can be adapted to optimize the speed of the first resource (memory) with respect to power and schedulability constraints (Section 6.1). Furthermore, we extend our results to jointly optimize the speed of both resources to minimize the total instantaneous power usage (Section 6.2).

6.1 Scaling the speed of the first resource

We now consider variations in the speed of the first resource, that is, we seek to find the function $F_M(t)$, assuming a fixed CPU speed while varying the DMA speed. This function can be simply derived once the list $\mathcal{P} = \{p_1, \ldots, p_\ell\}$ of

changing points of $F_C(t)$ is known[9]. Thus, we adapt the original system as follows.

First, we establish the initial parameters corresponding to the configuration $T_{ck} = 1$. Since we are interested in scaling memory-access times, we set the computation time C_i' of each job J_i by imposing a fixed CPU speed $\beta \geq 1$, such that $C_i' = \beta \cdot C_i$. Next, we select the starting point for the DMA speed as a fraction α of the original value, with $0 < \alpha \leq 1$, such that $M_i' = \alpha \cdot M_i$. In this new setting, computation times are kept constant, while memory-access times are scaled as $M_i' \cdot t$ for any value of $T_{ck} = t$.

The following theorem proves that the changing points in $\mathcal{P}'$ can be simply found as the reciprocal of those in $\mathcal{P}$, up to a multiplicative factor given by the choice of the initial parameters.

Theorem 6.1 *The list of changing points of $F_M(t)$ is given by $\mathcal{P}' = \{p_1', \ldots, p_\ell'\}$, whose generic element p_k' is equal to:*

$$p_k' = \frac{1}{p_{\ell-k+1}} \cdot \frac{\beta}{\alpha}.$$

Proof In correspondence of a generic changing point p_k' of $F_M(t)$, the optimal makespan is given by:

$$\mu^{p_k'} = \max_{i=1}^{n} \left(\sum_{j=1}^{i} M_j \cdot p_k' + \sum_{j=i}^{n} C_j \right).$$

By factorizing out the value of p_k', it becomes:

$$\mu^{p_k'} = p_k' \cdot \max_{i=1}^{n} \left(\sum_{j=1}^{i} M_j + \sum_{j=i}^{n} C_j / p_k' \right). \tag{6}$$

Equation (6) indicates that the optimal makespan obtained when the speed of the first resource is increased by p_k' is the same as the one obtained if the initial parameters are scaled by a factor p_k' and the speed of the second resource is decreased by a factor $1/p_k'$. Therefore, since the values of the changing points are in the form of ratios between task parameters, this means that if p_k' is a changing point for $F_M(t)$, then $1/p_k'$ is a changing point for $F_C(t)$. This proves that the changing points of $F_M(t)$ can be simply found as the reciprocal of those of $F_C(t)$, up to the scaling factor β/α applied to the initial task parameters. □

Note that the points in $\mathcal{P}'$ are indexed in reverse order with respect to $\mathcal{P}$. It is then necessary to discard all points < 1 from $\mathcal{P}'$ to restrict the domain of the function to the interval $[1, +\infty)$. As before, Equation (2) can be used to find

[9] Here, we refer to the *complete* list of changing points of $F_C(t)$, i.e., including all changing points in the interval $T_{ck} \in (0^+, +\infty)$. This means that, when running Algorithm 1, the function REINIT at line 22 should not be executed.

the value of $F_M(t)$ in correspondence of each changing point. Symmetrically, the slope of its last piece is given by $\sum_{i=1}^{n} M_i$.

An interesting observation that directly descends from Theorem 6.1 is that computing changing points in $\mathcal{P}'$ as the reciprocals of those in $\mathcal{P}$ is equivalent to exchanging the roles of the two resources and applying directly Algorithm 3. Indeed, the values in $\mathcal{P}'$ can be also obtained by running Algorithm 3 where computation times are relabeled as memory access times and vice versa.

6.2 Scaling the speed of both resources

This section discusses a possible strategy to optimize the speed of the two resources *jointly*. Specifically, we seek to find the optimal values of clock speed[10] s_M^* and s_C^* for the two resources that minimize the *maximum power usage*

$$P(s_M, s_C) = P_M(s_M) + P_C(s_C).$$

We recall that $P_M(s)$ and $P_C(s)$ represent the memory access power profile and the computation power profile, respectively. Also, in Section 3 we assumed that $P_M(s)$ is a linear function with slope α, while $P_C(s)$ is a monotonically increasing convex function.

Figure 7 reports an illustrative example of functions $P_M(s)$ and $P_C(s)$ respecting the assumptions discussed above.

In principle, if we are allowed to freely select the speed of both resources, there are infinite combinations of clock speeds that can produce a particular value of makespan. Among those, we are interested in all the combinations that produce an optimal makespan equal to the relative deadline D, because the minimum power usage will certainly be produced by one of those combinations. More formally, the total power usage will be minimized by a pair of clock periods (T_M, T_C) that verifies:

$$D = \max_{i=1}^{n} \left(T_M \cdot \sum_{j=1}^{i} M_j + T_C \cdot \sum_{j=i}^{n} C_j \right). \tag{7}$$

In order to find the optimal pair (T_M^*, T_C^*) that satisfies Equation (7) while minimizing power usage, we propose to approximate $P_C(s)$ as a piecewise linear function, which can be efficiently done by standard techniques [9, 30, 16]. Any desired level of accuracy can be met by simply increasing the total number of segments, which we denote by S. Each linear segment δ_j, $j = \{1, \ldots, S\}$, is defined by a triple (a_j, b_j, β_j), where $(a_j, b_j]$ is the considered interval and $\beta_j > 0$ is its slope. By the assumptions on $P_C(s_C)$, the slopes of the different pieces must be strictly increasing, i.e., $\beta_1 < \beta_2 < \ldots < \beta_S$ (see Figure 7 for an intuitive explanation).

[10] In this section, we reason in terms of clock speed instead of clock period for ease of understanding, and assume $s \in (0, 1]$.

