Preempt Less, Schedule Better: Revisiting PCG for Real-Time Uniform Processors

Yahya Hamdani
LIAS, ISAE-ENSMA, Université de Poitiers, France

Pascal Richard[1]
LIAS, Université de Poitiers, ISAE-ENSMA, France

Antoine Bertout
LIAS, Université de Poitiers, ISAE-ENSMA, France

Joël Goossens
Université libre de Bruxelles, Belgium

Emmanuel Grolleau
LIAS, ISAE-ENSMA, Université de Poitiers, France

Abstract

We address the problem of scheduling periodic implicit-deadline real-time tasks on m uniform processors. We introduce PCG*, an optimal TL-plane algorithm based on PCG [4], which guarantees at most $2(m-1)$ preemptions per TL-plane, matching the best-known theoretical bound for uniform platforms. The proposed algorithm advances the state of the art by offering an optimal real-time scheduling solution with a tight preemption bound within TL-planes. The numerical experiments presented in this work provide strong evidence that PCG* yields a substantial reduction in the number of preemptions relative to PCG. When applied to identical processor platforms, PCG* is also a best-possible polynomial time algorithm in terms of preemptions in a TL-plane, matching the $(m-1)$ preemption bound achieved by LRE-TL [9].

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

Keywords and phrases Real-Time Scheduling, Uniform Multiprocessor Platforms

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.2

Funding *Antoine Bertout*: ANR JCJC program, ANR-21-CE25-0023, project SHRIMP.

1 Introduction

Modern real-time systems are increasingly deployed at the core of safety-critical and performance-critical applications, from automotive and aerospace control to industrial automation and robotics. Guaranteeing that these systems always react within specified deadlines, while at the same time making effective use of the available processing capacity, requires the use of carefully designed scheduling algorithms supported by rigorous schedulability analysis. Such analysis techniques aim to characterize the worst-case timing behaviour of real-time tasks under a given scheduling policy, and thereby provide a proof that all timing constraints will be met during operation.

Multiprocessor platforms have become pervasive in real-time systems as modern workloads (e.g., automotive, avionics, robotics, multimedia) exceed the capabilities of single-core processors and power/thermal limits hinder further frequency scaling. Consequently, gaining performance now relies on parallelism, which real-time scheduling and analysis must explicitly account for to preserve timing guarantees. Moreover, heterogeneous architectures motivate

[1] Corresponding author

the uniform-speed multiprocessor model, which captures different effective core capacities in practical settings such as mixed-speed integration, reduced availability due to interference or background services, incremental platform upgrades [2], DVFS/DVS operating modes with speed–energy trade-offs [19, 24], and modern processors combining distinct core "types", reinforcing the need for schedulability methods that handle speed asymmetries [1, 21].

Among the large variety of real-time scheduling policies that have been proposed in the literature, particular attention has been devoted to optimal algorithms, which are capable of scheduling any task set that is feasible on a given platform. All currently known optimal polynomial-time algorithms assume that task migrations and preemptions are performed at no cost and incur no delay. The reason is that, once preemptions and migrations are explicitly modelled as incurring non-zero delays, the problem complexity changes dramatically: the resulting real-time scheduling problems become computationally hard, even on single-core platforms. For example, the uniprocessor real-time scheduling problem with explicit Cache-Related Preemption Delays (CRPD) has been shown to be strongly $\mathcal{NP}$-hard, and no online optimal algorithm exists [18]. This observation highlights why the zero-overhead assumption is adopted when designing optimal multiprocessor schedulers. It is not merely a technical convenience, but a fundamental requirement for obtaining optimal algorithms with tractable schedulability tests. Understanding the capabilities and limitations of optimal algorithms under this zero-overhead assumption is therefore a fundamental step both towards clarifying the core combinatorial difficulties of real-time scheduling and towards bridging the gap between theoretical optimality results and the behaviour of real systems, ultimately enabling the design of scheduling algorithms and analysis techniques that incorporate realistic operational constraints.

For multiprocessor systems, numerous optimal scheduling algorithms have been proposed [6]. Most of these algorithms target platforms with identical processors and task sets composed of periodic tasks with implicit deadlines (i.e., each task must complete before its next release). It is known that only global dynamic-priority scheduling algorithms (i.e., a job priority may change during its execution) can achieve optimality in this setting, in the sense that they can fully utilize all processors. In contrast, fixed-priority scheduling policies and partitioned scheduling strategies have worst-case utilization bounds asymptotically limited to 50%.

Notably, optimal algorithms based on DP-fair [15] principles (i.e., Deadline Partitioning fair scheduling) like those based on the TL-plane abstraction [4, 5, 9, 10] (i.e., a TL-plane defines a plane with Time and Local remaining execution times of jobs) decompose the periodic task scheduling problem into independent time intervals, each bounded by successive task releases/deadlines. Each interval then constitutes an independent subproblem, consisting of a set of local jobs derived from the original periodic tasks. The objective within each subproblem is to ensure that all local jobs complete before the next task deadline, which is common to all local jobs.

In this paper, we focus on uniform multiprocessor systems, in which each processor may operate at a different speed. Several optimal scheduling algorithms have been proposed in this context, each capable of meeting all task deadlines for any feasible task set. More precisely, we refine the PCG (Precaution Cut Greedy) online algorithm [4] so that it achieves the best-known lower bound on the number of preemptions within a TL-plane. In this paper, we introduce PCG*, an optimal TL-plane algorithm that guarantees at most $2(m-1)$ preemptions per TL-plane, matching the best-known theoretical bound for uniform platforms [12]. To the best of our knowledge, this is the first real-time scheduling algorithm that guarantees this preemption bound. Furthermore, when restricted to identical processors, PCG* achieves the

best possible preemption bound of $m-1$ [17], thereby matching the preemption bound of the LRE-TL algorithm [9], which is designed for identical multiprocessors. The proposed algorithm advances the state of the art by minimizing preemptions.

2 System model

We consider a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of independent sequential periodic real-time tasks to be scheduled on a multiprocessor platform. Each task $\tau_j = (C_j, T_j)$, for $1 \leq j \leq n$, is characterized by its worst-case execution time $C_j \in \mathbb{R}^+$ on a unit-speed processor and its period $T_j \in \mathbb{N}$, i.e., the interval between two consecutive job releases of τ_j. Each job of τ_j must complete execution before the release of its next job, implying an implicit deadline. The utilization factor of task τ_j is defined as $u_j = \frac{C_j}{T_j}$, representing the fraction of processing capacity, on a unit-speed processor, required by τ_j over time. Tasks are synchronously released at time 0.

An event-driven scheduler makes run-time decisions only when specific events occur: job releases, completions, or deadlines. Each such *scheduling event* defines a *scheduling point* at which the scheduler may select the jobs to execute.

A task set is said to be *feasible* if there exists a schedule in which every job completes execution by its deadline. A scheduler is an algorithm that assigns tasks to processors for execution. A scheduling algorithm is deemed *optimal* if it can successfully schedule every feasible task set.

Our main objective was to define an optimal polynomial-time scheduling algorithm with a guaranteed bound on the number of preemptions, which can only be achieved if preemption overheads are not taken into account in the task model. Accordingly, we propose a scheduling algorithm that does not consider such overheads.

In such scheduling algorithms, a job can be in one of three states: running if it is currently executing on a processor, ready if it is eligible to execute but not assigned to any processor, or completed. During execution, a job may be preempted on its current processor and later resumed – possibly on a different processor, without incurring any overhead, since otherwise no polynomial-time optimal scheduling algorithm can exist. Consequently, *preemptions and migrations are not distinguished* and are collectively referred to as *preemptions* throughout this paper, following the terminology used in job scheduling in operational research. This unified terminology enables a direct comparison of scheduling algorithms from both communities with respect to their preemption behaviour.

A processor π_i, $1 \leq i \leq m$, is characterized by its speed $s_i \in \mathbb{R}^+$, representing the amount of processing capacity provided per unit of time. If a task executes for d time units on processor π_i, it completes $d \times s_i$ units of execution. This model is referred to as a uniform multiprocessor platform. When all processor speeds are identical, the platform is called an identical multiprocessor platform, in which case we assume $s_i = 1$ for all i. We consider a continuous-time model.

3 Bridging real-time and job scheduling

3.1 DP-Fair Schedules

Scheduling periodic tasks can be done by ensuring proportionate fair progression of tasks. At any time, each task progresses at a constant rate corresponding to its utilization. This means that, over any time interval of length Δ, task τ_i executes for exactly $u_i \times \Delta$ time units, where u_i is its utilization factor. This theoretical abstraction of such an idealized execution is referred to as a fluid schedule.

While it is impossible to implement a truly fluid schedule at runtime, it is possible to approximate its behavior within fixed intervals, each bounded by consecutive releases/deadlines. At each deadline, the schedule must have completed the same amount of work as the fluid schedule. This approach is known as Deadline-Partitioning fair scheduling, or DP-fair for short [15]. The same principle underlies TL-plane-based schedulers [4,5], where each interval, called a TL-plane, ensures that tasks make proportionate progress toward their deadlines. This approach defines the *class of DP-fair schedules*.

The schedule to be defined within each interval is independent of the schedules in the other intervals. In a given interval of length Δ, the portion l of the k^{th} job, $\tau_{i,k}$, corresponding to task τ_i that must be executed is referred to as a *local job* $\tau^l_{i,k}$, with a *local execution time*, denoted by $p^l_{i,k}$, equal to $p^l_{i,k} = u_i \times \Delta$. It follows directly that, between the release time and deadline of a job, which may span several successive intervals, the sum of the local execution times of its corresponding local jobs is equal to the task's worst-case execution time. Each local job must be completed by time Δ, which is the common deadline for all local jobs belonging to the same interval.

