# Resource Speed Optimization for Two-Stage Flow-Shop Scheduling

Alessandra Melani*, Renato Mancuso†, Daniel Cullina†, Marco Caccamo†, Lothar Thiele‡
*Scuola Superiore Sant'Anna, Pisa, Italy, alessandra.melani@sssup.it
†University of Illinois at Urbana-Champaign, USA, {rmancus2, dcullina, mcaccamo}@illinois.edu
‡Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, thiele@ethz.ch

**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 either resource with the goal of minimizing power consumption while meeting schedulability constraints. We derive an algorithm that finds an exact solution to the problem in polynomial time, hence it is suitable for online operation even in the presence of variable real-time workload.

## I. 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 [1, 2] 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 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 [3]. 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 of the two resources (e.g., CPU) can be operated such that 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 variable-speed resource 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 and derive our results assuming fixed DMA speed and variable CPU speed (the same approach can be entirely reused for an equivalent system where the speed of the first resource is varied instead). In the following, speed is quantified in terms of variable clock period of the computing resource. As clarified in Section V, this allows us to reason on piecewise linear functions, hence 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 CPU (or memory) speed when single-rate periodic tasks that run across two stages of single-unit resources are considered.The selected optimal speed allows minimizing power consumption while ensuring that schedulability constraints are met.

**Organization of the paper.** The remainder of this paper is organized as follows. In Section II, we review related work. In Section III, we present the adopted system model and assumptions. Section IV establishes the necessary background. Next, we describe the proposed algorithm in Section V, while in Section VI we discuss its complexity. In Section VII we evaluate our algorithm by simulation studies. Finally, Section VIII concludes the paper and outlines future work.

## II. 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 [4]. More recently, the flow-shop problem has been shown to be strongly NP-hard if jobs consist of more than two stages [3], or if two or more resources are available for each stage [5]. To overcome such limitations, several heuristic solutions [6] and polynomial-time approximation schemes (PTAS) [7] 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. An attempt in this direction has been pursued by Pellizzoni et al. [1], 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 [2, 8, 9, 10].

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 [11, 12, 13]. 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 that satisfies schedulability constraints.

## III. 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 $\tau_i$ 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. The underlying assumption is that, in the considered platform, 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 operating frequency is known to have a directly proportional impact on power consumption [14], modern platforms are endowed with Dynamic Voltage and Frequency Scaling (DVFS) units that allow live adjustments of the operating frequencies at a high 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.

While frequency scaling is an effective way to scale performance and power consumption 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 high granularity of performance scaling. 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.

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

[1]Although reasoning in terms of clock period $T_{ck}$ describes well the performance of CPUs, it is more appropriate to reason in terms of bandwidth when describing the performance of memory subsystems. However, 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.

[3]See http://www.nvidia.com/object/tegra-k1-processor.html

[4]See http://ark.intel.com/products/75123/Intel-Core-i7-4770K

[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

happen at times $r + kT$, being $k$ any positive integer. Each job released by $\tau_i$ 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 task $\tau_i$ can progress in parallel with $C$-phases ($M$-phases) of a different task $\tau_j$.

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 [4]. More specifically, since one job of each task is released at multiples of $T$, it is enough to compute the (optimal) makespan of such collection of jobs and check it against the global relative deadline $D = T$ 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 [3] or when two or more resources are available for each stage [5]. Moreover, if tasks do not share the same period/deadline, the schedulability problem is no longer equivalent to that of makespan minimization, and since no optimal scheduling algorithm is known for the general schedulability of multi-stage tasks, it is nontrivial to extend our results to more general settings while preserving optimality.

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 is a first step to address the broader problem of speed selection for the class of multi-stage execution models. We envision that future research can extend this work to the scheduling of three-stage jobs (under certain assumptions [4]), and to optimally adjust speed of multiple resources. Finally, while extending the task model to arbitrary periods is not needed for practical purposes, we plan to extend it to support a limited set of task rates. For instance, our approach could be reused with different job rates on a multicore platform if each rate group is scheduled in isolation on private hardware resources (core with dedicated DMA channel).

