An optimal boundary fair scheduling

Geoffrey Nelissen · Hang Su · Yifeng Guo · Dakai Zhu ·
Vincent Nélis · Joël Goossens

Abstract Nowadays, many real-time operating systems discretize the time relying on a system time unit. To take this behavior into account, real-time scheduling algorithms must adopt a *discrete-time* model in which both timing requirements of tasks and their time allocations have to be integer multiples of the system time unit. That is, tasks cannot be executed for less than one time unit, which implies that they always have to achieve a minimum amount of work before they can be preempted. Assuming such a discrete-time model, the authors of Zhu et al. (Proceedings of the 24th IEEE inter- national real-time systems symposium (RTSS 2003), 2003, J Parallel Distrib Comput 71(10):1411–1425, 2011) proposed an efficient "boundary fair" algorithm (named BF) and proved its optimality for the scheduling of periodic tasks while achieving full system utilization. However, BF cannot handle *sporadic tasks* due to their inherent irregular and unpredictable job release patterns. In this paper, we propose an *optimal* boundary-fair scheduling algorithm for *sporadic* tasks (named BF^2), which follows the same principle as BF by making scheduling decisions only at the job arrival times and (expected) task deadlines. This new algorithm was implemented in Linux and we show through experiments conducted upon a multicore machine that BF^2 outperforms the state-of-the-art discrete-time optimal scheduler (PD^2), benefiting from much less scheduling overheads. Furthermore, it appears from these experimental results that BF^2 is barely dependent on the length of the system time unit while PD^2—the only other existing solution for the scheduling of sporadic tasks in discrete-time systems— sees its number of preemptions, migrations and the time spent to take scheduling decisions increasing linearly when improving the time resolution of the system.

Keywords Real-time · Scheduling · Optimal · Discrete-time · Multiprocessor · Fairness

1 Introduction

With the emergence of multicore/multiprocessor platforms in embedded devices, there is a reviving interest in scheduling algorithms for multiprocessor real-time systems. Unlike the uniprocessor scheduling theory which has been comprehensively studied, the scheduling of real-time tasks on multiprocessor is still an evolving research field and many problems remain open due to their intrinsic difficulties.

A problem that quickly became one of the major concerns in multiprocessor scheduling theory is the question of *“optimality”* (i.e., the ability of a particular scheduling algorithm to meet all the task deadlines for any feasible task set). Many optimal scheduling algorithms for multiprocessor platforms have been designed over the years (Zhu et al. 2011; Baruah et al. 1995, 1996; Srinivasan and Anderson 2002; Andersson and Tovar 2006; Funk 2010; Regnier et al. 2011; Nelissen et al. 2012). Most of them base their scheduling decisions on a continuous-time model. That is, a task can be scheduled for any amount of time, thereby authorizing arbitrarily short task executions. However, this model does not comply with many today’s real-time oper- ating systems which take all their scheduling decisions by relying on a system time unit (Wind River Systems, Inc. 2011; Krten and QNX Software Systems 2012). For examples, it is said in the documentations of the real-time operating systems RTEMS (2012), Lynux Works (2005), VxWorks (Wind River Systems, Inc. 2011) and QNX Neutrino (Krten and QNX Software Systems 2012; QNX Software Systems 2012) that delays imposed on tasks (i.e., delays defined in any function of the type *delay*() or *sleep*()) and timer initialization values must be defined as an integer multiple of the system time unit. As a result, one cannot delay a higher priority task for less than one system time unit nor program the end of the execution of a task with the help of a timer or any other kind of event before at least one system time unit. Therefore, it is quite unrealistic to schedule the execution of a task for less than one time unit as it is nonetheless the case with continuous-time algorithms (Andersson and Tovar 2006; Funk 2010; Regnier et al. 2011; Nelissen et al. 2012). One should however be careful while reading the documentation of these RTOS since the names of some functions may be misleading. For instance, even though QNX Neutrino provides a function named **nanosleep()** which is supposed to suspend the execution of the task during a time specified in nanoseconds, it is said on page 1553 in (QNX Software Systems Limited 2012) that "*The suspension time may be longer than requested because the argument value is rounded up to be a multiple of the system timer resolution*". Simi- larly, for VxWorks, it is said on Section 9.9 of Wind River Systems, Inc. (2011) that *"The POSIX nanosleep() routine provides specification of sleep or delay time in units of seconds and nanoseconds, in contrast to the ticks used by the VxWorks taskDe- lay() function. Nevertheless, the precision of both is the same, and is determined by the system clock rate; only the units differ"*. These functions have been implemented for compliancy reasons with the POSIX real-time standard (IEEE 2003) but do not actually provide a better resolution than the software timers based on the system time unit. Of course, it does not mean that a task will never execute for less than one time unit on the processing platform or that it will always run for natural multiples of the system time unit. A task can always need less time than initially expected to finish its execution. However, operating systems such as those previously cited do not allow the tasks to be *scheduled* for something different than natural multiples of the system time unit. They only provide mechanisms to adapt our scheduling decisions whenever an event such as the completion of a task occurs. This event is however completely independent of our will.

A solution to this problem consists in building the scheduling algorithm on a discrete-time model. In that case, both timing requirements of tasks and their time allocations have to be integer multiples of the system time unit.

The first optimal multiprocessor scheduling algorithm for periodic real-time tasks with *discrete* timing requirements was proposed in Baruah et al. (1993, 1996). This algorithm named PF is based on the notion of *proportionate fairness* (PFairness). The core idea of the Pfairness is to enforce proportional progress for all tasks by ensuring that the deviation from an ideal *fluid schedule* (see Definition 1 presented in Sect. 4 for a formal definition) *never* exceeds one system time unit. Several proportionate fair (PFair) algorithms have been proposed over the years (Baruah et al. 1993, 1995, 1996; Anderson and Srinivasan 2000a; Srinivasan and Anderson 2002). However, by making scheduling decisions at every time unit, PFair schedulers can incur high scheduling overheads as they generally produce an excessive amount of task preemptions and migrations.

Observing the fact that a periodic real-time task with implicit deadline can only miss its deadline at its period boundary (because the deadline of a task also corresponds to the end of its period when we consider periodic tasks with implicit deadlines), an optimal discrete-time based *boundary fair* scheduling algorithm (named BF) had previously been studied in Zhu et al. (2003, 2011). BF makes scheduling decisions *only* when a task reaches the end of its period (which corresponds also to its current deadline and the release of its next job). Specifically, at every such event henceforth called *boundary*, BF takes a scheduling decision for the whole time interval extending from the current boundary to the next one (the earliest next task deadline). Similar to PFair schedulers, BF ensures *fairness* for tasks at the period boundaries to avoid deadline misses. That is, at each period boundary, the deviation of any task from the theoretical fluid schedule is less than one time unit. It has been shown that BF can achieve full system utilization while guaranteeing all tasks to meet their deadlines (Zhu et al. 2003, 2011). Moreover, compared to PFair schedulers, BF substantially reduces the number of preemptions, migrations and scheduling points (Zhu et al. 2003, 2011). However, BF assumes that all the boundary instants (i.e., the deadlines and arrivals of jobs) are known *beforehand* and thus cannot handle sporadic tasks due to their irregular and unpredictable arrival patterns.

More recent works aimed at reducing the number of task preemptions and migra- tions for periodic task systems (Andersson and Tovar 2006; Funk 2010, 2011; Regnier et al. 2011; Nelissen et al. 2012). Most of these algorithms are also optimal for *spo- radic* tasks (Andersson and Bletsas 2008; Funk 2010, 2011; Nelissen et al. 2012). However, in spite of their ability to handle sporadic tasks, they adhere to a continuous- time model whose drawbacks have already been discussed earlier in the introduction. Hence, algorithms such as EKG (Andersson and Bletsas 2008), NPS-F (Bletsas and Andersson 2009; 2011), LRE-TL (Funk 2010), DP-Wrap (Levin et al. 2010; Funk et al. 2011) or RUN (Regnier et al. 2011) are *not* directly comparable to the discrete-time solutions such as PF (Baruah et al. 1996), PD^2 (Srinivasan and Anderson 2002) or BF (Zhu et al. 2003, 2011). An example motivating this statement will be provided in Sect. 3.

1.1 Contribution of this work

In this paper, we focus on *discrete-time* based systems and propose an optimal mul- tiprocessor *boundary-fair* scheduling algorithm for *sporadic* tasks named BF^2. BF^2 extends the principles and ideas of BF. Specifically, BF^2 makes scheduling decisions only at the arrival time and (expected) deadlines of tasks. However, unlike periodic tasks for which all boundaries are known at system design-time and coincide with the deadlines of periodic tasks, the irregular and unpredictable job arrival pattern of spo- radic tasks open non-trivial challenges. The arrival time of a sporadic task may indeed not coincide with a task deadline and such timing disparities entail an unexpected complexity for the scheduler.

As the main contribution of this work, we present BF^2 and prove its optimality for the scheduling of sporadic tasks with implicit deadlines.

As presented in Sect. 10, we implemented both BF^2 and PD^2 in Linux and run several experiments upon a six core machine. The obtained results show that BF^2 can substantially reduce the scheduling overheads such as the number of task preemptions and migrations and the time spent to take scheduling decisions when comparing to PD^2 (i.e., the only alternative for the scheduling of sporadic tasks in discrete-time systems). Furthermore, while the overheads caused by BF^2 are barely dependent on the length of the system time unit, PD^2 sees its overheads growing linearly with the time resolution adopted by the system.

1.2 Organization of this paper

System models are presented in Sect. 2 while Sect. 3 motivates the work explaining why a discrete-time schedule cannot simply be derived from a continuous-time scheduling solution. Sect. 4 reviews many fair schedulers for discrete-time systems. The basic steps of BF^2 are presented in Sect. 5 and Sect. 6 addresses the particularities inherent to the scheduling of sporadic tasks. The optimality of BF^2 is analyzed in Sects. 7 and 8. Implementation considerations and improvement techniques are discussed in Sect. 9. Finally, Sect. 10 presents our experimental results and Sect. 11 concludes the paper.

2 System model

We address the problem of scheduling a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of n independent sporadic tasks with implicit deadlines on a platform composed of m identical processors. Each task $\tau_i \overset{\text{def}}{=} \langle C_i, D_i, T_i \rangle$ is characterized by a worst-case computation requirement C_i, a relative deadline D_i, and a *minimum* inter-arrival time T_i. Hence, a task τ_i releases a (potentially infinite) sequence of jobs. Each job $J_{i,q}$ of τ_i that arrives at time $a_{i,q}$ must execute for at most C_i time units before its deadline occurring at time $a_{i,q} + D_i$ and the earliest possible arrival time of the next job of τ_i is at time $a_{i,q} + T_i$.

Since we are considering a discrete-time model, C_i, D_i and T_i are assumed to be natural multiples of the system time unit. The utilization of a task τ_i is defined as $U_i \overset{\text{def}}{=} \frac{C_i}{T_i}$. Informally, the utilization of a task represents the percentage of time the task may use a processor by releasing one job every T_i time units and executing each such job for C_i time units. The system utilization U is the sum of all task utilizations (i.e., $U \overset{\text{def}}{=} \sum_{i=1}^{n} U_i$). It gives the minimum computational capacity that must be provided by the platform to meet all the task deadlines.

We say that a job is active at time t if it has been released no later than t and has its deadline after t, i.e., the instant t lies between the arrival time of the job and its deadline. If a task τ_i has an active job at time t then we say that τ_i is active and we define $a_i(t)$ and $d_i(t)$ as the arrival time and absolute deadline of the currently active job of τ_i at time t. Since we consider tasks with implicit deadlines (i.e., $D_i = T_i$), at most one job of each task can be active at any time t. Therefore, without causing any ambiguity, we use the terms “tasks” and “jobs” interchangeably in the remainder of this paper.

The (worst-case) remaining execution time of an active job of a task τ_i at time t is denoted by $\text{ret}_i(t)$. It represents an upper-bound on the amount of time the active job of τ_i must still execute before its deadline $d_i(t)$.

3 Motivational example

In this section, we explain why creating a discrete-time scheduling algorithm is not straight-forward and cannot simply derive from a minor modification of existing scheduling algorithms for continuous-time systems.

Let for instance take the example of a real-time operating system such as VxWorks that uses a system time unit of 1 ms and let us assume that this operating system uses the algorithm RUN for the scheduling of the tasks. RUN is optimal for the scheduling of periodic tasks with implicit deadlines (Regnier et al. 2011). It makes use of a mechanism which consists in packing the tasks in what is called a *server*. A server inherits the deadlines of its component tasks and releases a *job* at each such

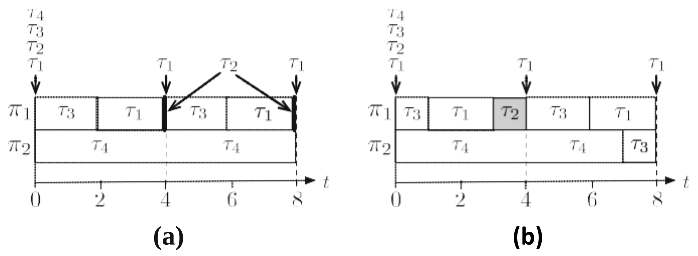


Fig. 1 Difference between a continuous (**a**) and a discrete-time (**b**) schedule

deadline. Let S_1 be one of these servers. It is composed of two tasks $\tau_1 = \langle 2, 4, 4\rangle$ and $\tau_2 = \langle 1, 50, 50\rangle$. S_1 has a utilization denoted $U(S_1) \overset{\text{def}}{=} \sum_{\tau_i \in S_1} U_i = 0.52$ and its first deadline is at time 4 (i.e., the first deadline of τ_1). S_1 releases a first job at time 0 with a deadline 4 and an execution time $U(S_1) \times 4 = 2.08$ (see Fig. 1a). That is, RUN would like to execute the server S_1 and hence its component tasks for 2.08 ms before instant 4. However, this is impossible since the system time unit imposed by the RTOS is of 1 ms implying that all tasks should be executed for an integral number of milliseconds. Worse, because the deadline of τ_1 is 4, RUN will execute τ_1 for 2 ms (i.e., for its worst-case execution time) between 0 and 4 and τ_2 for 0.08 ms in the same interval (as shown on Fig. 1a). This execution time computed for τ_2 might be of the same order of magnitude than the overheads caused by the execution of the scheduling algorithm, the preemptions and migrations. The performances of the application might therefore be severely impacted by such small execution times.

To comply with the discrete-time model imposed by the RTOS, we must round up or down the execution time allocated to τ_2. That is, either we execute τ_2 for 1 ms between instant 0 and 4, or we do not execute τ_2 at all. Both decisions have a strong impact on the system. Let us consider a system composed of two processors and constituted of two more tasks $\tau_3 = \langle 18, 25, 25\rangle$ and $\tau_4 = \langle 19, 25, 25\rangle$. The beginning of the schedule such as produced by RUN is presented on Fig. 1a. If we decide to round up the execution time of τ_2 in $[0, 4)$, we increase the resource demand of τ_2 in that interval. Hence, we must reduce the execution time of another task (say τ_3) such as illustrated on Fig. 1b. Consequently, in order to respect its deadline, τ_3 will need more time to execute after 4 than initially expected. A decision could be to increase the execution time of τ_3 in $[4, 8)$ and decrease the execution time of τ_4 (see Fig. 1b). However, this means that at time 8, τ_4 is now late on its initial schedule which was presented on Fig. 1a. τ_4 will therefore reclaim this missing execution time after time 8. The decision of rounding the execution time of τ_2 in $[0, 4)$ may therefore have a strong impact on the future scheduling decisions that should be taken after 4 and if unwisely made, it can even lead to the impossibility to respect all the future job deadlines.

In conclusion, the mechanism used to round up or down the execution time of the tasks will drastically change the behaviour of the scheduling algorithm. It might even be impossible to modify some continuous-time scheduling algorithm for the scheduling of discrete-time systems while keeping their main properties (e.g., optimality). To the best of our knowledge, no one ever successfully transformed an optimal continuous-

time scheduling algorithm for multiprocessor platforms to a discrete-time scheduling algorithm while keeping its optimality.

4 Related work: optimal schedulers for discrete-time systems

4.1 Proportionate and early-release fairness

The notion of fairness has been introduced by Baruah et al. (1996). As its name implies, the main idea is to fairly distribute the computational capacity of the platform between tasks. At any time t, each task τ_i is therefore executed on the processing platform for a time proportional to its utilization. This led to the concept of *fluid schedule* defined as follows:

Definition 1 (*Fluid schedule*) A schedule is said to be fluid if and only if at any time $t \geq 0$, the active job (if any) of every task τ_i arrived at time $a_i(t)$ has been executed for *exactly* $U_i \times (t - a_i(t))$ time units.

In discrete-time systems, tasks are always executed for an integer number of system time units. Consequently, task executions might deviate from the fluid schedule during the system lifespan. Indeed, consider a task τ_i with a utilization $U_i = 0.5$ and releasing
a job a time 0. At time $t = 3$, τ_i should have been executed for $0.5\times(3-0) = 1.5$ time units according to Definition 1. However, since τ_i can only be executed for integer multiples of the system time unit, it can only achieve 1 or 2 but certainly not 1.5 time units of execution. To measure this deviation from the fluid schedule, the *allocation error* (or *lag*) of a task is defined as follows (Baruah et al. 1996):

Definition 2 (*Allocation Error* (*lag*)) The lag of a task τ_i at time t is the difference between the amount of work $\text{exec}_i(a_i(t), t)$ executed by the active job of τ_i until time t in the actual schedule, and the amount of work that it would have executed in the fluid schedule by the same instant t. That is,

$$\text{lag}_i(t) \overset{\text{def}}{=} U_i \times (t - a_i(t)) - \text{exec}_i(a_i(t), t)$$

with $a_i(t)$ being the arrival time of the active job of τ_i.

Fair schedulers impose constraints on the lag of every task in order to bound the deviation from the fluid schedule. For instance, with a *Proportionate Fair* (PFair) scheduler the allocation errors of the tasks are always kept smaller than one system time unit (Baruah et al. 1993). That is,

Definition 3 (*Proportionate fair schedule*) A schedule is said to be proportionate fair (or PFair) if and only if

$$\forall \tau_i \in \tau,\ \ \forall t \geq 0 : \ |\ \text{lag}_i(t)\ | < 1$$

With an *Early-Release Fair* (ERFair) scheduler however, tasks can be ahead by more than one time unit, but never be late by more than one time unit on the fluid schedule (Anderson and Srinivasan 2001). Formally,

Definition 4 (*Early-Release fair schedule*) A schedule is said to be Early-Realease fair (or ERFair) if and only if

$$\forall \tau_i \in \tau, \quad \forall t \geq 0 \; : \mathrm{lag}_i(t) < 1$$

The intuition behind the PFair (or ERFair) approach can easily be understood; in discrete-time systems, each task must have been running for an integer number of time units before its deadline. Since the worst-case execution time C_i of any task τ_i is assumed to be an integer as well, if τ_i misses its deadline at time t, then it must have an integer number of remaining time units to execute. That is, there is at least a difference of one time unit between the fluid and the actual schedule, i.e., there is $\mathrm{lag}_i(t) \geq 1$. As a result, enforcing $\mathrm{lag}_i(t) < 1$ at every time t and therefore, by extension, at every task deadline, ensures that no task will ever miss its deadline.

In the remainder of this work, we will use the terms *behind*, *punctual* and *ahead* to qualify the state of a task at time t. Formally,

Definition 5 (*Task behind at time t*) A task τ_i is said to be behind at time t, if it has been executed for less time in the actual schedule than in the corresponding fluid schedule until time t. That is, $\mathrm{lag}_i(t) > 0$.

Definition 6 (*Task punctual at time t*) A task τ_i is said to be punctual at time t, if it has been executed for exactly the same amount of time in the actual schedule as in the corresponding fluid schedule until time t. That is, $\mathrm{lag}_i(t) = 0$.

Definition 7 (*Task ahead at time t*) A task τ_i is said to be ahead at time t, if it has been executed for more time in the actual schedule than in the corresponding fluid schedule until time t. That is, $\mathrm{lag}_i(t) < 0$.

A task should therefore always be punctual at each of its deadlines if the schedule respects all those deadlines.