Let us consider a simple example adapted from [25] with two processors, π_1 and π_2, with respective speeds $s_1 = 3$ and $s_2 = 1$, and two periodic tasks with implicit deadlines, $\tau_1 = (C_1 = 40, T_1 = 20)$ and $\tau_2 = (C_2 = 60, T_2 = 30)$. The total utilization is $U = 40/20 + 60/30 = 4$, which is exactly equal to the platform capacity (i.e., $s_1 + s_2$). Because of the processor speeds, this task set is not partitionable: neither task can be assigned entirely to π_2, and no priority-driven policy can schedule them without allowing a task to execute simultaneously on both processors over time. The only way to schedule these two tasks is to share both processors equally within each time interval delimited by successive deadlines.

This is illustrated in Figure 1, which depicts a DP-fair schedule over the hyperperiod (i.e., $H = \text{lcm}(T_1, T_2) = 60$). The set of intervals is: $[0, 20)$, $[20, 30)$, $[30, 40)$, and $[40, 60)$. Within the hyperperiod, tasks τ_1 and τ_2 generate 3 and 2 jobs, respectively. The first job of τ_1 is executed the first intervals and thus corresponds to one local jobs, $\tau^1_{1,1}$, with local execution times equal to $p^1_{1,1} = u_1 \times 20 = 40$. Its second job spans on the two next intervals with two local jobs, $\tau^1_{1,2}$ and $\tau^2_{1,2}$, with local execution times $p^1_{1,2} = p^2_{1,2} = u_1 \times 10 = 20$, respectively; its third job is executed in the last interval with one local job $\tau^1_{1,3}$, with respective local execution times equal to $p^1_{1,3} = u_1 \times 20 = 40$. The first job of τ_2 spans on the two first intervals with two local jobs, $\tau^1_{2,1}$ and $\tau^2_{2,1}$, with local execution time equal to $p^1_{2,1} = u_2 \times 20 = 40$ and $p^2_{2,1} = u_2 \times 10 = 20$, respectively; its second job corresponds to two local jobs $\tau^1_{2,2}$ and $\tau^2_{2,2}$ in the two subsequent intervals (i.e., $[30, 40)$ and $[40, 60)$), with respective local execution times equal to $p^1_{2,2} = u_2 \times 10 = 20$ and $p^2_{2,2} = u_2 \times 20 = 40$.

In Figure 1, the feasible DP-fair schedule pattern defined in $[0, 20)$ is reused in every subsequent interval, scaled according to the interval length. All local jobs belonging to the same interval share a common deadline, namely the end of that interval. In the next interval, $[20, 30)$, the same schedule pattern is stretched to fit an interval of length 10. Note that local job processing requirements are always proportional to the interval length. Therefore, the pattern defined in the first interval can be reused with the corresponding local execution times, and similarly in all subsequent intervals throughout the schedule. For this reason, the feasibility analysis of a DP-fair schedule reduces to validating a single interval, in which every local job shares a common deadline.

As a consequence, for a given interval of length Δ, the feasibility problem consists in checking whether a set of local jobs can be scheduled within a time interval of length Δ, i.e., whether there exists a preemptive schedule whose makespan is less than or equal to Δ, where the makespan is defined as the completion time of the last job in the schedule. Therefore, the feasibility problem is equivalent to the well-known makespan minimization problem in the operations research literature.

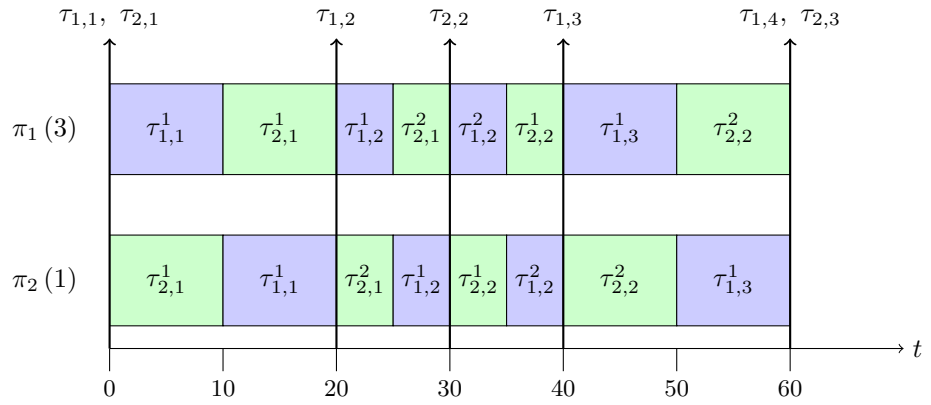


Figure 1 Example of DP-fair scheduling: the same schedule pattern is stretched between subsequent deadlines, which are also releases (implicit deadlines).

The principles of DP-fair schedules provide a straightforward decomposition method for managing infinite schedules that arise from task periodicity. The schedule of local jobs within each interval can be computed either online or offline. In the offline case, the schedule is precomputed for a single interval and then stretched or replicated at runtime to fit subsequent intervals. The offline schedule can also be computed at the beginning of each interval. In both situations, however, an offline algorithm cannot account for variations in job execution times at run-time. If an online algorithm is used instead, the computation can adapt to variations in job execution times. Regardless of the scheduling approach used within each interval, fairness is ensured at every deadline, since the local jobs have execution requirements proportional to the length of the interval. Crucially, this approach reduces the infinite-horizon feasibility problem to a finite makespan minimization problem within a single interval.

3.2 Feasibility and makespan minimization

The equivalence between the feasibility of a periodic task set and the makespan minimization of a corresponding set of local jobs allows the derivation of a closed-form formula for checking the feasibility of such periodic tasks. We next distinguish the feasibility tests between identical and uniform processors.

Identical platforms. A necessary and sufficient condition for a periodic task set to be schedulable on an identical multiprocessor platform is given by [3]: $u_i \leq 1, 1 \leq i \leq n$ and $\sum_{i=1}^{n} u_i \leq m$. These inequalities are equivalent to those arising in the makespan minimization problem on identical multiprocessor platforms, where task utilizations correspond to job execution times. This yields the well-known lower bound on the makespan, defined by McNaughton [17]:

$$\max\left(\frac{1}{m}\sum_{i=1}^{n} u_i,\ \max_{1\leq i\leq n} u_i\right) \leq 1. \tag{1}$$

Uniform platforms. The feasibility condition for uniform platforms generalizes the previous approach. A necessary and sufficient condition for feasibility is given by the minimum makespan scheduling problem [13], where task utilizations correspond to job execution times.

We now assume that tasks are indexed in non-increasing order of their utilization factors, i.e., $u_1 \geq u_2 \geq \cdots \geq u_n$, and that processors are indexed in non-increasing order of their speeds, i.e., $s_1 \geq s_2 \geq \cdots \geq s_m$.

In Horváth *et al.* [13], the minimum makespan for scheduling a set of jobs on a uniform platform is given by: (i) the average load of the k tasks with the largest utilization factors when scheduled on the k fastest processors; (ii) the average load of all n tasks when scheduled on the m processors. This in turn yields a necessary and sufficient condition for feasibility for periodic real-time tasks [8]:

$$\max\left(\max_{1\leq k\leq m-1}\frac{\sum_{j=1}^{k}u_j}{\sum_{i=1}^{k}s_i},\ \frac{\sum_{j=1}^{n}u_j}{\sum_{i=1}^{m}s_i}\right)\ \leq\ 1. \tag{2}$$

Feasibility and makespan minimization equivalence. Due to the established equivalence between the feasibility analysis of periodic task sets and the makespan minimization of local jobs scheduled between two successive releases/deadlines in a DP-fair schedule, an optimal real-time scheduling algorithm for periodic tasks with implicit deadlines must also optimally solve the corresponding makespan minimization problem. Conversely, any algorithm that optimally solves the makespan minimization problem can be used to construct a feasible schedule for periodic tasks by replicating its execution pattern across successive intervals of the infinite real-time schedule. This equivalence has been exploited to define DP-Wrap, a simple yet effective algorithm for scheduling periodic tasks on identical processors, which applies McNaughton's wrap-around scheduling rule within each interval [15].

3.3 Scheduling algorithms

We next describe the two main approaches for optimally scheduling local jobs within an interval bounded by two successive releases/deadlines.

3.3.1 Identical platforms

McNaughton's algorithm [17] is a simple wrap-around rule. To construct a schedule of length $C_{\max}$ (as defined in Equation (1)), jobs are first sequenced in an arbitrary order on a single virtual processor without any idle time. This single-processor schedule is then partitioned into slices of duration $C_{\max}$, where the i-th slice defines the schedule of processor π_i. Under this construction, a job is preempted if and only if it spans across two consecutive slices, meaning its execution is split between the end of the schedule on π_i and the beginning of the schedule on π_{i+1}, with no overlap.

The application of McNaughton's rule to periodic task scheduling is known as DP-Wrap [15]. McNaughton's algorithm guarantees that at most $m-1$ preemptions can occur [17], and this bound is tight. Moreover, it has been proven that constructing a makespan-optimal schedule with fewer than $(m-1)$ preemptions is $\mathcal{NP}$-hard [23].