## IV. 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.

### A. Johnson's algorithm

Johnson's algorithm [4] provides an optimal solution to schedule a collection of same-deadline, time-synchronized two-resource tasks. The following theorem defines the relative ordering between pairs of jobs in an optimal schedule.

*Theorem 4.1:* (from [4]) Given a collection $J_1, \ldots, J_n$ of two-stage flow-shop jobs, $J_i$ precedes $J_j$ in an optimal schedule if

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

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 increasing order, while jobs in $S_2$ are sorted in decreasing order;
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 $O(n\ log(n))$. In the rest of the paper, we will denote as $\sigma^t = \{\sigma_1^t, \ldots, \sigma_n^t\}$ the permutation of jobs that determines the optimal schedule when $T_{ck} = t$.

### B. Computing the optimal makespan

Given an optimal job ordering $\sigma^t = \{\sigma_1^t, \ldots, \sigma_n^t\}$, the corresponding value of makespan $\mu^t$ is given by:

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

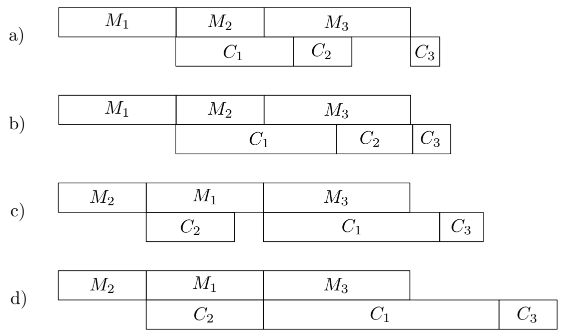

Fig. 1. Example of a task-set composed of three tasks with parameters $\tau_1 = (4,4)$, $\tau_2 = (3,2)$, $\tau_3 = (5,1)$. The four insets illustrate the optimal schedule when a) $T_{ck} = 1$; b) $T_{ck} = 1.33$; c) $T_{ck} = 1.5$; d) $T_{ck} = 2$.

This expression gives us many insights. Specifically, it indicates that an optimal schedule has the following characteristics: (i) there are no internal gaps in the use of either resource; (ii) there exists a *critical path* that determines the minimum makespan; (iii) such a critical path can be determined by identifying a *crossover job* $J_i^*$ such that jobs preceding it in the optimal schedule contribute to $\mu^t$ with their memory-access time, while subsequent jobs contribute with their computation time; (iv) $J_i^*$ contributes to $\mu^t$ with both $M_i$ and $C_i$.

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

## V. 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 IV-A) 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 jobs as depicted in Figure 1(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 1(b), and the makespan increases at a higher rate. Then, as soon as the computation time of $\tau_2$ 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 1(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 1(d)).

Figure 2 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 1. 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 the power consumption while ensuring the schedulability of the considered task-set.

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 1, changing points may be of two types, according to the following definitions.

*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]

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

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 $t = \hat{t}$).

[7]We recall that $\sigma_1^t, \ldots, \sigma_n^t$ is the permutation of jobs corresponding to the minimum makespan when $T_{ck} = t$.

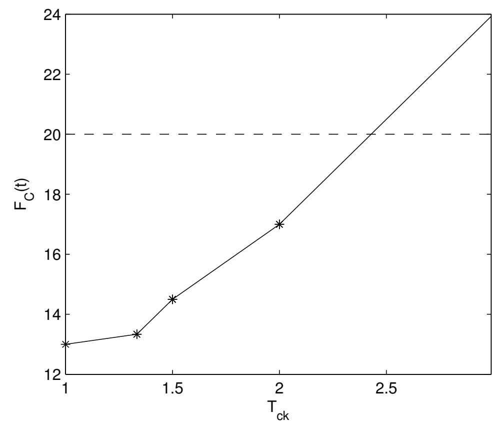

Fig. 2. Example of the function $F_C(t)$ associating the clock period $T_{ck}$ to the value of the optimal makespan for the task-set in Figure 1. The horizontal line corresponds to the relative deadline $D = 20$. The portion of the function lying below the line identifies the values of $T_{ck}$ that make the task-set schedulable.