We will now present in detail two of the most important PFair and ERFair algorithms: PF and PD^2. Indeed, we will prove in Sect. 7 that BF^2—our new boundary fair algorithm—while extending the principles and ideas of BF, is also a generalization of PD^2. A complete understanding of the PD^2 algorithm will therefore be needed to apprehend the various arguments developed in that section. Further, PD^2 has been build upon PF foundations. We therefore start by introducing PF before presenting PD^2 which can be seen as an improvement of PF.

4.1.1 *The PF algorithm*

Initially, PF took its scheduling decisions at any time t relying on *"characteristic strings"* expressing the future load request of each task in τ so as to respect the PFairness (Baruah et al. 1993, 1996). This procedure has however been simplified, first in Baruah et al. (1995) and then in Anderson and Srinivasan (1999), introducing the notion of *pseudo-deadline*. We therefore present this refined version of PF in this document.

Within a PFair scheduling, each task τ_i is divided into an infinite sequence of time units named *subtasks*. Each subtask has an execution time of one time unit and the *j*th

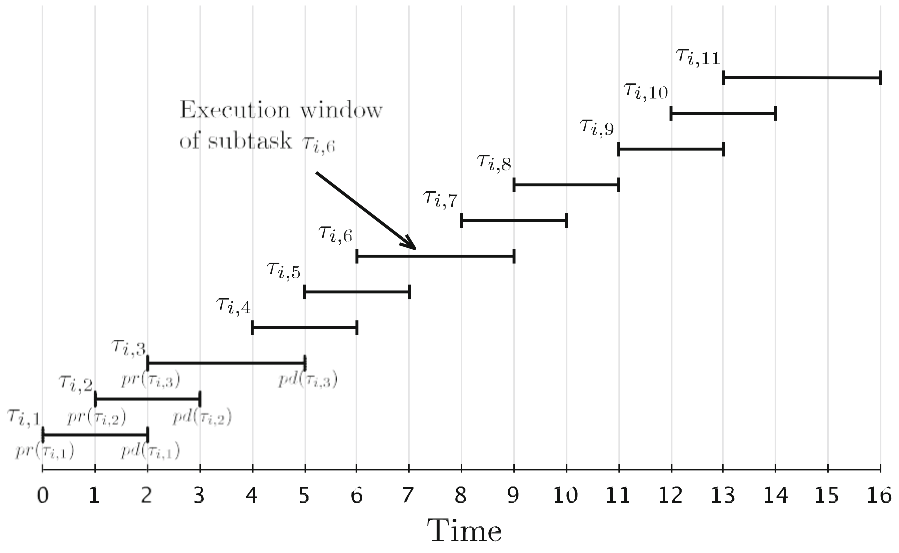


Fig. 2 Windows of the 11 first subtasks of a periodic task $\tau_i \overset{\text{def}}{=} (8, 11, 11)$

subtask of a task τ_i is denoted $\tau_{i,j}$ with $j \geq 1$. Note that a job $J_{i,q}$ is composed of C_i consecutive subtasks $\tau_{i,j}$.

To keep the lag of a task τ_i smaller than 1 and greater than −1, each subtask $\tau_{i,j}$ of job $J_{i,q}$ has to be executed in an associated *window*. This window extends from a *pseudo-release* $pr(\tau_{i,j})$ to a *pseudo-deadline* $pd(\tau_{i,j})$. In Anderson and Srinivasan (1999), it was shown for *periodic* tasks released at time 0 that[1]

$$pr(\tau_{i,j}) \overset{\text{def}}{=} \left\lfloor \frac{j-1}{U_i} \right\rfloor$$

and

$$pd(\tau_{i,j}) \overset{\text{def}}{=} \left\lceil \frac{j}{U_i} \right\rceil$$

Figure 2 shows, as an example, the repartition of the windows for a periodic task τ_i with $U_i = \frac{8}{11}$ releasing its first job at time 0.

More generally, for both *periodic* and *sporadic* tasks, the pseudo-release and pseudo-deadline of a subtask $\tau_{i,j}$ which is the *p*th subtask to execute in a particular

[1] In Anderson and Srinivasan (1999, 2000a) and Srinivasan and Anderson (2002) the pseudo-release and pseudo-deadlines were defined on a "slot" basis. In this document as in Anderson et al. (2005) and Srinivasan and Anderson (2005), the pseudo-release and pseudo-deadline refer to time-instants. This explain why the formula given here for $pd(\tau_{i,j})$ is slightly different to the one presented in Anderson and Srinivasan (1999, 2000a) and Srinivasan and Anderson (2002) but identical to those of Anderson et al. (2005) and Srinivasan and Anderson (2005).

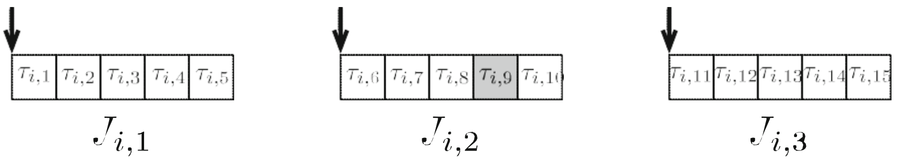


Fig. 3 Subtasks of the three first job of a task τ_i with a worst-case execution time $C_i = 5$. The *shaded* subtask $\tau_{i,9}$ is the ninth subtask of τ_i but the fourth subtask of the job $J_{i,2}$

job $J_{i,q}$ (see Fig. 3) is given by Eqs. 1 and 2, respectively.[2]

$$pr(\tau_{i,j}) = a_{i,q} + \left\lfloor \frac{p-1}{U_i} \right\rfloor \tag{1}$$

$$pd(\tau_{i,j}) = a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil \tag{2}$$

where $a_{i,q}$ is the arrival time of job $J_{i,q}$.

Note that because there are C_i consecutive subtasks in any job $J_{i,q}$, it holds that $q = \left\lceil \frac{j}{C_i} \right\rceil$ and $p = j - (q-1) \times C_i$. For example, in Fig. 3, the shaded subtask $\tau_{i,9}$ of task τ_i is the ninth subtask of τ_i but the fourth subtask of the job $J_{i,2}$. We therefore have for this particular subtask, $j = 9$, $p = 4$ and $q = 2$.

At each time t, the PF algorithm determines which subtasks are *eligible* to be scheduled. A subtask $\tau_{i,j}$ of τ_i is said to be eligible at time t under PF if it respects the following definition:

Definition 8 (*Eligible subtask under PF*) A subtask $\tau_{i,j}$ of a task τ_i is eligible to be scheduled at time t if the subtask $\tau_{i,j-1}$ has already been executed prior to t and $pr(\tau_{i,j}) \leq t < pd(\tau_{i,j})$, i.e., t lies within the execution window of $\tau_{i,j}$.

PF gives the highest priority to the active subtasks with the earliest pseudo-deadlines. If there is a tie between two subtasks with the same pseudo-deadline, an additional parameter named *successor bit* is used. Informally, the successor bit $b(\tau_{i,j})$ of a subtask $\tau_{i,j}$ is equal to 1 if and only if $\tau_{i,j}$'s window overlaps $\tau_{i,(j+1)}$'s window. $b(\tau_{i,j})$ is equal to 0 otherwise. For instance, in Fig. 2, $b(\tau_{i,2}) = 1$ while $b(\tau_{i,8}) = 0$. Using the definitions of the pseudo-deadline and pseudo-release, it was proven that

$$b(\tau_{i,j}) \overset{\text{def}}{=} \left\lceil \frac{j}{U_i} \right\rceil - \left\lfloor \frac{j}{U_i} \right\rfloor = \left\lceil \frac{p}{U_i} \right\rceil - \left\lfloor \frac{p}{U_i} \right\rfloor \tag{3}$$

Hence, PF orders the eligible subtasks at time t by their priorities using the following rules:

Prioritization Rules 1 *(Prioritization Rules of PF) With PF, a subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,\ell}$ (denoted $\tau_{i,j} \succ \tau_{k,\ell}$) iff:*

[2] It was shown in Anderson and Srinivasan (1999) that the window pattern is identical for each job of τ_i. Since the pseudo-deadlines of subtasks $\tau_{i,j}$ belonging to the first job of a task τ_i starting its execution at $t = 0$ is given by $pd(\tau_{i,j}) = \left\lceil \frac{j}{U_i} \right\rceil$ (see Anderson et al. 2005), the pseudo-deadlines of any other job $J_{i,q}$

is just translated by $a_{i,q}$ time units. The same reasoning applies to the pseudo-release.

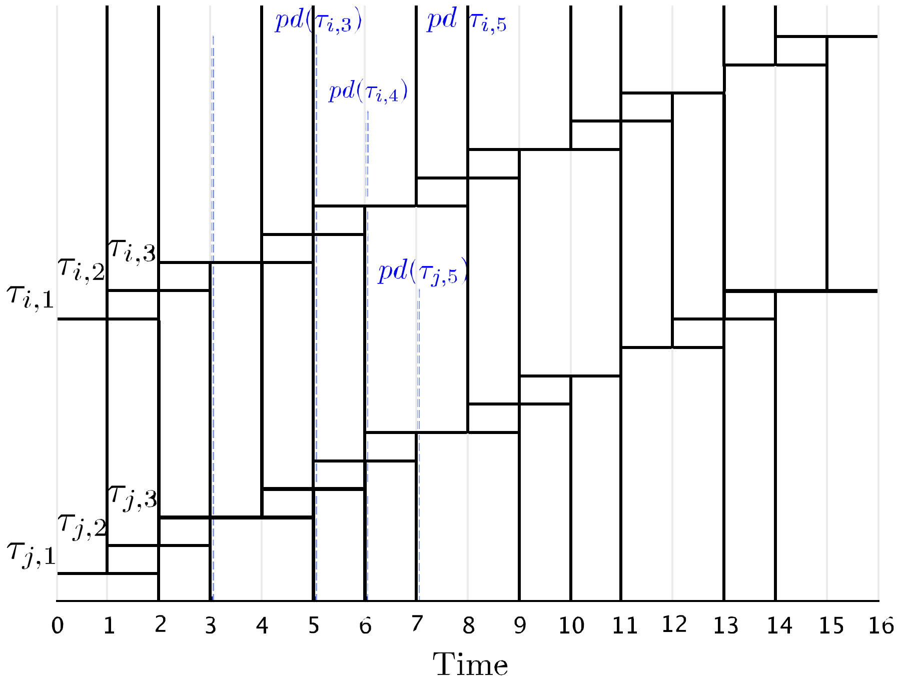


Fig. 4 Comparison of the priority of two periodic tasks $\tau_i \overset{\text{def}}{=} \langle 9, 13, 13\rangle$ and $\tau_j \overset{\text{def}}{=} \langle 8, 11, 11\rangle$ using the prioritization rules of PF

(i) $pd(\tau_{i,j}) < pd(\tau_{k,\ell})$ ***or***

(ii) $pd(\tau_{i,j}) = pd(\tau_{k,\ell}) \wedge b(\tau_{i,j}) > b(\tau_{k,\ell})$ ***or***

(iii) $pd(\tau_{i,j}) = pd(\tau_{k,\ell}) \wedge b(\tau_{i,j}) = b(\tau_{k,\ell}) = 1 \wedge \tau_{i,j+1} \succ \tau_{k,\ell+1}$

At each time t, the m eligible subtasks with the highest priorities according to Prioritization Rules 1 are chosen to be executed on the m processors of the platform. If $\tau_{i,j}$ and $\tau_{k,\ell}$ have the same priority (i.e., we neither have $\tau_{i,j} \succ \tau_{k,\ell}$ nor $\tau_{k,\ell} \succ \tau_{i,j}$) then the tie can be broken arbitrarily by the scheduler. Note that if the pseudo-deadline and successor bit are not enough to untie two subtasks $\tau_{i,j}$ and $\tau_{k,\ell}$ (i.e., they both have the same pseudo-deadline and a successor bit equal to 1), the priority of the next subtasks $\tau_{i,j+1}$ and $\tau_{k,\ell+1}$ released by the same tasks τ_i and τ_k are compared. This process is repeated iteratively until a subtask of τ_i or τ_k is eventually found to be of higher priority or both have a successor bit equal to 0. Indeed, in that case the schedule of a subtask $\tau_{i,j}$ do not interfere with the schedule of $\tau_{i,j+1}$ and comparing further subtasks does not make any sense anymore. It was proven in Anderson and Srinivasan (1999) that the recursion of this third rule of Prioritization Rules 1 always ends due to the fact that $b(\tau_{i,j}) = 0$ at least at the deadline of a job.

Example 1 Let τ_i and τ_j be two periodic tasks with implicit deadlines such that $U_i = \frac{9}{13}$ and $U_j = \frac{8}{11}$. Figure 4 presents the execution windows of the first subtasks of those two tasks. Let us compare the priority of τ_i and τ_j at time $t = 3$. As shown on

Fig. 4, at t = 3, $\tau_{i,3}$ and $\tau_{j,3}$ are the eligible subtasks of τ_i and τ_j, respectively. Using

Expressions 2 and 3, we get that $pd(\tau_{i,3}) = pd(\tau_{j,3}) = 5$ and $b(\tau_{i,3}) = b(\tau_{j,3}) = 1$(see Fig. 4). Therefore, we use the third rule of Prioritization Rules 1 and compare the priority of $\tau_{i,4}$ and $\tau_{j,4}$. Again, we have $pd(\tau_{i,4}) = pd(\tau_{j,4}) = 6$ and $b(\tau_{i,4}) = b(\tau_{j,4}) = 1$. Hence, we must compare the priorities of $\tau_{i,5}$ and $\tau_{j,5}$ to finally untie τ_i and τ_j. As shown on Fig. 4, this time we get $pd(\tau_{i,5}) = 8$ and $pd(\tau_{j,5}) = 7$ thereby implying that τ_j has a higher priority than τ_i at time $t = 3$.

4.1.2 The PD^2 algorithm

The PF algorithm performs very poorly in practice due to the third recursive rule in Prioritization Rules 1. This problem has been addressed by Baruah et al. (1995). They proposed PD, a new PFair algorithm which replaces the third rule of PF by the calculation of three new tie breaking parameters. Hence, PD makes use of five different rules, each being computed in a constant time.

In Anderson and Srinivasan (1999, 2000a), Anderson and Srinivasan proposed PD^2, an improvement of PD. They proved that the three additional prioritization rules of PD could be replaced by the computation of one quantity. Since PD^2 is a simplified version of PD, we only present PD^2 in this work.

Comparing with PF, PD^2 introduces a third parameter to compute the priority of a subtask $\tau_{i,j}$. This quantity is called the *group deadline* $GD(\tau_{i,j})$ of a subtask $\tau_{i,j}$. On the one hand, for light tasks (i.e., tasks such that $U_i < 0.5$), the group deadline is always equal to 0. On the other hand, for heavy tasks (i.e., tasks with $U_i \in [0.5, 1]$), the group deadline depends on the future load request of the task. Indeed, let us consider a sequence of subtasks $\tau_{i,j}$ to $\tau_{i,k}$ of task τ_i such that $\left(pd(\tau_{i,\ell+1}) - pd(\tau_{i,\ell})\right) = 1$ for all $j \leq \ell < k$ (see subtasks $\tau_{i,3}$ to $\tau_{i,5}$ in Fig. 5 for an example). If the subtask $\tau_{i,j}$ is scheduled in the last slot of its window, all the next subtasks of the sequence are forced to be scheduled in the last slot of their own windows. In Fig. 5 for instance, if the subtask $\tau_{i,3}$ is scheduled at the fourth time unit then subtasks $\tau_{i,4}$ and $\tau_{i,5}$ have to be scheduled at the fifth and sixth time unit, respectively. The group deadline is defined as the earliest time-instant such that such a cascading sequence ends.

Definition 9 (*Group Deadline*) The group deadline of any subtask $\tau_{i,j}$ of a task τ_i such that $U_i < 0.5$ (i.e., a light task), is $GD(\tau_{i,j}) \overset{\text{def}}{=} 0$.

The group deadline $GD(\tau_{i,j})$ of a subtask $\tau_{i,j}$ belonging to a heavy task τ_i (i.e., $U_i \geq 0.5$), is the earliest time t, where $t \geq pd(\tau_{i,j})$, such that either $(t = pd(\tau_{i,k}) \wedge b(\tau_{i,k}) = 0)$ or $\left(t = pd(\tau_{i,k}) + 1 \wedge \left(pd(\tau_{i,k+1}) - pd(\tau_{i,k})\right) \geq 2\right)$ for some subtask $\tau_{i,k}$ of τ_i such that $k \geq j$.

Informally, for heavy tasks, $GD(\tau_{i,j})$ is the earliest time instant greater than or equal to $pd(\tau_{i,j})$ that either finishes a succession of pseudo-deadlines separated by only one time unit or where the task τ_i becomes punctual.

In Fig. 5, the group deadline of $\tau_{i,3}$ is thereby $GD(\tau_{i,3}) = 8$. The interested reader may consult Anderson and Srinivasan (1999) for further information on the computation of the group deadlines in a constant time (see Equation (32) in Anderson and Srinivasan 1999).

With the definition of this new quantity, we can compare the priorities of two eligible subtasks at time *t* using the following set of three rules:

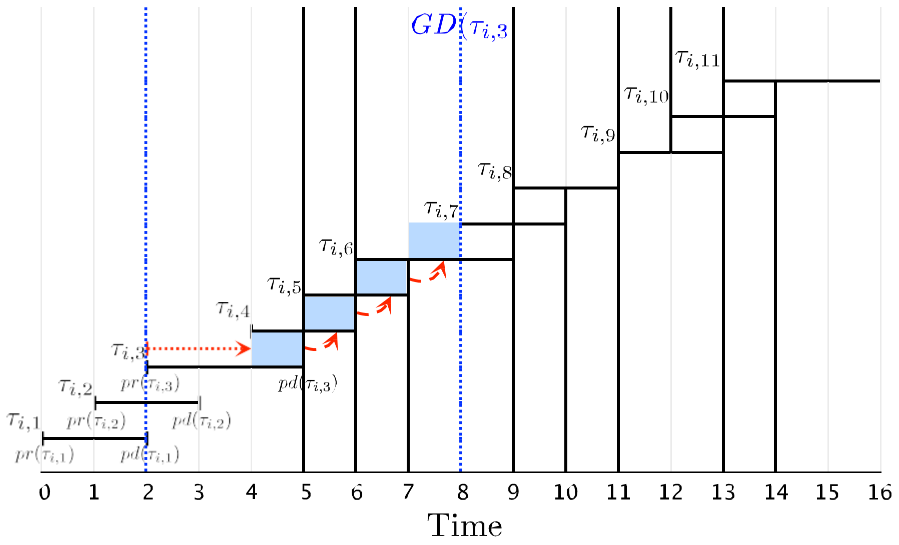


Fig. 5 Illustration of the group deadline of the subtask $\tau_{i,3}$ of a periodic task $\tau_i \overset{\text{def}}{=} \langle 8, 11, 11 \rangle$

Prioritization Rules 2 *(Prioritization Rules of PD^2) A subtask $\tau_{i,j}$ has a higherpri- ority than a subtask $\tau_{k,£}$ under PD^2 (denoted $\tau_{i,j} \succ \tau_{k,£}$) iff:*

(i) $pd(\tau_{i,j}) < pd(\tau_{k,£})$ **or**
(ii) $pd(\tau_{i,j}) = pd(\tau_{k,£}) \land b(\tau_{i,j}) > b(\tau_{k,£})$ **or**
(iii) $pd(\tau_{i,j}) = pd(\tau_{k,£}) \land b(\tau_{i,j}) = b(\tau_{k,£}) = 1 \land GD(\tau_{i,j}) > GD(\tau_{k,£})$

Again, if $\tau_{i,j}$ and $\tau_{k,£}$ have the same priority (i.e., neither $\tau_{i,j} \succ \tau_{k,£}$ nor $\tau_{k,£} \succ \tau_{i,j}$ holds) then the tie can be broken arbitrarily by the scheduler.

