J. Parallel Distrib. Comput. 71 (2011) 1411–1425

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

An optimal boundary fair scheduling algorithm for multiprocessor real-time systems

Dakai Zhu [a,*], Xuan Qi [a], Daniel Mossé [b], Rami Melhem [b]

[a] University of Texas at San Antonio, San Antonio, TX 78249, United States
[b] University of Pittsburgh, Pittsburgh, PA 15260, United States

ARTICLE INFO

Article history:
Received 6 September 2010
Received in revised form
12 May 2011
Accepted 16 June 2011
Available online 26 June 2011

Keywords:
Real-time systems
Multiprocessor
Periodic tasks
Optimal scheduling algorithms
Boundary fairness (Bfair) scheduling

ABSTRACT

With the emergence of multicore processors, the research on multiprocessor real-time scheduling has caught more researchers' attention recently. Although the topic has been studied for decades, it is still an evolving research field with many open problems. In this work, focusing on periodic real-time tasks with *quantum-based* computation requirements and implicit deadlines, we propose a novel optimal scheduling algorithm, namely *boundary fair (Bfair)*, which can achieve full system utilization as the well-known Pfair scheduling algorithms. However, different from Pfair algorithms that make scheduling decisions and enforce proportional progress (i.e., *fairness*) for all tasks at *each and every* time unit, Bfair makes scheduling decisions and enforces fairness to tasks *only* at tasks' period boundaries (i.e., deadlines of periodic tasks). The correctness of the Bfair algorithm to meet the deadlines of all tasks' instances is formally proved and its performance is evaluated through extensive simulations. The results show that, compared to that of Pfair algorithms, Bfair can significantly reduce the number of scheduling points (by up to 94%) and the overhead of Bfair at each scheduling point is comparable to that of the most efficient Pfair algorithm (i.e., PD^2). Moreover, by aggregating the time allocation of tasks for the time interval between consecutive period boundaries, the resulting Bfair schedule can dramatically reduce the number of required context switches and task migrations (as much as 82% and 85%, respectively) when compared to those of Pfair schedules, which in turn reduces the run-time overhead of the system.

1. Introduction

For different (e.g., periodic, sporadic and aperiodic) real-time tasks to be executed on systems with a single or multiple processing units, the scheduling problem of how to guarantee various hard and/or soft timing constraints has been studied extensively in the last few decades [42]. Although the scheduling theory for uniprocessor real-time systems has been well developed, such as the optimal EDF (earliest deadline first) and RM (rate-monotonic) scheduling algorithms [35], the scheduling for multiprocessor real-time systems is still an evolving research field and many problems remain open due to their intrinsic difficulties. With the emergence of multicore processors, there is a reviving interest in scheduling algorithms for multicore/multiprocessor real-time systems and many interesting results have been reported in recent years, such as [5,9–11,20,24,27,26,31,33,34].

Traditionally, there are two major approaches for scheduling real-time tasks in multiprocessor systems: *partitioned* and *global* scheduling [19,21]. In partitioned scheduling, each task is assigned to a specific processor and processors can only execute the tasks that are assigned to them. Although the well-established uniprocessor scheduling algorithms (such as EDF and RMS [35]) can be employed on each processor after partitioning tasks to processors, finding a feasible partition of tasks to processors has been shown to be NP-hard [19,21]. For the global scheduling, on the other hand, all ready tasks are put into a shared single queue and each idle processor fetches the next highest priority ready task from the global queue for execution. Despite its flexibility that allows tasks to migrate and execute on different processors, it has been shown that simple global scheduling policies (e.g., global-EDF and global-RMS) could fail to schedule task sets with extremely low system utilization [21]. In addition, neither partitioned nor global scheduling dominates one another as there are task sets that can be scheduled by one approach but not the other, and vice versa.

Recently, as a hierarchical approach, *cluster scheduling* has been investigated. In this approach, processors are grouped into clusters and tasks are partitioned among different clusters. For tasks that are allocated to a cluster, different global scheduling policies (e.g., global-EDF) can be adopted within the cluster [9,43]. Note that, cluster scheduling is a *general* approach, which will reduce to partitioned scheduling when there is only one processor in each cluster. For the case of a single cluster containing all the processors, it will reduce to the global scheduling.

* Corresponding author.
E-mail address: dzhu@cs.utsa.edu (D. Zhu).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.06.003

Since the shared cache architecture in multicore processors can significantly alleviate the task migration overhead in the global scheduling, in this work, we study an *optimal* global scheduling algorithm for a set of periodic real-time tasks with implicit deadlines (i.e., the relative deadlines of tasks equal their periods), which can achieve full system utilization. Fairness has been traditionally utilized to guarantee quality of service (QoS) in computing systems (e.g., in wireless networks [29]). In [12], Baruah et al. exploited fairness as a vehicle to design the first well-known quantum-time-based optimal global scheduling algorithm. Specifically, the *proportional fair (Pfair)* scheduler enforces proportional progress (i.e., *fairness*) for all tasks at *each and every* time unit, which can achieve full system utilization while guaranteeing that all tasks meet their deadlines. Several sophisticated Pfair variations have also been studied, such as *PD* [13] and PD^2 [3]. Recently, assuming that the time domain is *continuous*, the *T–L Plane* based algorithms were studied, which can also achieve full system utilization [15,23]. Following this line of research, a generalized deadline-partitioned fair (DP-Fair) scheduling model was investigated in [34]. However, since each task needs to get its time share within any allocation interval (e.g., time interval between adjacent deadlines of tasks), the time allocation to tasks can be arbitrarily small, which can lead to extremely high scheduling overhead for those scheduling algorithms.

Note that, the proportional fairness is actually a much more strict requirement than that of the original scheduling problem, where the only requirement is to have each task instance get enough time allocation and complete its execution before its deadline. Moreover, by making scheduling decisions at *each and every* quantum time unit, the Pfair algorithms can also lead to high scheduling overhead. Observing the fact that a periodic real-time task can only miss its deadline at its period boundary, in this work, we develop an optimal *boundary fair (Bfair)* scheduling algorithm, which makes scheduling decisions and ensures fairness for tasks *only* at their period boundaries (which are tasks' arrival times as well). Specifically, at each period boundary, the Bfair algorithm will allocate processors to tasks for the time units between the current period boundary and the next period boundary of tasks. Similar to the Pfair algorithms, to prevent deadline misses, Bfair also ensures *fairness* for tasks, but only at the period boundaries. That is, at each period boundary time, the allocation error for any task is less than one time unit.

We have formally proved the correctness of the Bfair algorithm on meeting the deadlines of all tasks' instances while achieving 100% system utilization. The time complexity for an efficient implementation of Bfair can be $O(n)$ (where n is the number of tasks in the system) at each scheduling point, which is the same as that of the Pfair algorithm [12]. However, Bfair can significantly reduce the number of scheduling points (by up to 94% in our simulations), and thus reduce the overall scheduling overhead. Moreover, our simulations show that the time overhead of Bfair for each scheduling point as well as for generating the whole schedule is comparable to that of the most efficient Pfair algorithm, PD^2 [3]. Furthermore, by aggregating the time allocation of tasks for the time interval between consecutive period boundaries, the resulting Bfair schedule can also dramatically reduce the number of context switches and task migrations, as much as 82% and 85%, respectively, when compared to those of Pfair schedules. Such reduction in context switches and task migrations can significantly reduce the run-time overhead, which is specially valuable for real-time systems.

There are several contributions of this work:

- First, we introduce the concept of *boundary fairness* for the periodic real-time scheduling problem, which is *fair enough* to get a feasible schedule while only requires scheduling decisions at tasks' period boundaries (i.e., deadlines and arrival times of tasks);
- Second, we propose an *optimal* and *efficient Bfair scheduling* algorithm and prove its correctness to generate a feasible schedule while achieving full system utilization;
- Finally, we evaluate the proposed Bfair algorithm and show its superior performance on reducing scheduling overhead when comparing to Pfair algorithms through extensive simulations.

The remainder of this paper is organized as follows. The related work is reviewed in Section 2. Section 3 defines the related notations and formulates the problem, which is further illustrated through a concrete motivating example. Section 4 presents the Bfair algorithm and its complexity analysis. The correctness of the Bfair algorithm is formally proved in Section 5. Simulation results are reported and discussed in Section 6. Section 7 gives out our conclusions.

2. Related work

Although rate-monotonic (RM) scheduling and earliest deadline first (EDF) have been shown to be optimal in uniprocessor periodic real-time systems, for static and dynamic priority assignments, respectively [35], neither of them is optimal for multiprocessor real-time systems [21]. Based on the partitioned scheduling, Oh and Baker studied the rate-monotonic first-fit (RMFF) heuristic and showed that RMFF can schedule any system of periodic tasks with total utilization bounded by $m(2^{1/2} - 1)$, where m is the number of processors in the system [41]. Later, a better bound of $(m + 1)(2^{1/(m+1)} - 1)$ for RMFF was shown in [37]. In [6], Andersson and Jonsson proved that the system utilization bound can reach 50% for a partitioned RM scheduling by exploiting the harmonicity of tasks' periods. For the partitioned scheduling with earliest deadline first (EDF) first-fit heuristic, Lopez et al. showed that any task set can be successfully scheduled if the total utilization is no more than $(\beta \cdot m + 1)/(\beta + 1)$, where $\beta = \lfloor 1/\alpha \rfloor$ and α is the maximum task utilization of the tasks considered [38]. Following similar techniques, the utilization bounds for partitioned-EDF in *uniform* multiprocessor systems (where the processors have the same functionalities but different processing speeds) were developed by Darera in [17].

For global scheduling based EDF, it has been shown that a task set is schedulable on m processors if the total utilization does not exceed $m(1 - \alpha) + \alpha$ [25]. Similarly, for global-RMS scheduling, system utilization of $(m/2)(1 - \alpha) + \alpha$ can be guaranteed [8]. Andersson et al. also studied one scheduling algorithm named RM-US, where tasks with utilization higher than some threshold θ have the highest priority [4]. For $\theta = \frac{1}{3}$, Baker showed that RM-US can guarantee a system utilization of $(m + 1)/3$ [8].

To further improve the achievable system utilization and reduce the scheduling overhead, Andersson et al. proposed the EKG algorithm based on the concept of *portion tasks* [7]. In EKG, a separator is defined as $SEP = \frac{k}{k+1}$ for cases where $k < m$ (m is the number of processors) and $SEP = 1$ for the case of $k = m$. For *heavy* tasks with utilization being more than *SEP*, they are allocated to their dedicated processors following the conventional partitioned-EDF scheduling. Each *light* task (with utilization being no more than *SEP*) is split into two portion tasks only if necessary, where the portion tasks are assigned to adjacent processors. The worst-case system utilization bound that can be achieved by EKG is 66% when it is configured with $k = 2$ that has few preemptions. Full (i.e., 100%) system utilization can be achieved by setting $k = m$, which can result in high scheduling overhead with many preemptions as EKG may allocate arbitrarily small share of time to tasks. Following the same line of research, several semi-partitioned based scheduling algorithms have been proposed very recently, which are different in how to handling portion tasks and thus achieve different system utilizations [26,30–32].

As an important concept, *fairness* has been widely utilized to guarantee quality of service (QoS) in computing systems (e.g., in wireless networks [29]). In [12], Baruah et al. exploited fairness as a vehicle to design the first well-known quantum-time-based optimal global scheduling algorithm for multiprocessor real-time systems, where the authors proposed the *proportional fair (Pfair)* that can achieve full system utilization. The basic idea of Pfair is to enforce proportional progress (i.e., *fairness*) for all tasks at each and every time unit, which actually puts a more strict requirement for the problem. That is, any Pfair schedule for a set of periodic real-time tasks will ensure that all task instances can complete their executions before the deadlines. By separating tasks as *light* (with task weight less or equal 50%) and *heavy* (with task weight larger than 50%) tasks, a more efficient Pfair algorithm, *PD*, is proposed in [13]. A simplified *PD* algorithm, PD^2, uses two less parameters than *PD* to compare the priorities of tasks [3]. However, both of them have the same with complexity of $O(\min\{n, m \lg n\})$, where n is the number of tasks and m is the number of processors. A variant of Pfair scheduling, early-release scheduling, was also proposed in [1]. By considering intra-sporadic tasks, where subtasks may be released later, the same authors proposed another polynomial-time scheduling algorithm, *EPDF*, that is optimal on systems of one or two processors [2], which was further extended to multiple processors in [44].