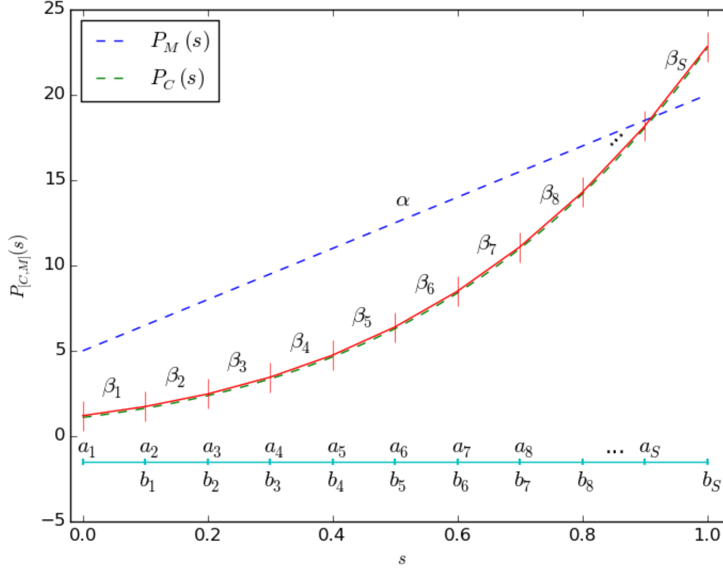


Fig. 7: Considered power profiles $P_M(s)$ and $P_C(s)$ for first (memory) and second (CPU) resource respectively.

In this setting, the optimal pair (T_M^*, T_C^*) can be found by an iterative procedure that takes into account the shape of the two power profiles in order to find the combination of speeds that leads to the minimum power usage. Algorithm 4 summarizes this procedure. The algorithm assumes the knowledge of the function $F_C^*(t)$, obtained by applying Algorithm 3 where the input argument is given by the original task-set where the minimum value of clock period is assumed for the first resource. Also, it takes as inputs the relative deadline D of the task-set, the slope α of $P_M(s)$ and the list of triples $\Delta = \{\delta_1, \ldots, \delta_S\}$ defining the S linear pieces that approximate $P_C(s)$.

Then, the *for* loop at lines 3-16 iterates over the list Δ, computing for each segment $(a_j, b_j]$, $j \in \{1, \ldots, S\}$, the ratio $r = \beta_j/\alpha$ (i.e., the ratio between $P_C(s)$ and $P_M(s)$ in the considered interval) at line 4. Then, the value of the optimal makespan in correspondence of $T_{ck} = r$ is computed. Specifically, the procedure ComputeMakespan at line 5 returns the value of $F_C^*(t)$ when $T_{ck} = r$. Next, the procedure calculates the scaling factor f_j that can be applied to the makespan in order to reach the desired value D. In other words, f_j represents the scaling factor that can be applied to the clock period of both resources (lines 7-8) to find a valid combination of speeds that yields the maximum value of makespan that guarantees the schedulability (i.e., that satisfies Equation (7)). However, we must consider this combination of speeds as valid only if the obtained clock period for the second resource falls in the considered range $(a_j, b_j]$, otherwise the linear approximation cannot be considered valid

(*if* statement at line 9). For any valid combination, we can calculate the power level based on the knowledge of the two power profiles $P_M(s)$ and $P_C(s)$, and update the total power P only if the sum $P_M(s_M) + P_C(s_C)$ is smaller than the current value of P.

After all segments have been scanned, the optimal values of clock periods for the two resources (previously stored in T_M^* and T_C^*) can be returned as output, as well as the the minimum power level stored in P.

Algorithm 4 Power optimization considering both resources

1: **procedure** PowerOpt(D, α, $\Delta = \{\delta_1, \ldots, \delta_S\}$)
2: $P \leftarrow +\infty$
3: **for** $j = 1$ **to** S **do**
4: $r \leftarrow \beta_j/\alpha$
5: $\mu^r \leftarrow$ ComputeMakespan(r)
6: $f_j \leftarrow D/\mu^r$
7: $s_M \leftarrow f_j$
8: $s_C \leftarrow r \cdot f_j$
9: **if** $s_C > a_j$ **and** $s_C \leq b_j$ **then**
10: **if** $P_M(s_M) + P_C(s_C) < P$ **then**
11: $P \leftarrow P_M(s_M) + P_C(s_C)$
12: $T_M^* \leftarrow 1/s_M$
13: $T_C^* \leftarrow 1/s_C$
14: **end if**
15: **end if**
16: **end for**
17: **return** P, T_M^*, T_C^*
18: **end procedure**

Clearly, assuming the knowledge of $F_C^*(t)$, which can be computed in $O(n^2)$ (Section 5.3), Algorithm 4 has a time complexity that is linear in the number of segments approximating $P_C(s)$, hence the total complexity is $O(n^2)+O(S)$.

7 Further Extensions

In this section, we discuss possible generalizations for our methodology. Specifically, Section 7.1 transforms the presented algorithm into an heuristic that can be used on a multiprocessor system to perform deadline-constrained speed optimization. Additionally, we provide the intuition about how our approach can be reused when energy consumption represents the main optimization objective (Section 7.2).

7.1 Heuristic approach for partitioned multiprocessor systems

We now briefly discuss how the proposed approach for speed selection can be reused in a multiprocessor scenario where tasks are statically partitioned to cores. Specifically, we assume a system composed of m identical cores that can

be operated at the same speed, and a single DMA engine that is in charge of loading the task image from the main memory (shared among the cores) to the local memory of each core. We denote as $\mathcal{T}_k$ the set of tasks assigned to the k-th CPU.

This scenario corresponds to a two-stage flow-shop scheduling problem where a single resource unit is available for the first stage, and m identical copies of the second resource are available for the second stage. As previously observed, the flow-shop scheduling problem has been demonstrated to be strongly NP-hard when two or more resources are available for some stage [15], hence no efficient algorithm exists to identify an optimal schedule.

Therefore, we reuse the approach presented in Section 5.1 to *heuristically* determine the speed at which to operate the memory and the m CPUs to minimize the maximum power usage, as defined in Section 6.2.