Several online algorithms have also been proposed for identical processor platforms. The Longest Remaining Processing Time (LRPT) rule is the simplest algorithm that is optimal for minimizing the makespan [14]; it can be seen as the preemptive version of the Longest Processing Time first scheduling rule. However, LRPT suffers from a major drawback: in practice, it requires an infinite number of preemptions. Cascades of preemptions arise when two or more jobs have exactly the same remaining execution time and must therefore be executed at the same rate, as in the fluid schedule model. This issue can be circumvented by applying the processor-sharing concept (for details, see [20]). Other algorithms have been

proposed in the real-time systems community, most notably TL-plane–based algorithms such as LLREF [5] and LRE-TL [9]; their principles will be described in the next section. In particular, LRE-TL guarantees no more than $m-1$ migrations per TL-plane [9], thereby matching the $m-1$ preemption bound achieved by McNaughton's algorithm.

3.3.2 Uniform platforms

Several offline algorithms have been proposed in the literature for minimizing the makespan. Gonzalez and Sahni [12] introduced an $\mathcal{O}(n)$-time algorithm for scheduling a set of jobs on uniform processors, which minimizes the makespan while guaranteeing at most $2(m-1)$ preemptions. The algorithm assumes jobs and processors are sorted by non-increasing execution time and speed, respectively. Jobs are then scheduled sequentially. Note that such an offline algorithm cannot account for variations in job execution times at run-time.

This approach was later revisited in two significant works. In Martel's paper [16], uniform processors are modeled as a single composite processor with time-varying speed, consisting of segments of constant speed within each interval of length $C_{\max}$ defined in Equation (2). A similar idea, based on multiple virtual processors with variable speeds, was proposed in [7]. Both algorithms are considerably easier to understand than the original method of Gonzalez and Sahni.

The $2(m-1)$ preemption bound is tight, as shown in [12]. Moreover, it has been proven that constructing a schedule with fewer than $2(m-1)$ preemptions is $\mathcal{NP}$-hard [22]. Therefore, this bound constitutes the best possible guarantee achievable by any polynomial-time scheduling algorithm.

For online scheduling algorithms, the LRPT variant that assigns the job with the longest remaining execution time to the fastest processors (LRPT-FM) is optimal for minimizing the makespan, but it suffers from the same limitations as standard LRPT. The Level algorithm [13] provides a better trade-off by ensuring at most $n^2(m-1)$ preemptions [12]. In the real-time systems community, two equivalent algorithms have been independently proposed: PCG [4] and U-LLREF [10]. However, as we demonstrate in this paper, these algorithms do not achieve the best possible preemption bound. In the following sections, we present a detailed overview of TL-plane–based event-driven scheduling algorithms.

4 Principles of TL-plane algorithms

In the remainder of this paper, we focus on DP-fair schedules based on the TL-plane abstraction [5]. As shown previously, both the feasibility analysis and the scheduling algorithms are concerned with a single interval delimited by two successive releases/deadlines. We formally proved in [11] that a deterministic DP-fair scheduler generates exactly the same schedule pattern in every interval. Accordingly, the periodic task model is fully represented by a set of local jobs, one for each original periodic task, together with their associated local execution times. *For the sake of simplicity, we hereafter refer to them simply as jobs and execution times since we focus on a single TL-plane.*

In the TL-plane abstraction, the progress of any job within an interval – bounded by two successive releases/deadlines – can be represented in a two-dimensional plane, referred to as the TL-plane. In this representation, the horizontal axis (T) denotes time, while the vertical axis (L) denotes the remaining execution time. As established earlier, the initial execution time of the (local) job is equal to the TL-plane length multiplied by the job's task utilization.

The first TL-plane–based algorithm, LLREF [5], is optimal for identical processor platforms. It follows the Longest Remaining Processing Time (LRPT) principle: at each scheduling point, the m jobs with the longest remaining local execution times are assigned

to the m processors until the next scheduling event occurs. These events define scheduling points at which tasks are rescheduled according to LRPT. LRE-TL [9] improves upon LLREF by applying LRPT only at the initialization of each TL-plane, rather than at every scheduling point. Precisely, two types of scheduling events are defined. These types of events are independent and may occur simultaneously[2]:

- **Bottom event (*B*-event):** occurs when a task completes, i.e., its local execution time reaches 0.
- **Critical event (*C*-event):** occurs when a task reaches zero laxity on a processor, i.e., the remaining execution time equals the remaining processor capacity available in the TL-plane.

These ideas were extended to uniform processor platforms in [4]. Before presenting that algorithm, called PCG, we first refine the definition of a C-event.

▶ **Definition 1.** *A job i triggers a C_k-event at time $t \geq 0$ if its remaining execution time at t is exactly equal to the remaining processing capacity of processor π_k over the interval $[t, L)$, where L denotes the schedule length. The remaining execution time at time t is defined by:*

$$p_i(t) = (L - t)\, s_k,$$

To avoid multiple preemptions, when a job triggers a C_k-event, PCG assigns the job to processor π_k for the remainder of the schedule. From that point on, it is never preempted. From an implementation perspective, the job and its assigned processor are removed from the corresponding lists of jobs and processors. The scheduling problem is thus decomposed into a simpler subproblem for completing the schedule. PCG operates as follows [4]:

- **Initialization:** Jobs are sorted in non-increasing order of their execution times, and processors are sorted in non-increasing order of their speeds.
- **Main loop:**
  - At any time $t \geq 0$, upon each C_k-event, the corresponding job is permanently assigned to π_k for the remainder of the TL-plane. Both the job and the processor are then removed from the scheduling problem.
  - At each B-event, the corresponding job is removed from the problem.
  - The remaining jobs are assigned to processors according to the LRPT-FM rule, until the next event occurs.

Jobs and processors are sorted in non-increasing order of execution times and processor speeds, respectively. Let $P_k = \sum_{j=1}^{k} p_j$ for $1 \leq k \leq m-1$, $P_n = \sum_{j=1}^{n} p_j$, and $S_k = \sum_{i=1}^{k} s_i$ for $1 \leq k \leq m$. Horváth's makespan formula can then be expressed as [13]:

$$C_{\max} = \max\left(\max_{1 \leq k \leq m-1}\left(\frac{P_k}{S_k}\right),\ \frac{P_n}{S_m}\right). \tag{3}$$

As an illustrative example, we consider the set of local jobs defined in Table 1. Horváth's formula yields a makespan defined in the Equation (3) of:

$$C_{\max} = \max\left(\frac{28}{10}, \frac{48}{18}, \frac{64}{22}, \frac{69}{24}, \frac{75}{25}\right) = 3$$

[2] LRE-TL also considers A-events to handle releases of sporadic tasks. These events are not considered in this paper, since we only focus on periodic tasks.

Table 1 Job set for illustrating Fig. 2.

	1	2	3	4	5	6
p_i	28	20	16	5	7/2	5/2
s_j	10	8	4	2	1	

Table 2 Job set for illustrating Fig. 3.

	1	2	3	4	5
p_i	20	19	18	17	2
s_j	4	3	2	1	1

Hence, this set of jobs is feasible if the interval has a length of at least 3 time units. Figure 2 depicts the corresponding TL-plane computed by PCG: dashed lines represent processor capacities decreasing over time, while solid lines represent the remaining execution times of jobs. Bullets denote scheduling points corresponding to events triggered by jobs. Initially, the five longest jobs are scheduled on the processors according to the LRPT-FM rule. At time 0.5, the ready Job 6 triggers a C_5-event, as it must be executed continuously on π_5 to complete by the makespan. Job 6 and processor π_5 are then removed from the scheduling problem. The relative order of the remaining execution times of the running jobs remains unchanged. Job 5, previously executing on π_5, is preempted at time 0.5 and enters the ready state until it triggers a C_4-event at time 1.5. Both Job 5 and processor π_4 are then removed. The remaining running jobs are 1, 3, and 2; they are reassigned, in this order, to the fastest processors according to LRPT-FM, while Job 4 waits for an available processor. At time 2, Job 1 triggers a C_2-event and is removed along with processor π_2. The updated assignments are: Job 3 on π_1, Job 2 on π_3, and Job 4 continues to wait. Job 3 subsequently triggers a C_3-event, leading to the removal of both Job 3 and processor π_3. In the final subproblem, π_1 is the only available processor, executing Job 2 until its completion, which triggers a B-event at time 2.8. At the same moment, the ready Job 4 triggers a C_1-event and resumes execution on π_1 until the makespan. The schedule is thus completed with a total of 8 preemptions.

The PCG algorithm was proven optimal for scheduling periodic tasks with implicit deadlines in [4], but no preemption bound is guaranteed. We now discuss the limitations of PCG in achieving the best-known bound of $2(m-1)$.

We analyze the number of preemptions triggered at B-events by the original PCG algorithm with LRPT-FM sorting applied at each scheduling point. A set of 5 jobs is executed on 5 processors, with parameters given in Table 2. As shown in Figure 3, Job 5 completes on processor π_5 at time 2, triggering a B-event. The remaining jobs are then resorted by PCG and reassigned, resulting in 4 preemptions. Around time 4, Jobs 1 and 4 simultaneously trigger C-events on π_2 and π_3, causing 4 additional preemptions. Around time 6, Jobs 3 and 2 trigger C-events on π_1 and π_4, adding 2 more preemptions. In total, 10 preemptions occur in the TL-plane, exceeding the theoretical bound of $2(m-1) = 8$.

No tight preemption bound is known for PCG. However, as previously shown, B-events can induce a significant number of preemptions, and this number may increase with the total number of jobs. As mentioned earlier, the best-known algorithms generate at most $2(m-1)$ preemptions, a bound that is independent of the number of jobs. Achieving this bound requires that (i) no preemptions are caused by B-events, and (ii) C-events yield at most two preemptions.