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.

*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 $T_{ck} = t'' = M_j/C_j$, provided that $t'' > t'$.

*Proof:* Without loss of generality, assume that, for $T_{ck} = t'$, the $C$-phases of the two jobs are both shorter than the corresponding $M$-phases, because otherwise Johnson's algorithm would sort them in ascending order by the length of their $M$-phase, and no reordering would be possible. Since in the initial ordering $J_i$ precedes $J_j$, it must also hold that $C_j < C_i$. Therefore, we just need to consider the two following initial configurations: i) $C_j < C_i < M_i < M_j$; ii) $C_j < C_i < M_j < M_i$.

*Case i).* The $C$-phases of the two jobs are sorted in descending order, while their $M$-phases are sorted in ascending order. It trivially follows by Johnson's rule that no job reordering is possible for any value of $t'' > t'$.

*Case ii).* By linearly increasing $C_i$ and $C_j$, no job swapping may occur until the relative ordering between task parameters changes. When $C_i$ reaches the value of $M_j$ (say, at $T_{ck} = t^1 = M_j/C_i$), a new configuration is produced, i.e., $C_j^{t^1} < M_j < C_i^{t^1} < M_i$. By Equation (1) we get: $\min(M_i, C_j^{t^1}) = C_j^{t^1} < \min(M_j, C_i^{t^1}) = M_j$, i.e., no reordering takes place. Next, a different configuration is produced at $T_{ck} = t^2 > t^1$ when either: a) the $C$-phase of $J_j$ becomes larger than $M_j$, leading to $M_j < C_j^{t^2} < C_i^{t^2} < M_i$; b) the $C$-phase of $J_i$ becomes larger than $M_i$, leading to $C_j^{t^2} < M_j < M_i < C_i^{t^2}$.

In case ii.a), it follows by Equation (1) that a job swapping occurs at $T_{ck} = t^2 = M_j/C_j$. From this point on, $M_j$ will always lead the minimum in Equation (1), hence no further update is possible.

In case ii.b), it follows by Equation (1) that no reordering takes place for $T_{ck} = M_i/C_i$. However, when at $T_{ck} = t^3 > t^2$ the $C$-phase of $J_j$ reaches the value of $M_j$, Johnson's rule imposes a job swapping. Arguing similarly as in case ii.a), we conclude that no further reordering is possible for $T_{ck} > t^3$.

Having proved that a job reordering may occur only at $T_{ck} = M_j/C_j$, the lemma follows. ■

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 Figures 1. 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, a larger number of jobs start contributing to the makespan increase rate, hence the slope of $F_C(t)$ can only increase. ■

The two lemmas above can be applied to the example of Figure 2 to visually identify schedule and crossover changing points.

## A. Finding changing points

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

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

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

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 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 $S_i = \sigma^t$ for $\mathcal{P}_{s,i} \le t < \mathcal{P}_{s,i+1}$.

**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($\sigma_{curr}, S$)
17:             if $f = -1$ and $r \ge 1$ 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 according to $M_i$. 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 $\sigma^{t^-}$ to $\sigma^t$. 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 1(b) and 1(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 $\sigma^t$ if $M_j < M_i$. However, if $C_j < M_j$, then $\tau_j$ will be computation-dominated and always scheduled after $J_i$.

Therefore, in the *for* loop at lines 7-21, we distinguish between jobs that have passed their schedule changing point and jobs that have not. It is enough to order the former class in ascending order by $M_i$ and the latter class in descending order by $C_i$. The concatenation of the two sets will represent the optimal schedule at $T_{ck} = t$. The array $L$, initialized at line 5, will progressively store 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 $L$ is needed to remove placeholder $null$ objects and construct a valid candidate schedule (line 12). 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.