The supertask approach was first proposed to support non-migratory tasks in [40]: tasks bound to a specific processor are combined into a single *supertask* which is then scheduled as an ordinary Pfair task. When a supertask is scheduled, one of its component tasks is selected for execution using earliest deadline first policy. Unfortunately, the supertask approach cannot ensure all the non-migratory component tasks to meet their deadline even when the supertask is scheduled in a Pfair manner.[1] Based on the concept of supertask, a *reweighting* technique has been studied, which inflates a supertask's weight to ensure that its component tasks meet their deadlines if the supertask is scheduled in a Pfair manner [28]. While this technique ensures that the supertask's non-migratory component tasks meet their deadlines, some system utilization is sacrificed.

Assuming that the time domain is *continuous* and the time allocation for tasks can be arbitrarily small, several optimal *T–L* plane based scheduling algorithms have been studied, which can also achieve full system utilization and guarantee all the timing constraints of tasks [15,16,23]. More recently, by adopting the idea of *deadline partitioning*, a general scheduling model named DP-Fair was studied for systems with continuous time by Levin et al. [34]. In addition to periodic tasks, the authors also showed how to extend DP-Fair to sporadic task sets with unconstrained deadlines. More detailed discussions for the existing work on multiprocessor real-time scheduling can be found in the recent survey paper [18].

The work reported in this paper is different from all existing results. For periodic real-time tasks with quantum-based timing parameters, the proposed Bfair scheduling algorithm can achieve full system utilization while guaranteeing to meet all tasks' deadlines. Different from the Pfair algorithms, which make scheduling decisions at every time unit, the Bfair algorithm makes scheduling decisions only at tasks period boundaries. Therefore, Bfair can significantly reduce the scheduling overhead as well as run-time overhead of the tasks due to reduced number of context switches and task migrations in the generated schedule, as shown later in this paper.

3. System models and problem formulation

In this section, we first present the system models and state our assumptions. Then, after defining boundary fairness as well as related notations, the scheduling problem of periodic real-time tasks on multiprocessor systems is formally stated. A motivation example is also presented to illustrate the idea of boundary fairness.

We consider a shared memory system with h identical processors. There are a set of n periodic real-time tasks, $\Gamma = \{T_1, \ldots, T_n\}$, where each task $T_i = (c_i, p_i)$ is characterized by its worst-case computation requirement c_i and its period p_i. Here, both c_i and p_i are assumed to be integer multiples of a system unit time. We consider real-time tasks with *implicit deadlines*. That is, p_i is also the relative deadline of task T_i. Assuming that the first *task instance* (or *job*) of each task arrives at time 0 (i.e., tasks are synchronous[2]), the j's task instance of task T_i arrives at time $(j-1)\cdot p_i$ and needs to complete its execution by its deadline at time $j \cdot p_i$ ($j \geq 1$).

The weight for task T_i is defined as $w_i = \frac{c_i}{p_i}$, and the system utilization is $U = \sum_{i=1}^{n} w_i$. Without loss of generality, we assume that $w_i < 1$ (note that there is $0 < w_i \leq 1$; if $w_j = 1$, we can dedicate one processor to task T_j and consider the smaller problem with the remaining tasks and the remaining processors). Moreover, it is assumed that $U = h$, the number of available processors. If $h - 1 < U < h$, we can add one dummy task $T_{n+1} = (c, p)$ such that $\sum_{i=1}^{n+1} w_i = h$. If $U \leq h-1$, we can just use $\lceil U \rceil$ processors [40]. For cases with $U > h$, the system will be overloaded and it is not possible to meet the deadlines of all tasks and will not be considered in this paper.

The *multiprocessor real-time scheduling problem* is to construct a *periodic schedule* for the above tasks, which allocates exactly c_i time units of a processor to task T_i within each interval $[(k-1)\cdot p_i, k \cdot p_i)$ for all $k \in \{1, 2, 3, \ldots\}$, subject to the following two constraints [13]:

- *C1:* A processor can only be allocated to one task at any time, that is, processors cannot be shared concurrently;
- *C2:* A task can only be allocated at most one processor at any time, that is, tasks are not parallel and thus cannot occupy more than one processor at any time.

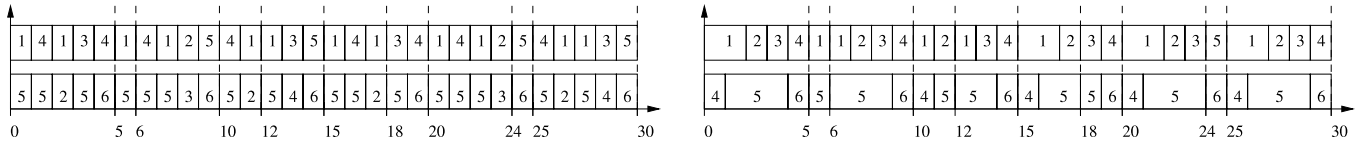
Assume that the *least common multiple* of all tasks' period is *LCM*. Because of the periodic property of the problem, we only consider the schedule from time 0 to time *LCM*. We define a set of period boundary time points as $B = \{b_0, \ldots, b_f\}$, where $b_0 = 0$, $b_f = LCM$ and $\forall c, \exists (i, k)$, $b_c = k \cdot p_i$ and $b_c < b_{c+1}$ ($c = 0, \ldots, f-1$). Define time unit (or slot) t as the real interval between time $t-1$ and time t (including $t-1$, excluding t), $t \in \{1, 2, 3, \ldots\}$. For convenience, we use $[b_k, b_{k+1})$ to denote time units between two boundaries, b_k and b_{k+1}, including time unit b_k and excluding time unit b_{k+1}. The *allocation error* of task T_i at boundary time b_k is defined as the difference between $b_k \cdot w_i$ and the time units allocated to T_i before b_k in a schedule. A schedule is *boundary fair* if and only if the absolute value of the allocation error for any task T_i at any boundary time b_k is less than one time unit.

Note that, if $U \leq h$, a proportional fair (Pfair) schedule is known to exist for the multiprocessor real-time scheduling problem [12]. From the above definition, we know that any Pfair schedule is also a boundary fair schedule (that is, a Pfair schedule also conforms to the allocation error requirements at boundaries). Therefore, we have the following lemma on the existence of a boundary fair schedule if $U \leq h$.

Lemma 1. *For the multiprocessor real-time scheduling problem, if the system utilization, U, is no bigger than h, the number of available processors, a boundary fair schedule exists.*

[1] It has been shown that Pfair schedules with such mapping constraints do exist [36], however, finding the efficient scheduling algorithm to get such Pfair schedule remains open.

[2] Although it is possible to have irregularly arrived tasks (such as sporadic tasks) in a system, it is well beyond the scope of this paper to consider such tasks due to the additional complexities involved to deal with such arrival irregularities [44]. In addition, it has been shown that, for multiprocessor real-time systems where tasks may arrive at any time, there is no optimal online scheduling algorithm [22].



a. A proportional fair schedule. b. A boundary fair schedule.

Fig. 1. Different fair schedules for an example task set $\Gamma = \{T_1(2, 5), T_2(3, 15), T_3(3, 15), T_4(2, 6), T_5(20, 30), T_6(6, 30)\}$ to be executed on a 2-processor system.

3.1. An example

To illustrate the idea of boundary fairness, we first consider an example task set that has 6 tasks: $T_1 = (2, 5)$, $T_2 = (3, 15)$, $T_3 = (3, 15)$, $T_4 = (2, 6)$, $T_5 = (20, 30)$, $T_6 = (6, 30)$. Here, $\sum_{i=1}^{6} w_i = 2$ and $LCM = 30$. Fig. 1a shows one proportional fair schedule generated by the Pfair scheduling algorithm [12], where the numbers in the rectangles denote the task numbers and the dotted lines are tasks' period boundaries. Note that this schedule is also a boundary fair schedule.

As mentioned earlier, due to the requirement of proportional progress (fairness) for all tasks, Pfair needs to make scheduling decisions at *every* time unit (i.e., there are 30 scheduling points for the Pfair scheduler). Moreover, from Fig. 1a, we can also see that the resulting Pfair schedule contains an excessive number of *context switches* and *task migrations*. Here, a context switch happens whenever a processor needs to execute a task that is different from the one it executed in the last time unit (e.g., there is a context switch on the top processor at time 2); a task migration happens when a task is executed on a different processor from the last one on which it runs (e.g., at time 9, task T_5 migrates from the bottom processor to the top processor). It can be easily found out that there are 52 context switches and 18 task migrations for the Pfair schedule within one LCM as shown in Fig. 1a.

Consider the *schedule section* between two consecutive period boundaries, for example, $[b_0, b_1) = [0, 5)$: here T_1 and T_4 get 2 time units each, T_2, T_3 and T_6 get 1 unit each and T_5 gets 3 units. If we follow the idea of McNaughton's algorithm [39] and pack tasks within this section on two processors *sequentially* (consecutively fill the time units on processors with tasks one by one), after T_1, T_2, T_3 are packed on the top processor, there is one time unit left and part of T_4's allocation (one time unit) is packed on the top processor; the rest of T_4's allocation (another time unit) is packed on the bottom processor followed by T_5 and T_6 as shown in Fig. 1b. Continuing the above process for the remaining schedule sections until *LCM*, we can get a boundary fair schedule within one *LCM*. From the figure, we can also see that, by aggregating the time allocation of tasks for the time interval between consecutive period boundaries, the resulting Bfair schedule has only 45 context switches and 9 task migrations, which is a dramatic reduction compared to those of the Pfair schedule in Fig. 1a. Such reduction of context switches and task migrations in the resulting schedule can significantly reduce run-time overhead, which is specially important for real-time systems.

Observing the fact that a deadline miss of a periodic real-time tasks can only happen at the end of the task's period, we propose a novel scheduling algorithm in this paper: at *any* boundary time point b_k ($k = 0, \ldots, f-1$), the algorithm allocates processors to tasks for the interval $[b_k, b_{k+1})$ *properly* to ensure that there is no deadline miss at the current as well as future boundary times. The details are discussed in the next section.

4. A boundary fair (Bfair) scheduling algorithm

The Bfair scheduling algorithm has the following high-level structure: at each boundary time, it allocates processors to tasks for time units between the current and the next period boundaries; each task T_i will have some *mandatory* integer time units that must be allocated to ensure fairness at the next boundary; if there are idle processors in some time slots after allocating the mandatory time units for every task, a dynamic priority is assigned to each of the *eligible* (as defined later) tasks and a few tasks with the highest priorities will get one *optional* time unit each. The key point in the Bfair algorithm is the allocation of optional time units based on tasks' priorities. Such priorities are derived by taking the future computation requirements of tasks into consideration, which guarantees that there is no deadline miss at not only the next but also the future boundary time points.

Before formally presenting the Bfair algorithm, we first give some definitions. The *remaining work* of task T_i at b_{k+1} after allocating the time units in the interval $[b_k, b_{k+1})$ is denoted as RW_i^{k+1} (which actually equals to the allocation error as defined in Section 3). The *mandatory* time units that have to be allocated to task T_i to keep its allocation error within one time unit (i.e., fairness) can be calculated as $m_i^{k+1} = \max\{0, \lfloor RW_i^k + (b_{k+1} - b_k) \cdot w_i \rfloor\}$, which is the *integer* part of the summation of the remaining work at b_k and the work to be done during $[b_k, b_{k+1})$. The *pending work* is the corresponding *decimal* part and denoted as $PW_i^{k+1} = RW_i^k + (b_{k+1} - b_k) \cdot w_i - m_i^{k+1}$. If task T_i gets an *optional* unit while allocating $[b_k, b_{k+1})$, we say that $o_i^{k+1} = 1$; otherwise $o_i^{k+1} = 0$. From these definitions, after allocating processors to tasks for the interval $[b_k, b_{k+1})$, we can get $RW_i^{k+1} = PW_i^{k+1} - o_i^{k+1}$.