5 PCG* for uniform platforms

This paper is motivated by two fundamental questions regarding the behavior of the PCG algorithm:

- Is the application of the LRPT-FM rule necessary at every scheduling point to ensure optimality? This rule requires sorting jobs in non-increasing order of execution times. Avoiding this sorting step can prevent unnecessary preemptions within a TL-plane.

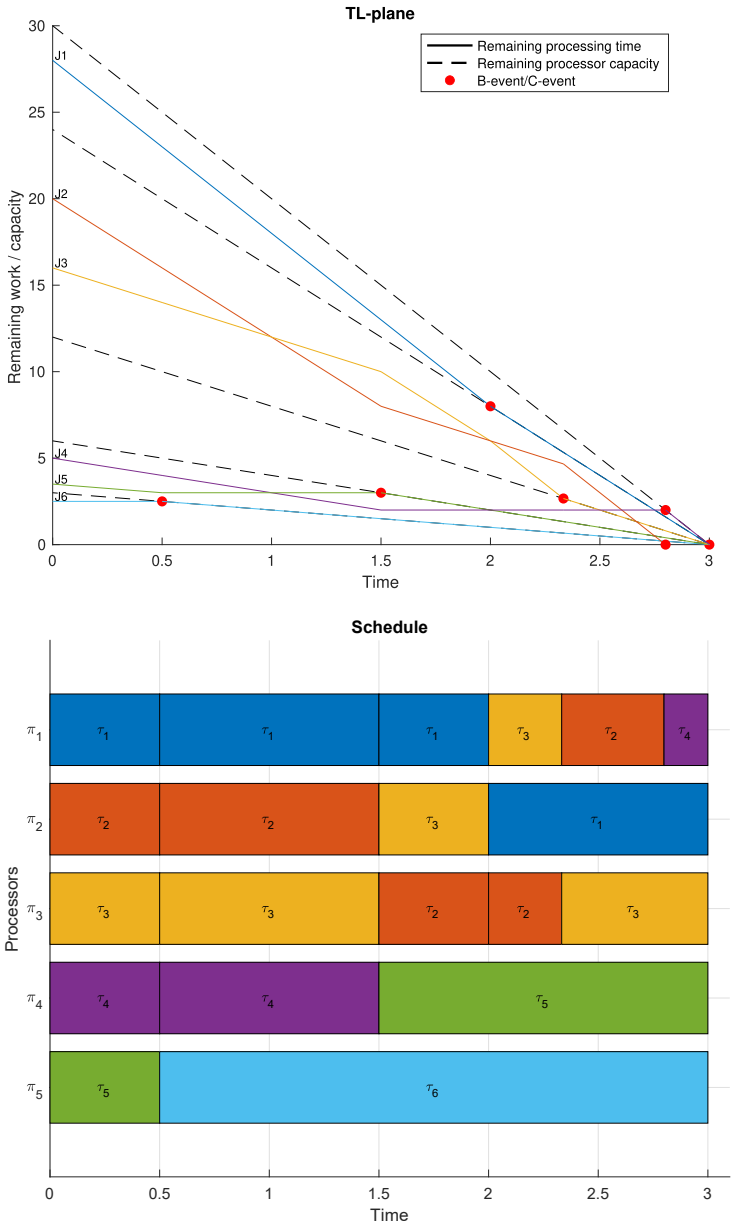


Figure 2 PCG TL-plane and associated schedule on Martel's job set [16].

- Is it possible for a real-time algorithm to attain the tight preemption bound of $2(m-1)$ within a TL-plane? Achieving this bound provides a guarantee equivalent to that of the best-known offline polynomial-time scheduler.

We now introduce PCG* and show that it achieves both objectives.

5.1 The PCG* algorithm

The PCG* algorithm schedules local jobs within the current TL-plane. TL-plane initialization consists of computing the processing requirements of local jobs to meet the DP-fair constraints: $p_j = \Delta u_j, j = 1 \ldots n$, where Δ is the TL-plane length. The initial local job ordering never changes from one TL-plane to another; hence, it can be computed once, before the system starts.

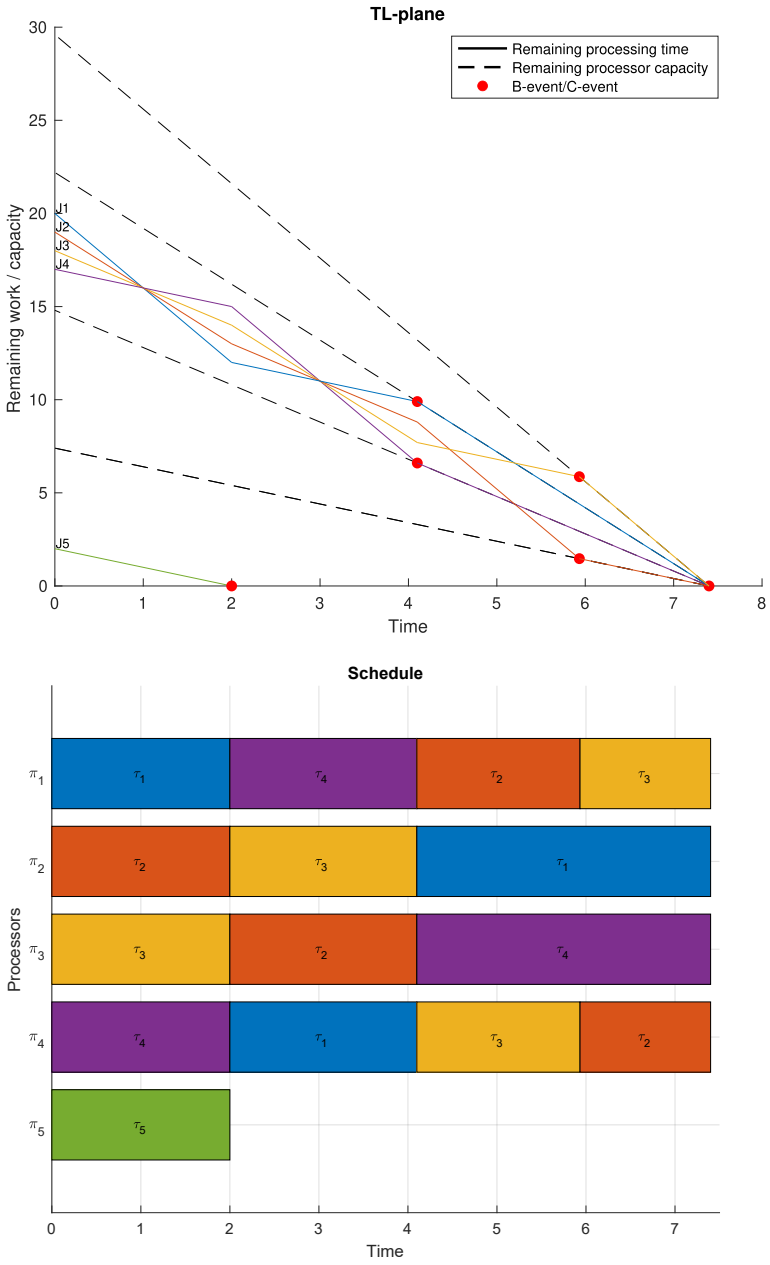


Figure 3 PCG algorithm generates preemptions at B-events.

As previously illustrated, PCG exceeds the preemption bound. To address this, PCG* redefines the event-handling logic: (i) B-events no longer trigger preemptions, and (ii) LRPT-FM is not applied. Assume that the optimal makespan satisfies $C^* = \frac{P_k}{S_k}$ in Equation (3). Then, in any makespan-optimal preemptive schedule of length C^*, the k fastest machines are fully occupied by the k longest jobs throughout the entire interval $[0, C^*]$ [20]. Therefore, to be optimal, PCG* must enforce this job-assignment constraint when handling C-events. This refined strategy is described hereafter.

Algorithm 1 summarizes the principles of PCG* applied on each TL-plane for a feasible system. It calls Algorithm 2 to construct the schedule between successive scheduling points. In the case of C-events, a job can be assigned to a processor until the end of the TL-plane;

Algorithm 1 PCG* on each TL-plane.

Input: list of n (local) jobs sorted in non-increasing order of their execution time; list of m processors sorted in non-increasing order of their speeds
Output: Feasible schedule on the TL-plane with at most $2(m-1)$ preemptions

```
Assign the m first jobs to the m first processors.
while the list of jobs is not empty do
    Compute the next instant at which jobs trigger events. Let E be the set of such
     events, keeping only one C-event per processor k (chosen arbitrarily).
    Call Algorithm 2 on E with current assignment
    Update jobs and processors lists
    Update schedule and execution times from the new assignment
end
```

once this occurs, both the job and the processor are removed from their respective lists. Execution times are updated based on the elapsed time between events and the processors' speeds. Note that, at each call of Algorithm 2, all jobs initially assigned to processors are by default in the running state. Also, jobs made assignable (eligible) by a C-event and all processors available for assignment are marked as *free*. Furthermore, C-events are fully handled (i.e., with the corresponding assignment of free jobs/processors) before B-events in order to guarantee the job-assignment constraint mentioned earlier. Lastly, a processor freed by a B-event is assigned only to a waiting job (i.e., one that was not previously running) and, consequently, does not generate any preemptions.