PD^2 has been first proven to be optimal for the scheduling of periodic tasks with implicit deadlines following a PFair scheduling policy (Anderson and Srinivasan 1999). That is, a subtask $\tau_{i,j}$ of a task τ_i is eligible to be scheduled only between its pseudo-release and its pseudo-deadline, thereby keeping the lag of τ_i within $(-1, 1)$. However, PD^2 was further extended over the years; first, for the scheduling of tasks under an ERFair scheduling policy (Anderson and Srinivasan 2000a), and then for the scheduling of more complex task models such as sporadic and dynamic task sets (Anderson and Srinivasan 2000b; Srinivasan and Anderson 2002, 2005). Srini- vasan and Anderson (2002) proved that

Theorem 1 *For any set τ of sporadic tasks with implicit deadlines executed on m identical processors, PD^2 respects all task deadlines provided that $\sum_{\tau_i \in \tau} U_i \leq m$ and $\forall \tau_i \in \tau : U_i \leq 1$.*

4.2 Boundary fairness

The authors of Zhu et al. (2003, 2011) showed that all deadlines can be respected by ensuring the fairness property only at task deadlines, rather than making scheduling decisions at every time unit. They proposed an optimal scheduling algorithm called BF for the scheduling of *periodic* tasks with implicit deadlines. This algorithm divides the time in slices bounded by two successive task deadlines. Then, the BF scheduler is invoked at every such boundary to make scheduling decisions for the next time slice. This algorithm is said to be *boundary fair* (BFair).

Formally, let the boundary b_k denote the kth time-instant in the schedule at which the scheduler is invoked. We say that b_k and b_{k+1} are the boundaries of the kth time slice denoted by TS^k. If $B \overset{\text{def}}{=} \{b_0, b_1, b_2, \ldots\}$ (with $b_k < b_{k+1}$ and $b_0 = 0$) denotes the set of boundaries encountered in the schedule, then a boundary fair schedule is defined as follows:

Definition 10 (*Boundary fair schedule*) A schedule is said to be boundary fair if and only if, at any boundary $b_k \in B$, it holds for every $\tau_i \in \tau$ that $\mathrm{lag}_i(b_k) < 1$.

The boundaries, i.e., the instants at which the scheduler is invoked, are determined by the scheduling algorithm itself. As a minimum, the fairness must be respected at the task deadlines and therefore, boundaries must be set at those instants. Nevertheless, extra boundaries could be added so as, for instance, to ease the scheduling decisions. Hence, we note that PFair and ERFair algorithms are also boundary fair (i.e., they also respect the fairness at all boundaries as they fulfill the fairness property at every time unit). However, the contrapositive is not necessarily true: a boundary fair algorithm may not ensure the fairness at every time unit.

Figure 6 shows the correspondence between a PFair and a Boundary Fair schedule. By simply regrouping all the time units of a same task executed within a time slice TS^k, it is possible to substantially reduce the amount of preemptions and migrations of this task between the two boundaries. This property is illustrated in Fig. 6 where, for instance, the task τ_0 is subject to 4 preemptions in the boundary fair schedule instead of 11 in the PFair schedule.

As previously mentioned, a boundary fair scheduler is invoked at every boundary b_k and makes scheduling decisions for the whole time slice extending from b_k to the

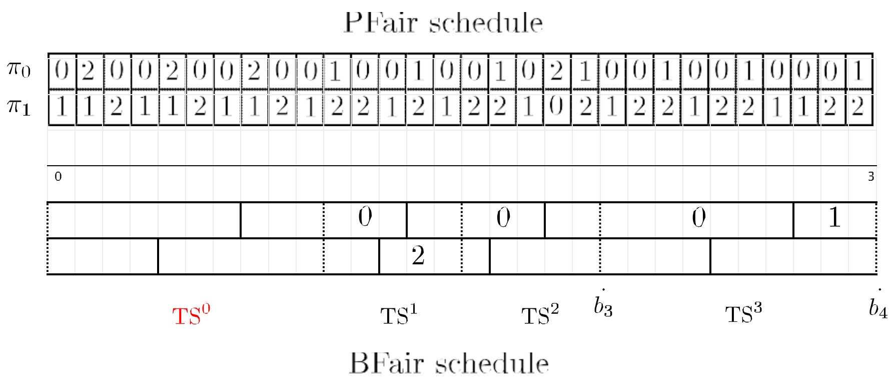


Fig. 6 Proportionate Fair and Boundary Fair schedules of three tasks τ_0, τ_1 and τ_2 on two processors. The periods and worst case execution times are defined as follows: T_0 = 15, T_1 = 10, T_2 = 30, C_0 = 10, C_1 = 7 and C_2 =19

Algorithm 1: Boundary fair scheduler.

```
Input:
$t$ := current time;
begin
    $b_{k+1} \leftarrow$ ComputeNextBoundary($\tau$);
    forall $\tau_i \in \tau$ do
        /*Allocate mandatory units for $\tau_i$                                */
        mand$_i(t, b_{k+1}) \leftarrow$ComputeMandatoryUnits($t$, $b_{k+1}$);
    end
    RU$(t,b_{k+1}) \leftarrow m \times (b_{k+1} - t) - \sum_{\tau_i \in \tau}$ mand$_i(t,b_{k+1})$;
    AllocateOptionalUnits(RU$(t, b_{k+1})$, $\tau$);
    GenerateSchedule($t$, $b_k$, mand$_i(t, b_{k+1})$, opt$_i(t, b_{k+1})$);
end
```

next boundary b_{k+1}. Any Bfair scheduler invoked at boundary b_k can be decomposed in three consecutive steps:

1. Determine the next boundary b_{k+1}, compute and allocate the minimum amount of time units each task τ_i must *mandatorily* execute in order to satisfy the condition lag$_i$ $(b_{k+1}) < 1$, $\forall \tau_i$ at the next boundary b_{k+1};
2. If all the available time units within the interval $[b_k, b_{k+1})$ have not been allotted to tasks during step 1, distribute the remaining time units amongst the tasks as *optional* time units;
3. Generate a schedule avoiding intra-job parallelism for the interval $[b_k, b_{k+1})$ according to the number of mandatory and optional time units allotted to each task.

Algorithm 1 shows the backbone of any boundary fair scheduler following these three steps.

Few BFair algorithms were developed over the years. We are actually aware of only two of them. The first one is BF which was introduced together with the BFair theory (Zhu et al. 2003). The second one is PL which was recently proposed in Kim and Cho (2011). Both were designed for the scheduling of *periodic* tasks with implicit deadlines.

Our new BFair algorithm called BF^2 will make use of many mechanisms already introduced with BF. These mechanisms are now detailed to facilitate the understanding of BF^2 presented in the next section.

4.2.1 The BF algorithm

The steps 1 to 3 of boundary fair algorithms are now detailed for BF, the optimal BFair algorithm for the scheduling of *periodic* tasks with implicit deadlines, which was proposed in Zhu et al. (2003,2011).

For periodic tasks with implicit deadlines the next boundary b_{k+1} is always defined as the earliest task deadline after the current boundary b_k. That is,

$$b_{k+1} \overset{\text{def}}{=} \min_{\tau_i \in \tau} \{d_i(b_k)\}$$

Once b_{k+1} has been determined, the three steps previously cited consist in the follow- ing:

Step 1: Allocation of the Mandatory Time Units. The BF scheduler first computes the minimum number of time units that each task τ_i has to execute within the interval
$[b_k, b_{k+1})$, in order to respect the fairness property at the next boundary b_{k+1} (i.e., $\text{lag}_i(b_{k+1}) < 1$). These time units are henceforth called *mandatory time units* and the number of such time units allotted to a task τ_i is denoted by $\text{mand}_i(b_k, b_{k+1})$. It was shown in Zhu et al. (2003, 2011), that $\text{mand}_i(b_k, b_{k+1})$ can be computed using the following equation:

$$\text{mand}_i(b_k, b_{k+1}) \overset{\text{def}}{=} \max\left\{0, \left\lfloor \text{lag}_i(b_k) + (b_{k+1} - b_k) \times U_i \right\rfloor\right\} \quad (4)$$

That is, within $[b_k, b_{k+1})$, each task τ_i must at least execute for the floor value of what it would have been executed in the fluid schedule, taking into account its
allocation error $\text{lag}_i(b_k)$ at boundary b_k. Here, the floor operator is used because in a discrete time system the execution time of a task must be an integer multiple of the system time unit.

Step 2: Allocation of the Optional Time Units. After receiving its mandatory time units, the lag of each task τ_i at the next boundary b_{k+1} is upper-bounded by 1. That is, for each individual task τ_i the fairness is ensured at time b_{k+1}. However, to avoid deadline misses in future time slices, the whole task set needs to make an appropriate
progress as well.

Example 2 Consider three periodic tasks $\tau_1 = \langle 1, 2, 2\rangle$, $\tau_2 = \langle 1, 4, 4\rangle$ and $\tau_3 = \langle 1, 4, 4\rangle$. The total utilization U is equal to 1 and the task set is therefore feasible on one processor. The two first time slices TS^0 and TS^1 extend from time 0 to 2 and from time 2 to 4, respectively. Using Expression 4 to calculate the number of mandatory time units that must be allocated to each task within TS^0, we get $\text{mand}_1(0, 2) = 1$, $\text{mand}_2(0, 2) = 0$ and $\text{mand}_3(0, 2) = 0$. If, as illustrated on Fig. 7, we do not allocate

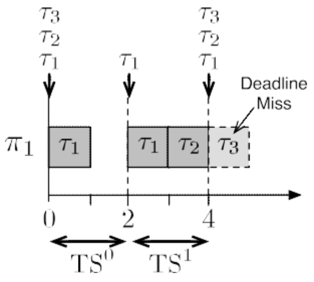


Fig. 7 Illustration of Example 2

a time unit either to τ_2 or τ_3, then all three tasks will need one mandatory time unit within TS^1. Since TS^1 spans only over two time units, this leads to a deadline miss.

Consequently, if the sum of all the mandatory time units allocated to all tasks differs from the total processing capacity of the platform in the time interval $[b_k, b_{k+1})$, then the spare processing time should be distributed among the tasks.

The total number of available time units on m processors in the time interval $[b_k, b_{k+1})$ is given by $m \times (b_{k+1} - b_k)$. Thus, after all the tasks received their mandatory time units in Step 1, the number of *remaining time units* $\mathrm{RU}(b_k, b_{k+1})$ within this interval is given by:

$$\mathrm{RU}(b_k, b_{k+1}) = m \times (b_{k+1} - b_k) - \sum_{\tau_i \in \tau} \mathrm{mand}_i(b_k, b_{k+1}) \tag{5}$$

Tasks compete for these $\mathrm{RU}(b_k, b_{k+1})$ time units, and each task τ_i possibly receives (at most) *one* of these remaining time units as an *optional time unit*. The number of optional time units allotted to τ_i (i.e., 0 or 1) is denoted by $\mathrm{opt}_i(b_k, b_{k+1})$. It is important to note that, even though these time units are optional for tasks (i.e., τ_i does not need to execute for this extra time to respect the fairness at boundary b_{k+1}), they actually have to be distributed amongst the tasks in order to guarantee an appropriate progress of the whole task set and avoid future deadline misses.

However, not all the tasks can receive an optional time unit. Specifically,

Definition 11 (*Eligible Task with BF*) A task τ_i is said to be *eligible* for an optional unit if

(i) its allocation error (i.e., $\mathrm{lag}_i(t)$) is greater than 0 *and*
(ii) $\mathrm{mand}_i(b_k, b_{k+1}) < (b_{k+1} - b_k)$, i.e., the number of time units already allocated to τ_i is strictly less than the length of the time slice. This second condition prevents a task from being executed concurrently on two (or more) processors.

Every eligible task competes for one optional time unit with an associated priority representing the future load requirement of the task. This priority of each task is based on two different parameters which can be computed in a constant time. We refer the reader to Zhu et al. (2003, 2011) for more informations.

The algorithm used by BF to distribute the optional time units at time-instant t ($b_k \leq t < b_{k+1}$) is shown in Algorithm 2. First, the algorithm identifies all the active tasks eligible for an optional time unit (line 1). Then, the $\mathrm{RU}(t, b_{k+1})$ unallocated time units are distributed amongst the eligible tasks with the help of the function GetHighestPriorityTask($E(t)$) (lines 6 to 12) where $E(t)$ is the set of eligible tasks at time t. Whenever a task τ_i is selected, it gets one optional unit (line 8) and its lag at boundary b_{k+1} is updated accordingly (line 9). The task τ_i is then removed from the eligible task set (line 10).

Step 3: Generation of the Schedule. Once we have determined the execution time of each task for the next time slice $[b_k, b_{k+1})$ (i.e., we have calculated $\mathrm{mand}_i(b_k, b_{k+1})$ and $\mathrm{opt}_i(b_k, b_{k+1}), \forall \tau_i \in \tau$), we still have to generate the schedule that will be exe- cuted in the time interval $[b_k, b_{k+1})$. It was shown in Zhu et al. (2003, 2011) that McNaughton's wrap around algorithm proposed in McNaughton (1959), can be used

Algorithm 2: AllocateOptionalUnits(RU(t, b_{k+1}), τ) which dispatches the remaining time units amongst the eligible tasks. General algorithm for BFair schedulers. The function GetHighestPriorityTask($E(t)$) might be implemented
differently for each particular scheduler.

```
1  $E(t)$ := set of eligible tasks;
2  **forall** $\tau_i \in E(t)$ **do**
3      Compute $\tau_i$'s priority at the next boundary;
4  **end**
   //Select RU$(t, b_{k+1})$ optional units
5  RU$(t,b_{k+1}) := m \times (b_{k+1} - t) - \sum_{\tau_i \in \tau}$ mand$_i(t,b_{k+1})$;
6  **while** RU$(t,b_{k+1}) > 0$ and $E(t) \neq \emptyset$ **do**
7      $\tau_i$ :=GetHighestPriorityTask($E(t)$)
8      opt$_i(t,b_{k+1}) := 1$;
9      lag$_i(b_{k+1})$ := lag$_i(b_{k+1}) - 1$;
10     $E(t) := E(t) \setminus \tau_i$;
11     RU$(t,b_{k+1})$ := RU$(t,b_{k+1}) - 1$;
12 **end**
```

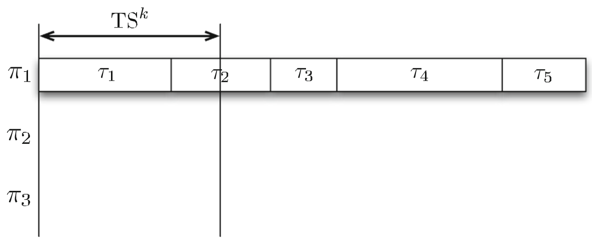
to schedule periodic tasks with implicit deadlines. That is, the execution times are assigned to the processors following a slight variation of the *next fit* heuristic. Tasks are assigned to a processor π_j as long as (i) the total execution time allocated to this processor is not greater than the length L^k of the time slice TS^k and (ii) the total exe-
cution time allocated to each processor π_1 to π_{j-1} is equal to L^k. Also, whenever the assignment of a task to π_j would cause the amount of execution time allocated to π_j to exceed the length L^k of TS^k , the task is "split" between π_j and the next processor π_{j+1} so that the total execution time assigned to π_j is exactly equal to L^k.

Example 3 Figure 8 depicts how the wrap around algorithm generates a schedule of five tasks τ_1 to τ_5 on three processors. The length of each box corresponds to the execution time (mand$_i$ (t, b_{k+1}) + opt$_i$ (t, b_{k+1})) that each task τ_i must execute within the current time slice TS^k. First, all the tasks are packed on the first processor π_1. Then, all boxes (or part of boxes) that overflow from TS^k are packed on the next processor
and this process continues until all the tasks are allocated.

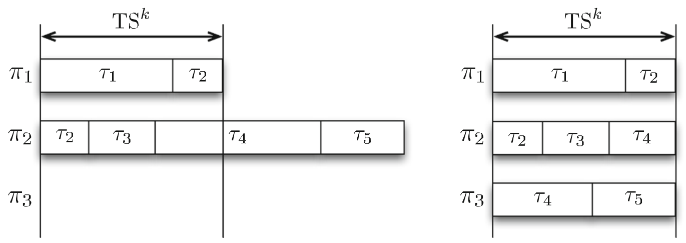
5 BF^2: a new BFair algorithm to schedule periodic tasks

Our proposed BFair scheduling algorithm named BF^2 is built on the same basics as BF (see Algorithm 1). It first determines the next boundary, then assigns mandatory and optional time units to tasks and finally generates a schedule for the time slice TS^k according to the number of time units allotted to each task.

For the scheduling of periodic tasks, the only difference between BF and BF^2 resides in the prioritization rules used to dispatch the RU(b_k, b_{k+1}) remaining time units as optional time units amongst the eligible tasks. The determination of the next boundary b_{k+1}, the computation of the mandatory time units allotted to each task



(a)



(b) **(c)**

Fig. 8 Illustration of the wrap around algorithm

and the generation of the schedule remain the same as in BF. On the other hand, the scheduling of sporadic tasks necessitates deeper modifications which will be presented in Sect. 6.

The main reason of using new prioritization rules in BF^2 is simply that , despite multiple attempts, we never succeeded to adapt the the prioritization rules of BF for the scheduling of sporadic tasks. This can easily be explained by the fact that the scheduling of sporadic tasks is more complicated than the scheduling of periodic tasks and BF was never designed while taking the sporadic task problem into consideration. Consequently, we created a completely new set of prioritization rules enabling the scheduling of sporadic tasks.

5.1 BF^2 prioritizations rules

To prioritize the eligible tasks during the optional time units dispatching phase, BF^2 uses two parameters reflecting the *future* execution requirement of each task. These parameters will further be shown to be a simple variation of the prioritization rules of PD^2.[3]

According to Definition 2, if a running task τ_i interrupts its execution at time t then its lag starts increasing gradually by an amount proportional to its utilization U_i.

[3] Note that PL uses the exact PD^2's prioritization rules (Kim and Cho 2011) while we propose a simplified version of these rules which makes use of only two parameters instead of three.

Specifically, after x time units without being executed, its lag becomes

$$\text{lag}_i(t+x) = \text{lag}_i(t) + x \times U_i \tag{6}$$

To respect the fairness property, the allocation error of τ_i can never exceed 1. Hence, the first parameter used by BF^2 to prioritize the tasks that are eligible for an optional
time unit, is the smallest number of time units x such that, if τ_i does not execute from the current time t to time $t + x$, then $\text{lag}_i\,(t + x)$ will exceed 1. This quantity can be computed by solving the following inequality

$$\text{lag}_i(t) + x \times U_i \geq 1 \tag{7}$$

We call this value of x the *urgency factor* of τ_i at time t (denoted by $\text{UF}_i(t)$) which can be formally defined as follows (solving Expression 7)

Definition 12 (*Urgency Factor*) The urgency factor $\text{UF}_i\,(t)$ of a task τ_i at time t is the minimum number of time units such that, if τ_i is not executed from time t to time $t + \text{UF}_i\,(t)$ then its allocation error $\text{lag}_i\,(t + \text{UF}_i\,(t))$ at time $t + \text{UF}_i\,(t)$ is greater than or equal to 1. That is,

$$\text{UF}_i(t) \overset{\text{def}}{=} \left\lceil \frac{1-\text{lag}_i(t)}{U_i} \right\rceil$$

From now on, we say that task τ_i is more urgent than task τ_j at time t if $\text{UF}_i\,(t) < \text{UF}_j\,(t)$, i.e., τ_i's lag would reach 1 before τ_j's lag if both tasks τ_i and τ_j were not executed anymore from time t.