If the schedule $\sigma_{curr}$ 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 accordingly (lines 15 and 16). Finally, since we are only interested in the changing points within $[1, +\infty)$, all points below 1 should be discarded. At line 22, the function $\textsc{Reinit}$ filters the points, based on the first element to consider (i.e., $f$ at line 18), setting 1 as first element of $\mathcal{P}_s$ and updating $S$ accordingly.

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

*b) 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 3(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 4(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 3(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 $\tau_2$ to be 4 instead of 6, as in Figure 4(a), we observe that when $T_{ck}$ reaches $M_2/C_1 = 1.33$ (Figure 4(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 4(c).

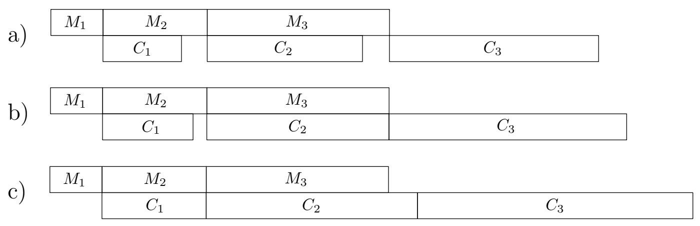

Fig. 3. Example of a task-set composed of three tasks with parameters $\tau_1 = (2, 3)$, $\tau_2 = (4, 6)$, $\tau_3 = (7, 8)$. The three insets illustrate the optimal schedule when a) $T_{ck} = 1$; b) $T_{ck} = 1.17$; c) $T_{ck} = 1.33$.

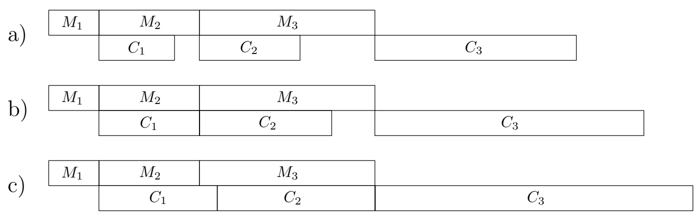

Fig. 4. Example of a task-set composed of three tasks with parameters $\tau_1 = (2, 3)$, $\tau_2 = (4, 4)$, $\tau_3 = (7, 8)$. The three insets illustrate the optimal schedule when a) $T_{ck} = 1$; b) $T_{ck} = 1.33$; c) $T_{ck} = 1.57$.

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

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 3, 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 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

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

```
1: procedure CrossoverPoints($\mathcal{T}$, $\mathcal{P}_s$, $i$, $S$)
2:     $t \leftarrow \mathcal{P}_{s,i}$; $\rho \leftarrow \emptyset$; $\sigma^t \leftarrow S_i$
3:     $Z \leftarrow$ AdjacentJobs($\mathcal{T}, \sigma^t$); $D \leftarrow 0$; $N \leftarrow 0$
4:     for $j =$ Size($Z$) to 1 do
5:         if $j \neq$ Size($Z$) and $t \cdot Z_j.ck > \varphi.ck$ then
6:             if $i ==$ Size($\mathcal{P}_s$) or $\varphi.ck < \mathcal{P}_{s,i+1}$ then
7:                 $\rho \leftarrow$ Append($\varphi, \rho$)
8:             end if
9:             $D \leftarrow 0$; $N \leftarrow 0$
10:        end if
11:        $D \leftarrow D + C(Z_j.jobs)$
12:        $N \leftarrow N + M(Z_j.jobs)$
13:        $\varphi.ck = t \cdot (N/D)$; $\varphi.x \leftarrow Z_j.jobs_1$
14:    end for
15:    if $i ==$ Size($\mathcal{P}_s$) or $\varphi.ck < \mathcal{P}_{s,i+1}$ then
16:        $\rho \leftarrow$ Append($\varphi, \rho$)
17:    end if
18:    return $\rho$
19: end procedure
```

of the interval, and the list of optimal schedules $S$. It produces in output a structure $\rho$ 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 VI, this is needed to efficiently compute the value of $F_C(t)$ once the list of changing points is known.