The implementation details are omitted due to space limitations. In our implementation, PCG runs in $\mathcal{O}(n \log n)$ time at each scheduling point because it re-sorts the jobs, whereas PCG* runs in $\mathcal{O}(nm)$ time because tasks are not re-sorted; consequently, determining the next event is more expensive (i.e., C-events can disrupt the initial job ordering). In both algorithms, there are at most n scheduling points in a TL-plane. Therefore, PCG* runs in $\mathcal{O}(n^2 m)$ time per TL-plane, whereas PCG requires $\mathcal{O}(n^2 \log n)$ time per TL-plane. However, we note that it is still possible to sort jobs in PCG* to accelerate event detection while preserving the behaviour of the presented algorithm, thereby yielding exactly the same $\mathcal{O}(n^2 \log n)$ complexity as PCG.

To illustrate PCG*, we reconsider the example from Martel [16], depicted in Figure 4. Unlike PCG, PCG* does not apply the LRPT-FM rule at every scheduling point. Consequently, at time 1.5, Jobs 2 and 3 are not preempted and continue executing on the same processors. At time 2, Jobs 1 and 2 respectively trigger a C_3-event and a C_2-event. The remaining scheduling subproblem consists of two jobs, Jobs 3 and 4. Job 3 is assigned to π_1, the last available processor. It completes at time 2.8, triggering a B-event. Simultaneously, Job 4 triggers a C_1-event and continues execution until completion, thereby finalizing the schedule. In this case, PCG* generates 5 preemptions in the TL-plane, whereas PCG yields 8.

5.2 Correctness

Like PCG, PCG* is optimal in the sense that, for any feasible task set, it produces a valid schedule in every TL-plane. This optimality is directly tied to the way C-events are handled.

The original optimality proofs of TL-plane–based algorithms rely on the feasibility condition stated in Equation 2. However, since we have established a direct equivalence between this feasibility condition and the makespan minimization problem within a TL-plane (in Section 3.2), we can provide a simpler optimality proof based on the latter.

Algorithm 2 Handling of events in E.

Input: E, jobs and processors lists, job-to-processor assignment
Output: jobs and processors lists, job-to-processor assignment, partial schedule.

```
foreach C_k-event in E do
    The job ℓ running on π_k is preempted by the job j triggering the event, ℓ is marked as free.
    If j was running, its assigned processor is marked as free. Job j is now assigned to π_k until the end of the TL-plane; both j and π_k are removed from their respective lists.
end
Assign free jobs to free processors (if any).
foreach B-event in E do
    The completed job is removed from the list and its processor is now marked as free.
    The first ready (not running) job is assigned to the free processor.
end
```

The central idea is to iteratively decompose the scheduling problem into simpler subproblems at each event (i.e., at each scheduling point). These subproblems are related through a mathematical relation that connects their respective makespans.

▶ **Definition 2.** *We use the following notation for a scheduling point at time t, with $0 \leq t \leq C_{\max}$:*

- *$p_i(t)$: the remaining execution time of job i at time t, with $p_i(0) = p_i$ by definition.*
- *$C_{\max}(t)$: the minimum remaining schedule length at time t, computed with respect to the remaining execution times of the jobs.*

The Horváth's makespan formula defined in Equation (3) can be adapted to cope with a TL-plane–based algorithm that makes scheduling decisions whenever events are triggered. Such decisions are based on the remaining execution times of the jobs, denoted $p_j(t)$ for $1 \leq j \leq n$. Let $P_k(t) = \sum_{j=1}^{k} p_j(t)$ for $1 \leq k \leq m-1$, and $P_n(t) = \sum_{j=1}^{n} p_j(t)$. Considering only the remaining portions of the jobs at time t, the corresponding schedule length is defined as:

$$C_{\max}(t) = \max\left(\max_{1\leq k\leq m-1}\left(\frac{P_k(t)}{S_k}\right), \frac{P_n(t)}{S_m}\right) \tag{4}$$

A schedule is *makespan-optimal* if it attains Horváth's makespan as defined in Equation (3). Consider the terms that attain the maximum in this formula. The corresponding subset of jobs and processors are *critical*: these jobs must execute continuously on these processors throughout the entire schedule in any makespan-optimal schedule [20]. Based on this property, we define an invariant that holds at all times.

We now define the invariant $C_{\max} = t + C_{\max}(t)$, where $C_{\max}(t)$ is computed by Equation (4). The underlying idea is as follows: the minimum makespan is determined by one of the two terms in Horváth's formula. The set of jobs associated with this critical term must be executed continuously up to the makespan in order for Horváth's makespan to be reached. If the condition is recomputed at any time t, using the remaining execution times of the jobs, the same term continues to determine the updated makespan $C_{\max}(t)$. We formally establish this invariant below.

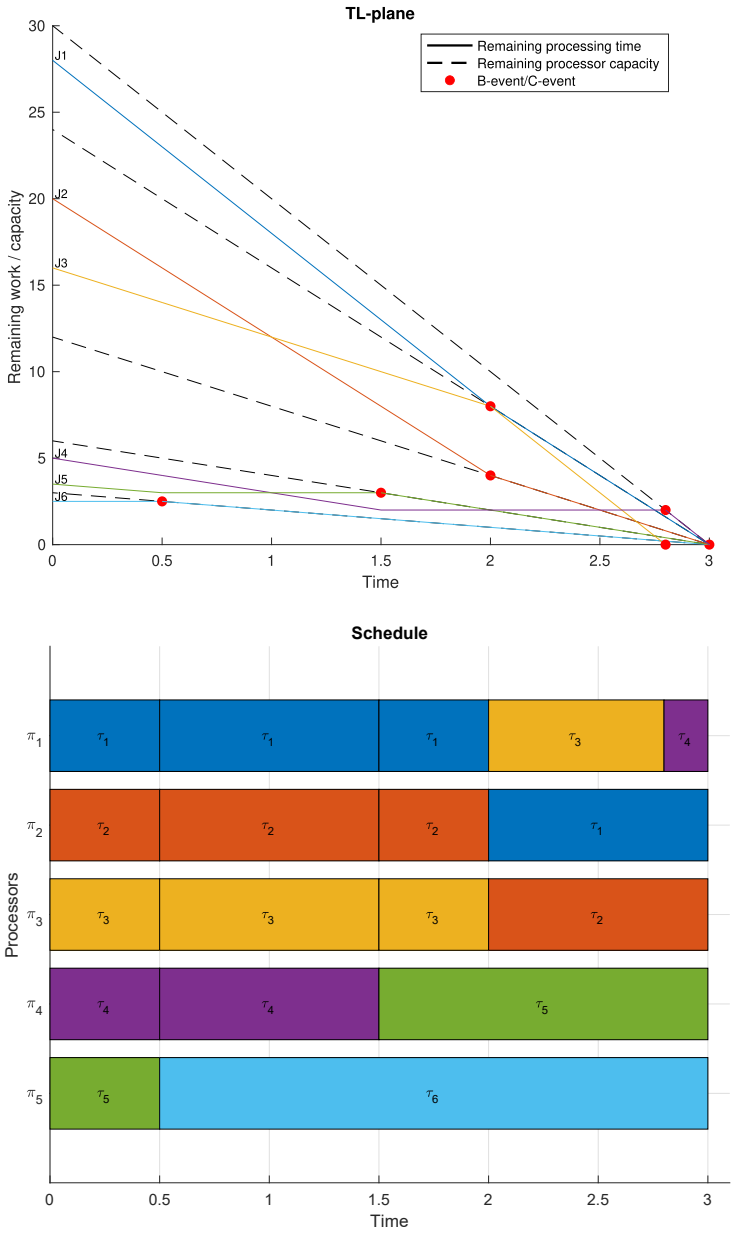


Figure 4 PCG* applied on the example of Figure 2.

▶ **Lemma 3.** *In any schedule of length $C_{\max}$, the application of PCG* satisfies the following invariant: for every time instant t such that $0 \leq t \leq C_{\max}$,*

$$C_{\max} = t + C_{\max}(t),$$

where $C_{\max}$ denotes the makespan computed at time 0 (i.e., Equation (3)), and $C_{\max}(t)$ denotes the makespan recomputed at time t based on the remaining execution times of the jobs and the set of remaining processors (i.e., Equation (4)).

Proof. Without loss of generality, we consider a variation of PCG* which assumes that jobs and processors are not removed, so as to preserve both the processing requirements and the processor capacities. Note that it does not change the makespan, since tasks assigned to processors by PCG* at C-events always end at Horváth's makespan. We consider two cases according to the two terms in Horváth's makespan formula given in Equation (3).

Case (1). Assume that the makespan is defined by the first term, for a given k: $C_{\max} = \frac{P_k}{S_k}$. There are k critical jobs. The remaining $n-k$ jobs will never be scheduled on the k fastest processors by PCG*. Hence, in the interval $[0, C_{\max})$, these k longest jobs execute *continuously* on the k fastest processors by PCG*. Therefore, $P_k(t) = P_k - tS_k$. By definition:

$$\begin{aligned} C_{\max}(t) &= \frac{P_k(t)}{S_k} = \frac{P_k - tS_k}{S_k} \\ &= \frac{P_k}{S_k} - t = C_{\max} - t \end{aligned}$$

Case (2). Similarly, assume that the makespan is defined by the second term: $C_{\max} = \frac{P_n}{S_m}$. In the interval $[0, C_{\max})$, there is no idle time in the schedule and all the jobs are continuously executed. Therefore, $P_n(t) = P_n - tS_m$. By definition:

$$\begin{aligned} C_{\max}(t) &= \frac{P_n(t)}{S_m} = \frac{P_n - tS_m}{S_m} \\ &= \frac{P_n}{S_m} - t = C_{\max} - t \end{aligned}$$

Hence, the invariant is proved in both cases, showing that PCG* reaches Horváth's makespan, and is thus makespan optimal. ◀

The previous result shows that PCG* can construct a valid schedule of length $C_{\max}$, and is used to establish the optimality of PCG*.