Similarly as in Section 6.2, we derive the function $F^*_{C,k}(t)$ for each core $k = \{1, \ldots, m\}$, which assumes the minimum value of clock period for the first resource. This can be done by the following steps, which must be executed for each of the k CPUs:

1. Compute the set $\mathcal{P}^k_s$ of schedule changing points by applying Algorithm 1 passing as input the complete task-set $\mathcal{T} = \bigcup_{k=1}^m \mathcal{T}_k$, given by the union of all tasks in the system[11];
2. For each interval $[\mathcal{P}^k_{s,i}, \mathcal{P}^k_{s,i+1})$ of adjacent points found at the previous step, find the crossover changing points by applying Algorithm 2 on a modified task-set $\mathcal{T}'_k$, constructed by setting to zero the computation time of tasks not assigned to the k-th CPU. More formally, the input task-set $\mathcal{T}'_k$ is constructed as follows:

$$\mathcal{T}'_k = \mathcal{T}_k \cup \bigcup \{\tau'_j \mid \tau_j \notin \mathcal{T}_k \textit{ and } \tau'_j = (M_j, 0)\}.$$

   Intuitively, the set $\mathcal{T}'_k$ is considered as input because the tasks not allocated to the k-th CPU only interfere with their memory access times, as the DMA engine is shared among all CPUs;
3. Identify the complete set of changing points $\mathcal{P}^k$ as the union of changing points found at the previous two steps;
4. Use Equation (2) to calculate the makespan in correspondence of each changing point, considering as input the modified task-set $\mathcal{T}'_k$.

The above procedure outputs a piecewise linear function $F^*_{C,k}(t)$ that represents the minimum makespan as a function of T_{ck} for each of the m CPUs. Finally, in order to perform speed estimation, we compute $F^*_C(t)$ as the pointwise maximum of the m curves. Considering that each curve has at most $O(n^2)$ changing points (equivalently, $O(n^2)$ segments), this can be done in

[11] Note that the set $\mathcal{P}^k_s$ constructed in this way is a superset of the actual schedule changing points of $\mathcal{T}_k$, hence the slope of $F^*_{C,k}(t)$ does not necessarily change for all points in $\mathcal{P}^k_s$. As we will see next, this simplification makes our results directly applicable to the multiprocessor case with no modifications.

$O(mn^2 \cdot \alpha(mn^2))$, where $\alpha(mn^2)$ represents the inverse of the Ackermann's function [12].

Using the upper-envelope computed above, the approach discussed in Section 6.2 can be applied with no modifications to compute the speed of the two types of resources.

In Section 9.3, the proposed heuristic approach is quantitatively evaluated in comparison to the theoretical optimum and other heuristics.

7.2 Energy consumption minimization

It has been shown that power usage as a function of CPU speed increases with a super-linear law [11]. A similar trend is observed for other classes of resources: for instance, the power usage of DRAM memories as a function of operating frequency and bandwidth follows a super-linear trend as well [32]. Conversely, the power usage of networking devices as a function of the number of processed packets per second follows a more linear law [21].

So far, we considered our optimization objective to be the minimization of instantaneous power usage. As shown in Section 6.2, this can be achieved by identifying the pair of speeds (s_M^*, s_C^*) that minimize the aggregated power usage (i.e., the function $P(s_M, s_C) = P_M(s_M) + P_C(s_C)$) and match the relative deadline D of the task-set. The proposed algorithm assumes the knowledge of the power profiles of both resources (see Figure 7).

Minimizing power usage, however, is not always the main goal of power-aware embedded real-time systems. In fact, many embedded devices are battery-operated and therefore have a limited energy supply. In such cases, the total energy consumed by a set of tasks during their execution represents the main objective function to minimize.

In this paper, we consider batches of two-stage jobs, where each stage requires to be executed on a specific resource. We assume that the time required to execute a task stage decreases linearly as the speed of the corresponding resource is linearly increased.

Additionally, in Section 6.2, we have assumed for the first resource a linear power model with a certain offset, and a (monotonically increasing) convex speed/power relation for the second resource. Therefore, the optimal speed in terms of energy consumption for the first stage is simply given by the maximal one, due to the presence of the offset. Without offset, the speed would not make a difference in terms of energy, thus, even in this case, it would be advisable to select the maximal speed for the first resource. This choice in fact provides the highest degree of freedom for selecting the speed of the second resource, which can be identified based on the knowledge of its energy profile.

Typically, the power profile of CPUs does not follow a linear trend [11]. In the literature, there is a consistent body of work showing that, with a convex power/speed relation, the minimal energy consumption is given by the lowest CPU frequency (e.g., [17]). In this case, the optimal speed allocation in terms

of overall energy is given by: (i) highest speed for the first stage, and (ii) lowest speed that just allows meeting the deadline for the second stage[12].

Other works [10, 18] show that, when leakage power is considered, there is a *critical speed* (generally different from minimum and maximum values) that optimizes energy consumption. Lowering the processor frequency below this threshold can have negative effects on the system-wide energy consumption, as it increases the contribution of the cumulative leakage power (i.e., leakage energy). More in detail, the typical situation is the following:

- low CPU speeds determine a long execution time and lead to high energy consumption;
- as the CPU speed increases linearly, a linear speed-up in the execution time of the task can be observed until a *critical speed* s^{crit} is reached;
- beyond this value, the increase in dynamic power reduces the performance gain. Hence, further increases in CPU speed produce marginally decreasing performance benefits, which in turn determine increasing values of energy consumption.

Hence, there is a major trade-off between increasing leakage power and decreasing dynamic power, which leads to a convex speed/energy profile. In this case, the optimal CPU speed that minimizes energy consumption falls somewhere in between minimum and maximum values. Concretely, our approach to power usage minimization can be adapted to minimize energy consumption as follows:

1. Select the highest speed s_M^{max} for the first resource;
2. Apply Algorithm 3 to derive the minimum speed $\bar{s} = \bar{T}_{ck}^{-1}$ for the second resource such that the collection of jobs completes within the deadline, being $\bar{s} \in [s_C^{min}, s_C^{max}]$;
3. Follow the energy profile to select the operating speed s_C^* for the second resource that minimizes energy consumption, such that $\bar{s} \leq s_C^* \leq s_C^{max}$. Specifically, if $s^{crit} \geq \bar{s}$, then $s_C^* = s^{crit}$; otherwise, $s_C^* = \bar{s}$.