Now, suppose that at time t, we stop executing a task τ_i and we wait the "very last instant" before resuming its execution, i.e., just before its lag reaches 1. That is, we do not execute τ_i from time t to time $t + \text{UF}_i\,(t) - 1$. According to Definition 2, its
lag at this instant $t + \text{UF}_i(t) - 1$ is given by

$$\text{lag}_i(t + \text{UF}_i(t) - 1) = \text{lag}_i(t) + (\text{UF}_i(t) - 1) \times U_i$$

Then, if we resume the execution of τ_i at time $t + \text{UF}_i(t) - 1$ and we execute τ_i for y consecutive time units, its lag at time $t + \text{UF}_i(t) - 1 + y$ becomes (using Definition 2)

$$\text{lag}_i(t + \text{UF}_i(t) - 1 + y) = \text{lag}_i(t) + (\text{UF}_i(t) - 1 + y) \times U_i - y \tag{8}$$

The second parameter used by BF^2 can be expressed as the exact amount of time y for which τ_i needs to execute to catch up its deviation from the fluid schedule at time $t + \text{UF}_i\,(t) - 1$, i.e., to get its lag equal to 0. According to Eq. 8, this amount of time is the value of y for which

$$\text{lag}_i(t) + (\text{UF}_i(t) - 1 + y) \times U_i - y = 0 \tag{9}$$

The value of y which satisfies the above equation is called the *recovery time* of τ_i at time t (denoted $\rho_i(t)$) and is formally defined as follows (using Expression 9):

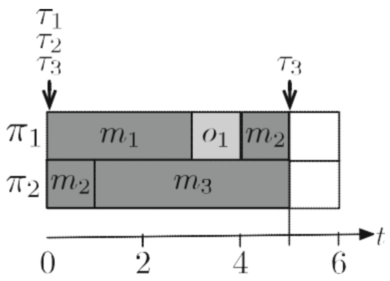


Fig. 9 Example of a schedule produced by BF^2 for three periodic tasks in a first time slice extending from time 0 to 5

Definition 13 (*Recovery Time*) The recovery time $\rho_i(t)$ of the task τ_i at time-instant t is the minimum execution time needed by τ_i to become punctual, assuming that τ_i is not executed during $\mathrm{UF}_i(t) - 1$ time units from time t. That is,

$$\rho_i(t) \overset{\text{def}}{=} \frac{\mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1) \times U_i}{1 - U_i}$$

If two eligible tasks τ_i and τ_j have the same urgency factor, BF^2 favors the task with the largest recovery time. Indeed, $\rho_i(t) > \rho_j(t)$ means that τ_i will need more execution time than τ_j to catch up its lateness on the fluid schedule, thereby constraining the future scheduling decisions during a longer period of time.

With these two parameters, a priority order between eligible tasks can now be defined:

Prioritization Rules 3 *(Prioritization Rules of BF^2) We say that an eligible task τ_i has a higher priority than a task τ_j in time slice* TS^k *if and only if*

(i) $\mathrm{UF}_i(b_{k+1}) < \mathrm{UF}_j(b_{k+1})$ *or*
(ii) $\mathrm{UF}_i(b_{k+1}) = \mathrm{UF}_j(b_{k+1}) \wedge \rho_i(b_{k+1}) > \rho_j(b_{k+1})$

If τ_i and τ_j have the same priority, then the scheduler can break the tie arbitrarily.

Note that, the priority of an eligible task does not depend on the current time t within the current time slice $[b_k, b_{k+1})$ at which the scheduler is invoked. Rather, it depends only on the next boundary b_{k+1}.

Example 4 Let us consider the schedule of three periodic tasks $\tau_1 = \langle 14, 20, 20\rangle$, $\tau_2 = \langle 5, 10, 10\rangle$ and $\tau_3 = \langle 4, 5, 5\rangle$ on two identical processors (note that $U = \sum_{i=1}^{3} U_i = 2$). The earliest deadline is the deadline of the first job of τ_3 at time $t = 5$ (see Fig. 9). Therefore, the first boundary b_1 is set to 5. According to Expression 4, there is $\mathrm{mand}_1(0, b_1) = \max\left\{0, \left\lfloor 0 + (5 - 0) \times \frac{14}{20} \right\rfloor\right\} = 3$, $\mathrm{mand}_2(0, b_1) = 2$ and $\mathrm{mand}_3(0, b_1) = 4$. Hence, the number $\mathrm{RU}(0, b_1)$ of remaining time units in the first time slice is $2 \times 5 - 9 = 1$ (from Expression 5). Therefore, one optional time unit must be given either to τ_1 or τ_2. Currently, using Definition 2, the allocation errors of τ_1 and τ_2 at the first boundary b_1 are given by $\mathrm{lag}_1(b_1) = \frac{14}{20} \times (5 - 0) - 3 = 0.5$ and $\mathrm{lag}_2(b_1) = \frac{5}{10} \times (5 - 0) - 2 = 0.5$. By Definition 12, we have $\mathrm{UF}_1(b_1) = \left\lceil \frac{1 - 0.5}{\frac{14}{20}} \right\rceil = 1$ and $\mathrm{UF}_2(b_1) = \left\lceil \frac{1 - 0.5}{\frac{5}{10}} \right\rceil = 1$. Moreover, from Definition 13, there is

$\rho_1(b_1) = \frac{0.5+(1-1)\times\frac{14}{20}}{1-\frac{14}{20}} = 1.67$ and $\rho_2(b_1) = 1$. Hence, according to Prioritization Rules 3, task τ_1 has the highest priority and must receive the optional time unit. The schedule in the first time slice can now be constructed as depicted on Fig. 9, using McNaughton's wrap around algorithm. Note that in Fig. 9, the mandatory units are represented by a dark shaded rectangle labeled with the task number and the optional time units are depicted by a light shaded rectangles.

6 Scheduling sporadic tasks with BF^2

We start this section by illustrating the challenges in scheduling sporadic tasks through a motivational example.

6.1 Challenges in scheduling sporadic tasks

Let us consider a task set composed of three sporadic tasks: $\tau_1 = \langle 1, 3, 3\rangle$, $\tau_2 = \langle 5, 6, 6\rangle$ and $\tau_3 = \langle 5, 6, 6\rangle$; scheduled on two identical processors ($U = \sum_{i=1}^{3} U_i = 2$). The first jobs of τ_2 and τ_3 arrive at time 0 while τ_1 releases its first job at time $t = 1$.

At time $t = 0$, there is two active jobs (the first job of τ_2 and τ_3) which have their deadlines at time 6. However, determining the appropriate *next boundary* comes to be the *first challenge*.

It has been shown in Zhu et al. (2003, 2011) and illustrated on Fig. 6 that considering a longer scheduling interval can help aggregate task execution time and then reduce the number of preemptions and task migrations in the produced schedule. On the other hand, to simplify the scheduling algorithm, no task deadline should occur between two successive boundaries. As a trade-off, BF^2 computes the earliest *expected* deadline. In this example, if we assume that τ_1 releases a job as soon as it can (i.e., at $t = 1$), then the deadlines of its job will occur at time $t = 4$. Hence, the earliest expected deadline is at time $b_1 = 4$.

Note that, even if task τ_1 does not arrive as expected at time $t = 1$, it is still safe to have the next boundary at time 4 since there will never be another deadline before that instant. Now that we have the first boundary b_1, BF^2 can use the algorithm presented in the previous section to compute the execution time to be allocated to the active tasks (see Fig. 10a for the produced schedule). In Fig. 10, the blank rectangles indicate the idle time of the corresponding processors.

As long as no new job is released, tasks τ_2 and τ_3 execute as shown in the schedule of Fig. 10a. However, if other jobs arrive during the interval $[0, 4)$, the schedule needs to

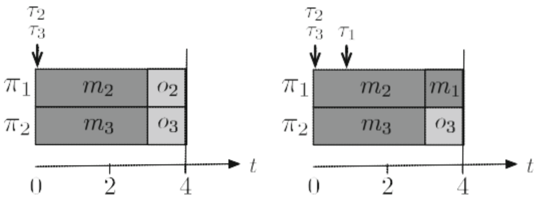


Fig. 10 Example of a schedule for a first time slice extending from 0 to 6

(a) (b)

be adjusted accordingly to ensure fairness to the newly activated tasks at the end of the interval (i.e., at boundary $b_1 = 4$). Therefore, after that task τ_1 released its first job at time $t = 1$ (as depicted on Fig. 10b), the allocation of the mandatory and optional units has to be revised for all active tasks and the schedule has to be updated accordingly.

Here comes the *second challenge*: how to schedule the allocated mandatory and optional time units, knowing that new jobs can arrive at any time. Note that, if there were only periodic tasks, the allocation of the optional time units would be *final* (since no job arrives before the next boundary) Zhu et al. (2003, 2011) and the optional unit of each task could be scheduled right after its mandatory units following McNaughton's algorithm (see Example 4). However, for sporadic tasks, the arrivals of new jobs before the next boundary may require the scheduler to *revoke* the optional time units allocated to the tasks as the newly arrived jobs may have higher priorities. Such an adjustment is crucial to ensure that higher priority tasks always get the optional time unit that they need to meet their deadlines. Indeed, in our example, if we do not revoke one of the two optional units allocated to τ_2 and τ_3 (Fig. 10a), τ_1 cannot execute before time 4 and consequently misses its deadline. Therefore, the optional time units of the tasks have to be scheduled *separately* from the mandatory time units. Optional time units are executed when no mandatory units can be executed anymore, thereby enabling optional units to be revoked if needed.

The adjusted schedule for the interval extending from $t = 1$ to $b_1 = 4$ is presented in Fig. 10b. As you can see, the optional time unit allocated to τ_2 has been revoked
and has been allocated to task τ_1 instead.

From the above example depicted in Fig. 10, we can see that, the arrival time of sporadic tasks can be different from the pre-computed boundaries. This is the major difference with periodic task sets. Therefore, the BF^2 scheduler needs to be invoked on two different events: (i) the expected task deadlines; and (ii) the arrival instants of sporadic tasks.

In the remainder of this section, we first discuss how to determine the time slice boundaries with sporadic tasks. Then, we present the detailed algorithm to schedule tasks in each time slice TS^k according to the execution time granted to each individual task. Finally, we explain how to adjust the schedule if a new job arrives between the boundaries of a time slice TS^k.

6.2 How to determine the next boundary

Boundaries for periodic tasks can be readily determined as the job release pattern of each task is predictable (Zhu et al. 2003, 2011). However, for sporadic tasks, jobs can be released at *any* instant provided that they are separated by (at least) their minimum inter-arrival times. Therefore, at a time-instant t, finding the *next boundary* b_{k+1} is not straightforward.

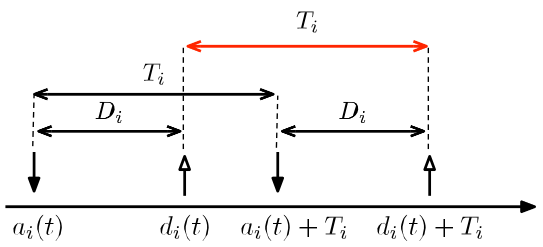
We first define three disjoint task sets at time t:

- the *ready task set <P(t)* contains active tasks (i.e., $t \in [a_i(t), d_i(t))$) whose execution has not been completed yet (i.e., $\text{ret}_i(t) > 0$);
- the *early-completion task set \JI(t)* contains tasks whose active jobs have

deadlines later than t (i.e. $d_i(t) > t$) but have already finished their executions;

- the *delayed task set* $Q(t)$ contains tasks that do not have an active job at time t.

Fig. 11 Arrival times and deadlines of two consecutive jobs of τ_i separated by exactly T_i time units



Note that, there is $\tau = {<}P(t) \cup \backslash JI(t) \cup Q(t)$ and the intersection between any two of these sets is empty.

From the previous work proposed in Zhu et al. (2003, 2011), we know that longer scheduling intervals can help aggregate task allocations and thus reduce the number of resulting preemptions and task migrations. Therefore, it is preferred to have the next boundary as late as possible. However, to simplify the scheduling algorithm, no other task deadline should ever occur before the next boundary. Following these principles, we compute the *earliest expected deadline* $d_i^e(t)$ for any task τ_i at time t as follows:

– For any task τ_i belonging to the ready task set ${<}P(t)$, the deadline of its active active job is $d_i(t)$. Thus, $d_i^e(t) \overset{\text{def}}{=} d_i(t)$;
– For an early-completion task τ_i in $\backslash JI(t)$, its active job has finished its execution and therefore cannot miss its deadline. Hence, we should consider its next job that can arrive no earlier than $a_i(t) + T_i$ and its deadline which cannot occur before
$d_i(t)+T_i$ (see Fig. 11). Therefore, the earliest expected deadline of this new instance is $d_i^e(t) \overset{\text{def}}{=} d_i(t) + T_i$;
– For a delayed task τ_i in the delayed task set $Q(t)$, its next job can arrive as early as $t + 1$, which gives $d_i^e(t) \overset{\text{def}}{=} t + 1 + D_i$;

Formally, at any time t, the next boundary b_{k+1} can be determined as follows:

$$b_{k+1} = \min\{d_i^e(t) \mid \tau_i \in \tau\} \tag{10}$$

where

$$d_i^e(t) \overset{\text{def}}{=} \begin{cases} d_i(t) & \text{if } \tau_i \in {<}P(t) \\ d_i(t) + T_i & \text{if } \tau_i \in \backslash JI(t) \\ (t+1) + D_i & \text{if } \tau_i \in Q(t) \end{cases} \tag{11}$$

Remember that $\tau = {<}P(t) \cup \backslash JI(t) \cup Q(t)$.

6.3 How to generate the schedule

As explained in Sect. 4.2, there are two types of execution time units allocated to tasks: the mandatory and optional time units. Mandatory time units have to be executed before the next boundary b_{k+1}, and as their name implies, optional time units are optional (even though they have to be distributed to highest priority tasks to ensure appropriate progress of the whole task set). Therefore, if we need to allocate time for the

Algorithm 3: Generation of the schedule in a time slice extending from t to b_{k+1}.

```
1  $\Gamma$ := tasks with allocated mandatory units;
2  $p$ := number of processors;
3  ct$(\Gamma, p)$ := earliest completion time of tasks in $\Gamma$ on $p$ processors;
   //Schedule the tasks that need a dedicated processor
4  while $\exists \tau_x \in \Gamma$ | mand$_x(t, b_{k+1}) \geq ct(\Gamma, p)$ do
5      Schedule mandatory units of $\tau_x$ on processor $\pi_p$ from time $t$ to $t$ + mand$_x(t, b_{k+1})$;
6      $p := p - 1$;
7      $\Gamma := \Gamma \setminus \{\tau_x\}$;
8  Recompute earliest completion time ct$(\Gamma, p)$ of tasks in $\Gamma$ on the $p$ last processors;
9  end
10 if $(\Gamma /= \emptyset)$ then
11     Schedule mandatory units of tasks in $\Gamma$ according to McNaughton's wrap around algorithm in a
       decreasing priority order on the $p$ last processors;
12 end
13 Schedule optional units in a decreasing priority order at the earliest available time slot
   without parallelism with their mandatory parts;
```

execution of a new task arriving during the time interval $[b_k, b_{k+1})$, optional time units that were already distributed could be revoked and reallocated to other tasks, whereas
mandatory time units cannot be unassigned. Hence, as another fundamental difference from the periodic case, the mandatory and optional time units of a same task cannot be scheduled consecutively in BF^2. Instead, mandatory time units must be executed as early as possible, while optional time units must be scheduled after the mandatory time units, i.e, we start executing what is mandatory before considering running the
optional part. Furthermore, if a delayed task τ_i released within $[b_k, b_{k+1})$, has a higher priority than an other active task τ_j which already received an optional time unit, then τ_j's optional time unit must be reallocated to τ_i's execution. Consequently, optional time units must be scheduled in a decreasing priority order, thereby ensuring that a
delayed task with higher priority (regardless of its arrival time) can always obtain an optional time unit before a low priority task does. Such a property is crucial to ensure the correctness of BF^2 proven in Sect. 8.

Algorithm 3 summarizes the steps to schedule the allocated mandatory and optional time units when BF^2 is invoked at any time t such that $b_k \leq t < b_{k+1}$. Let Γ be the set of tasks that need to execute mandatory units within the interval $[t, b_{k+1})$ (i.e., mand$_i(t, b_{k+1}) > 0$). We first schedule all mandatory units of tasks in Γ as soon as possible in the interval $[t, b_{k+1})$. To that end, we use the approach proposed by McNaughton to minimize the completion time of a set of jobs (McNaughton 1959).
First, we compute the earliest completion time ct(Γ, p) for the execution of all the mandatory time units in Γ on a number p of processors. This quantity is obtained by dividing the total workload to execute, by the number of available processors. That is,

$$\text{def} \sum_{\tau_i \in \Gamma} \text{mand}_i(t, b_{k+1})$$

$$\mathrm{ct}(\Gamma, p) = \frac{}{p}$$

If a task τ_i in Γ requires ct(Γ, p) time units[4] or more for its mandatory part, we dedicate one processor for the execution of τ_i's mandatory units from time t to t + mand$_i(t, b_{k+1})$ (line 5). The task τ_i is then removed from Γ and the earliest completion time of mandatory units that are not allocated yet, is updated taking into account that
one processor is not available anymore (lines 6 to 8). Once all remaining tasks in Γ have mand$_i(t, b_{k+1}) <$ ct(Γ, p), McNaughton's wrap around algorithm is used to schedule the mandatory units of the remaining tasks in Γ on the p last processors, within the time slice of length ct(Γ, p). That is, mandatory time units are assigned in a non-increasing priority order, and whenever the number of time units assigned to a processor π_j would exceed ct(Γ, m), then the task is split between π_j and π_{j+1} (see Fig. 8). However, in a discrete time environment, every execution time must be an integer, which is probably not the case of ct(Γ, p). We therefore have two options:
split the task when the workload assigned to a processor exceeds either $\lfloor$ct$(\Gamma, p)\rfloor$ or $\lceil$ct$(\Gamma, p)\rceil$. Let us first assume that we split it when we reach $\lceil$ct$(\Gamma, p)\rceil$. In this case, there are smt less time units assigned to the last processor where smt is given by

$$smt \overset{\text{def}}{=} p \times \lceil ct(\Gamma, p)\rceil - \sum_{\tau_i \in \Gamma} \text{mand}_i(t, b_{k+1})$$

Hence, we instead assign $\lfloor$ct$(\Gamma, p)\rfloor$ time units to the smt first processors and $\lceil$ct$(\Gamma, p)\rceil$ to the $(p - smt)$ last processors (see the following example for a detailed illustration).

Finally, all optional units are scheduled in a decreasing priority order at the earliest available time slot without parallelism with their mandatory parts (line 13).

Example 5 Suppose that, when BF[2] is invoked at time t, the next boundary is at time $b_{k+1} = t + 7$ (see Fig. 12a). We assume seven active tasks τ_1 to τ_7 and four processors. The mandatory units allocated to the seven tasks are 7, 2, 2, 3, 3, 0 and 0,
respectively. Moreover, every task receives one optional unit and we assume that the tasks have a priority inversely proportional to their associated index (i.e., task τ_1 has the highest priority and τ_7 has the lowest). We initially have a total of 17 mandatory units to schedule. Since the platform has 4 processors, the earliest completion time of all mandatory units is ct$(\Gamma, 4) = \frac{17}{4} = 4.25$. However, since τ_1 needs to execute for 7 mandatory units (which is greater than ct$(\Gamma, 4)$), we dedicate one processor for the schedule of τ_i's mandatory units (see Fig. 12a). With one processor and 7 mandatory units less, we must recompute the earliest completion time of tasks in
$\Gamma = \{\tau_2, \tau_3, \tau_4, \tau_5\}$ which is now equal to ct$(\Gamma, 3) = \frac{3+3+2+2}{3} = 3.33$. Since all remaining tasks in Γ have a number of mandatory units smaller than 3.33, we can use McNaughton's wrap around algorithm proposed in McNaughton (1959). Note that, for the remaining tasks in Γ, there are only a total of 10 mandatory units to schedule. Thus, $smt = p \times \lceil ct(\Gamma, p)\rceil - \sum_{\tau_i \in \Gamma} \text{mand}_i(t, b_{k+1}) = 3 \times 4 - 10 = 2$.
Therefore, as shown on Fig. 12b, mandatory units of tasks τ_2 to τ_5 are scheduled using McNaughton's wrap around algorithm, reserving $\lfloor$ct$(\Gamma, 3)\rfloor = 3$ time units on the two

[4] Remember that the time unit is simply a measure of the time and does not impose to ct(Γ, p) to be an integer.