Algorithm 1 The Bfair algorithm at b_k

```
1: for (T_1, ..., T_n) do
2:    /*allocate mandatory units for T_i */
3:    m_i^{k+1} = max{0, ⌊RW_i^k + (b_{k+1} − b_k) · w_i⌋};
4:    PW_i^{k+1} = RW_i^k + (b_{k+1} − b_k) · w_i − m_i^{k+1};
5: end for
6: RU = h · (b_{k+1} − b_k) − Σ m_i^{k+1};
7: /*allocate optional processor-time units if any*/
8: if ( RU > 0 ) then
9:    SelectedTaskSet = TaskSelection(RU); /*Pick up high priority tasks*/
10:   for (T_i ∈ SelectedTaskSet) do
11:      o_i^{k+1} = 1; /*allocate an optional unit for T_i*/
12:   end for
13:   for (T_i ∉ SelectedTaskSet) do
14:      o_i^{k+1} = 0;
15:   end for
16: else
17:   for (T_1, ..., T_n) do
18:      o_i^{k+1} = 0;
19:   end for
20: end if
21: for (T_1, ..., T_n) do
22:   RW_i^{k+1} = PW_i^{k+1} − o_i^{k+1};
23: end for
24: GenerateSchedule(b_k, b_{k+1});
```

Similar to the notations used in [12], at boundary time b_{k+1}, task T_i is said to be *ahead* if $RW_i^{k+1} < 0$, *punctual* if $RW_i^{k+1} = 0$ and *behind* if $RW_i^{k+1} > 0$. Moreover, for easy discussions, we define a task T_i to be *pre-behind*, *pre-punctual* and *pre-behind* at b_{k+1} if $PW_i^{k+1} > 0$, $PW_i^{k+1} = 0$ and $PW_i^{k+1} < 0$, respectively. Note that,

to ensure fairness and limit the allocation error of tasks within one time unit, no optional unit should be allocated to the pre-ahead and pre-punctual tasks. In addition, to satisfy the second constraint (C2) of the problem, the total allocated time units (including both mandatory and optional) for any task should not exceed the length of the interval under consideration. Therefore, after allocating mandatory time units, a task T_i is said to be *eligible* to compete for optional units if there are $PW_i^{k+1} > 0$ (it is pre-behind) and $m_i^{k+1} < b_{k+1} - b_k$ (it is not fully allocated). Note that, to ensure fairness, any eligible task can only get one optional unit.

The Bfair algorithm is presented in Algorithm 1. Initially, there are $RW_i^0 = 0$ $(i = 1, \ldots, n)$ at time 0. At time b_k, to allocate processors to tasks for the interval of $[b_k, b_{k+1})$, the algorithm first allocates mandatory units for each task T_i (lines 1 to 5). Then, the number of *remaining time units RU* is determined (line 6), which indicates the number of optional units that need to be allocated. If $RU > 0$, the first RU highest priority eligible tasks are selected with the help of the function *TaskSelection*() (which will be discussed in details next) and each of them will get an optional unit (lines 9 to 15). Otherwise, no task will get optional units (lines 17 to 19). After allocating all time units, the remaining work of tasks are updated (lines 21 to 23) and the schedule for the interval $[b_k, b_{k+1})$ is generated by the function *GenerateSchedule*(), which can sequentially pack tasks to processors in linear time following the idea of McNaughton's algorithm [39] (see Fig. 1b in Section 3).

As mentioned earlier, the key part of the Bfair algorithm is to select a few high priority eligible tasks to claim one optional unit each that can guarantee there is no deadline miss at the current and future boundary time points. To determine the priority of eligible tasks, we extend the idea in the Pfair algorithm [12] and define a *characteristic string* of task T_i at boundary time b_k as a finite string over $\{-, 0, +\}$:

$$\alpha(T_i, k) = \alpha_{k+1}(T_i)\alpha_{k+2}(T_i), \ldots, \alpha_{k+s}(T_i)$$

where $\alpha_k(T_i) = sign[b_{k+1} \cdot w_i - \lfloor b_k \cdot w_i \rfloor - (b_{k+1} - b_k)]$ and $s(\geq 1)$ is the minimal integer such that $\alpha_{k+s}(T_i) \neq$ '+'. From the definition, when $\alpha_{k+1}(T_i) =$ '+', it means that if task T_i becomes behind at boundary time b_{k+1} after allocating the interval of $[b_k, b_{k+1})$, T_i will be behind at b_{k+2} as well since it will not be eligible to compete for optional units with its mandatory units being $m_i^{k+2} = b_{k+2} - b_{k+1}$ (i.e., fully allocated) during the allocation of the next interval $[b_{k+1}, b_{k+2})$. In comparison, when $\alpha_{k+1}(T_j) =$ '−', even if task T_j does not get an optional unit and become behind at b_{k+1}, it will still be eligible to compete for optional units during the next interval since there will be $m_j^{k+2} < b_{k+2} - b_{k+1}$. Therefore, to ensure that there are enough eligible tasks to claim optional units during the allocation of the future intervals, tasks with '+' character should have higher priority than those with '−' character when competing for optional units during the allocation of an interval.

Moreover, for tasks with the '−' character at boundary time b_{k+s}, we further define the *urgency factor* of task T_i as $UF_i^k = \frac{1-(b_{k+s} \cdot w_i - \lfloor b_{k+s} \cdot w_i \rfloor)}{w_i}$, which indicates the minimal time needed for the task to collect enough work demand to receive one unit allocation and become punctual after b_{k+s}. Combined with the characteristic string, the priority for task T_i at time b_k is defined as a tuple $\eta_i^k = \{\alpha(T_i, k), UF_i^k\}$.

As shown in Algorithm 2, the function *Compare*(T_i, T_j) presents the basic steps to compare the priorities of two eligible tasks, which is needed by the function *TaskSelection*() (in Algorithm 1) to choose a few higher priority tasks to receive optional units. To compare tasks priorities, the tasks' characteristic strings are first compared, followed by the comparison of their urgency factors if necessary. When comparing the characteristic strings, the comparison is done by comparing characters starting from b_{k+1} until one task's character does not equal to '+' at the boundary time point b_{k+s} (lines 3 to 5). If there is a difference, the task with higher character (where '−' < '0' < '+') has higher priority; if both of them equal to '0', the task with smaller identifier is assumed to have higher priority; if both of them equal '−', the urgency factors are compared and the task with smaller urgency factor has higher priority; when there is still a tie, again the task with the smaller identifier is assumed to have higher priority. The correctness of the Bfair algorithm to meet all deadlines of tasks with such a priority comparison function is formally proved in Section 5.

Algorithm 2 The function *Compare*(T_i, T_j) at b_k

```
1:  /*Input: tasks T_i and T_j, where i < j;*/
2:  s = 1;
3:  while (α_{k+s}(T_i) = α_{k+s}(T_j) ='+') do
4:      s = s + 1;
5:  end while
6:  if (α_{k+s}(T_i) > α_{k+s}(T_j)) then
7:      return (T_i > T_j);
8:  else if (α_{k+s}(T_i) < α_{k+s}(T_j)) then
9:      return (T_i < T_j);
10: else if (α_{k+s}(T_i) = α_{k+s}(T_j) ='0') then
11:     return (T_i > T_j);
12: else if (UF_i^{k+1} > UF_j^{k+1}) then
13:     return (T_i < T_j);
14: else
15:     return (T_i > T_j);
16: end if
```

4.1. Complexity of the Bfair algorithm

Assume that the maximum period for all tasks is $p_{\max}$, that is, $p_{\max} = \max(p_i)$ $(i = 1, \ldots, n)$. In the function *Compare*(), there are at most $p_{\max}$ iterations of character comparison for any two tasks in the *WHILE* loop (lines 3 to 5 in Algorithm 2). This comes from the fact that, at the end of a period, the character for a task cannot be '+'. So, the complexity of the function *Compare*() is $O(p_{\max})$. Using any linear-comparison selection algorithm [14], *TaskSelection*() used in Algorithm 1 needs to make $O(n)$ calls to the function *Compare*() to decide which RU-subset of the eligible tasks to receive the optional units. Note that the function *GenerateSchedule*() (line 16 in Algorithm 1) has a complexity of $O(n)$ by sequentially packing all tasks onto processors. Therefore, the overall complexity of the Bfair algorithm is $O(n \cdot p_{\max})$, which is the same as that of the Pfair algorithm [12].

4.2. Constant time priority comparison

In this section, we present an improved priority comparison function *ConstantCompare*() that can compare two tasks' priorities in constant time (i.e., with the complexity of $O(1)$) and thus reduce the overall complexity of the Bfair algorithm. We also show the equivalence of the two comparison functions (that is, for any given pair of tasks, the constant time function always returns the same high priority task as that of *Compare*()).

Following the similar idea in [13], when the first character in the characteristic strings of two tasks T_i and T_j $(i < j)$ at boundary time b_k is $\alpha_{k+1}(T_i) = \alpha_{k+1}(T_j) =$ '+', instead of looking forward to future boundaries, we can derive the order of the tasks' priorities through the comparison of the priorities of their *counter tasks*. Here, a counter task CT_i of task T_i is constructed with the following timing parameters $(p_i - c_i, p_i)$. That is, the weight of CT_i is $\frac{p_i - c_i}{p_i} = 1 - w_i$, where w_i is task T_i's weight. From the definition of task's characteristic string, we can easily obtain the following lemma on the characters of a task T_i and its counter task CT_i.

Lemma 2. *If the first character in the characteristic string of a task T_i at boundary time b_k is a '+', the first character in the characteristic string of its counter task CT_i will be a '−'.*

Proof. Since $\alpha_{k+1}(T_i) =$ '+', there is:

$$b_{k+2} \cdot w_i - \lfloor b_{k+1} \cdot w_i \rfloor - (b_{k+2} - b_{k+1}) > 0$$
$$b_{k+2} \cdot w_i - \lfloor b_{k+1} \cdot w_i \rfloor > (b_{k+2} - b_{k+1}) \geq 1.$$

Therefore, for the counter task CT_i, we have

$$\begin{aligned} & b_{k+2} \cdot (1 - w_i) - \lfloor b_{k+1} \cdot (1 - w_i) \rfloor - (b_{k+2} - b_{k+1}) \\ & \quad = b_{k+2} - b_{k+2} \cdot w_i - b_{k+1} - \lfloor b_{k+1} \cdot w_i \rfloor - (b_{k+2} - b_{k+1}) \\ & \quad = -(b_{k+2} \cdot w_i - \lfloor b_{k+1} \cdot w_i \rfloor) < 0. \end{aligned}$$

That is, $\alpha_{k+1}(CT_i) =$ '−', which concludes the proof. □

From Lemma 2, we know that if the first character of the characteristic strings of two tasks is a '+', their corresponding counter tasks will have a '−' as the first character. Therefore, we can re-define the priority of a task T_i at boundary time b_k as $\beta_i^k = \{\alpha_{k+1}(T_i), NUF_i^k\}$, where the new urgency factor $NUF_i^k = UF_i^k$ if $\alpha_{k+1}(T_i) =$ '−'; otherwise, if $\alpha_{k+1}(T_i) =$ '+', we have $NUF_i^k = \frac{1-(b_{k+1}\cdot(1-w_i)-\lfloor b_{k+1}\cdot(1-w_i)\rfloor)}{1-w_i}$, which is actually the urgency factor of its counter task CT_i. Based on such new priority, the function *ConstantCompare*() that can compare two tasks in constant time is shown in Algorithm 3.