▶ **Theorem 4.** *PCG* is an optimal scheduling algorithm for periodic implicit-deadline tasks on a uniform multiprocessor platform.*

Proof. Lemma 3 proves that PCG* can construct a schedule of length $C_{\max} \leq \Delta$, where $C_{\max}$ is given by Horváth's formula (Equation (3)) and Δ is the length of a TL-plane.

$$\begin{aligned} \max\left(\max_{1\leq k\leq m-1}\left(\frac{P_k}{S_k}\right), \frac{P_n}{S_m}\right) &= C_{\max} \leq \Delta \\ \max\left(\max_{1\leq k\leq m-1}\frac{\sum_{j=1}^{k} p_j}{\sum_{i=1}^{k} s_i}, \frac{\sum_{j=1}^{n} p_j}{\sum_{i=1}^{m} s_i}\right) &= C_{\max} \leq \Delta \\ \max\left(\max_{1\leq k\leq m-1}\frac{\sum_{j=1}^{k} u_j}{\sum_{i=1}^{k} s_i}, \frac{\sum_{j=1}^{n} u_j}{\sum_{i=1}^{m} s_i}\right) &\leq 1 \qquad \text{since } u_j = p_j/\Delta \end{aligned}$$

This is Equation 2, and consequently PCG* is optimal. ◀

5.3 Preemption bound

A TL-plane scheduling algorithm is designed to schedule jobs within an interval delimited by two consecutive releases/deadlines in the periodic task schedule. The same algorithm is reused in every interval. This organization inherently defines two kinds of preemptions:

- internal preemptions, generated by the TL-plane algorithm. The same number of preemptions is incurred in every TL-plane, since the schedule follows exactly the same pattern, scaled by the length of the TL-plane;
- transition preemptions, generated by the initialization of the next TL-plane through the resorting of (local) jobs according to their (local) execution times. TL-plane algorithms do not handle these preemptions per se. PCG and PCG* yield closely related numbers of transition preemptions, since they share the same initialization phase (i.e., sorting jobs in non-increasing order of their execution times).

The total number of preemptions generated by a TL-plane algorithm is repeated in every TL-plane for all deterministic TL-plane scheduling algorithms (i.e., as stated above, they always define the same scheduling pattern in every TL-plane). Hence, the total number of preemptions generated when scheduling periodic tasks over the hyperperiod is directly linked to the number of preemptions within a single TL-plane (internal + transition), multiplied by the total number of TL-planes within the hyperperiod (i.e., the number of distinct absolute deadlines within the hyperperiod H minus 1). Since the scheduling decisions of PCG and PCG* do not handle transition preemptions, taking them into account is not relevant for establishing a preemption bound suitable for comparing scheduling algorithms according to this criterion. Consequently, we focus only on internal preemptions within a single TL-plane, which further enables comparison with the best-known worst-case preemption bound (i.e., $2(m-1)$ [12]).

We now show that PCG* guarantees at most $2(m-1)$ preemptions per TL-plane. Preemption analysis at the TL-plane level is our primary focus, as it captures the algorithm's intrinsic behaviour; preemptions over a hyperperiod can then be obtained by aggregating across TL-planes and by accounting for those incurred during TL-plane switching. Hereafter, we only focus on preemptions that occur within a single TL-plane. Furthermore, this allows us to compare PCG* with existing preemption bounds of offline algorithms from job scheduling theory.

As described in Section 5.1, PCG* triggers preemptions only on C-events, never on B-events. Hence, we focus solely on the effect of C-events. Since each C-event removes one processor, there can be at most m such events. The next lemma shows that PCG* generates at most two preemptions per C_k-event.

▶ **Lemma 5.** *PCG* generates at most 2 preemptions per C_k-event in any feasible schedule.*

Proof. By application of PCG*, a job that triggers a C_k-event incurs one preemption when reassigned to π_k, and the preempted job may incur another when resumed on a different processor. Hence, each C-event causes at most two preemptions. ◀

Lemma 5 guarantees at most $2(m-1)$ preemptions if at most $m-1$ C-events are triggered in a feasible schedule. Hence, proving the result requires studying the case of schedules with exactly m C-events. In that case, a key observation is that, in any execution of PCG*, at least one C-event generates only one preemption when triggered simultaneously with another event, either a B-event or a C-event. While we demonstrate this through an example hereafter, this property is formally proven in Theorem 6. For instance, in Figure 4, at time 2, three C-events occur simultaneously, each generating only one preemption. Specifically, Job 1 triggers a C_2-event and is assigned to π_2, freeing π_1. At the same time, Job 2 triggers a C_3-event and is assigned to π_3, preempting Job 3. Finally, the free Job 3 is then reassigned to the free processor π_1. Thus, each of these C-events generates only one preemption. Now consider time 2.8 in the same figure, where a C-event coincides with a B-event. When Job 3 completes, the ready Job 4, which triggers the C-event, is immediately resumed on the processor released by the B-event. This results in zero preemptions, since Job 4 was not running previously.

With the previous lemma, we can now establish the preemption bound of PCG*.

▶ **Theorem 6.** *PCG* produces at most $2(m-1)$ preemptions in a feasible schedule on a uniform platform.*

Proof. Note that PCG* removes exactly one processor at each C-event; hence, there are at most m C-events in any TL-plane of a feasible schedule. Let q be the number of C-events in a TL-plane. Since PCG* generates preemptions only at C-events, and since each C-event generates at most two preemptions by Lemma 5, the theorem immediately follows if $q \leq m-1$. In the following, we therefore assume that $q = m$.

Let t be the last scheduling instant at which at least one C-event occurs, and let r be the number of C-events occurring at time t. Before time t, exactly $(m-r)$ C-events have been triggered, and therefore they generate at most $2(m-r)$ preemptions by Lemma 5. Furthermore, these $m-r$ processors have been removed by PCG*, and exactly r processors remain to be scheduled at time t.

At time t, all remaining processors are removed by the simultaneous r C-events. Hence, any job that is running immediately before t and does not itself trigger one of these C-events must complete at t, thereby triggering a B-event. Otherwise, it would have remaining work but no processor left on which to execute, contradicting feasibility.

If $r = 1$, the unique job triggering the last C-event is necessarily waiting, and the event coincides with a B-event on the last remaining processor; hence no preemption is generated at time t. The total number of preemptions is therefore at most $2(m-1)$. Assume now that $r \geq 2$. At time t, only jobs that trigger one of the r C-events can be preempted and were previously running on another processor. Since each job can be preempted at most once at a given instant, these r C-events generate at most r preemptions. Therefore, the total number of preemptions is at most $2(m-r)+r$. Since $r \geq 2$, we have $r \leq 2(r-1)$, and thus $2(m-r)+r \leq 2(m-r)+2(r-1) = 2(m-1)$. ◀

To illustrate that this bound is tight, consider the system from [12] with $n = k$ critical jobs and $m = k$ processors: all jobs have processing requirements $p = k-(k-1)/k$, with processor speeds $s_i = k$ for $i < k$ and $s_k = 1$. In this setting, Horváth's condition is maximized by the last term, yielding an optimal makespan of 1. As observed in [12], there is no idle time and all m jobs must run on π_m in any makespan-optimal schedule to ensure simultaneous completion at time 1. Hence, two jobs are preempted once (the one initially running on the slowest processor and the one completing on it), while the remaining jobs are preempted twice, thereby illustrating that the bound is tight. Figure 5 shows the schedule produced by PCG* for $k = 5$, which attains exactly $2(k-1) = 8$ preemptions. The first three C-events on faster processors each yield two preemptions, while the last one, occurring simultaneously with the C-event on the slowest processor, yields a single preemption for each.

6 PCG* for identical platforms

Obviously, identical multiprocessors are a particular case of uniform processor platforms in which all processor speeds are identical. As a consequence, any optimal algorithm for the uniform case is also optimal for identical processor platforms.

Horváth's feasibility condition is a proper generalization of McNaughton's condition. We formally exhibit that point in the next result.