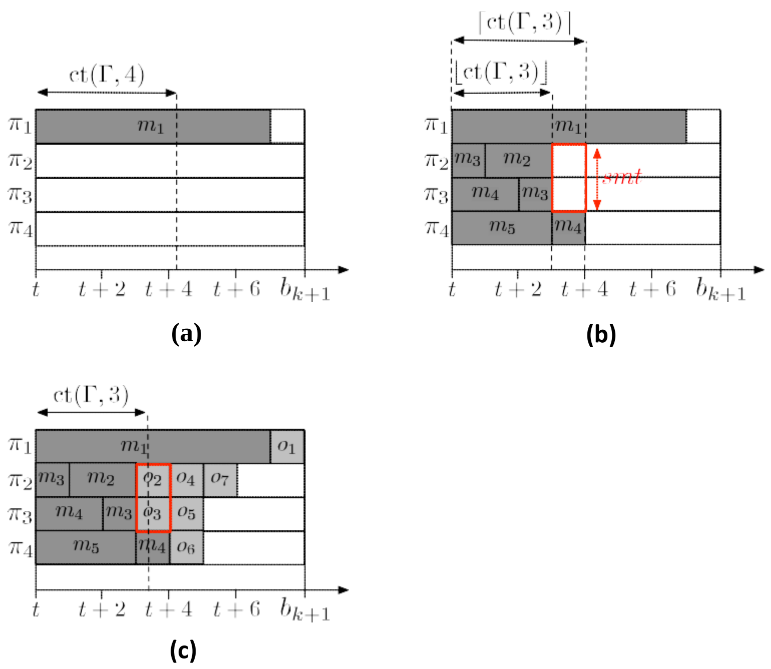


Fig. 12 Example of a schedule slice generation in BF^2

first processors (i.e., smt = 2) and $1ct(\Gamma, 3)l$ = 4 time units on the last processor. Finally, the optional time units o_1 to o_7 are scheduled at the earliest time without intra-job parallelism in a decreasing priority order (see Fig. 12c).

6.4 How to deal with arrival times of delayed tasks

As explained earlier, when new jobs arrive at time t ($b_k < t < b_{k+1}$), the BF^2 scheduler needs to be invoked to adjust the schedule within the remaining interval $[t, b_{k+1})$. In particular, the previously assigned optional units that have not yet been executed at time t should be revoked. Then, together with the newly arrived tasks, all eligible tasks will re-compete for optional units based on their priorities.

Suppose that the previous invocation of the scheduler was at time t' ($b_k \le t' < t < b_{k+1}$). For an active task τ_i at time t', its mandatory and optional units allocation for the interval $[t', b_{k+1})$ were given by $\mathrm{mand}_i(t', b_{k+1})$ and $\mathrm{opt}_i(t', b_{k+1})$, respectively. Moreover, the number of executed units of task τ_i during the interval $[t', t)$ is denoted $\mathrm{exec}_i(t', t)$. For every task τ_i, the number of mandatory units that τ_i still has to execute within the interval $[t, b_{k+1})$ is given by

$$\mathrm{mand}_i(t, b_{k+1}) = \max\{0, \mathrm{mand}_i(t', b_{k+1}) - \mathrm{exec}_i(t', t)\}$$

If the task τ_i received an optional time unit at time t' (i.e., $\mathrm{opt}_i(t', b_{k+1}) = 1$), we must consider two different situations:

– The optional unit has already been executed within $[t^i, t)$ (that is, $\text{exec}_i(t^i, t) = \text{mand}_i(t^i, b_{k+1}) + \text{opt}_i(t^i, b_{k+1})$). This leads to

$$\text{mand}_i(t, b_{k+1}) = \text{opt}_i(t, b_{k+1}) = 0$$

– The optional unit has not been executed yet (that is, $\text{exec}_i(t^i, t) < \text{mand}_i(t^i, b_{k+1}) + \text{opt}_i(t^i, b_{k+1})$). Then we must revoke the optional unit allocated to τ_i (that is, $\text{opt}_i(t, b_{k+1}) = 0$). Since the number of time units allocated to τ_i is decreased by 1, the allocation error of τ_i at the next boundary b_{k+1} must be incremented by 1:

$$\text{lag}_i(b_{k+1}) \leftarrow \text{lag}_i(b_{k+1}) + 1$$

  Task τ_i will then compete against other eligible tasks to regain its optional time unit in the interval extending from t to b_{k+1}.

For any newly arrived task τ_x, its number of mandatory time units within $[t, b_{k+1})$ can be computed as $\text{mand}_x(t, b_{k+1}) = \lfloor (b_{k+1} - t) \times U_x \rfloor$ (from Eq. 4). Moreover, there is $\text{lag}_x(b_{k+1}) = (b_{k+1} - t) \times U_x - \text{mand}_x(t, b_{k+1})$ (from Definition 2).

Finally, the distribution of optional units to eligible tasks and the generation of the schedule for $[t, b_{k+1})$ is carried out using Algorithms 2 and 3, respectively.

Due to this optional time unit redistribution at each new job arrival, and because Algorithm 3 schedules the execution of the allocated optional time units in a decreasing priority order, the following lemma holds:

Lemma 1 *Let τ_i be a task that either released a job within the interval $\lceil b_k, b_{k+1})$ or was already active at boundary b_k. Let $a_i(t)$ be the arrival time of τ_i and let the earliest activation time of τ_i in TS^k be given by $\text{act}_{i,k} \overset{\text{def}}{=} \max\{a_i(t), b_k\}$. At any time $t > b_k$ at which a task τ_x executes an optional time unit, for every task τ_i such that $\text{act}_{i,k} \leq t$, if τ_i has a higher priority than τ_x according to Prioritization Rules 3, then either τ_i also received an optional time unit no later than t or $\text{mand}_i(\text{act}_{i,k}, t) \geq (t - \text{act}_{i,k})$.*

7 BF^2: a generalization of PD^2

As stated in Sect. 4.2, the PFair theory is a particular case of the BFair approach. In particular, BF^2 is the generalization of PD^2, the simplest known PFair algorithm. We will show that the two rules of BF^2 (Prioritization Rules 3) are equivalent to the three rules of a slight variation of PD^2 presented in Sect. 7.1. That is, for a given state of the system, the rules of BF^2 and the slight variation of PD^2 named PD^{2*} provide the same task priority order (at the exception of the ties of PD^{2*} that could be broken differently in BF^2).

There are two major differences between BF^2 and PD^2:

– Unlike PD^2, BF^2 does not make any distinction between "light" and "heavy" tasks (i.e., tasks with $U_i \leq 0.5$ and tasks with $U_i > 0.5$, respectively). Indeed, the parameters $\text{UF}_i(t)$ and $\rho_i(t)$ used in the computation of the task priorities are defined for *all* tasks, irrespective of their utilization.

– Only two parameters are needed to prioritize tasks with BF^2, while there are three parameters in PD^2.

In spite of these differences, we show in this section that the definition of the group deadline proposed in PD^2 (Definition 9) can be slightly modified so that Prioritization Rule 3.(ii) of BF^2 can replace both Prioritization Rules 2.(ii) and 2.(iii) of PD^2, while Prioritization Rule 3.(i) of BF^2 provides identical results as Prioritization Rule 2.(i) of PD^2. Note that it can easily be shown that the number of basic operations (i.e., addition, subtraction, multiplication, division) needed for the computation of BF^2 parameters is slightly less than that for PD^2. Hence, similar to PD^2 which simplified the algorithm PD suppressing two tie breaking parameters, BF^2 can be seen as a simplification of PD^2 in the sense that it suppresses one tie breaking parameter.

7.1 PD^{2*}: a new slight variation of PD^2

One particularity of PD^2 is that the group deadline $GD(\tau_{i,j})$ is not similarly defined for light and heavy tasks (i.e., tasks with $U_i < 0.5$ and $U_i \geq 0.5$, respectively). Indeed, for light tasks $GD(\tau_{i,j})$ is always equal to 0 whereas for heavy tasks, it is the earliest time instant after or at the pseudo-deadline $pd(\tau_{i,j})$ following a succession of pseudo-deadlines separated by only one time unit.

We propose a slight variation of PD^2 where the group deadline is defined identically for all tasks (whatever their utilization). Hence, the group deadline of a light task τ_i is not systematically equal to 0. This *generalized group deadline* denoted by $GD^*(\tau_{i,j})$ is defined as follows:

Definition 14 (*Generalized Group Deadline*) The generalized group deadline $GD^*(\tau_{i,j})$ of any subtask $\tau_{i,j}$ of a task τ_i, is the earliest time t, where $t \geq pd(\tau_{i,j})$, such that either $(t = pd(\tau_{i,k}) \wedge b(\tau_{i,k}) = 0)$ or $(t = pd(\tau_{i,k}) + 1 \wedge \left(pd(\tau_{i,k+1}) - pd(\tau_{i,k})\right) \geq 2)$ for a subtask $\tau_{i,k}$ of τ_i such that $k \geq j$.

This slight variation of PD^2 is named PD^{2*} and the new set of rules prioritizing the subtasks becomes:

Prioritization Rules 4 *(Prioritization Rules of PD2) With PD^{2*}, a subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,\pounds}$ iff:*

(i) $pd(\tau_{i,j}) < pd(\tau_{k,\pounds})$

(ii) $pd(\tau_{i,j}) = pd(\tau_{k,\pounds}) \wedge b(\tau_{i,j}) > b(\tau_{k,\pounds})$

(iii) $pd(\tau_{i,j}) = pd(\tau_{k,\pounds}) \wedge b(\tau_{i,j}) = b(\tau_{k,\pounds}) = 1 \wedge GD^*(\tau_{i,j}) > GD^*(\tau_{k,\pounds})$

As for PD^2, if both $\tau_{i,j}$ and $\tau_{k,\pounds}$ have the same priority, then the tie can be broken arbitrarily by the scheduler.

Note that, with this new definition of the group deadline, the proof of optimality of PD^2 given in Srinivasan and Anderson (2002) has only one lemma which is impacted by the modification of the rule (iii). The updated proof of this lemma is provided in Appendix, thereby proving that the proposed variation of PD^2 is still optimal for the scheduling of sporadic tasks (as well as for more complete models of tasks such as intra-sporadic tasks and dynamic task sets) under a PFair or ERFair policy. Hence, we can write (consequence of Theorem 1)

Theorem 2 *For any set τ of sporadic tasks with implicit deadlines executed on m identical processors, PD^{2*} respects all task deadlines provided that $\sum_{\tau_i \in \tau} U_i \leq m$ and $\forall \tau_i \in \tau : U_i \leq 1$.*

7.2 Equivalence between $pd(\tau_{i,j})$ and UF$_i(t)$

Let $\tau_{i,j}$ denote the next subtask that must be executed by the task τ_i at time t. As proven below in Lemma 2, UF$_i(t)$ is a measure of the *relative* pseudo-deadline of the subtask $\tau_{i,j}$ from time t, while $pd(\tau_{i,j})$ denotes the *absolute* pseudo-deadline of $\tau_{i,j}$. That is,

$$\mathrm{UF}_i(t) = pd(\tau_{i,j}) - t \tag{12}$$

Lemma 2 *Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. If $\tau_{i,j}$ is the subtask of τ_i ready at time t, then* UF$_i(t) = pd(\tau_{i,j}) - t$.

Proof Let $\tau_{i,j}$ be the pth subtask of the current active job $J_{i,£}$ of τ_i released at time $a_{i,£}$. By definition of the pseudo-deadline (Eq. 2), $pd(\tau_{i,j}) = a_{i,£} + \left\lceil \frac{p}{U_i} \right\rceil$.

Since t and $a_{i,£}$ are both integers (i.e. the time is discrete in a PFair schedule), it holds that:

$$\begin{aligned} pd(\tau_{i,j}) - t &= a_{i,£} + \left\lceil \frac{p}{U_i} \right\rceil - t \\ &= \left\lceil \frac{p}{U_i} - t + a_{i,£} \right\rceil \\ &= \left\lceil \frac{p - U_i \times (t - a_{i,£})}{U_i} \right\rceil \end{aligned} \tag{13}$$

Moreover, because $\tau_{i,j}$ is the pth subtask which must be scheduled, it means that $p - 1$ subtasks of $J_{i,£}$ have already been executed. By definition of the allocation error (Definition 2), there is

$$\mathrm{lag}_i(t) = U_i \times (t - a_{i,£}) - (p - 1) \text{ and}$$

rearranging the terms

$$p - U_i \times (t - a_{i,£}) = 1 - \mathrm{lag}_i(t) \tag{14}$$

Then, using Eq. 14 on the right-hand side of Eq. 13, we get

$$pd(\tau_{i,j}) - t = \left\lceil \frac{1 - \mathrm{lag}_i(t)}{U_i} \right\rceil$$

Since, by Definition 12, UF$_i(t) = \left\lceil \frac{1-\mathrm{lag}_i(t)}{U_i} \right\rceil$, we finally obtain that UF$_i(t) = pd(\tau_{i,j}) - t$ which states the Lemma. $\square$

Example 6 Let us consider the schedule of the third subtask of a periodic task $\tau_i \overset{\text{def}}{=} \langle 8, 11, 11 \rangle$ and let us assume that we are at time $t = 3$ (see Fig. 13a). Since the utilization of τ_i is $U_i = \frac{8}{11}$ and the two first subtasks of τ_i have already been scheduled, using Definition 2, we get $\text{lag}_i(3) \overset{\text{def}}{=} \frac{8}{11} \times (3 - 0) - 2 = 0.1819$. Hence, according to Definition 12, the urgency factor of τ_i at time t is given by $\text{UF}_i(3) \overset{\text{def}}{=} \left\lceil \frac{1-0.1819}{\frac{8}{11}} \right\rceil = 2$. Moreover, there is $pd(\tau_{i,3}) \overset{\text{def}}{=} 0 + \left\lceil \frac{3}{\frac{8}{11}} \right\rceil = 5$ using Expression 2. Therefore, we in effect have $\text{UF}_i(3) = pd(\tau_{i,3}) - t$.

Lemma 2 proves that Prioritization Rules 2.(i) and 3.(i) are equivalent. Indeed, if we have $pd(\tau_{i,j}) < pd(\tau_{p,\ell})$ (i.e., Rule 2.(i)), then $pd(\tau_{i,j}) - t < pd(\tau_{p,\ell}) - t$, thereby leading to $\text{UF}_i(t) < \text{UF}_p(t)$ (i.e., Rule 3.(i)).

7.3 Equivalence between $b(\tau_{i,j})$ and $\rho_i(t)$

We now prove in Lemma 3 below, that $\rho_i(t) = 1$ if $b(\tau_{i,j}) = 0$ and $\rho_i(t) > 1$ if $b(\tau_{i,j}) = 1$. Note that from Definition 13, the recovery time $\rho_i(t)$ is a real number while the successor bit $b(\tau_{i,j})$ can only be equal to 0 or 1.

Lemma 3 *Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. Let $\tau_{i,j}$ be the subtask of τ_i ready at time t. If $b(\tau_{i,j}) = 0$ ($b(\tau_{i,j}) = 1$, respectively) then $\rho_i(t) = 1$ ($\rho_i(t) > 1$, respectively).*

Proof Let $\tau_{i,j}$ be the pth subtask of the current active job $J_{i,\ell}$ of τ_i released at time $a_{i,\ell}$. By definition of the successor bit (Eq. 3), we have

$$b(\tau_{i,j}) = \left\lceil \frac{p}{U_i} \right\rceil - \left\lfloor \frac{p}{U_i} \right\rfloor \tag{15}$$

Now, suppose that the successor bit $b(\tau_{i,j})$ equals 0. Then, according to Eq. 15, it holds that $\frac{p}{U_i}$ is an integer. Hence, Equation 2 implies that $pd(\tau_{i,j}) = a_{i,\ell} + \frac{p}{U_i}$ and applying Lemma 2 we get that

$$\begin{aligned} \text{UF}_i(t) &= pd(\tau_{i,j}) - t \\ &= a_{i,\ell} + \frac{p}{U_i} - t \end{aligned} \tag{16}$$

Similarly to the reasoning proposed in Lemma 2, because $\tau_{i,j}$ is the pth subtask which must be scheduled, $p - 1$ subtasks of $J_{i,\ell}$ have already been executed. By definition of the allocation error (Definition 2), there is

$$\text{lag}_i(t) = U_i \times (t - a_{i,\ell}) - (p - 1)$$

and rearranging the terms

$$p - U_i \times (t - a_{i,\ell}) = 1 - \text{lag}_i(t)$$
$$\Leftrightarrow a_{i,\ell} + \frac{p}{U_i} - t = \frac{1 - \text{lag}_i(t)}{U_i} \tag{17}$$

Then, using Eq. 17 with Eq. 16, we get

$$\text{UF}_i(t) = \frac{1 - \text{lag}_i(t)}{U_i}$$

Replacing $\text{UF}_i(t)$ in the expression of $\rho_i(t)$ given by Definition 13, we obtain

$$\begin{aligned}\rho_i(t) &= \frac{\text{lag}_i(t) + (\text{UF}_i(t) - 1) U_i}{1 - U_i}\\ &= \frac{\text{lag}_i(t) + \left(\frac{1-\text{lag}_i(t)}{U_i} - 1\right) U_i}{1 - U_i}\\ &= \frac{\text{lag}_i(t) + 1 - \text{lag}_i(t) - U_i}{1 - U_i}\\ &= \frac{1 - U_i}{1 - U_i}\end{aligned} \tag{18}$$

thereby implying that $\rho_i(t) = 1$ if $b(\tau_{i,j}) = 0$.

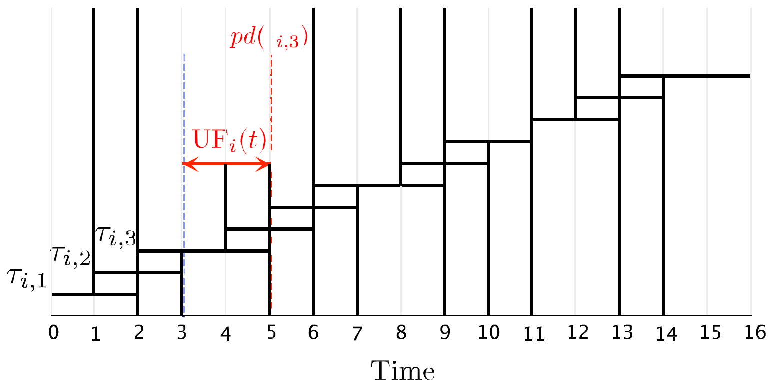
Similarly if $b(\tau_{i,j}) = 1$ then, by Eq. 15, $\frac{p}{U_i}$ is not an integer and therefore neither is $\frac{p}{U_i} - t + a_{i,\ell}$. Then, using Eq. 17 with this last expression, we get that $\frac{1-\text{lag}_i(t)}{U_i}$ is not an integer either and because $\text{UF}_i(t) = \left\lceil \frac{1-\text{lag}_i(t)}{U_i} \right\rceil$ (Definition 12), it holds by the properties of the ceil operator that $\text{UF}_i(t) > \frac{1-\text{lag}_i(t)}{U_i}$. Hence, using Eq. 18, we get

$$\rho_i(t) > \frac{\text{lag}_i(t) + \left(\frac{1-\text{lag}_i(t)}{U_i} - 1\right) U_i}{1 - U_i}$$

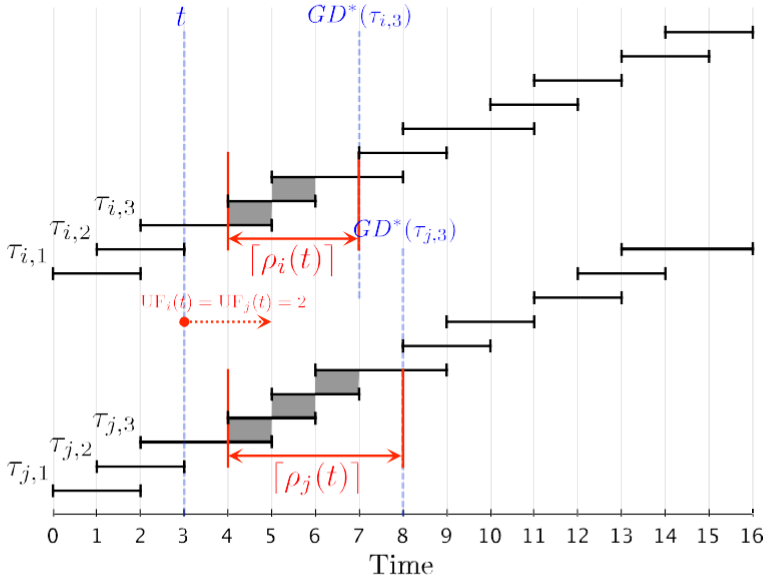
thereby leading to $\rho_i(t) > 1$ when being simplified. ∎

7.4 Equivalence between $GD^*(\tau_{i,j})$ and $\rho_i(t)$

As shown in Sect. 5, the floor value of the recovery time gives the number of successive subtasks that a task τ_i will mandatorily have to execute contiguously if the current subtask is executed in the last slot of its window, i.e., at time $t + \text{UF}_i(t) - 1$ (see Fig. 13b for an illustration). This claim is proven in Lemma 4. Then, in Lemma 5, we prove that $GD^*(\tau_{i,j})$ and $\lfloor \rho_i(t) \rfloor$ are equivalent when we compare the priorities of two tasks τ_i and τ_p such that $\text{UF}_i(t) = \text{UF}_p(t)$. Specifically, we prove that



(a)



(b)

Fig. 13 Comparison between: **a** the pseudo-deadline $pd(\tau_{i,3})$ of the third subtask of τ_i and its urgency factor $\mathrm{UF}_i(t)$ at time t. **b** The generalized group deadlines of $\tau_{i,3}$ and $\tau_{j,3}$, and their recovery times at time t

$$GD^*(\tau_{i,j}) = t + \mathrm{UF}_i(t) + \lceil\rho_i(t)\rceil - 1$$

Hence, if $\tau_{i,j}$ and $\tau_{p,q}$ are the two subtasks eligible at time t, then, assuming that $\mathrm{UF}_i(t) = \mathrm{UF}_p(t)$, it results that $\lceil\rho_i(t)\rceil > \lceil\rho_p(t)\rceil$ when $GD^*(\tau_{i,j}) > GD^*(\tau_{p,q})$.