Note that, for cases where either task T_i or T_j does not have their first character as a '+', both functions *Compare*() and *ConstantCompare*() will operate on the same first characters (and the same urgency factors, if necessary) of the two tasks, which leads to the same result. Now, let us consider the case of both tasks T_i and T_j having their first character as a '+'. Note that, *a task will have the same punctual time as its counter task*. Therefore, if CT_i's urgency factor (i.e., NUF_i^k) is less than that of CT_j (i.e., NUF_j^k), it means that CT_i will become punctual earlier than CT_j, which in turn indicates that task T_j will run behind longer than T_i and become punctual later than T_i. That is, T_j will have higher priority than T_i (line 18 in Algorithm 3). Moreover, in this case, we will get either $\alpha_{k+s}(T_j) > \alpha_{k+s}(T_i) \neq$ '+' or $\alpha_{k+s}(T_j) = \alpha_{k+s}(T_i) =$ '−' with $UF_j^k < UF_i^k$ (where $s > 1$), and the same result $T_j > T_i$ will be returned by *Compare*(). Similar analysis can be applied to other cases. Therefore, for any two tasks T_i and T_j, the two priority comparison functions always return the same result.

With the above constant time priority comparison function and any linear-comparison selection algorithm (e.g., [14]), the complexity of our *BF* algorithm can be reduced to $O(n)$, which is comparable to that of the *PD* algorithm, $O(\min\{n, m \lg n\})$ [13]. The existence of the $O(n)$ Bfair algorithm is further confirmed independently in a recent paper [24].

4.3. Sample execution of the Bfair algorithm

Before we present the proof of the correctness of the Bfair algorithm, we illustrate the execution of Bfair for the example in Section 3.1. Note that, there are 10 boundary time points within [0, 30). The parameters used by the Bfair algorithm are calculated as shown in Table 1, where a '∗' means the corresponding item is not eligible or does not need to be calculated.

As we can see, initially, $RW_i^0 = 0, (i = 1, \ldots, 6)$. When allocating the first section (from b_0 to b_1), the mandatory units for each task are allocated in the first step, where $T_1, \ldots, T_6$ get 2, 1, 1, 1, 3, 1 units, respectively. Since $\sum_{i=1}^{6} m_i^1 = 9$ and the total available time units are $(b_1 - b_0) \cdot h = (5 - 0) \cdot 2 = 10$, there is 1 time unit left (i.e., $RU = 1$). To allocate it, one task is to be selected to receive an optional unit. Notice that there are 2 eligible tasks T_4 and T_5 (at time b_1, $PW_i^1 > 0$ and $m_i^1 < 5, i = 4, 5$), and their characteristic strings are $\alpha(T_4, 0) =$ '0' and $\alpha(T_5, 0) =$ '0'. Task T_4 has the highest priority (T_4 and T_5 have same character '0' at b_1, so the task with smaller identifier has higher priority) and will get an optional time unit. After that, the allocation for [0, 5) is complete and $RW_i^1 (i = 1, \ldots, 6)$ values are calculated accordingly. The schedule for the section [0, 5) will be generated by packing tasks to processors sequentially as shown in Fig. 1b (see Section 3).

Algorithm 3 The function *ConstantCompare*(T_i, T_j) at b_k

```
1: /*Input: tasks T_i and T_j where i < j;*/
2: if (α_{k+1}(T_i) > α_{k+1}(T_j)) then
3:    return (T_i > T_j);
4: else if (α_{k+1}(T_i) < α_{k+1}(T_j)) then
5:    return (T_i < T_j);
6: else if (α_{k+s}(T_i) = α_{k+s}(T_j) ='0') then
7:    return (T_i > T_j);
8: else if (α_{k+s}(T_i) = α_{k+s}(T_j) ='−') then
9:    if (NUF_i^{k+1} > NUF_j^{k+1}) then
10:      return (T_i < T_j);
11:   else
12:      return (T_i > T_j);
13:   end if
14: else if (α_{k+s}(T_i) = α_{k+s}(T_j) ='+') then
15:   if (NUF_i^{k+1} ≥ NUF_j^{k+1}) then
16:      return (T_i > T_j);
17:   else
18:      return (T_i < T_j);
19:   end if
20: end if
```

For section [5, 6), only T_5 gets one mandatory unit ($m_5^2 = \max\{0, \lfloor RW_5^1 + 2 \cdot w_5 \rfloor\} = 1$) and there is one additional unit to be allocated. T_1, T_2, T_3 and T_6 are eligible tasks because $PW_i^2 > 0$ and $m_i^2 < b_2 - b_1$ $(i = 1, 2, 3, 6)$. All these tasks have $\alpha(T_i, 1) =$ '−'. T_1 has the highest priority with the smallest urgency factor and will get an optional unit. These steps are repeated until after allocating section [25, 30), and we get a boundary fair schedule within the *LCM*. Note that the schedule generated by our Bfair algorithm is happened to be the same as shown Fig. 1b, which is also generated from the proportional fair schedule as explained in Section 3.1. However, compared to the Pfair scheduling algorithm that has 30 scheduling points [12], there are only 10 scheduling points for the Bfair algorithm. Furthermore, as mentioned earlier, the schedule generated by the Bfair algorithm (Fig. 1b) will also incur much less context switches and task migrations.

From the above example, we can also see that when allocating the processors in the interval $[b_k, b_{k+1})$:

- The summation of the mandatory units is less than or equal to the time units available (see Lemma 5): $\sum_{i=1}^{n} m_i^{k+1} \leq (b_{k+1} - b_k) \cdot h$;
- There are enough eligible tasks to claim the remaining units if any (see Lemma 6): *the number of eligible tasks* $\geq (b_{k+1} - b_k) \cdot h - \sum_{i=1}^{n} m_i^{k+1}$;
- After allocation, $\sum_{i=1}^{n} RW_i^{k+1} = 0$ (which means processors are fully allocated) and $\forall i | RW_i^{k+1} | < 1$ (which means the schedule is fair).

These observations will be used to present the correctness of the Bfair algorithm as shown in the next section.

5. Analysis of the Bfair algorithm

First, we recall that $PW_i^k = RW_i^{k-1} + (b_k - b_{k-1}) \cdot w_i - m_i^k$ (where $m_i^k = \max\{0, \lfloor RW_i^{k-1} + (b_k - b_{k-1}) \cdot w_i \rfloor\}$), and $RW_i^k = PW_i^k - o_i^k$.

Table 1
The execution of the Bfair algorithm for the example with six tasks $\Gamma = \{T_1(2,5), T_2(3,15), T_3(3,15), T_4(2,6), T_5(20,30), T_6(6,30)\}$ to be executed on a 2-processor system.

Time	0	5	6	10	12	15	18	20	24	25	30
b_k	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}
RW_1^k	0	0	−3/5	0	−1/5	0	−4/5	0	−2/5	0	0
RW_2^k	0	0	1/5	0	−3/5	0	−2/5	0	−1/5	0	0
RW_3^k	0	0	1/5	0	2/5	0	3/5	0	−1/5	0	0
RW_4^k	0	−1/3	0	1/3	0	0	0	−1/3	0	1/3	0
RW_5^k	0	1/3	0	−1/3	0	0	0	1/3	0	−1/3	0
RW_6^k	0	0	1/5	−0	2/5	0	3/5	0	4/5	0	0
m_1^k	*	2	0	1	0	1	1	0	1	0	2
m_2^k	*	1	0	1	0	0	0	0	0	0	1
m_3^k	*	1	0	1	0	1	0	1	0	0	1
m_4^k	*	1	0	1	1	1	1	0	1	0	2
m_5^k	*	3	1	2	1	2	2	1	3	0	3
m_6^k	*	1	0	1	0	1	0	1	0	1	1
PW_1^k	*	0	2/5	0	4/5	0	1/5	0	3/5	0	0
PW_2^k	*	0	1/5	0	2/5	0	3/5	0	4/5	0	0
PW_3^k	*	0	1/5	0	2/5	0	3/5	0	4/5	0	0
PW_4^k	*	2/3	0	1/3	0	0	0	2/3	0	1/3	0
PW_5^k	*	1/3	0	2/3	0	0	0	1/3	0	2/3	0
PW_6^k	*	0	1/5	0	2/5	0	3/5	0	4/5	0	0
$\alpha_k(T_1)$	*	–	–	–	–	–	–	–	0	–	–
$\alpha_k(T_2)$	*	–	–	–	–	–	–	–	0	–	–
$\alpha_k(T_3)$	*	–	–	–	–	–	–	–	0	–	–
$\alpha_k(T_4)$	*	0	–	–	–	–	–	–	–	–	–
$\alpha_k(T_5)$	*	0	–	0	–	–	–	–	–	–	–
$\alpha_k(T_6)$	*	–	–	–	–	–	–	–	0	–	-
UF_1^k	*	*	3/2	*	1/2	*	2	*	*	*	*
UF_2^k	*	*	4	*	3	*	2	*	*	*	*
UF_3^k	*	*	4	*	3	*	2	*	*	*	*
UF_4^k	*	*	*	*	*	*	*	1	*	2	*
UF_5^k	*	*	*	*	*	*	*	1	*	1/2	*
UF_6^k	*	*	4	*	3	*	2	*	*	*	*
o_1^k	*	0	1	0	1	0	1	0	1	0	0
o_2^k	*	0	0	0	1	0	1	0	1	0	0
o_3^k	*	0	0	0	0	0	0	0	1	0	0
o_4^k	*	1	0	0	0	0	0	1	0	0	0
o_5^k	*	0	0	1	0	0	0	0	0	1	0
o_6^k	*	0	0	0	0	0	0	0	0	0	0

In addition, for convenience, we define three task sets at boundary time b_k [12]:

$AS^k = \{T_i | RW_i^k < 0\}$: ahead task set at b_k;

$BS^k = \{T_i | RW_i^k > 0\}$: behind task set at b_k;

$PS^k = \{T_i | RW_i^k = 0\}$: punctual task set at b_k.

Furthermore, a task T_i is said to be *pre-ahead* at b_k if $PW_i^k < 0$, which means that even if T_i does not get any mandatory unit ($m_i^k = 0$; otherwise, there will be $PW_i^k \geq 0$, a contradiction) in $[b_{k-1}, b_k)$ it will still be ahead at b_k. The *pre-ahead task set* is defined as:
$PAS^k = \{T_i | PW_i^k < 0\}$.

A task T_i is said to be *pre-behind* at b_k if $PW_i^k > 0$ after allocating m_i^k, but it may "recover" after the allocation of optional units. The *pre-behind task set* is defined as:
$PBS^k = \{T_i | PW_i^k > 0\}$.

Notice that, if a task T_i is punctual after mandatory units allocation, it will not be eligible to get any optional unit and will still be punctual after optional units allocation; thus, there is no need to define a *pre-punctual task set*. Moreover, we define the *eligible task set* as:
$ES^k = \{T_i | (T_i \in PBS^k)$ AND $(m_i^k < b_k - b_{k-1})\}$.

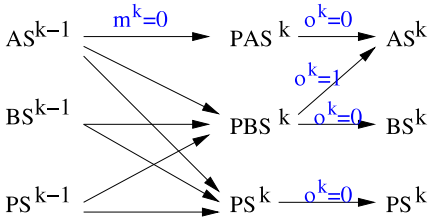


Fig. 2. Task transitions from b_{k-1} to b_k.

From these definitions, we can get task transitions between b_{k-1} and b_k as shown in Fig. 2. For example, $\forall T_i \in PAS^k$, T_i was ahead at b_{k-1} and got no mandatory unit. Moreover, T_i will get no optional unit (since it is not an eligible task) and still be ahead at b_k. That is, $PAS^k \subseteq AS^{k-1}$ and $PAS^k \subseteq AS^k$. For task $T_i \in AS^{k-1}$, it is also possible for T_i to have enough work demand during $[b_{k-1}, b_k)$ and belong to PS^k or PBS^k. For task $T_i \in PBS^k$, T_i may get an optional unit to become *ahead* or get no optional unit and remain *behind* at b_k.

From the Bfair algorithm, we can easily get the following properties of the defined task sets.

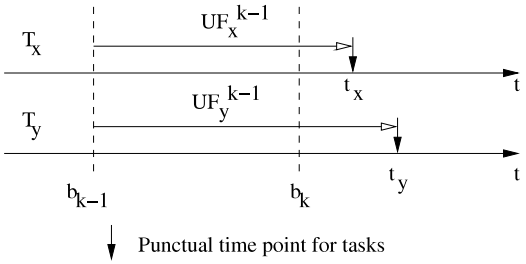


Fig. 3. Urgency factors for different tasks.