▶ **Theorem 7.** *For unit-speed processor platforms, Horváth's condition reduces to McNaughton's condition:*

$$C_{\max} = \max\left(\max_{1\leq j\leq n} p_j, \frac{1}{m}\sum_{j=1}^{n} p_j\right)$$

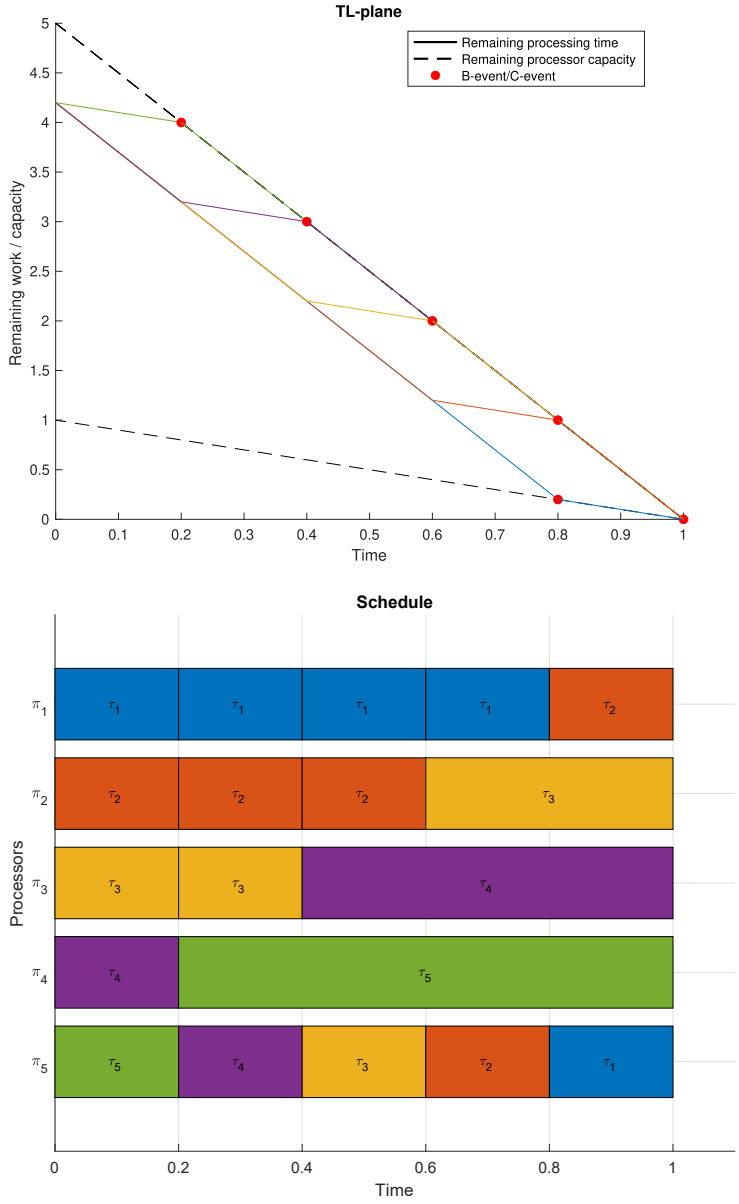


Figure 5 PCG* applied on the example from [12].

Proof. Horváth's condition assumes $p_1 \geq p_2 \geq \cdots \geq p_n$ and $s_1 \geq s_2 \geq \cdots \geq s_m$. We show that, in the case of unit-speed multiprocessors, the two terms in Horváth's formula reduce to the two terms in McNaughton's formula. Consider the first term $\frac{P_k}{S_k}$. Since we consider unit-speed processors: $S_k = k$. $P_k = \sum_{j=1}^{k} p_j$. Since $p_1 \geq p_2 \geq \cdots \geq p_n$:

$$p_1 \geq \frac{p_1 + p_2}{2} \geq \cdots \geq \frac{p_1 + p_2 + \cdots + p_{m-1}}{m-1}$$

$p_1 = \max_{i \leq j \leq n} p_j$. Hence, for any k, $1 \leq k \leq m-1$,

$$\max_{1 \leq k \leq m-1} \frac{P_k}{S_k} = \max_{i \leq j \leq n} p_j$$

The first term in Horváth's condition is equivalent to the first term of McNaughton's condition. Consider now the second term of Horváth's condition with unit-speed processors:

$$\frac{P_n}{S_m} = \frac{1}{m} P_n = \frac{1}{m} \sum_{j=1}^{n} p_j$$

This completes the proof. ◀

We prove hereafter that PCG* generates at most $m-1$ preemptions and reaches the lower bound of [17]. A key observation for identical processors is that C-events are triggered only by non-running jobs. We first prove a technical lemma.

▶ **Lemma 8.** *For identical multiprocessor systems, PCG* ensures that every C-event generates at most one preemption.*

Proof. Since all processors have the same speed, a C-event can only be triggered by the longest job that is not currently running. If there are several jobs with the longest remaining execution times, the following principle is applied several times. Two cases need to be considered.

Case (1): the C-event is not triggered simultaneously with a B-event. In this case, PCG* preempts one running job, which may later be resumed on a (possibly different) processor. Thus, the C-event generates at most one preemption.

Case (2): the C-event is triggered simultaneously with a B-event. In this case, the processor that becomes free at the B-event is allocated by PCG* to the job that triggered the C-event. Hence, no preemption occurs. ◀

PCG* removes one processor each time a C-event is triggered. The following lemma focuses on the last remaining processor.

▶ **Lemma 9.** *Consider a feasible identical multiprocessor system: no preemption can be triggered on the last remaining processor considered by PCG*.*

Proof. For the last processor of a feasible system, two situations can arise:
(1) no C-event occurs;
(2) a C-event occurs and it necessarily coincides with a B-event.

Case (1): If there is no C-event, then no preemption is ever triggered on the last processor.

Case (2): Assume that the last processor experiences a C-event. This means that the job triggering the C-event has no remaining laxity to complete before the common deadline of all jobs in the TL-plane. By definition, the job that triggers a C-event is a ready job waiting to be executed because no processor is available. Since PCG* removes one processor at each C-event, the last C-event arises when all processors but one have already been removed. At that time, the last processor is executing another job.
Suppose, by contradiction, that the job running on the last processor is not completed at the time of the C-event (i.e., it does not trigger a B-event). Then there is no free processor on which to execute the job that has zero laxity, and therefore the schedule cannot be completed by the deadline. This contradicts the optimality of PCG*. Hence, the C-event on the last processor must occur simultaneously with a B-event, which means that no preemption is required. Thus, no preemption can be triggered on the last processor. ◀

▶ **Theorem 10.** *PCG* defines a schedule with at most $(m-1)$ preemptions when applied to a feasible identical multiprocessor system.*

Proof. For m identical processors, there are at most m C-events. Lemma 8 proves that each C-event generates at most one preemption. By Lemma 9, the last processor considered by PCG* never experiences a preemption. Hence, there are at most $(m-1)$ preemptions in the schedule. ◀

As previously indicated, LRE-TL achieves the $(m-1)$ preemption bound [9], which is the best possible for any polynomial-time algorithm [23]. It requires $\mathcal{O}(n \log n)$ execution time to handle events within a TL-plane [9]. The implementation relies on two heaps: one for running tasks and another for waiting tasks. The next event occurrence is computed in $\mathcal{O}(1)$ time, since only the longest job is considered to determine the next C-event among the tasks waiting for a processor, and only the shortest task among the running tasks is used to determine the next B-event.

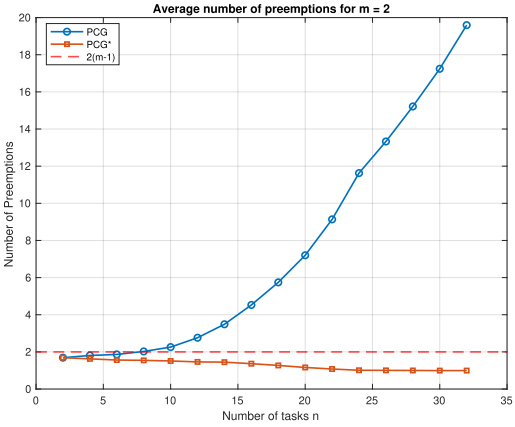
These principles do not apply when processors have different speeds, and all jobs must be considered to determine the next event. Hence, the next event can be computed in $\mathcal{O}(nm)$ time. Since there are at most n events within a TL-plane, PCG* requires $\mathcal{O}(n^2 m)$ operations to compute the schedule of a TL-plane.

7 Experiments

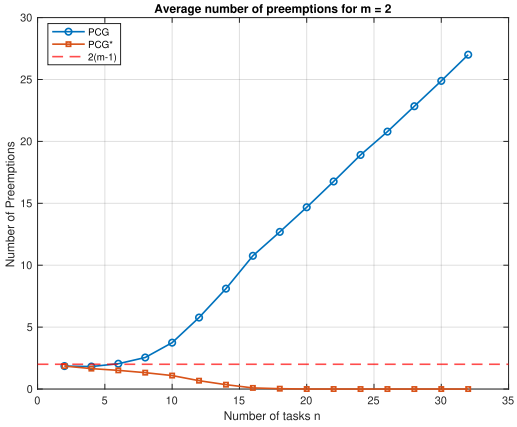
We present an experimental comparison of PCG and PCG* on synthetic job sets. Preemptions related to TL-plane switching are not taken into account. Those preemptions do not depend on the scheduling algorithm but on the DP-fair decomposition into independent scheduling subproblems. Hence, to enable a clear comparison on preemptions generated by PCG and PCG*, we only focus on preemption within a TL-plane, excluding those generated by TL-plane switching.

We generate synthetic job sets by randomly creating both the job processing requirements and the processor speeds as integers. Each job is characterized by a processing requirement drawn uniformly at random from the interval $[1, 100]$. All jobs are released at time 0, corresponding to the beginning of a TL-plane. The platform is modeled as a set of uniform processors: for each processor, we set speeds independently and uniformly from the interval $[1, 10]$. For each problem, job processing requirements and processor speeds are sorted in non-increasing order for computing the makespan using Horváth's formula (i.e., Equation 3). For a given number of jobs n and processors m, we generate 1000 synthetic systems, leading to robust statistical results. In Figure 6, the plots on the left show the results for TL-planes whose length is equal to the makespan, while the plots on the right correspond to the same problem sizes but with a TL-plane of length $1.5\times$ the makespan. Even when the TL-plane length equals the makespan, if the maximum in Horváth's formula is achieved by one of the first $m-1$ terms, some idle times are inserted into the schedule. We consider three problem sizes with $m \in \{2, 8, 16\}$. As depicted in the plots, the number of preemptions generated by PCG increases as the number of jobs grows. This fact is logical since as the number of jobs increases, more and more B-events occur. On each event, PCG sorts jobs according to LRPT-FM, resulting in numerous preemptions. The same phenomenon is amplified whenever the length of the TL-plane increases in comparison with the optimal makespan (i.e., plots on the right in Figure 6). Hence, lower task utilization can lead to more preemptions using PCG. This is not the case for PCG* since B-events do not generate any preemption. Using PCG*, if the number of jobs increases, then more B-events are experienced by the jobs and consequently the number of C-events decreases, generating fewer preemptions. That is why fewer preemptions are observed using PCG* when the TL-plane is longer in comparison with the optimal makespan that is the smallest possible TL-plane length in a feasible schedule. This is important when using TL-plane-based scheduling for real-time systems as in most cases, the TL-planes lengths obtained by the DP-fair decomposition are longer than the minimal makespan.