Lemma 4 *Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. Then, τ_i has exactly $\lfloor\rho_i(t)\rfloor$ successive pseudo-deadlines separated by one time unit following t. Formally, if $\ell \overset{\text{def}}{=} \lfloor\rho_i(t)\rfloor$ and $\tau_{i,j}$ is the subtask of τ_i ready at time t, then*

$$pd(\tau_{i,j+k+1}) - pd(\tau_{i,j+k}) = 1, \quad 0 \le k < £ - 1 \tag{19}$$

$$pd(\tau_{i,j+£}) - pd(\tau_{i,j+£-1}) \ge 2 \tag{20}$$

Proof Let $\tau_{i,j}$ be the subtask of τ_i ready at time t. Hence, the first pseudo-deadline of τ_i after t is $pd(\tau_{i,j})$.

Let us assume that we are at time t_k ($> t$), that we have executed the subtasks $\tau_{i,j}$ to $\tau_{i,j+k-1}$ within $[t, t_k)$ and that the active subtask of τ_i at time t_k is $\tau_{i,j+k}$. That is, k time units have been executed between t and t_k and assuming that $\tau_{i,j}$ is a subtask of the job $J_{i,q}$ released at time $a_{i,q}$, it holds that

$$\mathrm{exec}_i\left(a_{i,q}, t_k\right) - \mathrm{exec}_i\left(a_{i,q}, t\right) = k \tag{21}$$

Using Definition 2, the difference between the lag of τ_i at time t_k and t is given by

$$\mathrm{lag}_i(t_k) - \mathrm{lag}_i(t) = U_i \times (t_k - t) - \left(\mathrm{exec}_i(a_{i,q}, t_k) - \mathrm{exec}_i(a_{i,q}, t)\right)$$

Hence, using Expression 21

$$\mathrm{lag}_i(t_k) = \mathrm{lag}_i(t) + U_i \times (t_k - t) - k \tag{22}$$

There are two cases that must be studied regarding the value of the recovery time

$\rho_i(t)$ at time t. (a) If $\rho_i(t) < k + 1$ (i.e., $\lfloor \rho_i(t) \rfloor \le k$), then by Definition 13,

$$\rho_i(t) = \frac{\mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1) \times U_i}{1 - U_i} < k + 1$$

leading to

$$\mathrm{lag}_i(t) < k + 1 - (k+1) \times U_i - (\mathrm{UF}_i(t) - 1) \times U_i$$

Using this last expression to replace $\mathrm{lag}_i(t)$ in Expression 22, we get

$$\begin{aligned}
\mathrm{lag}_i(t_k) &= \mathrm{lag}_i(t) + U_i \times (t_k - t) - k \\
&< k + 1 - (k+1) \times U_i - (\mathrm{UF}_i(t) - 1) \times U_i + U_i \times (t_k - t) - k \\
&< 1 - U_i \times \left(\mathrm{UF}_i(t) + k - (t_k - t)\right)
\end{aligned}$$

Rearranging the terms, we obtain

$$\frac{1 - \mathrm{lag}_i(t_k)}{U_i} > \mathrm{UF}_i(t) + k - (t_k - t)$$

Because by definition of the urgency factor (Definition 12), $\mathrm{UF}_i(t_k) \overset{\text{def}}{=} \frac{1 - \mathrm{lag}_i(t_k)}{U_i}$, this leads to

$$\mathrm{UF}_i(t_k) > \mathrm{UF}_i(t) + k - (t_k - t)$$

Finally, applying Lemma 2 to UF$_i(t_k)$ and UF$_i(t)$, we get

$$pd(\tau_{i,j+k}) - t_k > pd(\tau_{i,j}) - t + k - (t_k - t)$$

and simplifying

$$pd(\tau_{i,j+k}) > pd(\tau_{i,j}) + k \tag{23}$$

That is, there are more than k time units between the first and the $(k + 1)$th pseudo-deadline.

(b) If $\rho_i(t) \geq k + 1$ (i.e., $\lfloor \rho_i(t) \rfloor > k$) then by Definition 13,

$$\rho_i(t) = \frac{\text{lag}_i(t) + (\text{UF}_i(t) - 1) \times U_i}{1 - U_i} \geq k + 1$$

leading to

$$\text{lag}_i(t) \geq (k+1) - (k+1) \times U_i - (\text{UF}_i(t) - 1) \times U_i$$

Using this last expression to replace lag$_i(t)$ in Expression 22, we get

$$\begin{aligned} \text{lag}_i(t_k) &= \text{lag}_i(t) + U_i \times (t_k - t) - k \\ &\geq k + 1 - (k+1) \times U_i - (\text{UF}_i(t) - 1) \times U_i + U_i \times (t_k - t) - k \\ &\geq 1 - U_i \times \left(\text{UF}_i(t) + k - (t_k - t)\right) \end{aligned}$$

Consequently,

$$\frac{1 - \text{lag}_i(t_k)}{U_i} \leq \text{UF}_i(t) + k - (t_k - t)$$

Since $\text{UF}_i(t_k) = \left\lceil \frac{1 - \text{lag}_i(t_k)}{U_i} \right\rceil$ (Definition 12) and UF$_i(t)$, k, t_k and t are natural numbers (remember that the time is discrete), by the ceil operator property, it holds that

$$\text{UF}_i(t_k) \leq \text{UF}_i(t) + k - (t_k - t)$$

Therefore, applying Lemma 2 to UF$_i(t_k)$ and UF$_i(t)$, we get

$$pd(\tau_{i,j+k}) \leq pd(\tau_{i,j}) + k \tag{24}$$

However, since $pd(\tau_{i,j}) \overset{\text{def}}{=} a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil$ if $\tau_{i,j}$ is the pth subtask of a job $J_{i,q}$ released at time $a_{i,q}$ (Expression 2), the pseudo-deadlines of two different subtasks $\tau_{i,j}$ and $\tau_{i,j+r}$ of τ_i ($r > 0$) are given by $pd(\tau_{i,j}) = a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil$ and $pd(\tau_{i,j+r}) = a_{i,q} + \left\lceil \frac{p+r}{U_i} \right\rceil$, respectively. Because, p and r are natural numbers and U_i is assumed to be smaller than or equal to 1, it holds that

$$\begin{aligned} pd(\tau_{i,j+r}) &\geq a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil + \left\lceil \frac{r}{U_i} \right\rceil \\ &\geq a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil + r \\ &\geq pd(\tau_{i,j}) + r \end{aligned}$$

Using this last expression in conjunction with Expression 25, we obtain

$$pd(\tau_{i,j+k}) = pd(\tau_{i,j}) + k \tag{25}$$

That is, there are exactly k time units separating the first and the $(k + 1)$th pseudo-deadline.

Now assuming that $0 \leq k < \ell - 1$, then both k and $k + 1$ are smaller or equal to $\ell \overset{\text{def}}{=} \lfloor \rho_i(t) \rfloor$. We are therefore in case (b) for both subtasks $\tau_{i,j+k}$ and $\tau_{i,j+k+1}$. Hence, Expression 25 yields

$$\begin{aligned} pd(\tau_{i,j+k+1}) - pd(\tau_{i,j+k}) &= \left(pd(\tau_{i,j}) + k + 1\right) - \left(pd(\tau_{i,j}) + k\right) \\ &= 1 \end{aligned}$$

thereby proving Expression 19.

If $k = \ell \overset{\text{def}}{=} \lfloor \rho_i(t) \rfloor$, the corresponding subtask $\tau_{i,j+\ell}$ is in case (a). Therefore, $pd(\tau_{i,j+\ell}) > pd(\tau_{i,j}) + \ell$ (Expression 23). On the other hand, $\tau_{i,j+\ell-1}$ is in case (b) leading to $pd(\tau_{i,j+\ell-1}) = pd(\tau_{i,j}) + \ell - 1$ (Expression 25). Hence,

$$\begin{aligned} pd(\tau_{i,j+\ell}) - pd(\tau_{i,j+\ell-1}) &> \left(pd(\tau_{i,j}) + \ell\right) - \left(pd(\tau_{i,j}) + \ell - 1\right) \\ &> 1 \end{aligned}$$

and because $pd(\tau_{i,j+\ell})$ and $pd(\tau_{i,j+\ell-1})$ are both integers, this proves Expression 20.

∎

This property can now be used to provide an expression of the generalized group deadline of a subtask $\tau_{i,j}$ as a function of the urgency factor and the recovery time of the task τ_i at time t.

Lemma 5 *Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. If $\tau_{i,j}$ is the subtask of τ_i ready at time t, then*

$$GD^*(\tau_{i,j}) = t + \mathrm{UF}_i(t) + \lceil \rho_i(t) \rceil - 1$$

Proof From Definition 14, the generalized group deadline $GD^*(\tau_{i,j})$ of a subtask $\tau_{i,j}$ is defined as the earliest time t_G, where $t_G \geq pd(\tau_{i,j})$, such that either

$$t_G = pd(\tau_{i,j+k}) \ \wedge \ b(\tau_{i,j+k}) = 0 \tag{26}$$

or

$$t_G = pd(\tau_{i,j+k}) + 1 \ \wedge \ \left(pd(\tau_{i,j+k+1}) - pd(\tau_{i,j+k}) \right) \geq 2 \tag{27}$$

for some $k \geq 0$.

As stated in Lemma 2, $pd(\tau_{i,j}) = t + \mathrm{UF}_i(t)$. Since from Definition 14, $GD^*(\tau_{i,j}) \geq pd(\tau_{i,j})$, it holds that

$$GD^*(\tau_{i,j}) \geq t + \mathrm{UF}_i(t)$$

Moreover, from Lemma 4, the instant t is followed by exactly $\lfloor \rho_i(t) \rfloor$ pseudo-deadlines of τ_i separated by one time unit.[5] Formally, if $\ell \overset{\text{def}}{=} \lfloor \rho_i(t) \rfloor$, then

$$pd(\tau_{i,j+k+1}) - pd(\tau_{i,j+k}) = 1, \qquad 0 \leq k < \ell - 1 \tag{28}$$

$$pd(\tau_{i,j+\ell}) - pd(\tau_{i,j+\ell-1}) \geq 2 \tag{29}$$

Therefore,

(A) *The condition expressed by* (27) *to have a generalized group deadline at the time instant* $pd(\tau_{i,j+k}) + 1$ *is not respected for* $0 \leq k < \ell - 1$.

Furthermore, since $pd(\tau_{i,j+k+1}) - pd(\tau_{i,j+k}) = 1$ for $0 \leq k < \ell - 1$ (Expres- sion 28), the pseudo-deadline of $\tau_{i,j+k}$ is given by

$$pd(\tau_{i,j+k}) = pd(\tau_{i,j}) + k \tag{30}$$

Now, let us assume that we are at time t_k ($\geq t$), that we have executed the subtasks $\tau_{i,j}$ to $\tau_{i,j+k-1}$ within $[t, t_k)$ and that the active subtask of τ_i at time t_k is $\tau_{i,j+k}$. That is, k time units have been executed between t and t_k and assuming that $\tau_{i,j}$ is a subtask of the job $J_{i,q}$ released at time $a_{i,q}$, it holds that

$$\mathrm{exec}_i\left(a_{i,q}, t_k\right) - \mathrm{exec}_i\left(a_{i,q}, t\right) = k \tag{31}$$

Using Definition 2, the difference between the lag of τ_i at time t_k and t is given by

$$\mathrm{lag}_i(t_k) - \mathrm{lag}_i(t) = U_i \times (t_k - t) - \left(\mathrm{exec}_i(a_{i,q}, t_k) - \mathrm{exec}_i(a_{i,q}, t) \right)$$

Hence, using Expression 31

$$\mathrm{lag}_i(t_k) = \mathrm{lag}_i(t) + U_i \times (t_k - t) - k \tag{32}$$

[5] Note that if $\lfloor \rho_i(t) \rfloor = 1$ then Lemma 4 says that there is exactly one pseudo-deadline "separated" by one time unit which means that the two pseudo-deadlines $pd(\tau_{i,j})$ and $pd(\tau_{i,j+1})$ are separated by more

thanone time unit, i.e., $pd(\tau_{i,\,j+1}) - pd(\tau_{i,\,j}) \geq 2$.

Moreover, since $\tau_{i,j+k}$ is the subtask active at time t_k, applying Lemma 2, we have that

$$\mathrm{UF}_i(t_k) = pd(\tau_{i,j+k}) - t_k$$

and using Expression 30 and Lemma 2

$$\begin{aligned} \mathrm{UF}_i(t_k) &= pd(\tau_{i,j}) + k - t_k \\ &= \mathrm{UF}_i(t) + t + k - t_k \end{aligned} \tag{33}$$

Applying Expression 33 to Definition 13, we get that

$$\begin{aligned} \rho_i(t_k) &= \frac{\mathrm{lag}_i(t_k) + U_i \times (\mathrm{UF}_i(t_k) - 1)}{1 - U_i} \\ &= \frac{\mathrm{lag}_i(t_k) + U_i \times (\mathrm{UF}_i(t) - t + k - t_k - 1)}{1 - U_i} \end{aligned}$$

and Expression 32 leads to

$$\begin{aligned} \rho_i(t_k) &= \frac{\mathrm{lag}_i(t) + U_i \times (t_k - t) - k + U_i \times (\mathrm{UF}_i(t) + t + k - t_k - 1)}{1 - U_i} \\ &= \frac{\mathrm{lag}_i(t) + U_i \times (\mathrm{UF}_i(t) - 1) + U_i \times k - k}{1 - U_i} \end{aligned}$$

Finally, Definition 13 yields

$$\begin{aligned} \rho_i(t_k) &= \rho_i(t) + \frac{U_i \times k - k}{1 - U_i} \\ &= \rho_i(t) - k \end{aligned} \tag{34}$$

Therefore, since $£ \overset{\text{def}}{=} \mathrm{L}\rho_i(t)\mathrm{J}$, we have that $\rho_i(t_k) > 1$ for every k such that $0 \le k < £ - 1$. Hence, using Lemma 3, it holds that $b(\tau_{i,j+k}) = 1$ for every subtask $\tau_{i,j+k}$ such that $0 \le k < £ - 1$ (remember that $\tau_{i,j+k}$ is the subtask active at time t_k). It therefore results that

(B) *The condition expressed by* (26) *to have a generalized group deadline of* τ_i *at time* $pd(\tau_{i,j+k})$ *is not respected for* $0 \le k < £ - 1$.

Hence, by (A) and (B), none of the conditions to have a generalized group deadline is encountered before $pd(\tau_{i,j+£-1})$. That is,

$$GD^*(\tau_{i,j}) \ge pd(\tau_{i,j+£-1})$$

Since $£ \overset{\text{def}}{=} \mathrm{L}\rho_i(t)\mathrm{J}$, Expression 34 implies that two different situations can hold at time $t_{£-1}$ (i.e., when $\tau_{i,£-1}$ is the active subtask); either $\rho_i(t_{£-1}) = 1$ or $1 < \rho_i(t_{£-1}) < 2$.

– If $\rho_i(t_{\ell-1}) = 1$ (i.e., $\rho_i(t) \overset{\text{def}}{=} \ell = \lfloor\rho_i(t)\rfloor$) then because $\tau_{i,j+\ell-1}$ is the active subtask at time $t_{\ell-1}$, we get from Lemma 3 that

$$b(\tau_{i,j+\ell-1}) = 0$$

Therefore, from Expression 26, $GD^*(\tau_{i,j}) = pd(\tau_{i,j+\ell-1})$. Consequently, using Expression 30 for $k = \ell - 1$ and applying Lemma 2,

$$\rho_i(t) = \lfloor\rho_i(t)\rfloor \Rightarrow GD^*(\tau_{i,j}) = t + \text{UF}_i(t) + \lfloor\rho_i(t)\rfloor - 1 \tag{35}$$

– If $\rho_i(t_{\ell-1}) > 1$ (i.e. $\rho_i(t) > \ell \overset{\text{def}}{=} \lfloor\rho_i(t)\rfloor$) then we get from Lemma 3 that

$$b(\tau_{i,j+\ell-1}) = 1$$

Moreover, we know from Expression 29 that $pd(\tau_{i,j+\ell}) - pd(\tau_{i,j+\ell-1}) \geq 2$. Therefore, from Expression (27), $GD^*(\tau_{i,j}) = pd(\tau_{i,j+\ell-1}) + 1$. Consequently, using Expression 30 for $k = \ell - 1$ and applying Lemma 2,

$$\rho_i(t) > \lfloor\rho_i(t)\rfloor \Rightarrow GD^*(\tau_{i,j}) = t + \text{UF}_i(t) + \lfloor\rho_i(t)\rfloor - 1 + 1 \tag{36}$$

From the properties of the ceil and floor operators, Expressions 35 and 36 can be simplified in

$$GD^*(\tau_{i,j}) = t + \text{UF}_i(t) + \lceil\rho_i(t)\rceil - 1$$

which states the Lemma. □

Assuming that $\tau_{i,j}$ and $\tau_{p,q}$ are two subtasks eligible at time t, if $\text{UF}_i(t) = \text{UF}_p(t)$ and $\lceil\rho_i(t)\rceil > \lceil\rho_p(t)\rceil$ then using Lemma 5, there is $GD^*(\tau_{i,j}) > GD^*(\tau_{p,q})$. Since $\text{UF}_i(t)$ gives the distance of $\tau_{i,j}$'s pseudo-deadline from time t, the ceil value of the recovery time $\rho_i(t)$ can be seen as a measure of the generalized group deadline of $\tau_{i,j}$ relatively to $\tau_{i,j}$'s pseudo-deadline.

Example 7 Let us consider the schedule of the third subtask of two periodic tasks $\tau_i \overset{\text{def}}{=} (9, 13, 13)$ and $\tau_j \overset{\text{def}}{=} (8, 11, 11)$ pictured on Fig. 13b. Using a similar reasoning as in Example 6, it can be shown that $\text{UF}_i(t) = \text{UF}_j(t) = 2$ at time $t = 3$. Furthermore, using Definition 2, we have $\text{lag}_i(3) = \frac{9}{13}\times(3-0)-2 = 0.077$ and $\text{lag}_j(3) = \frac{8}{11}\times(3-0)-2 = 0.1819$. Therefore, from Definition 13, we get $\rho_i(3) = \frac{0.077+(2-1)\times\frac{9}{13}}{1-\frac{9}{13}} = 2.42$ and $\rho_j(3) = \frac{0.1819+(2-1)\times\frac{8}{11}}{1-\frac{8}{11}} = 3.15$. By Definition 14, it also holds that $GD^*(\tau_{i,3}) = 7$ and $GD^*(\tau_{j,3}) = 8$. Hence, as expected, we have $GD^*(\tau_{i,3}) = t+\text{UF}_i(t)+\lceil\rho_i(t)\rceil-1 = 3+2+3-1 = 7$ and $GD^*(\tau_{j,3}) = t+\text{UF}_i(t)+\lceil\rho_i(t)\rceil-1 = 3+2+4-1 = 8$.