Property 1. *For the defined task sets:*

(a) *If* $T_i \in BS^{k-1}$ *and* $m_i^k = 0$, $T_i \in PBS^k$.
(b) *If* $\alpha_{k-1}(T_i) =$ '+' *and* $T_i \in AS^k$, $m_i^k + o_i^k = b_k - b_{k-1}$.
(c) $\forall T_i \in PAS^k$, $m_i^k = o_i^k = 0$ *and* $RW_i^{k-1} < RW_i^k < 0$.
(d) *If* $T_i \in BS^{k-1}$ *and* $m_i^k = o_i^k = 0$, $T_i \in BS^k$ *and* $0 < RW_i^{k-1} < RW_i^k$.
(e) *If* $T_i \in AS^k$ *and* $m_i^k = o_i^k = 0$, $T_i \in AS^{k-1}$ *and* $\alpha_{k-1}(T_i) =$ '−'.
(f) *If* $T_i \in BS^k$ *and* $m_i^k = b_k - b_{k-1}$, $T_i \in BS^{k-1}$ *and* $\alpha_{k-1}(T_i) =$ '+'.
(g) *If* $T_i \in AS^{k-1} \cap AS^k$ *and* $m_i^k + o_i^k = b_k - b_{k-1}$, $RW_i^k < RW_i^{k-1} < 0$.
(h) *If* $T_i \in BS^{k-1} \cap BS^k$ *and* $m_i^k + o_i^k = b_k - b_{k-1}$, $0 < RW_i^k < RW_i^{k-1}$. □

Note that, for task $T_x \in PAS^k \subseteq AS^k$, from Property 1c and e, we have $m_x^k = o_x^k = 0$ and $\alpha_{k-1}(T_x) =$ '−'. If T_x gets an optional unit during last iteration when allocating $[b_{k-2}, b_{k-1})$ (i.e., $o_x^{k-1} = 1$), for any task T_y ($x \neq y$) that is behind at b_{k-1} and is not fully allocated during last iteration (i.e., $T_y \in ES^{k-1}$), from the *BF* algorithm, we have that T_y's priority is lower than that of T_x; that is, $\alpha_{k-1}(T_y) = \alpha_{k-1}(T_x) =$ '−' and T_y's urgency factor (UF_y^{k-1}) is bigger than or equal to that of T_x (UF_x^{k-1}). Since $T_x \in PAS^k \subseteq AS^k$, we have $UF_x^{k-1} > b_k - b_{k-1}$ (otherwise, there will be $PW_x^k \geq 0$ and $T_x \notin PAS^k$, a contradiction).

This scenario is further illustrated in Fig. 3, where t_x and t_y are the nearest punctual time points after b_{k-1} for T_x and T_y, respectively. Recall that, the urgency factor is the minimal time needed for a task to collect enough work and become punctual after b_{k-1}. We have $UF_y^{k-1} = t_y - b_{k-1} \geq UF_x^{k-1} = t_x - b_{k-1} > b_k - b_{k-1}$, that is, $t_y \geq t_x > b_k$. Thus, we have Lemma 3.

Lemma 3. *If* $\exists T_x \in PAS^k$ *and* $o_x^{k-1} = 1$, *then* $\forall T_y \in (ES^{k-1} \cap BS^{k-1})$, *there are* $m_y^k = 0$, $\alpha_{k-1}(T_y) = \alpha_{k-1}(T_x) =$ '−' *and* $UF_y^{k-1} \geq UF_x^{k-1} > b_k - b_{k-1}$. □

The above result can be extended to across more than one boundary intervals. That is, suppose that a task T_x receives an optional unit and becomes ahead at the boundary time b_{k-s-1}. If such a task T_x belongs to the pre-ahead task sets at all the following the boundary time points until b_k, then for any eligible task T_y that has a lower priority than that of T_x and does not get an optional unit during the interval of $[b_{k-s-2}, b_{k-s-1})$, it will not get any mandatory unit during the following intervals until time b_k. More formally, we can get the following Lemma 4.

Lemma 4. *If* $\exists T_x \in (PAS^k \cap PAS^{k-1} \cap \cdots \cap PAS^{k-s})$ $(s \geq 1)$ *and* $o_x^{k-s-1} = 1$, *then* $\forall T_y \in (ES^{k-s-1} \cap BS^{k-s-1})$, *there are* $m_y^{k-s} = m_y^{k-s+1} = \cdots = m_y^k = 0$, $\alpha_{k-s-1}(T_y) = \alpha_{k-s-1}(T_x) =$ '−' *and* $UF_y^{k-s-1} \geq UF_x^{k-s-1} > b_k - b_{k-s-1}$. □

To prove that the Bfair algorithm correctly generates a boundary fair schedule, first we show that two conditions are always satisfied during the allocation of $[b_{k-1}, b_k)$: (1) the summation of all tasks' mandatory integer units is no more than the available time units on the h processors; (2) there are enough eligible tasks to claim any available optional units. The proof for these conditions is by contradiction, that is, if any one of these two conditions is not met, we can show that there will be at least one task ahead and one task behind in every one of the previous boundaries; this will contradict with the fact that there is at least one boundary (i.e., b_0) at which every task is punctual. The conditions are formally presented in the following two lemmas and their proofs are in the Appendix.

Lemma 5. *If* $\sum_i w_i = h$ *and Algorithm 1 is followed at boundary time* $b_0, \ldots, b_{k-1}$ *and for* $v = 0, \ldots, k-1$, $\sum_i RW_i^v = 0$ *and* $|RW_i^v| < 1$, *then when allocating processors in* $[b_{k-1}, b_k)$, *we have* $\sum_i m_i^k \leq (b_k - b_{k-1}) \cdot h$. □

Lemma 6. *If* $\sum_i w_i = h$ *and Algorithm 1 is followed at boundary time* $b_0, \ldots, b_{k-1}$ *and for* $v = 0, \ldots, k-1$, $\sum_i RW_i^v = 0$ *and* $|RW_i^v| < 1$, *then when allocating processors in* $[b_{k-1}, b_k)$, *we have* $|ES^k| \geq (b_k - b_{k-1}) \cdot h - \sum_i m_i^k$; *that is, the number of eligible tasks is no less than the number of remaining units (RU) to be allocated.* □

From Lemmas 3 to 6, we can get the following theorem for Algorithm 1.

Theorem 1. *The schedule generated by Algorithm 1 is boundary fair, that is, at boundary time* b_k *(after allocating* $[b_{k-1}, b_k)$*), we have* $\sum_i RW_i^k = 0$ *and* $|RW_i^k| < 1$ $(i = 1, \ldots, n)$.

Proof. The proof is by induction on boundary time b_k.

Base case: At time b_0, we have $RW_i^0 = 0$, $i = 1, \ldots, n$, that is, $\sum_i RW_i^0 = 0$ and $|RW_i^0| < 1$.

Induction step: Assume that for boundary time $b_0, \ldots, b_{k-1}$, we have $\sum_i RW_i^v = 0$ and $|RW_i^v| < 1$ $(v = 0, \ldots, k-1, i = 1, \ldots, n)$.

When allocating $[b_{k-1}, b_k)$, from Lemmas 5 and 6, the following two conditions are satisfied:

(1) $\sum_i m_i^k \leq (b_k - b_{k-1}) \cdot h$; and
(2) $|ES^k| \geq (b_k - b_{k-1}) \cdot h - \sum_i m_i^k$.

After allocating m_i^k, task T_i will belong to one of the sets in the middle column of Fig. 2. Below we consider the four possible transitions (arrows from the middle sets to the sets on the right):
$\forall T_i \in PAS^k \cap AS^k$, $-1 < RW_i^k = PW_i^k < 0$;
$\forall T_i \in PBS^k \cap AS^k$, $-1 < RW_i^k = PW_i^k - 1 < 0$;
$\forall T_i \in PBS^k \cap BS^k$, $0 < RW_i^k = PW_i^k < 1$;
$\forall T_i \in PS^k$, $RW_i^k = PW_i^k = 0$.
Hence, $\forall T_i$, $|RW_i^k| < 1$. Next,

$$\begin{aligned}\sum_i RW_i^k &= \sum_i (PW_i^k - o_i^k) \\ &= \sum_i (RW_i^{k-1} + (b_k - b_{k-1}) \cdot w_i - m_i^k - o_i^k) \\ &= 0 + (b_k - b_{k-1}) \cdot h - \sum_i (m_i^k + o_i^k).\end{aligned}$$

Since $\sum_i (m_i^k + o_i^k) = (b_k - b_{k-1}) \cdot h$, we will have, at time b_k, $\sum_i RW_i^k = 0$.

Thus, the schedule generated by the Bfair algorithm in Algorithm 1 is boundary fair, which concludes the proof. □

As we noted above, a boundary fair schedule maintains fairness for tasks at the boundaries, which means that there is no deadline miss and the Bfair algorithm generates a feasible schedule. Moreover, the Bfair algorithm is optimal in the sense that it utilizes 100% of the processors in a system.

6. Simulations and discussions

In this section, we will experimentally evaluate the performance of the Bfair algorithm (denoted by *BF*) on reducing the number of scheduling points as well as the overall scheduling overhead. Note that, both *T*–*L* plane based algorithms [15,16,23] and DP-Fair [34] are based on continuous time, which are not comparable to the quantum-time-based Bfair algorithm. Moreover, other recent semi-partition based scheduling algorithms [26,30–32] cannot achieve full system utilization, which are not comparable as well. Therefore, for meaningful comparison, we implemented three quantum-time-based Pfair algorithms: the original algorithm *PF* [12], an improved algorithm *PD* [13] and the most efficient algorithm PD^2 [3].

For the Bfair (*BF*) algorithm, eligible tasks are first divided into three categories based on their characters (corresponding to '+', '0' and '−') for the current boundary time. Tasks are selected from high to low priority categories, which effectively reduces the number of priority comparison needed. If not all tasks in a category can be selected, a simple $O(k \cdot n)$ task selection function is used, where k is the number of tasks needed in that category. Here, we implement the constant time priority comparison function that is utilized by the task selection process as described in Section 4. The original Pfair (*PF*) algorithm is implemented as described in [12]. For the *PD* algorithm, tasks are first divided into 7 priority categories with the complexity of $O(n)$ [13]; then the same task selection approach as that of *BF* is exploited. We also implement the constant time priority comparison function for *PD*. While the complexity of implemented *PD* algorithm is still $O(h \cdot n)$, where h is the number of processors, its performance (as shown below) is almost linear with the number of tasks. For the PD^2 algorithm, the window of tasks is generated online and the group deadline of each window is computed with a constant time approach following the scheme in [3].

Moreover, in our experiments, each task set contains 20 to 100 tasks. The period for a task is uniformly distributed within the minimum period $p_{\min}$ and the maximum period $p_{\max}$. We vary the values of $p_{\min}$ and $p_{\max}$ from 10 to 100. However, due to limitation in the simulations, we consider only the task sets with $LCM < 2^{32}$. Note that, for task sets with $LCM > 2^{32}$, Bfair will actually perform better as the number of scheduling points for Pfair algorithms increases much faster than that of the Bfair algorithm. For each data point in the following figures, the result is the average of 100 randomly generated task sets.

6.1. Number of scheduling points

First, we show that Bfair can significantly reduce the required scheduling points compared to that of Pfair algorithms. Note that, as Pfair algorithms need to make scheduling decisions at each time unit, the number of scheduling points for Pfair algorithms is essentially equal to LCM of the tasks' periods. However, the scheduling points for Bfair are only the period boundaries of all tasks. For easy presentation, we report the *normalized* number of scheduling points, which is the ratio of the number of Bfair scheduling points to that of Pfair algorithms.