Therefore, the pair of speeds (s_M^{max}, s_C^*) is the one that minimizes the overall energy consumption.

Intuitively, at step 3. above, if $s^{crit} < \bar{s}$, then s_C^* is set to $\bar{s}$, because this is the lowest possible speed value that allows meeting the deadline. In this case, the completion time of the collection of job just matches the deadline. If instead $s^{crit} \geq \bar{s}$, s_C^* is set to s^{crit}, which determines a non-null slack time before the deadline. Dynamic Power Management (DPM) techniques [6] aim at reducing energy consumption by allowing a selective resource shutdown during inactivity intervals. As a part of our future work, we plan to extend our system to support DPM techinques by considering a number of additional parameters, such as the overheads associated with shutdown and wakeups, the transition times to and from low-power states, etc.

[12] This value can be found by applying Algorithm 3 with no modifications.

8 Complexity of flow-shop with intermediate deadlines

In Section 3, we have restricted our system model to task-sets that are subject to the same temporal constraints, that is, all tasks share the same relative deadline D and period T, being $D \leq T$. For any task-set complying with this assumption, an optimal schedule can be derived by exploiting the result in [20]. In this section, we show that the problem of deriving an optimal schedule for a set of tasks sharing the same period but with different deadlines is NP-complete. Hereby, we assume that a generic task τ_i, $i = 1, \dots, n$, is expressed as a triple (M_i, C_i, D_i), where D_i is the relative deadline of task τ_i, and $D_i \leq T$ for any such task.

First, we need to prove that the considered problem is in NP, and then that a problem that is already known to be NP-complete can be reduced to our problem. Our result is proven via reduction from the three-stage flow-shop scheduling problem, which has been demonstrated to be NP-complete in [14].

Lemma 8.1 *The two-stage flow-shop scheduling problem with intermediate deadlines is in NP.*

Proof We must show that there exists an efficient (polynomial-time) verifier such that for any yes instance of our problem (any feasible task-set in our case), there exists a certificate that the verifier will accept, and for any no instance (any unfeasible task-set in our case), there is no such certificate. For a feasible instance of our problem, the certificate is a feasible schedule $\sigma = \{\sigma_1, \dots, \sigma_n\}$. Given such a job ordering σ, the verifier can check in linear time whether each job J_i, $i = 1, \dots, n$, is feasibly scheduled to complete within D_i time units. Clearly, for any no instance of the problem, the verifier cannot accept any schedule as a valid certificate. □

Theorem 8.1 *The problem of deriving an optimal schedule for a set of n two-stage flow-shop tasks with intermediate deadlines is NP-complete.*

Proof The proof is via reduction from the three-stage flow-shop scheduling problem where the third resource is *non-exclusive*, i.e., there are infinitely many copies of the third type of resource. An instance of this latter problem is expressed as a set of n tasks specified as triples $(\alpha, \beta_i, \gamma_i)$ that represent the execution times on the three resources, and a common relative deadline $D \leq T$. We can construct an instance of our problem by defining a set of two-stage jobs such that task τ_i has a relative deadline equal to $D - \gamma_i$ and the execution time on the first two resources remains the same. This is clearly a polynomial-time transformation. It remains to prove that the two problems are indeed equivalent.

First, suppose that a set of three-stage tasks can be feasibly scheduled to complete within D time units. Then, there exists a schedule σ such that the completion time of all jobs in σ is smaller than or equal to D. Given that infinite copies of the third resource exist in the system, any such job can start executing its last stage as soon as the second stage has completed. Therefore,

if the third stage of each job completes within D time units, it must be that the second stage completes within $D - \gamma_i$ time units, hence our algorithm must return yes.

Finally, assume that a set of two-stage tasks with intermediate deadlines can be feasibly scheduled such that all deadlines are met, and let $D = \max_{i=1}^{n} D_i$. Equivalently, each job completes its second stage within D_i time units. Then, the three-stage scheduling problem where the jobs execute for $D - D_i$ time units on the third resource has a feasible solution, because each job completes its execution on the third resource within $D_i + (D - D_i) = D$ time units. □

9 Evaluation

In this section, we quantitatively evaluate the performance of our approach to speed optimization. In Section 9.1, we describe representative examples showing the power and energy savings that can be achieved when our algorithm is used to determine the optimal operating speed of the CPU, in comparison with simple heuristic strategies. Then, by means of extensive simulations, we quantify the schedulability advantage of the proposed scheduling strategy over heuristic solutions, both in the single-core case (Section 9.2) and the partitioned multi-core setting (Section 9.3).

9.1 Power and energy optimization

The first experiment has been performed assuming specific energy and power models. Such models have been empirically derived using an embedded platform and a realistic image processing benchmark. Since we are focusing on the optimization of power usage and energy consumption for the second resource (CPU), we first derive the power/energy profile of the considered benchmark on the hardware platform. Next, we assume that a generic synthetic task-set exhibits the same power/energy profiles. This approach allows us to reason on a realistic power-frequency relationship while presenting results that have nicer mathematical properties.

More in detail, we have considered an image processing benchmark, called Disparity, from the San Diego Vision Benchmark Suite [31]. The Disparity benchmark represents a data-intensive application that processes two images taken from different locations and extracts depth information to construct the position of objects depicted in the input images. For the measurements, we have used a Odroid-U3 embedded platform. The system features a Samsung Exynos 4412 processor with four ARM Cortex-A9 cores. Although the nominal maximum speed is 1.7 GHz, the CPU can be overclocked up to a frequency of 2 GHz. Moreover, the power management circuitry allows a minimum operating frequency of 200 MHz, with a scaling granularity of 100 MHz. Thereby, 20 different levels of frequency scaling can be selected. In our experiments, for

each available scaling frequency, we execute the aforementioned benchmark and record: (a) the current I flowing to the system; and (b) the execution time R of the benchmark under analysis. Finally, we derive the consumed power as: $P = I \cdot V$ (see Figure 8), where V is the fixed system voltage. Moreover, we derive energy consumption as: $E = P \cdot R$ (see Figure 9). Not surprisingly, our findings are in line with the usual trend for power/energy observed in literature [10, 18, 11]. In order to acquire energy and power traces to construct the presented trends, the considered platform has been configured as follows. First, no external peripheral is plugged in. Next, on-board peripherals (e.g. Ethernet) are powered-off or configured in sleep mode. Since we study a single-core system, 3 out of 4 cores are switched off. The observed execution time of each task is recorded using cycle-accurate on-board performance counters[13]. As the task under analysis executes, the power consumed by the platform is recorded using an oscilloscope. Specifically, we measure the voltage drop across a small (1 ohm) resistor placed in series with the Odroid board. Since the size of the resistor is known, current and finally power are derived.