7.5 Equivalence between PD^{2*} and BF^2 prioritization rules

By literally translating Prioritization Rules 4.(i), (ii) and (iii) using Lemmas 2, 3 and 4, we obtain:

(i) $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$
(ii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \wedge \rho_i(t) > 1 \wedge \rho_p(t) = 1$
(iii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \wedge \rho_i(t) > 1 \wedge \rho_p(t) > 1 \wedge \mathrm{UF}_i(t) + \lceil\rho_i(t)\rceil - 1 > \mathrm{UF}_p(t) + \lceil\rho_p(t)\rceil - 1$

which can be rewritten as

(i) $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$
(ii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \wedge \rho_i(t) > 1 \wedge \rho_p(t) = 1$
(iii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \wedge \rho_i(t) > 1 \wedge \rho_p(t) > 1 \wedge \lceil\rho_i(t)\rceil > \lceil\rho_p(t)\rceil$

Because the ties can be broken arbitrarily, this can be simplified in:

(i) $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$
(ii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \wedge \rho_i(t) > \rho_p(t)$

proving that Prioritization Rules 3 of BF^2 are equivalent to Prioritization Rules 4 of the slight variation of PD^2 presented in Sect. 7.1.

Hence, we proved that the rules to prioritize the tasks under BF^2 are equivalent to the rules of the slight variation of PD^2 named PD^{2*}. Since we proved that PD^{2*} maintains the optimality of PD^2 for the scheduling of sporadic task sets with implicit deadlines provided that $U \leq m$ and $U_i \leq 1$ for every task τ_i (Theorem 2), BF^2 is optimal for the scheduling of the same class of systems when the scheduler is invoked
at each quantum of time.

Lemma 6 *If the prioritization rules of BF^2 (Prioritization Rules 3) are used at every time unit to decide which eligible tasks must be scheduled in the next time slot, then there is* $\mathrm{lag}_i(t) < 1$ *for all* τ_i *at every time* t*, ensuring that all deadlines are respected*
for sporadic tasks with implicit deadlines, provided that $U \leq m$ *and* $U_i \leq 1$ *for every* τ_i *in* τ.

Note that PD^2 and thus BF^2 when invoked at each time unit, is actually optimal for the scheduling of *generalized intrasporadic tasks*. This model of tasks supposes that (i) some subtasks $\tau_{i,j}$ of a task τ_i may arrive later than their expected pseudo-release, and (ii) some subtasks $\tau_{i,j}$ may be absent during the scheduling of the tasks set τ. The sporadic task model is obviously a particular case of this more general model.

8 Optimality of BF^2

Lemma 6 states that BF^2 is optimal if it is invoked at each and every time unit. In that case, the produced schedule is similar to that of PD^{2*} and therefore respects the fairness or early-release fairness at every time unit. Hence, we will refer to this version of BF^2 as the ERfair one. However, as explained in the previous sections, in order to reduce the number of preemptions and migrations, BF^2 should be invoked only at boundaries and job arrivals. Let S_{BFair} denote the schedule produced by BF^2 when invoked at boundaries and job arrivals, and let $S_{ERf\ air}$ be the schedule produced by BF^2 when invoked at every time units. To prove the optimality of BF^2 when invoked only at boundaries and job arrivals, we show in the remainder of this section that, even though tasks are not scheduled in the same order in S_{BFair} and $S_{ERf\ air}$, the same tasks are executed for the same amount of time between two instants corresponding to two boundaries. Therefore, since all deadlines are respected in $S_{ERf\,air}$ (Lemma 6) and because deadlines occur only at boundaries (see Sect. 6.2), BF^2 must also meet all deadlines in S_{BFair}. Hence, BF^2 is optimal for the scheduling of sporadic tasks with implicit deadlines when it is invoked only at boundaries and job arrivals.

Let us first provide some precisions on the schedules S_{BFair} and $S_{ERf\,air}$. According to Sect. 4.2.1,a task τ_i may receive a time unit in S_{BFair} at time $t \in [b_k, b_{k+1})$

- as a mandatory unit. In this case, according to Step 1 in Sect. 4.2.1, we have $\text{lag}_i(b_{k+1}) \geq 1$ without executing this time unit;
- as an optional unit. Then, according to the definition of an eligible task for an optional time unit given at Step 2 in Sect. 4.2.1, we have $\text{lag}_i(b_{k+1}) > 0$ if τ_i does not receive this time unit.

Therefore, a task τ_i is eligible for a time unit (either mandatory or optional) in S_{BFair} whenever $\text{lag}_i(b_{k+1}) > 0$. Hence, we assume from this point onward, that, in $S_{ERf\ air}$, a task τ_i is also eligible for a time unit at any time $t \in [b_k, b_{k+1})$ if $\text{lag}_i(b_{k+1}) > 0$.[6] The goal of this definition of the eligibility of a task in $S_{ERf\ air}$ is the following: if τ_i is eligible for a time unit in S_{BFair} then it is also eligible for a time unit in $S_{ERf\ air}$.

The proof of the optimality is made by induction on the boundaries. Assuming that every task in τ has been executed for the same amount of time in both S_{BFair} and $S_{ERf\ air}$ until boundary b_k, then we prove that every task in τ is scheduled for the

same amount of time between b_k and the next boundary b_{k+1} in S_{BFair} and $S_{ERf\ air}$. Notice that the induction hypothesis must be true at boundary b_0 (i.e., at the start of the schedule), since nothing has been executed yet. Hence, we only have to prove the

induction step.

We start by proving an interesting property on the urgency factor and the recovery time of a task τ_i (Lemmas 7, 8). Then, the induction step is proven in Lemma 9 and the optimality is stated in Theorem 3.

Lemma 7 *Let* $\text{exec}_i(t)$ *and* $\text{exec}_i(t')$ *be the amount of time the task* τ_i *has been executed until time* t *and* t' ($t' > t$), *respectively. If* $\text{exec}_i(t) = \text{exec}_i(t')$ *then* $\text{UF}_i(t') = \text{UF}_i(t) - (t' - t)$ *and* $\rho_i(t') = \rho_i(t)$.

Proof Let us first consider the urgency factor of τ_i. From Definition 12, $\text{UF}_i(t') = \frac{1-\text{lag}_i(t')}{U_i}$.

[6] Since $b_{k+1} > t$, we have $\text{lag}_i(b_{k+1}) > \text{lag}_i(t)$ if τ_i is not executed (Expression 6), thereby implying that $\text{lag}_i(b_{k+1}) > 0$ if $\text{lag}_i(t) > 0$. Therefore, τ_i is always eligible for a time unit when $\text{lag}_i(t) > 0$ which is a sufficient condition (Srinivasan and Anderson 2002) to have a correct schedule with an optimal ER-Fair

scheduler such as BF^2 (Lemma 6).

Furthermore, from Definition 2

$$\text{lag}_i(t') = \text{lag}_i(t) + U_i \times (t' - t) \tag{37}$$

Hence, using both expressions together, it holds that

$$\begin{aligned}
\text{UF}_i(t') &= \left\lceil \frac{1 - \text{lag}_i(t) - U_i \times (t' - t)}{U_i} \right\rceil \\
&= \left\lceil \frac{1 - \text{lag}_i(t)}{U_i} - (t' - t) \right\rceil \\
&= \left\lceil \frac{1 - \text{lag}_i(t)}{U_i} \right\rceil - (t' - t)
\end{aligned}$$

and from Definition 12

$$\text{UF}_i(t') = \text{UF}_i(t) - (t' - t) \tag{38}$$

Regarding the recovery time of τ_i, we have from Definition 13 that $\rho_i(t') = \frac{\text{lag}_i(t') + (\text{UF}_i(t') - 1) \times U_i}{1 - U_i}$. Using Equations 37 and 38, and re-applying Definition 12 afterward, we get

$$\begin{aligned}
\rho_i(t') &= \frac{\text{lag}_i(t) + U_i \times (t' - t) + (\text{UF}_i(t) - (t' - t) - 1) \times U_i}{1 - U_i} \\
&= \frac{\text{lag}_i(t) + (\text{UF}_i(t) - 1) \times U_i}{1 - U_i} \\
&= \rho_i(t)
\end{aligned}$$

Hence, the lemma. □

Lemma 8 *Let* exec$_i(t)$ *and* exec$_i(t')$ *be the amount of time the task* τ_i *has been executed until time* t *and* t' $(t' > t)$*, respectively. If* exec$_i(t)$ = exec$_i(t')$ *for all* $\tau_i \in \tau$ *then the priority order between all tasks at time* t' *is identical to the priority order computed at time* t.

Proof Let τ_k and $\tau_{£}$ be any two distinct tasks in τ and let assume that τ_k has a higher priority than $\tau_{£}$ at time t . This means that either UF$_k(t)$ < UF$_{£}(t)$ or UF$_k(t)$ = UF$_{£}(t) \wedge \rho_k(t) \geq \rho_{£}(t)$ (see Prioritization Rules 3). Since, by assumption, exec$_k(t)$ = exec$_k(t')$ and exec$_{£}(t)$ = exec$_{£}(t')$, we can use Lemma 7. This leads to UF$_k(t')$ = UF$_k(t) - (t' - t)$, UF$_{£}(t')$ = UF$_{£}(t) - (t' - t)$, $\rho_k(t') = \rho_k(t)$ and $\rho_{£}(t') = \rho_{£}(t)$. Hence, either UF$_k(t')$ < UF$_{£}(t')$ or UF$_k(t')$ = UF$_{£}(t')$ $\wedge$ $\rho_k(t') \geq \rho_{£}(t')$, thereby implying that τ_k has a higher priority than $\tau_{£}$ at time t' (see Prioritization Rules 3). Applying this argument to every pair of tasks in τ states the lemma. □

Lemma 9 *If every task in* τ *was executed for the same amount of time in both* S_{BFair} *and* S_{ERfair} *until boundary* b_k*, then every task in* τ *executes for the same amount of time between* b_k *and the next boundary* b_{k+1} *in* S_{BFair} *and* S_{ERfair} *.*

Proof In this proof, we build the schedules S_{ERfair} and S_{BFair} in parallel, and, for each decision taken in S_{ERfair} we verify that the same decision is taken in S_{BFair}.

Let us assume that for each time unit allocated to a task τ_i before time t ($b_k \leq t < b_{k+1}$) in S_{ERfair}, a time unit is also allocated to τ_i in S_{BFair}. Note that this claim is true at boundary b_k by the lemma assumption.

At time t, in S_{ERfair}, the m highest priority tasks are selected to execute. Let τ_i be the task with the *highest* priority executed in S_{ERfair} which is not executed in S_{BFair} yet. That is, we assume that the execution time allocated to every task in S_{ERfair} until time $t+1$ is identical to the execution time allocated to the same tasks in S_{BFair}, except for τ_i which received one more time unit in S_{ERfair}. There are two cases considering the urgency factor of τ_i:

1. If $\text{UF}_i(t) \leq (b_{k+1}-t)$ (i.e., the urgency factor is not greater than the time separating b_{k+1} from t), then, according to Definition 12, it holds that the allocation error of τ_i would be at least equal to 1 at the next boundary b_{k+1} if we do not execute this time unit of τ_i before b_{k+1} (i.e., $\text{lag}_i(b_{k+1}) \geq 1$). Therefore, according to Step 1 in Sect. 4.2.1, this time unit is allocated as a mandatory time unit to τ_i in the interval $[t, b_{k+1})$ in S_{BFair}.
2. $\text{UF}_i(t) > (b_{k+1} - t)$: Since τ_i is the highest priority task at time t, applying Lemma 8, we get that τ_i also has the highest priority at time b_{k+1}. Hence, according to Algorithm 2, τ_i is chosen for an optional time unit in S_{BFair}. Note that, whatever the arrival time of τ_i, BF^2 ensures that the optional time units are always allocated to tasks with the highest priority in S_{BFair} (Lemma 1). Hence, we can be sure that S_{BFair} actually executes this optional time unit.

In conclusion, for each time unit allocated to a task τ_i in S_{ERfair}, a time unit is also granted to τ_i in S_{BFair} in the interval $[b_k, b_{k+1})$. ∎

Theorem 3 *For any set τ of sporadic tasks with implicit deadlines executed on m identical processors, S_{BFair} respects all task deadlines provided that $U = \sum_{\tau_i \in \tau} U_i \leq m$ and $U_i \leq 1, \forall \tau_i \in \tau$.*

Proof By iteratively applying Lemma 9 to the intervals $[0, b_1)$, $[b_1, b_2)$, $[b_2, b_3)$, *etc.*, we prove that every task $\tau_i \in \tau$ is executed for the same number of time units in S_{ERfair} and S_{BFair} in any interval bounded by two boundaries. Since Lemma 6 implies that S_{ERfair} respects all task deadlines, and because Expression 10 imposes
that task deadlines coincide with boundaries, it must hold that all task deadlines are also met in S_{BFair}. ∎

9 Improvements

In this section, we discuss two improvements of BF^2 that can help to reduce its scheduling overheads.

9.1 Work conservation

After having allocated the optional time units for active tasks, some processors can still have idle time units since the platform may not be necessarily fully utilized as well as not all sporadic tasks are always active during the schedule. To efficiently exploit these processor idle times, a work conserving technique can be added to BF^2. That is, no processor should be idle if at least one non-executed task has some remaining work. As an example of such work conserving technique; whenever a processor π_j is idle, the non-running task with the earliest deadline (if any) is executed on processor π_j.

Note that this improvement of BF^2 does not impact its optimality. Indeed, it was proven in Srinivasan and Anderson (2005) that PD^2 is optimal for the scheduling of *generalized intra-sporadic tasks*. This model of tasks assumes that some subtasks $\tau_{i,j}$ of some tasks τ_i can be absent during the scheduling of the tasks set τ . As stated in Appendix, PD^{2*} — i.e., the slight variation of PD^2 we introduced in Sect. 7.1 — is also optimal for the scheduling of such tasks and the proofs of Sects. 7 and 8 can be used to prove the optimality of BF^2 for the scheduling of generalized intra-sporadic tasks.

Now, let us assume that the task set τ has a total utilization smaller than m. There exists a correct schedule as stated by Theorem 3. However, the platform is not fully utilized ant it must exists instants where processors remains idle in the schedule pro- duced by BF^2. Let us execute a subtask $\tau_{i,j}$ of the task τ_i in one of these idle times. This subtask $\tau_{i,j}$ will not be present anymore in the system when it should be exe-
cuted in the original non work conserving BF^2 schedule. Hence, BF^2 will have to take other scheduling decisions relying on the actual state of the system. This new system state is identical to the state that the system would have had if τ_i was a generalized intra-sporadic task where the subtask $\tau_{i,j}$ would have been absent.

Because BF^2 is optimal for the scheduling of generalized intra-sporadic tasks, adding a work conservation technique which executes subtasks earlier than initially
expected, does not jeopardize the optimality of BF^2.

9.2 Clustering

Clustering techniques as discussed in Qi et al. (2011) and Andersson and Tovar (2006) can also help reducing the number of task migrations and preemptions when the system is not fully utilized. They divide the platform into clusters of size k (i.e., subsets of k processors). Then, a bin-packing algorithm can be applied to dispatch the tasks among the clusters such that the total utilization in each cluster does not exceed k. An optimal scheduling algorithm is then executed on each cluster independently.

Additionally, in order to minimize the amount of preemptions and migrations, every task with a utilization greater than or equal to $\left(\frac{k}{k+1}\right)$ receives its own processor.

Using this approach, an optimal scheduling algorithm such as BF^2 can correctly schedule any task set with a total utilization $U \leq \frac{k}{k+1} \times m$ (Andersson and Tovar 2006).

Hence, we can compute the smallest value for k such that a given task set τ with a total utilization U remains schedulable. By minimizing the number of processors *k* in each cluster, we also reduce the number of tasks which interact with each other, thereby decreasing the number of preemptions and migrations.

Note that this clustering technique reduces to a fully partitioned scheduling algo- rithm when the total utilization of the platform is smaller than 50 %.

10 Experimental results

The BF^2 and PD^2 scheduling algorithms were both implemented into the Linux kernel in its 2.6.34 version. Experiments were then carried out on a Lenovo ThinkStation containing two Intel chips Xeon E5405. Each chip is build around four cores running at a frequency of 2 GHz. Each core has a 32 kB 8-way set associative L1 instruction cache and a 32 kB 8-way set associative write-back L1 data cache. Two on-chip cores share a unified 6 MB 24-way set associative L2 cache and the machine used for these experiments was configured with 16 GB off-chip memory.

Even though our machine is composed of 8 identical cores, we only used six of them for the execution of the real-time tasks. The two other cores were utilized for the debugging and the monitoring of the experiments.

The implemented BF^2 scheduling algorithm makes use of the work conserving technique discussed in Sect. 9.1. In its current implementation, whenever a processor π_j becomes idle, the work conserving technique favors the execution of a non currently running job that were previously running on π_j during its last execution. Hence, the number of task migrations is reduced.

The PD^2 scheduler was implemented in its early-release version. Indeed, according to Anderson and Srinivasan (2000a), this is the best performing implementation of PD^2. Furthermore, the ERFair version of PD^2 is also work conserving, which enables a fair comparison between PD^2 and BF^2.

Finally, a mechanism reducing the number of preemptions and migrations by keep- ing running the tasks on the same processors even when the theoretical scheduler asks for an instantaneous migration, has been implemented for both PD^2 and BF^2. Indeed, the exact processor upon which a task is executed on the platform is not important from a theoretical perspective. These unneeded preemptions and migrations can therefore be avoided.

For each experiment, the number of tasks is fixed to a certain number n. The utilization of each task is then randomly generated in an interval extending from
$0.7 \times \frac{6}{n}$ to $1.3 \times \frac{6}{n}$ (remember that we have 6 processors in the platform). In order to not exceed a total utilization of 6, the last task may potentially have a utilization smaller than $0.7 \times \frac{6}{n}$. The minimum inter-arrival time of each task is randomly chosen between 1 s and 2 s with a granularity of 10 ms. Moreover, the release of each job of
a task τ_i is delayed by an amount of time randomly chosen between 0 and 500 ms. Each task set is then executed during 50 s on the computing platform.

Figure 14a–f show the results obtained for different number of tasks. Each point in these graphs is the average of 10 experiments. Figure 14a–d propose normalized values (i.e., the results obtained with BF^2 have been divided by those of PD^2) for various lengths of the system time unit.

Figure 14a–c show that the number of preemptions, migrations and scheduling points are always smaller for BF^2 than for PD^2. Hence, for 20 tasks and a time unit of 10 ms, we have three times less preemptions and migrations and almost 10 times less scheduling points in BF^2 than in PD^2. For 90 tasks, which makes an average of 15 tasks per processor, the number of preemptions and migrations in BF^2 still represent only 2/3 of the preemptions and migrations of PD^2. The number of scheduling points on its side does not exceed half of the number of scheduling points imposed by PD^2.

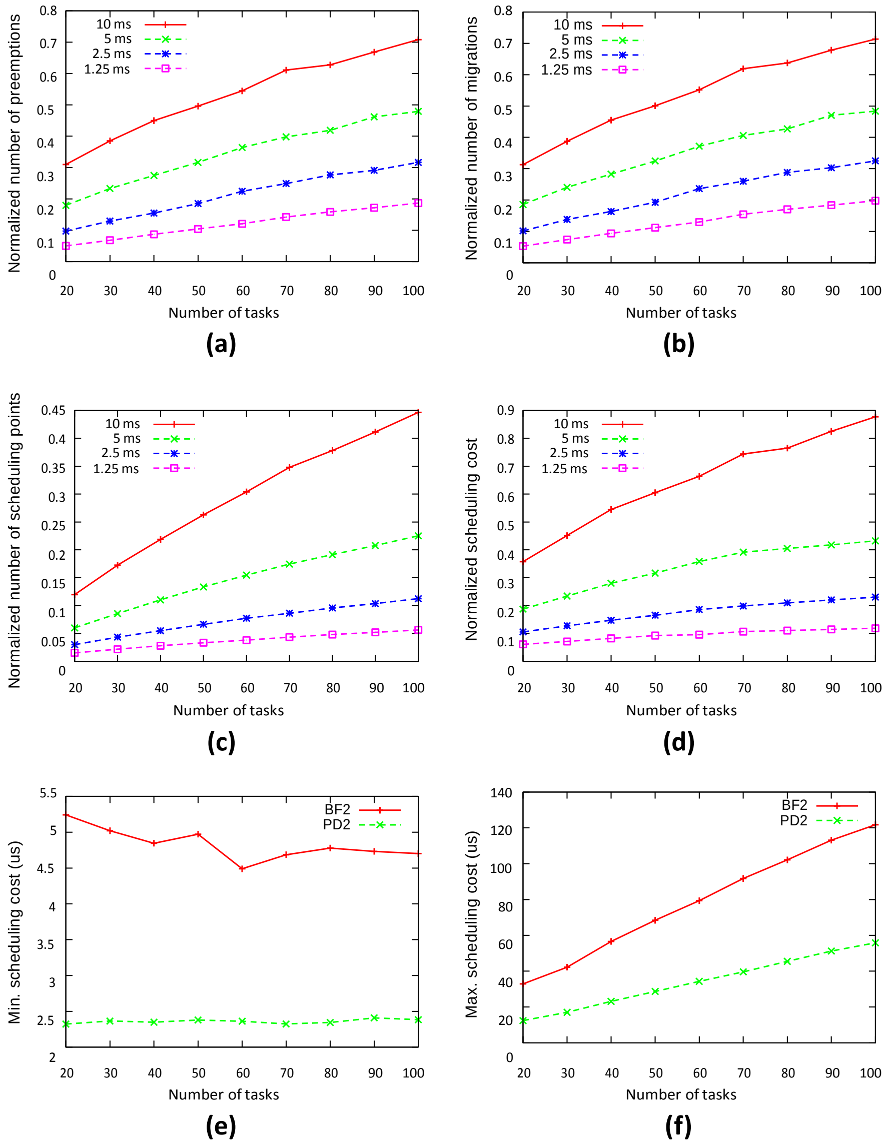


Fig. 14 Experimental results when comparing BF^2 to PD^2 on a six cores platform

However, we should note that because PD^2 makes scheduling decisions at each and every time unit, having a number of scheduling points equal to 50 % of those of PD^2 means that BF^2 calls the scheduler every two time units in average when we have 100 tasks and a time unit of 10 ms.