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

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

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

Note that the index shift in Equation (5) (i.e., $\sigma^t_{h+1}$ instead of $\sigma^t_h$) 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 1(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 3(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 $\varphi.ck$, it means that the gap corresponding to $Z_j$ will be filled *later* than the one relative to $\varphi$ (i.e., *in order*), and the two groups of jobs will give rise to two distinct crossover changing points. Hence, $\varphi$ is appended to $\rho$, 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, 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 $\varphi$, 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 $\rho$ (subject to the same check performed at line 6), which is finally returned as output.

## B. 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($\mathcal{T}$)
 2:     $\{\mathcal{P}_s, S\} \leftarrow$ SchedulePoints($\mathcal{T}$); $\mathcal{P} \leftarrow \mathcal{P}_{s,1}$
 3:     $\mathcal{P} \leftarrow$ Append(CrossoverPoints($\mathcal{T}, \mathcal{P}_s, 1, S$), $\mathcal{P}$)
 4:     for $i = 2$ to Size($\mathcal{P}_s$) do
 5:         $\mathcal{P} \leftarrow$ Append($\mathcal{P}_{s,i}, \mathcal{P}$)
 6:         $\mathcal{T}' \leftarrow$ ScaleJobs($\mathcal{T}, \mathcal{P}_{s,i}$)
 7:         $\mathcal{P} \leftarrow$ Append(CrossoverPoints($\mathcal{T}', \mathcal{P}_s, i, S$), $\mathcal{P}$)
 8:     end for
 9:     return $\mathcal{P}$
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 1(d)).

## C. 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 $\mathcal{P} = \{p_1, \dots, p_\ell\}$ 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 $\alpha$ 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 list of changing points of $F_M(t)$ is then given by $\mathcal{P}' = \{p_1', \dots, p_\ell'\}$, whose generic element $p_j'$ is equal to:

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

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

[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.

## VI. Complexity

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

*Lemma 6.1:* $F_C(t)$ 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 6.2:* 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 $\mu^t$ 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.

## VII. Evaluation

In this section, we evaluate the performance of our approach by simulation studies. First, we describe a representative example that shows the power and energy savings that can be achieved when our algorithm is used to determine the optimal operating speed of the CPU. Specifically, we perform a comparison with simple heuristic-based scheduling strategies. Then, we quantify the schedulability advantage of the proposed optimal scheduling strategy by means of extensive simulations.

The first experiment has been performed by assuming specific energy and power models. Such models have been derived using an embedded platform and a realistic image processing benchmark. Since we are interested in optimizing power and energy consumption for the second resource (CPU), we first derive extract the power/energy profile of the considered benchmark on the hardware platform. Next, we assume that a generic synthetic taskset exhibits the same energy profile. This approach allows us to both reason about 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 [15]. 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 5), where $V$ is the fixed system voltage voltage. Moreover, we derive energy consumption as: $E = P \cdot R$ (see Figure 6). Not surprisingly, our finding are in line with the usual trend for power/energy observed in literature [16].

In order to exemplify how the proposed algorithm can be used to optimize the power consumption, we considered 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 compared 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 5 shows how the makespan varies according to the scheduling strategies compared. 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, such a value corresponds to a power consumption of about 1520 mW. Instead, the heuristic scheduling strategies identify a 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 higher power consumption (1614 mW, 1636 mW, and 1760 mW, respectively).