Fig. 4 shows the results. Here, with a fixed minimum task period $p_{\min} = 10$, we vary the maximum period $p_{\max}$ from 20 to 100 (as shown in the *X*-axis) and *Y*-axis represents the normalized number of scheduling points. From the figure, we can see that the number of scheduling points of the Bfair algorithm varies from 25% to 48% of that for the Pfair algorithms. For a given maximum period, when there are more tasks in a task set, a time point is more likely to be a period boundary and there are more scheduling points for the Bfair algorithm. Note that, for a task set with fixed number of tasks, the periods of tasks are more separated with

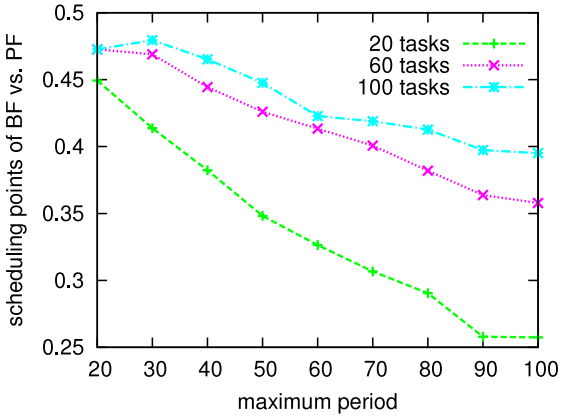


Fig. 4. The number of scheduling points with varying $p_{\max}$; $p_{\min} = 10$.

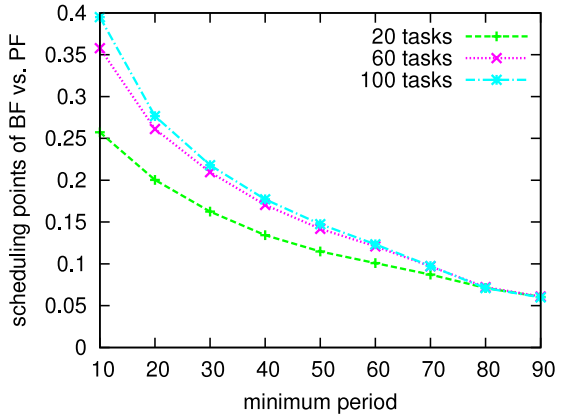


Fig. 5. The number of scheduling points with varying $p_{\min}$; $p_{\max} = 100$.

larger values of $p_{\max}$. Therefore, for larger values of $p_{\max}$, there are fewer number of period boundaries and thus fewer number scheduling points for the Bfair algorithm. When the maximum period is fixed at $p_{\max} = 100$, Fig. 5 further shows the normalized number of scheduling points for the Bfair algorithm when the minimum period of tasks $p_{\min}$ varies from 10 to 90. For the larger values of $p_{\min}$, the periods of tasks become more regular and the number of scheduling points become much less. For the case of $p_{\min} = 90$, only 6% of the time points within LCM are period boundaries (and thus scheduling points) even for task sets with 100 tasks. That is, the Bfair algorithm can save up to 94% of the scheduling points for the task sets considered.

6.2. Time overhead of Bfair and Pfair algorithms

Next, we compare the run-time overhead of the Bfair algorithm with that of the Pfair algorithms by measuring their execution times at each scheduling point as well as the overall execution times to generate the whole schedule within LCM. The algorithms are implemented in C and run on a Linux box with two Intel quad-core 2.4 GHz processors and 16 GB memory.

Fig. 6 shows the execution time (on average, in microsecond) of the algorithms at each scheduling point for task sets with different number of tasks. Here, we set $p_{\min} = 10$ and $p_{\max} = 100$. For the worst execution time of a task (c_i), it is uniformly distributed between 1 and its period (p_i). Therefore, on average, the utilization of tasks is around 0.5 and the number of processors is around half of the number of tasks. Note that, the execution time of *PF* is much more than the other algorithms and increases very quickly as the number of tasks increases. For the cases with 100 tasks, *PF* uses more than 9 times execution time than that of the other

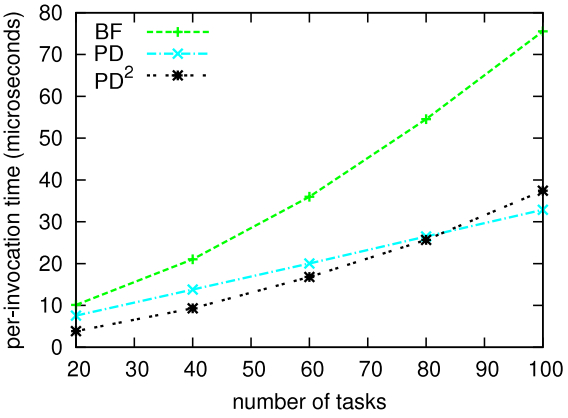


Fig. 6. Execution time (in microseconds) at each scheduling point.

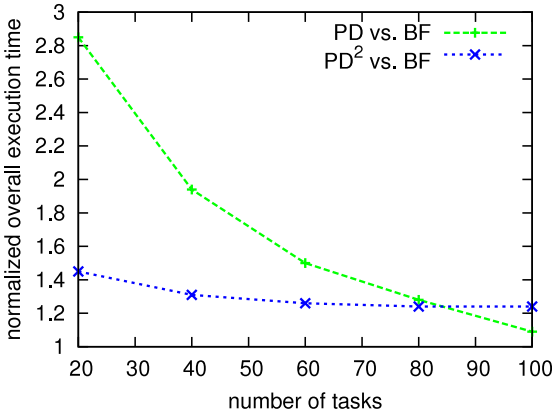


Fig. 7. Normalized overall execution time to generate the schedule within LCM.

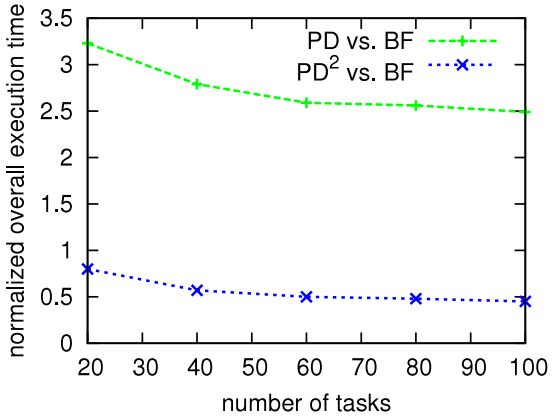


Fig. 8. Normalized overall execution time to generate the schedule within LCM for light tasks.

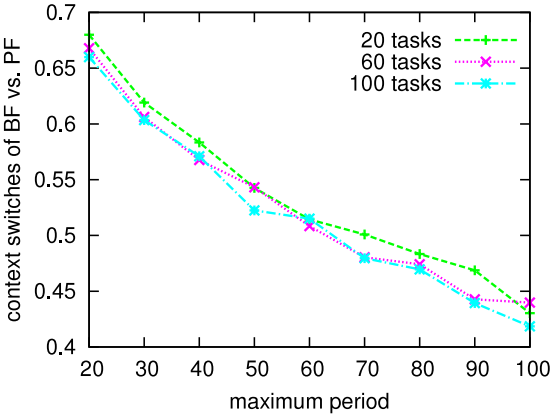


Fig. 9. Normalized number of context switches with varying p_{max}; $p_{min} = 10$.

algorithms, which is not shown in the figure for clear illustration of other algorithms. From the figure, we can see that all algorithms only take a few microseconds for each scheduling points when the number of tasks is 20. Moreover, both *PD* and PD^2 perform better than the Bfair (*BF*) algorithm and use less time at each scheduling point. When the number of tasks increases, the execution time of *PD* and PD^2 increases almost linearly while the execution time of *BF* increases slightly fast. The reason comes from the increased number of optional units that take more time to be allocated in *BF* algorithms. In addition, PD^2 performs slightly better than *PD* when the number of tasks is less than 80, however it takes more time than that of *PD* when more tasks are in a task set. The possible reason is related to the different approach on handling of *heavy tasks* (with utilization more than 0.5) in PD^2, which requires the computation of window and group deadlines. When the number of such tasks increases, PD^2 needs more time to handle them at each scheduling point.

Although *BF* takes more time at each scheduling point, as shown earlier, there are much less scheduling points for *BF* within one LCM. Fig. 7 further shows the normalized execution time (on average) for *PD* and PD^2 to generate the whole schedule within LCM of tasks' periods, where the overall execution time of *BF* is used as the baseline. Again, as *PF* takes too much time when compared with other algorithms, its overall execution time is not shown in the figure for easy comparison of others. Here, we can see that, for all the cases considered, *PD* and PD^2 take more time (on average) to generate the whole schedules when comparing to *BF*. However, as the number of tasks increases, the difference between the overall execution time of *PD* and *BF* become smaller. There are two reasons: first, there are more scheduling points for *BF* when the number of tasks increases; second, the execution time of *BF* at each scheduling point increases faster than that of *PD* with more tasks being in a task set.

Note that, the performance of PD^2 depends closely on the characteristics of the tasks in a task set. When task sets contain only *light tasks* (with utilization less than 0.5), PD^2 performs much better as it does not need to calculate the group deadlines for heavy tasks anymore. The normalized overall execution time for such task sets is further shown in Fig. 8. Here, we can see that PD^2 only needs 50% of the time to generate the whole schedule. However, for task sets with only heavy tasks, both *PD* and PD^2 perform much worse as more units need to be allocated for the same number of tasks. See [46] for more detailed results.

6.3. Number of context switches and task migrations

In addition to less run-time overhead, as shown in the example in Section 3.1, by aggregating the time allocation of tasks together for the time interval between consecutive period boundaries, the schedule generated by the Bfair algorithm can also reduce the number of required context switches as well as task migrations. Note that, all Pfair algorithms enforce proportional progress for tasks at every time unit. Therefore, we expect that the resulting Pfair schedules will be roughly the same and we consider only the one generated by *PF*. In what follows, for randomly generated task sets with different number of tasks, we experimentally compare the number of context switches and task migrations for the schedules generated by the *BF* and *PF* algorithms.

For fixed $p_{min} = 10$, Fig. 9 first shows the normalized number of context switches (represented by the *Y*-axis) for the schedule of *BF* when the one of *PF* is used as the baseline with varying p_{max}

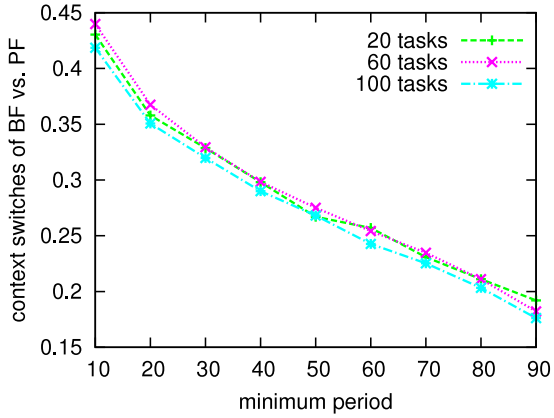


Fig. 10. Normalized number of context switches with varying p_{min}; $p_{max} = 100$.

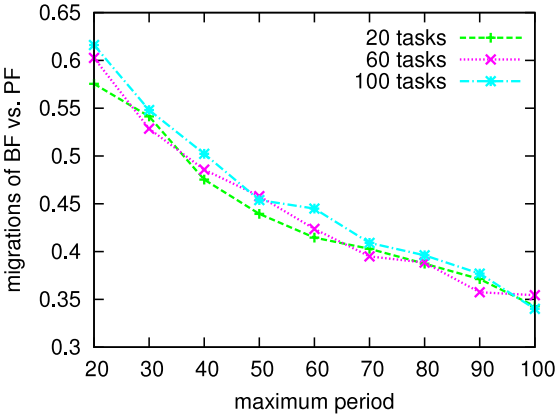


Fig. 11. Normalized number of task migrations with varying p_{max}; $p_{min} = 10$.