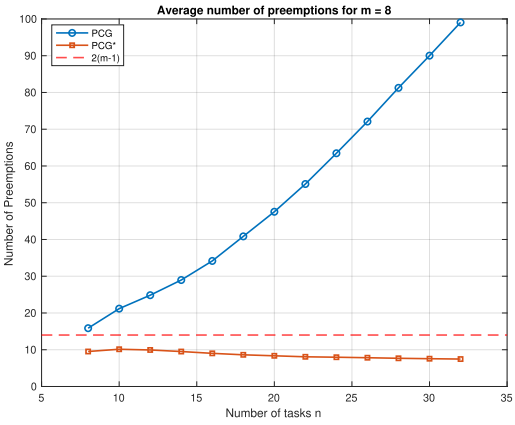
Overall, across all tested configurations, PCG* consistently yields fewer preemptions than PCG, and the gap widens as the TL-plane length increases relative to the makespan.



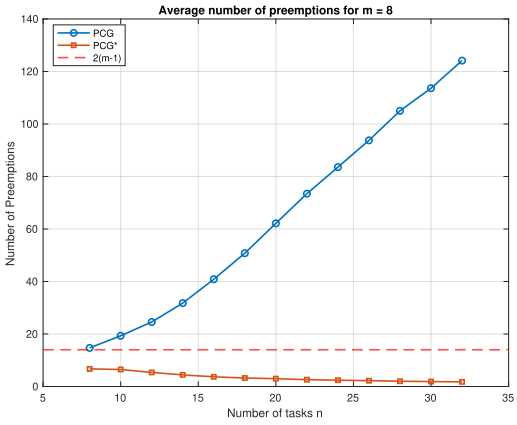
(a) $m = 2$ TL-plane length is the makespan.



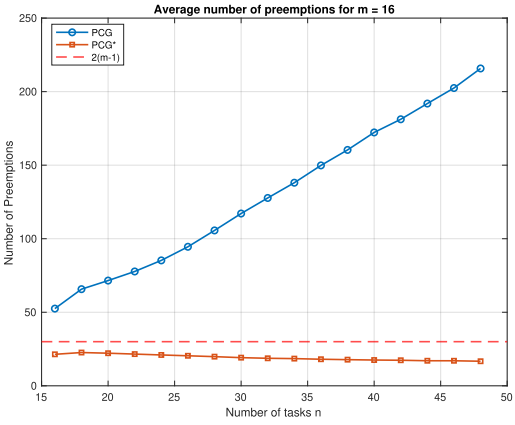
(b) $m = 2$ TL-plane length is the makespan$\times 1.5$.



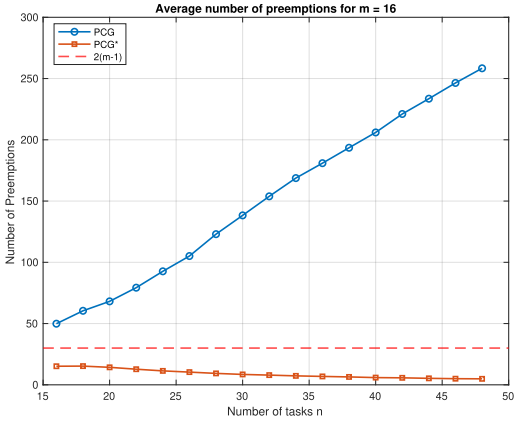
(c) m=8 TL-plane length is the makespan.



(d) $m = 8$ TL-plane length is the makespan$\times 1.5$.



(e) $m = 16$ TL-plane length is the makespan.



(f) $m = 16$ TL-plane length is the makespan$\times 1.5$.

Figure 6 Experimentation results.

8 Conclusion

We have presented PCG*, a variant of the Precaution Cut Greedy (PCG) algorithm [4], which is optimal for scheduling periodic tasks on uniform multiprocessor platforms. Our algorithm guarantees at most $2(m-1)$ preemptions per TL-plane on uniform platforms, which constitutes the tightest preemption bound achievable by any polynomial-time algorithm. This is the first real-time scheduling algorithm that guarantees this tight preemption bound for uniform platforms. Both algorithms have similar computational complexity. The numerical experiments presented in this work provide strong evidence that PCG* yields a substantial reduction in the number of preemptions relative to PCG. PCG* also achieves McNaughton's tighter bound of $m-1$ preemptions [17] when applied to identical platforms. This bound was previously reached by the LRE-TL algorithm [9], which is specifically designed for identical platforms.

In future work, we plan to extend the proposed approach to sporadic task systems, which would allow the scheduler to support genuinely event-driven workloads in which job releases are not strictly periodic. We also intend to develop a work-conserving variant, in order to better accommodate execution-time variability and to avoid leaving processors idle when runnable work is available. Such a variant would additionally facilitate a tighter integration of aperiodic-task servers, thereby improving responsiveness to aperiodic demand while preserving the key structural properties of the TL-plane abstraction.

References

1 Matteo Bambagini, Marko Bertogna, Mauro Marinoni, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems*, 15(1):7:1–7:34, 2016. doi:10.1145/2808231.

2 Sanjoy Baruah. Scheduling periodic tasks on uniform multiprocessors. *Information Processing Letters*, 80(2):97–104, 2001. doi:10.1016/S0020-0190(01)00148-X.

3 Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. doi:10.1007/BF01940883.

4 Shih-Ying Chen and Chih-Wen Hsueh. Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. In *2008 Real-Time Systems Symposium*, pages 147–156, 2008. doi:10.1109/RTSS.2008.35.

5 Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 101–110. IEEE, 2006. doi:10.1109/RTSS.2006.10.

6 Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011. doi:10.1145/1978802.1978814.

7 Tomáš Ebenlendr and Jiří Sgall. Optimal and online preemptive scheduling on uniformly related machines. *J. of Scheduling*, 12(5):517–527, October 2009. doi:10.1007/s10951-009-0119-7.

8 S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 183–192, 2001. doi:10.1109/REAL.2001.990609.

9 Shelby Funk. LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst.*, 46(3):332–359, December 2010. doi:10.1007/s11241-010-9109-2.

10 Shelby H Funk and Archana Meka. U-LLREF: An optimal scheduling algorithm for uniform multiprocessors. In *The 9th Workshop on Models and Algorithms for Planning and Scheduling Problems*, page 262, 2009.

11 Thomas Gaspard, Antoine Bertout, Pascal Richard, Joël Goossens, and Emmanuel Grolleau. An unfair optimal scheduling algorithm for uniform multiprocessors. In *2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2025. doi:10.1109/ETFA65518.2025.11205723.

12 Teofilo Gonzalez and Sartaj Sahni. Preemptive scheduling of uniform processor systems. *J. ACM*, 25(1):92–101, January 1978. doi:10.1145/322047.322055.

13 Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *J. ACM*, 24(1):32–43, 1977. doi:10.1145/321992.321995.

14 Leonard Kleinrock. *Queueing Systems, Volume II: Computer Applications*. Wiley-Interscience, New York, 1976.

15 Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 3–13, 2010. doi:10.1109/ECRTS.2010.34.

16 Charles U. Martel. A parallel algorithm for preemptive scheduling of uniform machines. *Journal of Parallel and Distributed Computing*, 5(6):700–715, 1988. doi:10.1016/0743-7315(88)90037-8.

17 Robert McNaughton. Scheduling with deadlines and loss functions. *Manage. Sci.*, 6(1):1–12, October 1959. doi:10.1287/mnsc.6.1.1.

18 Guillaume Phavorin, Pascal Richard, Joël Goossens, Claire Maiza, Laurent George, and Thomas Chapeaux. Online and offline scheduling with cache-related preemption delays. *Real-Time Systems*, 54(3):662–699, 2018. doi:10.1007/s11241-017-9275-6.

19 Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, pages 89–102. ACM, 2001. doi:10.1145/502034.502044.

20 Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 6th edition, 2022. doi:10.1007/978-3-031-05921-6.

21 Gurulingesh Raravi, Björn Andersson, and Konstantinos Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. *Real-Time Systems*, 49(1):29–72, 2013. doi:10.1007/s11241-012-9161-1.

22 Hadas Shachnai, Tami Tamir, and Gerhard J Woeginger. Minimizing makespan and preemption costs on a system of uniform machines. *Algorithmica*, 42:309–334, 2005. doi:10.1007/s00453-005-1171-0.

23 Evgeny V Shchepin and Nodari Vakhania. On the geometry, preemptions and complexity of multiprocessor and shop scheduling. *Annals of Operations Research*, 159(1):183–213, 2008. doi:10.1007/s10479-007-0266-1.

24 Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA, November 1994. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1267640.

25 Kecheng Yang and James H. Anderson. An optimal semi-partitioned scheduler for uniform heterogeneous multiprocessors. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 199–210, 2015. doi:10.1109/ECRTS.2015.25.