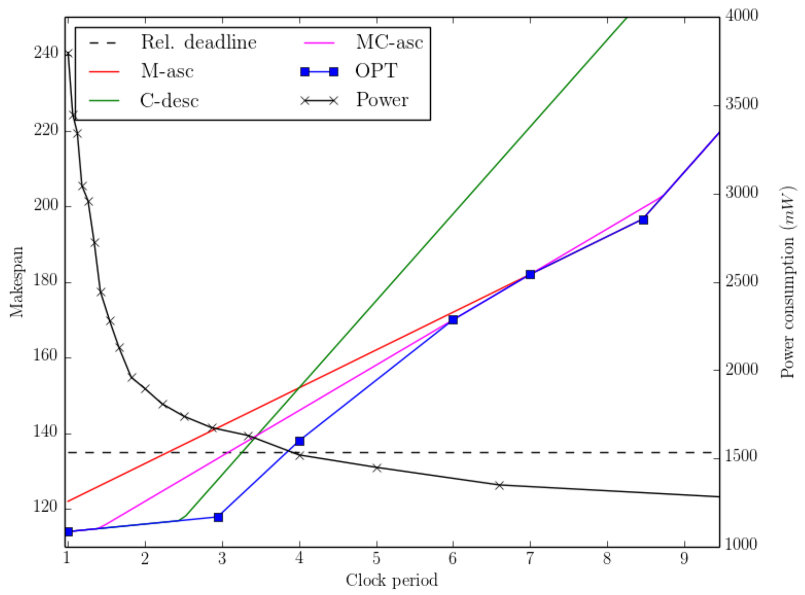

Fig. 5. Example of a task-set composed of five tasks with parameters $\tau_1 = (24, 4)$, $\tau_2 = (14, 2)$, $\tau_3 = (2, 4)$, $\tau_4 = (60, 10)$, $\tau_5 = (12, 3)$ and relative deadline $D = 135$.

The performance benefit of our algorithm is also evident whenever the energy consumption is the parameter to be optimized. 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 6). 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 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 larger energy consumption.

Finally, we performed an extensive simulation experiment to quantify the schedulability advantage that can be gained if an optimal scheduling strategy is adopted with respect to the heuristic approaches mentioned above. In particular, we generated 10000 task-sets in total; for each of them, the number of tasks is chosen in the interval $n \in [2, 10]$. The task parameters are then 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 measured for each value $T_{ck} = t$ its *average schedulability advantage*, defined as the average ratio

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

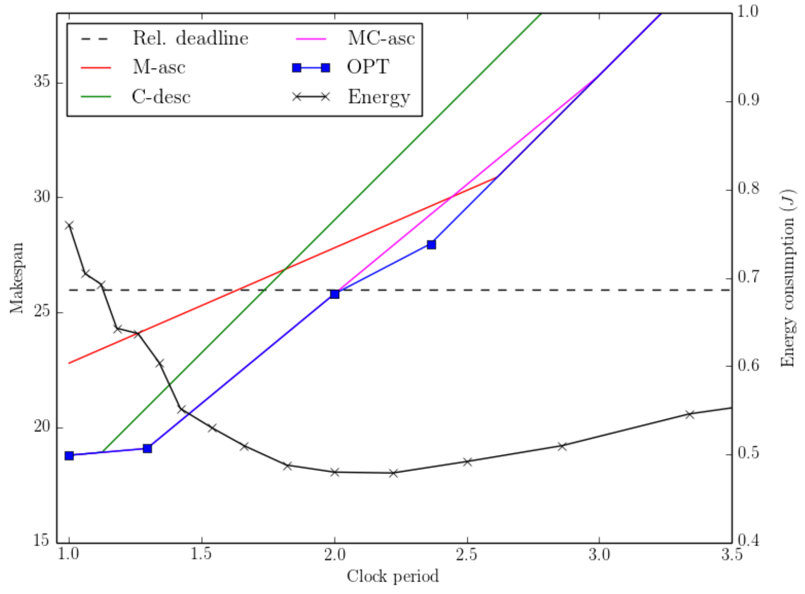

Fig. 6. Example of a task-set composed of five tasks with parameters $\tau_1 = (4, 2)$, $\tau_2 = (4, 1)$, $\tau_3 = (8, 2)$, $\tau_4 = (6, 5)$, $\tau_5 = (3, 1.5)$ and relative deadline $D = 26$.

where $\mu_t^{alg}$ 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 7. As expected, the curve corresponding to *OPT* dominates the other ones. 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 ordering the jobs at increasing values of $M_i$.