(represented by the X-axis). From the figures, we can see that, the normalized number of context switches generally decreases as p_{max} increases. The reason is that, as p_{max} increases, there are fewer number of period boundaries within LCM of tasks' periods (see Fig. 4) and the time interval between consecutive boundaries becomes larger, which provides better opportunities for tasks to aggregate their time allocations and thus leads to reduced number of context switches. For task sets with different number of tasks, we can also see that the normalized number of context switches is roughly the same for a given value of p_{max}. The reason comes from the average utilization of tasks, which is around 0.5 as mentioned earlier. In this case, for task sets with more tasks, more processors will be deployed, which results in increased number of context switches for both *BF* and *PF*, but in roughly the same rate. Therefore, the normalized number of context switches stays roughly the same for task sets with different number of tasks with a given pair of p_{min} and p_{max}.

The same reasonings apply to the case of varying p_{min} with fixed $p_{max} = 100$ as shown in Fig. 10. From these results, we can see that, even with $p_{min} = 10$ and $p_{max} = 100$, there are only 44% of context switches in the schedules generated by *BF* compared to that of the *PF*. For the case of $p_{min} = 90$ and $p_{max} = 100$, the periods of tasks are more regular and the time interval between consecutive boundaries is larger, the normalized number of context switches for the schedules of *BF* is only 18% of that for *PF*. That is, up to 82% of context switches can be saved, which will be very helpful to reduce the run-time overhead of real-time systems.

As another metric, Figs. 11 and 12 further show the normalized number of task migrations required for the resulting Bfair schedule when compared to that of the Pfair schedule. With the same reasonings as those for context switches, we can see that the

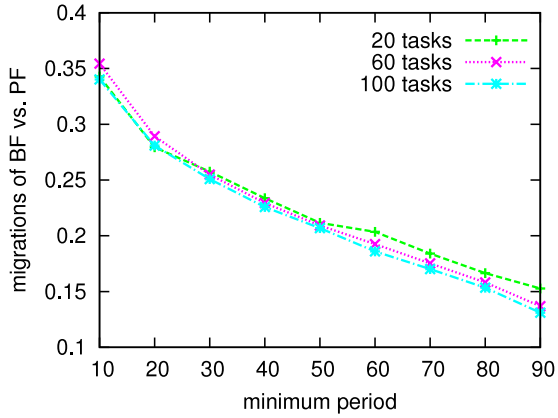


Fig. 12. Normalized number of task migrations with varying p_{min}; $p_{max} = 100$.

number of task migrations is also significantly reduced, up to 85% for the case with $p_{min} = 90$ and $p_{max} = 100$. Such reduction is very important to reduce the run-time overhead of real-time systems, especially considering the cache effects (e.g., due to cold start) of task migrations.

7. Conclusions

In this paper, we present a *novel* optimal scheduling algorithm, *boundary fair (Bfair)*, for multiprocessor real-time systems. Unlike its predecessor, the *Pfair* scheduling [12], which makes scheduling decisions at every time unit to ensure proportional progress for all tasks at *any* time, our Bfair scheduling algorithms makes scheduling decisions and maintains fairness for tasks *only* at the period boundaries, which effectively reduces the number of scheduling points compared to that of the Pfair algorithms. Moreover, by aggregating the time allocation of tasks for the time interval between consecutive period boundaries, the resulting Bfair schedule needs dramatically reduced number of context switches and task migrations, which are very important to reduce the run-time overhead for real-time systems.

The correctness of the Bfair algorithm to meet the deadlines of all tasks' instances is formally proved and the run-time performance of Bfair is evaluated through extensive simulations. The results show that, compared to that of the Pfair algorithms, Bfair can significantly reduces the number of scheduling points (up to 94%) and the time overhead of Bfair for each scheduling point is comparable to that of the most efficient Pfair algorithm, PD^2 [3]. For task sets with fewer number (e.g., 20) of tasks, Bfair use much less time to generate the whole schedule within LCM when compared to *PD* and PD^2. However, for task sets with more (e.g., 100) tasks where more scheduling points exist for the Bfair algorithm, PD^2 uses around half of the time to generate the whole schedule when compared to Bfair. Moreover, for the number of context switches and task migrations, the resulting Bfair schedule can reduce them by 82% and 85%, respectively, when compared to those of Pfair schedules.

For our future work, we will investigate how to extend the *BF* algorithm for sporadic tasks as well as systems with heterogeneous processors that have different processing speed.

Acknowledgments

First, the authors would like to thank the reviewers for their insightful comments and valuable suggestions to help improve the quality of this paper. In addition, the authors want to thank Geoffrey Nelissen for his help to identify the gaps and problems in the original proof of the Bfair algorithm. Moreover, the authors

thank US National Science Foundation for the support of this work through the awards CNS-0855247, CNS-1016974, and CAREER award CNS-0953005. A preliminary version of this paper has been presented at the 24th IEEE Real-Time System Symposium (RTSS) in 2003 [45].

Appendix. The proofs of Lemmas 5 and 6

Lemma 5. *If $\sum_i w_i = h$ and Algorithm 1 is followed at boundary time $b_0, \ldots, b_{k-1}$ and for $v = 0, \ldots, k-1$, $\sum_i RW_i^v = 0$ and $|RW_i^v| < 1$, then when allocating processors in $[b_{k-1}, b_k)$, we have $\sum_i m_i^k \le (b_k - b_{k-1}) \cdot h$.*

Proof. The proof is by contradiction, that is, if the equation does not hold, we will show that both *ahead set* and *behind set* are not empty for each of the previous boundaries, which contradicts the fact that there is at least one boundary (i.e., b_0) in which every task is punctual.

Suppose $\sum_i m_i^k > (b_k - b_{k-1})\cdot$ By assumption, $\sum_i RW_i^{k-1} = 0$ and $\sum_i w_i = h$. Thus:

$$\sum_i PW_i^k = \sum_i (RW_i^{k-1} + (b_k - b_{k-1}) \cdot w_i - m_i^k) = (b_k - b_{k-1}) \cdot h - \sum_i m_i^k \le -1.$$

Define two task sets *PWAS* (possibly-wrong ahead set) and *PWBS* (possibly-wrong behind set) for boundary b_{k-1} as follows:

$PWAS^{k-1} = \{T_x | T_x \in PAS^k\}$;
$PWBS^{k-1} = \{T_y | (T_y \in BS^{k-1})\ AND\ (m_y^k \ge 1)\}$.

Notice that, both $PWAS^{k-1}$ and $PWBS^{k-1}$ are not empty. Otherwise, if $PWAS^{k-1} = \emptyset$, we have $\sum_i PW_i^k \ge 0$, which contradicts $\sum_i PW_i^k \le -1$. If $PWBS^{k-1} = \emptyset$, we have $\forall T_i \in BS^{k-1}$, $m_i^k = 0$. From Property 1a, $T_i \in PBS^k$ and $BS^{k-1} \subseteq PBS^k$. Therefore:

$$\sum_{T_i \in PBS^k} PW_i^k \ge \sum_{T_i \in BS^{k-1}} PW_i^k > \sum_{T_i \in BS^{k-1}} RW_i^{k-1} > 0.$$

Notice that $PAS^k \subseteq AS^{k-1}$, therefore:

$$\sum_{T_i \in AS^{k-1}} RW_i^{k-1} < \sum_{T_i \in PAS^k} RW_i^{k-1} < \sum_{T_i \in PAS^k} PW_i^k < 0.$$

By assumption, $\sum_i RW_i^{k-1} = 0$. Since $\forall T_i \in PS^{k-1}$, $RW_i^{k-1} = 0$, thus $\sum_{T_i \in AS^{k-1}} RW_i^{k-1} = -\sum_{T_i \in BS^{k-1}} RW_i^{k-1}$. Therefore:

$$\sum_{T_i \in PBS^k} PW_i^k > \sum_{T_i \in BS^{k-1}} RW_i^{k-1} = -\sum_{T_i \in AS^{k-1}} RW_i^{k-1} > -\sum_{T_i \in PAS^k} PW_i^k$$

hence $\sum_i PW_i^k > 0$ that contradicts $\sum_i PW_i^k \le -1$.

So, both $PWAS^{k-1}$ and $PWBS^{k-1}$ are not empty. From Lemma 3 and the Bfair algorithm, we will get either:

(a) $\forall T_x \in PWAS^{k-1}$, $T_x \in PAS^{k-1}$; or
(b) $\forall T_y \in PWBS^{k-1}$, $m_y^{k-1} = b_{k-1} - b_{k-2}$.

Otherwise, $\exists T_x \in PWAS^{k-1}$ and $\exists T_y \in PWBS^{k-1}$ such that $T_x \in PBS^{k-1}$, $o_x^{k-1} = 1$ and $m_y^{k-1} < b_{k-1} - b_{k-2}$. From Lemma 3, there will be $UF_y^{k-1} > UF_x^{k-1} > b_k - b_{k-1}$. Notice that from the definition of $PWAS^{k-1}$ and $PWBS^{k-1}$, we have $UF_y^{k-1} < UF_x^{k-1}$, which is a contradiction.

Below, we extend the construction of the non-empty sets PWAS and PWBS to earlier boundaries. Recall that our intuition behind this proof is that there will at least one boundary (e.g. b_0) in which every task is punctual.

If (a) is true, $\forall T_x \in PWAS^{k-1}$, $T_x \in PAS^{k-1}$. Define:

$$PWAS^{k-2} = \{T_x | T_x \in PWAS^{k-1}\} \ne \emptyset;$$
$$PWBS^{k-2} = \left\{ T_y | (T_y \in BS^{k-2})\ AND\ \left(\sum_{l=k-1}^{k} (m_y^l + o_y^l) > 0 \right) \right\}.$$

Suppose $PWBS^{k-2} = \emptyset$, that is, $\forall T_y \in BS^{k-2}$, $\sum_{l=k-1}^{k}(m_y^l + o_y^l) = 0$. From Property 1a and d, $T_y \in PBS^k$ and $BS^{k-2} \subseteq PBS^k$. Since $PWAS^{k-2} = PWAS^{k-1} = PAS^k \subseteq PAS^{k-1} \subseteq AS^{k-2}$, from Property 1c, we have $\forall T_x \in PWAS^{k-2}$, $m_x^{k-1} = o_x^{k-1} = m_x^k = o_x^k = 0$. Therefore:

$$\sum_{T_y \in PBS^k} PW_i^k > \sum_{T_y \in BS^{k-2}} PW_i^k > \sum_{T_y \in BS^{k-2}} RW_i^{k-1} > \sum_{T_y \in BS^{k-2}} RW_i^{k-2} = -\sum_{T_x \in AS^{k-2}} RW_i^{k-2} \ge -\sum_{T_x \in PAS^{k-1}} RW_i^{k-2} \ge -\sum_{T_x \in PAS^k} RW_i^{k-2} > -\sum_{T_x \in PAS^k} PW_i^k$$

that is, $\sum_i PW_i^k > 0$, which is a contradiction.

So, both $PWAS^{k-2}$ and $PWBS^{k-2}$ are not empty. Similarly, this will lead to either:

(i) $\forall T_x \in PWAS^{k-2}$, $T_x \in PAS^{k-2}$; or
(ii) $\forall T_y \in PWBS^{k-2}$, $m_y^{k-2} = b_{k-2} - b_{k-3}$.