In order to exemplify how the proposed algorithm can be used to optimize power usage, we consider a specific task-set consisting of five tasks with the following parameters: $\tau_1 = (24, 4)$, $\tau_2 = (14, 2)$, $\tau_3 = (2, 4)$, $\tau_4 = (60, 10)$, $\tau_5 = (12, 3)$. The relative deadline of the task-set is $D = 135$. To determine the operating frequency of the CPU, we compare our optimal algorithm, referred to as OPT, against three simple heuristic scheduling strategies:

- *M-asc*: tasks are scheduled by ascending values of M_i;
- *C-desc*: tasks are scheduled by descending values of C_i;
- *MC-asc*: tasks are scheduled by ascending values of M_i/C_i.

Figure 8 shows how the makespan varies as a function of the clock period for each of the considered scheduling strategies. The adopted power model is also reported in the same figure. The optimal operating frequency of the CPU can be identified as the value of clock period where the function OPT intersects the horizontal line corresponding to the relative deadline of the task-set, i.e., $T_{ck} = 3.842$. According to our power model, this value corresponds to a power consumption of about 1520 mW. Instead, the heuristic scheduling strategies identify much lower clock period values for the CPU (3.27 for *C-desc*, 3.08 for *MC-asc*, 2.3 for *M-asc*), which correspond to a significantly higher power levels (1614 mW, 1636 mW, and 1760 mW, respectively).

The performance benefit of our algorithm is also evident whenever the energy consumption is the optimization objective. Consider an example task-set consisting of five tasks with $D = 26$ and the following parameters: $\tau_1 = (4, 2)$, $\tau_2 = (4, 1)$, $\tau_3 = (8, 2)$, $\tau_4 = (6, 5)$, $\tau_5 = (3, 1.5)$. In our model, the energy consumption has a convex shape, and reaches its global minimum (0.48 J) in correspondence of $T_{ck} \approx 2$ (see Figure 9). For this value, the task-set is schedulable according to our optimal scheduling strategy, and also by *MC-asc*, as the two functions coincide until $T_{ck} = 2$. However, the other heuristic

[13] ARM Cortex-A9 CPUs implement a clock cycle counter that is accessible on a dedicated per-core register [3].

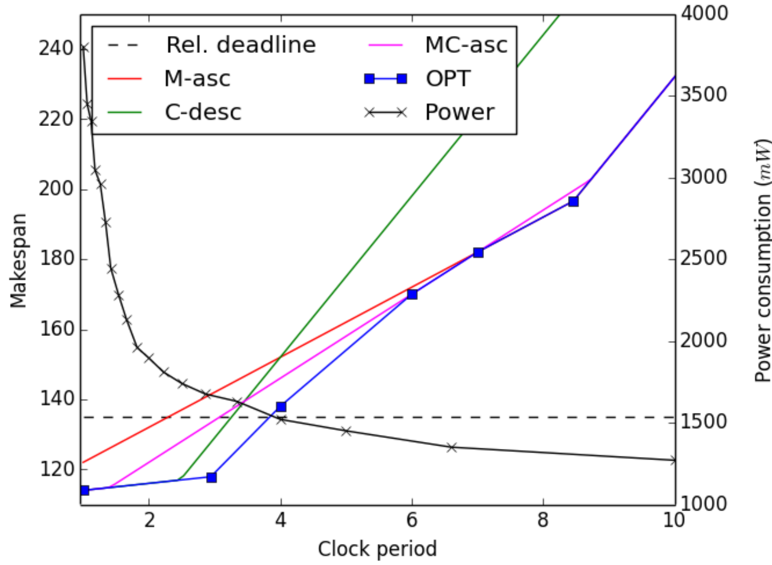


Fig. 8: Power minimization for a task-set composed of five tasks with parameters $\tau_1 = (24, 4)$, $\tau_2 = (14, 2)$, $\tau_3 = (2, 4)$, $\tau_4 = (60, 10)$, $\tau_5 = (12, 3)$ and relative deadline $D = 135$.

solutions do not allow selecting the optimal operating frequency, because that value would render the task-set unschedulable. Hence, *M-asc* and *C-desc* would require to select a lower value of clock period (1.75 and 1.66, respectively), determining a much higher energy consumption.

9.2 Schedulability in the single-core setting

We now evaluate the schedulability advantage attainable with an optimal scheduling strategy with respect to the simple heuristic approaches mentioned above. In this set of experiments, we generated 10000 task-sets in total; for each task-set, the number of tasks is chosen in the interval $n \in [2, 10]$. The task parameters are selected as follows: first, C_i is uniformly chosen in $[1, 10]$, and then M_i is selected in the interval $[C_i, C_i \cdot 50]$. In this way, we ensure that, for $T_{ck} = 1$, computation times are smaller than the respective memory-access times, which represents the most general configuration. By varying the clock period T_{ck} in the interval $[1, 20]$, we measure for each scheduling strategy its *average schedulability advantage*, defined as the average ratio