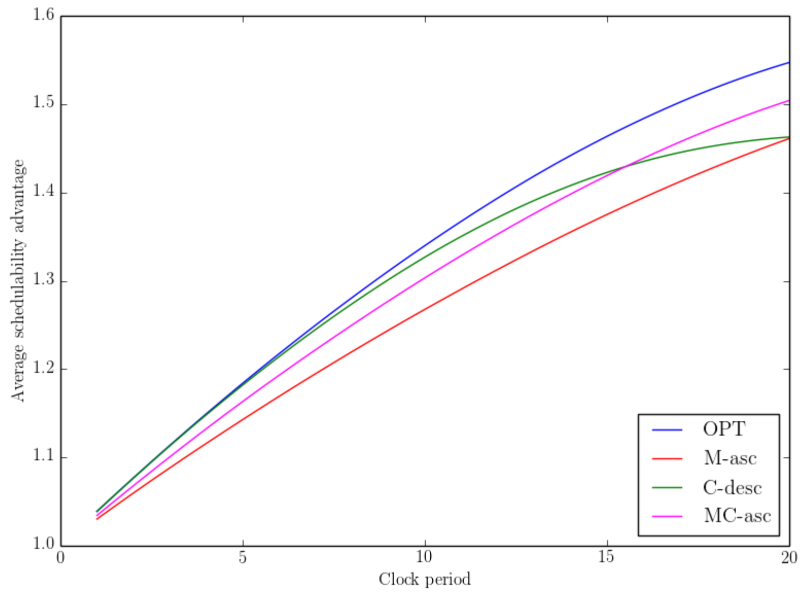

Fig. 7. Average schedulability advantage of the optimal scheduling strategy in comparison to heuristic approaches.

## VIII. 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 either resource as the one that minimizes power consumption while ensuring the schedulability of the considered task-set.

The algorithm leverages the seminal results on flow-shop scheduling to propose an exact solution for the problem. In addition, the 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.

As future work, we plan to extend our results by considering simultaneous variations in the speed of the two resources. We also intend to apply our approach to three-stage task systems in special sub-cases of interest, widening its applicability to more general task models. Finally, we envision that by introducing additional assumptions or constraints to the problem, the time complexity of our algorithm could be further improved.

## Acknowledgment

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1035736, 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.

## References

[1] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2011.

[2] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. In *Real-Time Systems*. Springer, 2011.

[3] M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operation Research*, 1(2):117–129, 1976.

[4] S.M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Res. Logist. Q.*, 1:61–68, 1954.

[5] J.A. Hoogeveen, J.K. Lenstra, and B. Veltman. Preemptive scheduling in a two-stage multiprocessor shop is NP-hard. *European J. Oper. Res.*, 89: 172–175, 1996.

[6] B. Chen. 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, 1995.

[7] P. Schuurman and G. J. Woeginger. A polynomial time approximation scheme for the two-stage multiprocessor flow shop problem. *Theor. Comput. Sci.*, 237(1-2):105–122, 2000.

[8] A. Alhammad and R. Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *14th International Conference on Embedded Software (EMSOFT)*, October 2014.

[9] A. Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2015.

[10] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *23rd International Conference on Real-Time Networks and Systems (RTNS)*, November 2015.

[11] B. Egger, J. Lee, and H. Shin. Scratchpad memory management in a multitasking environment. In *8th ACM International Conference on Embedded Software (EMSOFT)*, October 2008.

[12] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013.

[13] S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[14] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits (2nd Edition)*. Prentice Hall electronics and VLSI series. Prentice Hall, January 2003. ISBN 0130909963.

[15] S.K. Venkata, I. Ahn, Donghwan Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M.B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64, Oct 2009.

[16] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *Parallel Processing and Applied Mathematics*, volume 8384 of *Lecture Notes in Computer Science*, pages 793–803. Springer, 2014.