More importantly, even though the time needed for BF^2 to make scheduling decisions at each scheduling point is greater than for PD^2 (see Fig. 14e, f), since we have fewer scheduling points, the total time spent to schedule the system remains shorter with BF^2 (see Fig. 14d). The cost of the scheduling is therefore less with BF^2 than PD^2.

Note that if it might be the case that a job is released every time unit, the algorithm would reduce to PD^{2*}, this slight variation of PD^2 presented in this paper. Hence, the results in terms of preemptions and migrations would be similar to those of PD^2 but the total scheduling overhead should be worse. However, we believe that for most realistic applications, BF^2 will have a smaller number of scheduling points and thus smaller total overhead since it is really unlikely that new jobs would be released each and every time unit.

Furthermore, while the number of preemptions, migrations and scheduling points increase linearly with the time resolution of the system with PD^2 (i.e., they almost double when the length of the system time unit is divided by two), these values are barely dependent on the length of the system time unit with BF^2. Hence, Fig. 14a–d show that the normalized number of preemptions, migrations, scheduling points and the scheduling cost decrease when the length of the system time unit is reduced. These results seem to imply that we could increase the time resolution of our system without paying any cost in terms of schedulability under BF^2. Although this last assertion should still be verified with other studies, this is a great argument to favor the utilization of BF^2 rather than PD^2 for the scheduling of discrete-time systems. Furthermore, in Brandenburg et al. (2008), PD^2 has been compared with other non-optimal scheduling algorithms in terms of overheads and schedulability. The conclusions were based on real implementations of the algorithms (using a Linux extension called LITMUS^{RT} running on a 32 logical CPUs Niagara platform). It was shown that PD^2 is a valid competitor against EDF algorithms in terms of schedulability while taking overheads into account. Since, according to our experiments, BF^2 better performs than PD^2 in terms of overheads, the same conclusion should hold.

11 Conclusions

In this work, we addressed the problem of scheduling sporadic tasks in *discrete-time* systems. We proposed a new optimal boundary-fair scheduling algorithm for sporadic tasks (named BF^2) and proved its optimality. BF^2 makes scheduling decisions only at expected task deadlines and new job releases while PD^2 — the only alternative for the scheduling of sporadic tasks in discrete-time systems — reschedule the system at every time unit.

Our experimental results obtained through experiments conducted on a six core machine, show that BF^2 outperforms PD^2 with respect to the number of preemptions, migrations and time spent to take scheduling decisions. Furthermore, contrarily to PD^2, BF^2 is barely dependent on the system time unit length. A better time resolution could therefore be used without impacting the schedulability of the system.

Acknowledgments This work was supported, in part, by US National Science Foundation CAREER Award CNS-0953005. This work was also partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within Projects Ref. FCOMP-01-0124-FEDER-037281 (CISTER) and ref. FCOMP-01-0124-FEDER-020447 (REGAIN);

by FCT and the EU ARTEMIS JU funding, within CONCERTO Project, ref. ARTEMIS/0003/2012, JU Grant No. 333053; also by ERDF (European Regional Development Fund), through ON2 - North Portugal Regional Operational Programme, under the National Strategic Reference Framework (NSRF), within project ref. NORTE-07-0124-FEDER-000063 (BEST-CASE).

Appendix: Proof of optimality for PD^{2*}

In this appendix, we present the updated proof of the only lemma of the proof of opti- mality of PD^2 for the scheduling of sporadic tasks (Lemma 6 in Srinivasan and Ander- son 2002), which is impacted by the new definition of the group deadline proposed in Sect. 7.1. For the sake of clarity, every modification to the initial proof presented in Srinivasan and Anderson (2002) is made to bolditalics in the current document. Note that Lemma 6 of Srinivasan and Anderson (2002) is identical to Lemma 12 intro- duced in Srinivasan and Anderson (2005) which proves the optimality of PD^2 for the scheduling of dynamic task sets based on the generalized sporadic task model under some constraints. Hence, PD^{2*} is also optimal for the scheduling of dynamic task sets based on the generalized sporadic tasks model under the same constraints (the other lemmas are not impacted in Srinivasan and Anderson 2005).

Remember that in Srinivasan and Anderson (2002), the proof of optimality of PD^2 is made by contradiction. Hence, they assume that there exists a task set τ respecting some properties noted (T1), (T2) and (T3) such that a deadline is missed during the schedule S produced with PD^2. The first time-instant after the beginning of the schedule corresponding to a deadline miss is denoted by t_d.

Note that in the following proof, we use the notion of displacement defined in Srinivasan and Anderson (2002). A displacement is denoted by $\text{�}_i = \left(X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \right)$ and means that the subtask $X^{(i)}$ which was initially scheduled at time t_i is now replaced by the subtask $X^{(i+1)}$ which was scheduled at time t_{i+1}. A displacement �_i is there- fore valid if and only if $X^{(i+1)}$ is eligible to be scheduled at time t_i (which is denoted by $e(X^{(i+1)}) \leq t_i$).

Lemma 10 *Let $\tau_{i,j}$ be a subtask of a light task scheduled at $t^{\text{i}} < pd(\tau_{i,j})$ in S. If the eligibility time of the successor of $\tau_{i,j}$ is at least $pd(\tau_{i,j}) + 1$, then there cannot be holes in both $pd(\tau_{i,j})$ and $pd(\tau_{i,j}) + 1$.*

Proof Note that by part (c) of Lemma 2 (in Srinivasan and Anderson 2002), $pd(\tau_{i,j}) \leq t_d$. Therefore, $t^{\text{i}} < t_d$. Let $pd(\tau_{i,j}) = t$. If $t = t_d$, then t satisfies the stated require- ments because there is no holes in slot t_d (by part (d) of Lemma 2 in Srinivasan and Anderson 2002). In the rest of the proof we assume that $t < t_d$, and hence $t + 1 \leq t_d$. Suppose that there are holes in both t and $t + 1$. Because there is a hole in slot t and (from the statement of the lemma) the eligibility time of the successor of $\tau_{i,j}$ is at least $t + 1$, by Lemma 5 (in Srinivasan and Anderson 2002), either $pd(\tau_{i,j}) < t$ or $pd(\tau_{i,j}) = t \wedge b(\tau_{i,j}) = 1$. Because $pd(\tau_{i,j}) = t$, the latter in fact must hold, i.e., $pd(\tau_{i,j}) = t \wedge b(\tau_{i,j}) = 1$. We now show that $\tau_{i,j}$ can be removed without causing the missed deadline to be met, contradicting (T2) from Srinivasan and Anderson 2002.

In particular, we show that the sequence of left-shifts caused by removing $\tau_{i,j}$ does not extend beyond slot $t + 1$.

Let the chain of displacements caused by removing $\tau_{i,j}$ be $\text{�}_1, \text{�}_2, \ldots, \text{�}_k$, where $\text{�}_i = \left(X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \right)$, $X^{(1)} = \tau_{i,j}$ and $t_1 = t^{\text{i}}$. By Lemma 3 (in Srinivasan and Anderson 2002), we have $t_{i+1} > t_i$ for all $i \in [1, k]$. Also, the priority of $X^{(i)}$ is greater than the priority of $X^{(i+1)}$ at t_i, because $X^{(i)}$ was chosen over $X^{(i+1)}$ in S. Because $pd(\tau_{i,j}) = t \wedge b(\tau_{i,j}) = 1$, this implies the following property (***from Prioritization Rules*** 4 ***presented in Sect.*** 7.1):

(P) For all $i \in [1, k+1]$, **(i)** $pd(X^{(i)}) > t$ ***or (ii)*** $\boldsymbol{pd(X^{(i)}) = t}$ ***and*** $\boldsymbol{b(X^{(i)}) = 0}$ ***or (iii)*** $\boldsymbol{pd(X^{(i)}) = t}$ ***and*** $\boldsymbol{GD^*(X^{(i)}) \leq GD^*(\tau_{i,j})}$

Suppose this chain of displacements extends beyond $t+1$, i.e., $t_{k+1} > t+1$. Let h be the smallest $i \in [1, k+1]$ such that $t_i > t+1$. Then, $t_{h-1} \leq t+1$.

If $t_{h-1} < t+1$, then $X^{(h)}$ is eligible to be scheduled in slot $t+1$ because $e(X^{(h)}) \leq t_{h-1}$ (by the validity of displacement �$_{h-1}$). Because there is a hole in slot $t+1$ in S, $X^{(h)}$ should have been scheduled there in S (which is a contradiction with the assumption that $t_h > t+1$). Therefore, $t_{h-1} = t+1$ and by Lemma 4 (in Srinivasan and Anderson 2002), $X^{(h)}$ must be a successor of $X^{(h-1)}$. By similar reasoning, because there is a hole in slot t, $t_{h-2} = t$ and $X^{(h-1)}$ must be the successor of $X^{(h-2)}$ (see Figure 6(b) in Srinivasan and Anderson 2002).

By (P), either $pd(X^{(h-2)}) > t$ or $\boldsymbol{pd(X^{(h-2)}) = t \wedge b(X^{(h-2)}) = 0}$ ***or*** $\boldsymbol{pd(X^{(h-2)}) = t \wedge GD^*(X^{(h-2)}) \leq GD^*(\tau_{i,j})}$. In either case, $\boldsymbol{pd(X^{(h-1)}) > t+1}$ ***or*** $\boldsymbol{pd(X^{(h-1)}) = t+1 \wedge b(X^{(h-1)}) = 0}$. To see this, note that if $pd(X^{(h-2)}) > t$, then because $X^{(h-1)}$ is the successor of $X^{(h-2)}$, by (6) in Srinivasan and Anderson (2002), $pd(X^{(h-1)}) > t+1$. ***If*** $\boldsymbol{GD^*(X^{(h-2)}) = t}$ ***and*** $\boldsymbol{b(X^{(h-2)}) = 0}$***, then using (6) (in Srinivasan and Anderson*** 2002***), it holds that*** $\boldsymbol{pd(X^{(h-1)}) > t+1}$***. Moreover, because*** $\boldsymbol{b(\tau_{i,j}) = 1}$***, Definition*** 14 ***yields*** $\boldsymbol{GD^*(\tau_{i,j}) = pd(\tau_{i,j}) + 1 = t+1}$***. Hence, if*** $\boldsymbol{pd(X^{(h-2)}) = t}$ ***then*** $\boldsymbol{GD^*(X^{(h-2)}) \leq t+1}$***. By Defini- tion*** 14***,*** $\boldsymbol{GD^*(X^{(h-2)}) \geq pd(X^{(h-2)}) = t}$ ***implying that*** $\boldsymbol{GD^*(X^{(h-2)}) = t}$ ***or*** $\boldsymbol{GD^*(X^{(h-2)}) = t+1}$***. If*** $\boldsymbol{GD^*(X^{(h-2)}) = t}$ ***then Definition*** 14 ***imposes that*** $\boldsymbol{b(X^{(h-2)}) = 0}$***. Therefore, using (6) (in Srinivasan and Anderson*** 2002***), it holds that*** $\boldsymbol{pd(X^{(h-1)}) > t+1}$***. On the other hand, if*** $\boldsymbol{GD^*(X^{(h-2)}) = t+1}$ ***then, according to Definition*** 14***, either*** $\boldsymbol{GD^*(X^{(h-2)}) = pd(X^{(h-2)}) + 1 \wedge pd(X^{(h-1)}) - pd(X^{(h-2)}) \geq 2}$***, or*** $\boldsymbol{GD^*(X^{(h-2)}) = pd(X^{(h-1)}) \wedge b(X^{(h-1)}) = 0}$***. Hence, in all cases,*** $\boldsymbol{pd(X^{(h-1)}) > t+1}$ ***or*** $\boldsymbol{pd(X^{(h-1)}) = t+1 \wedge b(X^{(h-1)}) = 0}$***.***

Now, because $X^{(h-1)}$ is scheduled at $t+1$, by part (b) of Lemma 2 (in Srini- vasan and Anderson 2002), the successor of $X^{(h-1)}$ is not eligible before $t+2$, i.e., $e(X^{(h)}) \geq t+2$. This implies that the displacement �$_{h-1}$ is not valid. Thus, the chain of displacements cannot extend beyond $t+1$ and because $t+1 \leq t_d$, removing $\tau_{i,j}$ cannot cause a missed deadline at t_d to be met. This contradicts (T2) (in Srinivasan and Anderson 2002). Therefore, there cannot be holes in both t and $t+1$. ꝺ

References

Anderson JH, Srinivasan A (1999) A new look at pfair priorities. Technical Report TR00-023, Departement of Computer Science, University of North Carolina

Anderson JH, Srinivasan A (2000a) Early-release fair scheduling. In: Proceedings of the 12th Euromicro conference on real-time systems (ECRTS 2000). IEEE, Computer Society. Stockholm, Sweden, pp 35–43. doi:10.1109/EMRTS.2000.853990

Anderson JH, Srinivasan A (2000b) Pfair scheduling: beyond periodic task systems. In: Proceedings of the 7th IEEE international conference on embedded and real-time computing systems and applications

(RTCSA 2000). IEEE Computer Society. Cheju Island, South Korea, pp 297–306. ISBN 0-7695-0930-4. doi:10.1109/RTCSA.2000.896405

Anderson JH, Srinivasan A (2001) Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In: Proceedings of the 13th Euromicro conference on real-time systems (ECRTS 2001). IEEE Computer Society. Delft, The Netherlands, pp 76–85

Anderson JH, Holman P, Srinivasan A (2005) Fair scheduling of real-time tasks on multiprocessors. In: Leung JY-T (ed) Handbook of Scheduling, chapter 31. Chapman & Hall/CRC, Boca Raton

Andersson B, Bletsas K (2008) Sporadic multiprocessor scheduling with few preemptions. In: Proceedings of the 20th Euromicro conference on real-time systems (ECRTS 2008). IEEE Computer Society. Prague, Czech Republic, pp 243–252. ISBN 978-0-7695-3298-1. doi:10.1109/ECRTS.2008.9

Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In; Proceedings of the 12th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2006). IEEE Computer Society. Sydney, Australia, pp 322–334. doi:10.1109/RTCSA.2006.45

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1993) Proportionate progress: a notion of fairness in resource allocation. In: Proceedings of the 25th annual ACM symposium on theory of computing (STOC 1993). ACM. San Diego, California, USA, pp 345–354. ISBN 0-89791-591-7. doi:10.1145/167088.167194

Baruah SK, Gehrke J, Plaxton CG (1995) Fast scheduling of periodic tasks on multiple resources. In: Proccedings of the 9th international parallel processing sSymposium (IPPS '95). IEEE Computer Society. Santa Barbara, California, USA, pp 280–288. ISBN 0-8186-7074-6

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica 15(6):600–625

Bletsas K, Andersson B (2009) Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In: Proceedings of the 30th IEEE real-time systems symposium. pp 447–456. ISBN 978-0-7695-3875-4. doi:10.1109/RTSS.2009.16

Bletsas K, Andersson B (2011) Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. Real-Time Syst 47:319–355. ISSN 0922–6443

Brandenburg BB, Calandrino JM, Anderson JH (2008) On the scalability of real-time scheduling algorithms on multicore platforms: a case study. In: Proceedings of the29th IEEE real-time systems symposium (RTSS 2008). IEEE Computer Society. pp 157–169

Funk S (2010) LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. Real-Time Syst 46:332–359. ISSN 0922–6443

Funk S, Levin G, Sadowski C, Pye I, Brandt S (2011) DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling. Real-Time Syst 47:389–429. ISSN 0922–6443

IEEE (2003) IEEE standard for information technology—standardized application environment profile (aep)—posix realtime and embedded application support. Technical Report Std 1003.13-2003. IEEE Computer Society

Kim H, Cho Y (2011) A new fair scheduling algorithm for periodic tasks on multiprocessors. Inf Process Lett 111(7):301–309

Krten R, QNX Software Systems (2012) Getting started with QNX neutrino: a guide for realtime programmers. Technical report, QNX Software Systems

Levin G, Funk S, Sadowski C, Pye I, Brandt S (2010) DP-Fair: a simple model for understanding optimal multiprocessor scheduling. In: Proceedings of the 22nd Euromicro conference on real-time systems (ECRTS 2010). IEEE Computer Society. Brussels, Belgium, pp 3–13

LynuxWorks (2005) Lynxos user's guide. lynxos release 4.0. Technical Report DOC-0453-02, LynuxWorks

McNaughton R (1959) Scheduling with deadlines and loss functions. Manag Sci 6(1):1–12. ISSN 00251909

Nelissen G, Berten V, Nélis V, Goossens J, Milojevic D (2012) U-EDF: an unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: Proceedings of the 24th Euromicro conference on real-time systems (ECRTS 2012). IEEE Computer Society. Pisa, Italy, pp 13–23

Qi X, Zhu D, Aydin H (2011) Cluster scheduling for real-time systems: utilization bounds and run-time overhead. Real-Time Syst 47(3):253–284. ISSN 0922–6443. doi:10.1007/s11241-011-9121-1

QNX Software Systems (2012) Qnx neutrino realtime operating system: library reference. Technical report, QNX Software Systems

QNX Software Systems Limited (2012) Qnx neutrino realtime operating system library reference. Technical report, QNX Software Systems Limited

Regnier P, Lima G, Massa E, Levin G, Brandt S (2011) RUN: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: Proceedings of the 32th IEEE real-Time systems symposium (RTSS 2011). IEEE Computer Society. Vienna, Austria, pp 104–115

RTEMS (2012) URL http://www.rtems.org. Accessed March 2014

Srinivasan A, Anderson JH (2002) Optimal rate-based scheduling on multiprocessors. In: Proceedings on 34th annual ACM symposium on theory of computing (STOC 2002). ACM. Montréal, Quebec, Canada, pp 189–198. ISBN 1-58113-495-9. doi:10.1145/509907.509938

Srinivasan A, Anderson JH (2005) Fair scheduling of dynamic task systems on multiprocessors. J Syst Softw 77(1): 67–80, Avril 2005. ISSN 0164–1212. doi:10.1016/j.jss.2003.12.041

Wind River Systems, Inc. (2011) VxWorks: Application programmer's guide 6.9, 2nd edn. Technical report, Wind River Systems Inc.

Zhu D, Mossé D, Melhem R (2003) Multiple-resource periodic scheduling problem: how much fairness is necessary? In: Proceedings of the 24th IEEE international real-time systems symposium (RTSS 2003). IEEE Computer Society. Cancun, Mexico, pp 142–151. ISBN 0-7695-2044-8

Zhu D, Qi X, Mossé D, Melhem R (2011) An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. J Parallel Distrib Comput 71(10):1411–1425. ISSN 0743–7315. doi:10.1016/j.jpdc.2011.06.003