$$\frac{\sum_{i=1}^{n} M_i + t \cdot \sum_{i=1}^{n} C_i}{\mu_t^{alg}},$$

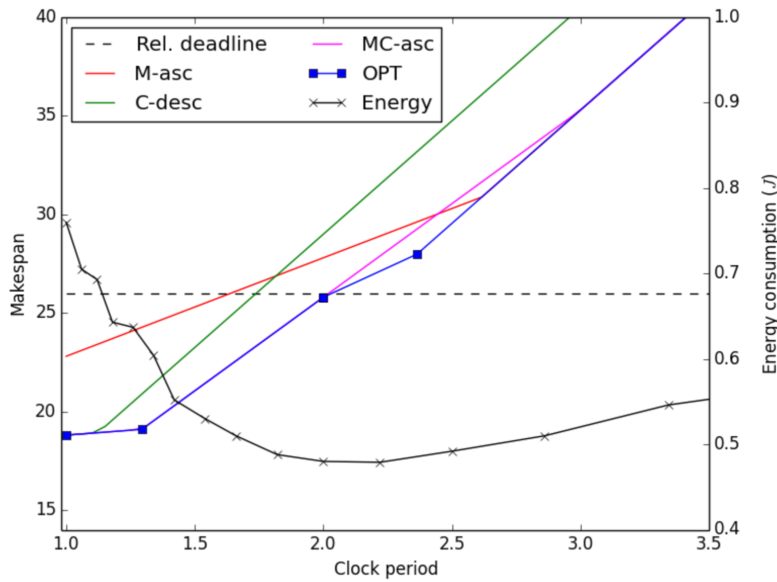


Fig. 9: Energy minimization for a task-set composed of five tasks with parameters $\tau_1 = (4, 2)$, $\tau_2 = (4, 1)$, $\tau_3 = (8, 2)$, $\tau_4 = (6, 5)$, $\tau_5 = (3, 1.5)$ and relative deadline $D = 26$.

where μ_t^{alg} represents the value of the makespan at $T_{ck} = t$ when alg is the adopted scheduling strategy. Intuitively, this quantity represents how much the considered scheduling strategy is able to gain over a sequential execution. The results are shown in Figure 10. As expected, the curve corresponding to OPT dominates the other ones for all values of T_{ck}. The schedulability advantage of OPT is much more evident for larger values of T_{ck}, where all heuristic strategies show a significant performance degradation. Notably, the performance of *C-desc* exhibits the worst performance, since for large value of T_{ck} the optimal ordering is achieved by sorting the jobs in ascending order of M_i.

9.3 Schedulability in the partitioned multi-core setting

Although the scheduling strategy described in Section 7.1 for the partitioned multi-core scenario yields sub-optimal solutions, it is interesting to study its performance with respect to the theoretical optimum and other heuristics. Hence, we briefly study the schedulability properties of our heuristic, based on the Johnson's scheduling strategy (referred to as $JOHN$), with respect to (i) the theoretical optimum, computed by brute-force search, and (ii) a simple heuristic, namely MC, that schedules jobs by sorting them in ascending order of the ratio M_i/C_i.

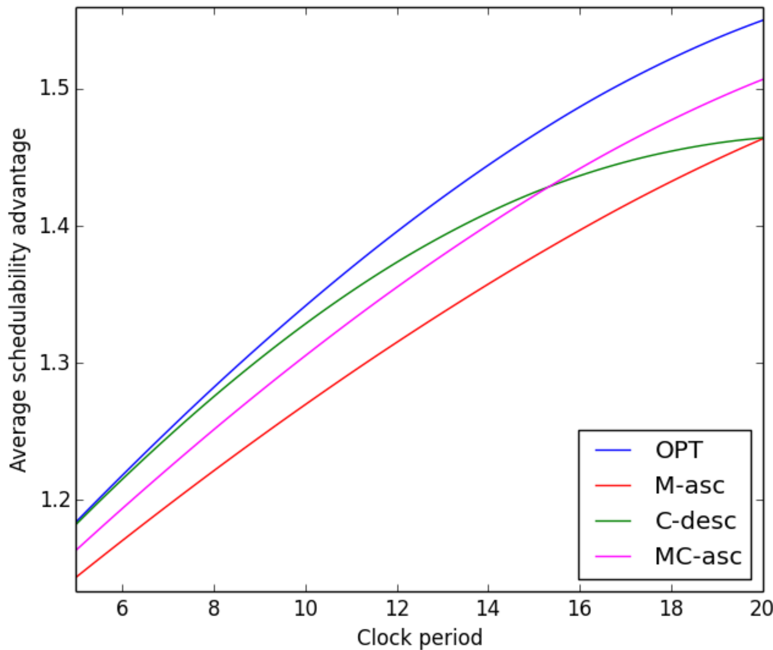
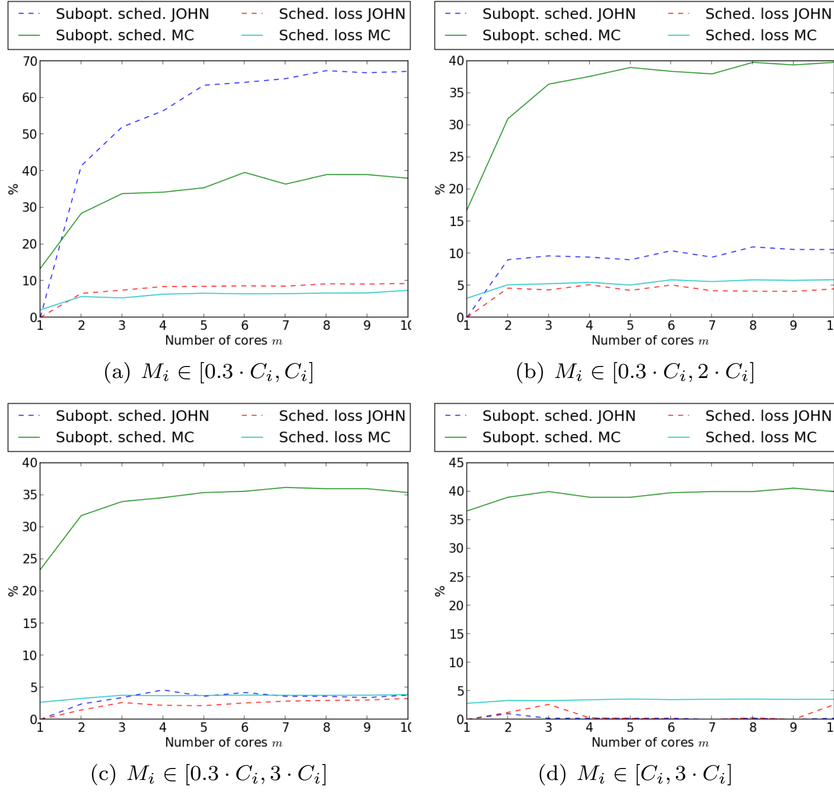


Fig. 10: Average schedulability advantage of the optimal scheduling strategy in comparison to heuristic approaches.

Figure 11 reports the obtained results as a function of the number m of cores available. In this set of experiments, 10000 task-sets have been generated for each value of $m \in [1, 10]$; for each task-set, the number of tasks is uniformly chosen as $n \in [4, 9]$. Then, computation times C_i are uniformly selected as integers in the interval $[1, 10]$. Finally, memory-access times M_i are uniformly selected in the interval $[M_{min}, M_{max}]$, whose values are reported in the figure captions. For each experiment, we recorded (i) the percentage of suboptimal schedules of $JOHN$ and MC w.r.t. the theoretical optimum (Suboptimal sched.), and (ii) the average additional makespan length in percentage for suboptimal schedules (Sched. loss).

The results show that when the memory-access time of each task is smaller than its corresponding C_i ($M_i \in [0.3 \cdot C_i, C_i]$, Figure 11(a)), the *JOHN* heuristic detects more suboptimal schedules than *MC*, although the schedulability loss remains always below 10% for both strategies. However, when memory-access times are selected in a wider interval (Figures 11(b) and 11(c)), the results show the opposite trend, i.e., in average our heuristic *JOHN* outperforms *MC* for all values of m. The improvement of *JOHN* over *MC* is even magnified when memory-access times are greater than their respective computation times (Figure 11(d)). In the latter case, *JOHN* detects <5% suboptimal



(a) $M_i \in [0.3 \cdot C_i, C_i]$

(b) $M_i \in [0.3 \cdot C_i, 2 \cdot C_i]$

(c) $M_i \in [0.3 \cdot C_i, 3 \cdot C_i]$

(d) $M_i \in [C_i, 3 \cdot C_i]$

Fig. 11: Scheduling performance of *JOHN* and *MC* multicore heuristics with respect to theoretical optimum: percentage of task-sets with suboptimal schedule with respect to the total number of generated task-sets (suboptimal sched.); average additional length of makespan in percentage for suboptimal schedules (sched. loss).

schedules, with a negligible schedulability loss. Conversely, *MC* detects ∼40% suboptimal schedules, with a schedulability loss of around 5%.

To explain this trend, Figure 12 depicts an example of a task-set for which the *MC* heuristic performs better than our heuristic.

The task-set is composed of five tasks with the following parameters: $\tau_1 = (8, 14), \tau_2 = (2, 2), \tau_3 = (7, 12), \tau_4 = (4, 8), \tau_5 = (1, 2)$. Tasks τ_2, τ_3 and τ_5 are assigned to CPU 1, while tasks τ_1 and τ_4 are assigned to CPU 2. It can be noted that *JOHN* optimizes the overlapping between computation and memory operations, achieving an overlapping of 24 time units. Unfortunately, it does not optimize the overlapping of computation performed on different CPUs, which results in 5 time units of CPU-to-CPU overlapping. Conversely, the *MC* heuristic, despite the lower memory-computation parallelism (21 time

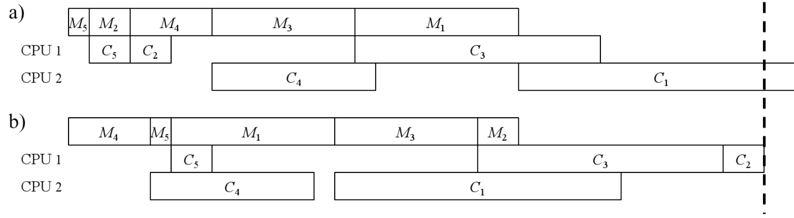


Fig. 12: Example of a task-set for which *MC* (inset (b)) produces a better schedule than the proposed *JOHN* heuristic (inset (a)).

units), achieves a CPU-to-CPU parallelism of 9 time units, yielding a smaller makespan.

Intuitively, when memory-access times are smaller than computation times, as in the example above, the *JOHN* heuristic simply orders jobs in ascending order of M_i, without considering the CPU-to-CPU overlapping. Vice versa, the *MC* heuristic achieves better performance because, considering the ratio M_i/C_i, the resulting schedule is generally more balanced in terms of CPU-to-CPU overlapping. This situation exemplifies the trend observed in Figure 11(a). When M_i is selected in a wider interval (Figures 11(b) and (c)), or is greater than its respective C_i (Figure 11(d)), our heuristic achieves better performance, because Johnson's algorithm considers the value of C_i to order jobs whenever $M_i > C_i$ (i.e., it sorts them in descending order of C_i), typically producing a larger CPU-to-CPU overlapping than *MC*.

10 Conclusion and Future Work

Co-scheduling algorithms are increasingly being developed to exploit the great potential of modern architectures, and particularly to coordinate the access to memory and computing resources. In this paper, we considered a system composed of DMA-CPU tasks executing sequentially on the two resources. We developed an algorithm that optimally determines the speed of one or both resources with the objective of minimizing power usage while ensuring the schedulability of a given task-set. The algorithm leverages the seminal results on flow-shop scheduling to propose an exact solution for the problem. The proposed algorithm is shown to have a quadratic complexity in the task-set size, hence it can be efficiently applied for both offline and online operations. Moreover, we proved that the two-stage flow-shop problem in the more general setting with intermediate deadlines is NP-complete. In addition, applications to multiprocessor systems and energy minimization were considered. Finally, a quantitative evaluation of the proposed approaches in comparison to common heuristics showed that a significant performance gain can be achieved in terms of both schedulability and power/energy savings.

As future work, we intend to perform a more extensive evaluation of the proposed methodologies using a commercially-available hardware platform.

Additionally, we plan to study the two-resource scheduling problem under more generic settings. This includes task models that represent parallel workload with precedence constraints by means of a directed acyclic graph (DAG). Finally, we envision that by introducing additional assumptions or constraints to the problem, the time complexity of our algorithm could be further improved.

References

1. Alhammad A, Pellizzoni R (October 2014) Schedulability analysis of global memory-predictable scheduling. In: 14th International Conference on Embedded Software (EMSOFT)
2. Alhammad A, Wasly S, Pellizzoni R (April 2015) Memory efficient global scheduling of real-time tasks. In: 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)
3. ARM Holdings (2016) ARM Cortex-A9 - technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388e/index.html
4. Barcelo N, Kling P, Nugent M, Pruhs K, Scquizzato M (2015) On the complexity of speed scaling. In: Mathematical Foundations of Computer Science 2015, Springer, pp 75–89
5. Baruah S (2014) Improved multiprocessor global schedulability analysis of sporadic dag task systems. In: 2014 26th Euromicro Conference on Real-Time Systems, pp 97–105
6. Benini L, De Micheli G (2012) Dynamic power management: design techniques and CAD tools. Springer Science & Business Media
7. Bonifaci V, Marchetti-Spaccamela A, Stiller S, Wiese A (2013) Feasibility analysis in the sporadic dag task model. In: 2013 25th Euromicro Conference on Real-Time Systems, pp 225–233
8. Chen B (1995) Analysis of classes of heuristics for scheduling a two-stage flow shop with parallel machines at one stage. Journal of the Operational Research Society 46(2):234–244
9. Cox MG (1971) An algorithm for approximating convex functions by means by first degree splines. The Computer Journal 14(3):272–275
10. De Vogeleer K, Memmi G, Jouvelot P, Coelho F (2014) The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In: Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol 8384, Springer, pp 793–803
11. Devadas V, Aydin H (2012) On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. Computers, IEEE Transactions on 61(1):31–44
12. Edelsbrunner H, Guibas LJ, Sharir M (1989) The upper envelope of piecewise linear functions: algorithms and applications. Discrete & Computational Geometry 4(1):311–336

13. Egger B, Lee J, Shin H (October 2008) Scratchpad memory management in a multitasking environment. In: 8th ACM International Conference on Embedded Software (EMSOFT)
14. Garey M, Johnson D, Sethi R (1976) The complexity of flowshop and jobshop scheduling. Mathematics of Operation Research 1(2):117–129
15. Hoogeveen J, Lenstra J, Veltman B (1996) Preemptive scheduling in a two-stage multiprocessor shop is NP-hard. European J Oper Res 89:172–175
16. Imamoto A, Tang B (October 2008) Optimal piecewise linear approximation of convex functions. In: World Congress on Engineering and Computer Science (WCECS)
17. Ishihara T, Yasuura H (August 1998) Voltage scheduling problem for dynamically variable voltage processors. In: International Symposium on Power Electronics and Design (InLow)
18. Jejurikar R, Pereira C, Gupta R (June 2004) Leakage aware dynamic voltage scaling for real-time embedded systems. In: 41st Design Automation Conference (DAC)
19. Jia Z, Li Y, Wang Y, Wang M, Shao Z (2015) Temperature-aware data allocation for embedded systems with cache and scratchpad memory. ACM Trans Embed Comput Syst 14(2):30:1–30:24
20. Johnson S (1954) Optimal two- and three-stage production schedules with setup times included. Naval Res Logist Q 1:61–68
21. Kaup F, Melnikowitsch S, Hausheer D (2014) Measuring and modeling the power consumption of openflow switches. In: Network and Service Management (CNSM), 2014 10th International Conference on, pp 181–186
22. Melani A, Mancuso R, Cullina D, Caccamo M, Thiele L (March 2016) Speed optimization for tasks with two resources. In: 19th International Conference on Design, Automation and Test in Europe (DATE)
23. Melani A, Bertogna M, Bonifaci V, Marchetti-Spaccamela A, Buttazzo G (November 2015) Memory-processor co-scheduling in fixed priority systems. In: 23rd International Conference on Real-Time Networks and Systems (RTNS)
24. Pellizzoni R, Betti E, Bak S, Yao G, Criswell J, Caccamo M, Kegley R (April 2011) A predictable execution model for COTS-based embedded systems. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)
25. Rabaey JM, Chandrakasan A, Nikolic B (2003) Digital Integrated Circuits (2nd Edition). Prentice Hall electronics and VLSI series, Prentice Hall
26. Schranzhofer A, Chen JJ, Thiele L (April 2010) Timing analysis for TDMA arbitration in resource sharing systems. In: 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)
27. Schuurman P, Woeginger GJ (2000) A polynomial time approximation scheme for the two-stage multiprocessor flow shop problem. Theor Comput Sci 237(1-2):105–122

28. Serrano MA, Melani A, Bertogna M, Quinones E (2016) Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1066–1071
29. Tabish R, Mancuso R, Wasly S, Alhammad A, Phatak SS, Pellizzoni R, Caccamo M (2016) A real-time scratchpad-centric OS for multi-core embedded systems. In: 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 1–11
30. Vandewalle J (1975) On the calculation of the piecewise linear approximation to a discrete function. IEEE Transactions on computers (8):843–846
31. Venkata S, Ahn I, Jeon D, Gupta A, Louie C, Garcia S, Belongie S, Taylor M (2009) SD-VBS: The San Diego Vision Benchmark Suite. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp 55–64
32. Vogelsang T (2010) Understanding the energy consumption of dynamic random access memories. In: Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on, pp 363–374
33. Wang Y, Shao Z, Chan HCB, Liu D, Guan Y (2014) Memory-aware task scheduling with communication overhead minimization for streaming applications on bus-based multiprocessor system-on-chips. IEEE Transactions on Parallel and Distributed Systems 25(7):1797–1807
34. Wasly S, Pellizzoni R (2013) A dynamic scratchpad memory unit for predictable real-time embedded systems. In: 25th Euromicro Conference on Real-Time Systems (ECRTS)
35. Wasly S, Pellizzoni R (April 2014) Hiding memory latency using fixed priority scheduling. In: 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)
36. Yao G, Pellizzoni R, Bak S, Betti E, Caccamo M (2011) Memory-centric scheduling for multicore hard real-time systems. In: Real-Time Systems, Springer