If (b) is true, $\forall T_y \in PWBS^{k-1}$, $m_y^{k-1} = b_{k-1} - b_{k-2}$. From Property 1f, $T_y \in BS^{k-2}$ and $\alpha_{k-2}(T_y) =$ '+'. Define:

$$PWAS^{k-2} = \{T_x | (T_x \in AS^{k-2})\ AND\ (\exists T_y \in PWBS^{k-2}, \alpha(T_x, k-3) < \alpha(T_y, k-3))\};$$
$$PWBS^{k-2} = PWBS^{k-1} \ne \emptyset.$$

If $PWAS^{k-2} = \emptyset$, then $\forall T_x \in AS^{k-2}$ and $\forall T_y \in PWBS^{k-2}$, $\alpha(T_x, k-3) \ge \alpha(T_y, k-3)$. Thus $\alpha_{k-2}(T_x) = \alpha_{k-2}(T_y) =$ '+'. Since $m_y^k \ge 1$, whatever the value of $\alpha_{k-1}(T_x)$ is, we will have $T_x \in (PBS^k \cup PS^k)$. Notice that $PAS^k \ne \emptyset$, we have $\exists T_z \in ((BS^{k-2} - PWBS^{k-2}) \cup PS^{k-2})$, $T_z \in PAS^k \subseteq AS^{k-1}$ (i.e., $o_z^{k-1} = 1$). Note that $\alpha_{k-2}(T_z) <$ '+' (otherwise, $T_z \in PWBS^{k-2}$, a contradiction). Since $\forall T_x \in AS^{k-2}$, $\alpha_{k-2}(T_x) =$ '+' (i.e., T_x's priority is higher than T_z's) and $m_x^{k-1} < b_{k-1} - b_{k-2}$, there will be $o_x^{k-1} = 1$ and $T_x \in AS^{k-1}$ (notice that $m_x^{k-1} + o_x^{k-1} = b_{k-1} - b_{k-2}$ because of Property 1b), that is $AS^{k-2} \subseteq AS^{k-1}$. Then, there is $AS^{k-1} \supseteq (AS^{k-2} \cup PAS^k)$. Since

$$\sum_{T_i \in BS^{k-1}} RW_i^{k-1} = -\sum_{T_i \in AS^{k-1}} RW_i^{k-1} = \sum_{T_i \in PWBS^{k-1}} RW_i^{k-1} + \sum_{T_i \in (BS^{k-1} - PWBS^{k-1})} RW_i^{k-1} \ge -\sum_{T_i \in AS^{k-2}} RW_i^{k-1} - \sum_{T_i \in PWAS^{k-1} = PAS^k} RW_i^{k-1}.$$

Notice that $PWBS^{k-2} = PWBS^{k-1}$. From Property 1h:

$$\sum_{T_i \in PWBS^{k-1}} RW_i^{k-1} < \sum_{T_i \in PWBS^{k-2}} RW_i^{k-2} \le \sum_{T_i \in BS^{k-2}} RW_i^{k-2} = -\sum_{T_i \in AS^{k-2}} RW_i^{k-2} < -\sum_{T_i \in AS^{k-2}} RW_i^{k-1}$$

then, from last two equations, we will have:

$$\sum_{T_i \in (BS^{k-1} - PWBS^{k-1})} RW_i^{k-1} > - \sum_{T_i \in PAS^k} RW_i^{k-1}.$$

Since $(BS^{k-1} - PWBS^{k-1}) \subseteq PBS^k$, we will have:

$$\begin{aligned}\sum_{T_i \in PBS^k} PW_i^k &\geq \sum_{T_i \in (BS^{k-1} - PWBS^{k-1})} PW_i^k \\ &> \sum_{T_i \in (BS^{k-1} - PWBS^{k-1})} RW_i^{k-1} \\ &> - \sum_{T_i \in PAS^k} RW_i^{k-1} > - \sum_{T_i \in PAS^k} PW_i^k\end{aligned}$$

that is $\sum_i PW_i^k > 0$, a contradiction.

So, both $PWAS^{k-2}$ and $PWBS^{k-2}$ are not empty. The same as before, we will get either:

(i) $\forall T_x \in PWAS^{k-2}, T_x \in PAS^{k-2}$; or
(ii) $\forall T_y \in PWBS^{k-2}, m_y^{k-2} = b_{k-2} - b_{k-3}$.

Continue the above steps to the boundary time b_{k-w}, where $\forall T_i, RW_i^{k-w} = 0$ (note that $\forall T_i, RW_i^0 = 0$). At that boundary we will have two non-empty sets *PWBS* and *PWAS*, which is a contradiction.

Therefore, when allocating $[_{k-1}, b_k)$, we have $\sum_i m_i^k \leq (b_k - b_{k-1}) \cdot h$. □

Lemma 6. *If $\sum_i w_i = h$ and Algorithm 1 is followed at boundary time $b_0, \ldots, b_{k-1}$ and for $v = 0, \ldots, k-1$, $\sum_i RW_i^v = 0$ and $|RW_i^v| < 1$, then when allocating processors in $[b_{k-1}, b_k)$, we have $|ES^k| \geq (b_k - b_{k-1}) \cdot h - \sum_i m_i^k$; that is, the number of eligible tasks is no less than the number of remaining units (RU) to be allocated.*

Proof. The proof follows a similar approach as that for Lemma 5. That is, suppose that $|ES^k| < (b_k - b_{k-1}) \cdot h - \sum_i m_i^k$, we will show that there are two non-empty sets associated with each of the previous boundaries, contradicting with the fact that every task is punctual at the most recent system punctual boundary time.

Suppose that $|ES^k| < (b_k - b_{k-1}) \cdot h - \sum_i m_i^k$. After allocating m_i^k and o_i^k, we will have $\forall T_i \in ES^k, o_i^k = 1$ and $\forall T_y \in BS^k, m_y^k = b_k - b_{k-1}$. From Property 1, we have $BS^k \subseteq BS^{k-1}$. Note that,

$$\begin{aligned}\sum_i (m_i^k + o_i^k) &< (b_k - b_{k-1}) \cdot h \\ \sum_i RW_i^k &= \sum_i (PW_i^k - o_i^k) \\ &= \sum_i (RW_i^{k-1} + (b_k - b_{k-1}) \cdot w_i - m_i^k - o_i^k) \\ &= 0 + (b_k - b_{k-1}) \cdot h - \sum_i (m_i^k + o_i^k) \geq 1.\end{aligned}$$

Therefore $BS^k \neq \emptyset$. Define two task sets for boundary time b_{k-1} as follows:

$$\begin{aligned}PWAS^{k-1} &= \{T_x | (T_x \in AS^{k-1}) \; AND \; (\eta_x^{k-2} < \eta_y^{k-2}, \\ &\quad \forall T_y \in PWBS^{k-1})\}; \\ PWBS^{k-1} &= \{T_y | T_y \in BS^{k-1} \cap BS^k\}.\end{aligned}$$

Note that $BS^k \neq \emptyset$ and $BS^k \subseteq BS^{k-1}$. Therefore, $PWBS^{k-1} = BS^k \neq \emptyset$. Recall that η_x^k is the priority of a task T_x at the boundary time b_k. Suppose that $PWAS^{k-1} = \emptyset$, we will have $\forall T_x \in AS^{k-1}$ and $\forall T_y \in PWBS^{k-1}$, there is $\alpha(T_x, k-2) \geq \alpha(T_y, k-2)$. From Property 1, $\alpha_{k-1}(T_y) =$ '+'. Thus, $\alpha_{k-1}(T_x) =$ '+'. Therefore, $\forall T_x \in AS^{k-1}$, there are $m_x^k < b_k - b_{k-1}$ and $T_x \in PBS^k$. That is, T_x is an eligible task. Thus $o_x^k = 1$ and $T_x \in AS^k$. That is, we have $AS^{k-1} \subseteq AS^k$. Note that,

$$\begin{aligned}- \sum_{T_i \in AS^{k-1}} RW_i^{k-1} &< - \sum_{T_i \in AS^{k-1}} RW_i^k \\ &\leq - \sum_{T_i \in AS^k} RW_i^k < \sum_{T_i \in BS^k} RW_i^k \\ &< \sum_{T_i \in BS^k} RW_i^{k-1} < \sum_{T_i \in BS^{k-1}} RW_i^{k-1}.\end{aligned}$$

That is, we will have $\sum_i RW_i^{k-1} > 0$, which contradicts with the assumption that $\sum_i RW_i^{k-1} = 0$. Therefore, $PWAS^{k-1} \neq \emptyset$.

So, both $PWAS^{k-1}$ and $PWBS^{k-1}$ are not empty. With the similar reasonings as in the proof of Lemma 5, depending on whether or not the tasks in $PWBS^{k-1}$ are eligible to compete for optional units during the interval of $[b_{k-2}, b_{k-1})$, we will get either

(a) $\forall T_x \in PWAS^{k-1}, T_x \in PAS^{k-1}$ (if $\exists T_y \in PWBS^{k-1} \cap ES^{k-1}$); or
(b) $\forall T_y \in PWBS^{k-1}, m_y^{k-1} = b_{k-1} - b_{k-2}$.

If (a) is true, we will have $\exists T_y \in PWBS^{k-1}, m_y^{k-1} < b_{k-1} - b_{k-2}$; and $\forall T_x \in PWAS^{k-1}, T_x \in PAS^{k-1} \subseteq AS^{k-2}$. Define two task sets for boundary time b_{k-2} as follows:

$$\begin{aligned}PWAS^{k-2} &= \{T_x | T_x \in AS^{k-2} \cap PWAS^{k-1}\}; \\ PWBS^{k-2} &= \{T_y | (T_y \in BS^{k-2}) \; AND \; (\eta_x^{k-3} < \eta_y^{k-3}, \forall T_x \in PWAS^{k-2})\}.\end{aligned}$$

It is easy to see that $PWAS^{k-2} \neq \emptyset$. In what follows, we show that $PWBS^{k-2} \neq \emptyset$, again, through contradiction.

Suppose that $PWBS^{k-2} = \emptyset$. That is, $\forall T_y \in BS^{k-2}$, we have $\exists T_x \in PWAS^{k-2}$ and $\eta_y^{k-3} \leq \eta_x^{k-3}$. From the definition of $PWAS^{k-2}$, we have that, $\forall T_x \in PWAS^{k-2}, \alpha_{k-2}(T_x) =$ '−' (see Property 1e) and $UF_{k-2}(T_x) > b_{k-1} - b_{k-2}$. Therefore, from Lemma 3, we have that, $\forall T_y \in BS^{k-2}$, there is $m_y^{k-1} = 0$. Moreover, since $\exists T_z \in ES^{k-1} \cap PWBS^{k-1}$ and $\forall T_x \in PWAS^{k-2} = PWAS^{k-1}$, such that $\eta_z^{k-2} > \eta_x^{k-2}$ and $o_z^{k-1} = 0$, we have $o_y^{k-1} = 0$. Therefore, $\forall T_y \in BS^{k-2}$, there is $T_y \in BS^{k-1}$.

For the boundary time b_{k-1}, we can define another task set as $AS_{high}^{k-1} = AS^{k-1} - PWAS^{k-1}$, which contains tasks that have higher priorities than that of tasks in $PWBS^{k-1}$ during the allocation of the interval $[b_{k-2}, b_{k-1})$. For any task $T_h \in AS_{high}^{k-1}$, we can get that $T_h \in AS^k$ and $m_h^k + o_h^k = b_k - b_{k-1}$. Suppose that $PWBS^{k-1}$ contains v_1 tasks and AS_{high}^{k-1} contains v_2 tasks (that is, $|PWBS^{k-1}| = v_1$ and $|AS_{high}^{k-1}| = v_2$).

Now, define $\Phi^{k-1} = PWAS^{k-1} \cup AS_{high}^{k-1} \cup PWBS^{k-1} \cup BS^{k-2}$. We can see that Φ^{k-1} may not include all the ahead tasks at b_{k-2} and all the behind tasks at b_{k-1}. Therefore, we have $\sum_{T_i \in \Phi^{k-1}} RW_i^{k-2} \geq 0$ and $\sum_{T_i \in \Phi^{k-1}} RW^{k-1} \leq 0$. That is,

$$\begin{aligned}\sum_{T_i \in \Phi^{k-1}} RW^{k-1} &= \sum_{T_i \in \Phi^{k-1}} (RW^{k-2} + u_i \cdot (b_{k-1} - b_{k-2}) \\ &\quad - m_i^{k-1} - o_i^{k-1}) \leq 0 \\ \sum_{T_i \in \Phi^{k-1}} (RW^{k-2} + u_i \cdot (b_{k-1} - b_{k-2})) &\leq \sum_{T_i \in \Phi^{k-1}} (m_i^{k-1} + o_i^{k-1}) \\ \sum_{T_i \in \Phi^{k-1}} (u_i \cdot (b_{k-1} - b_{k-2})) &\leq \sum_{T_i \in \Phi^{k-1}} (m_i^{k-1} + o_i^{k-1}).\end{aligned}$$

Note that $\forall T_i \in PWAS^{k-1} \cup BS^{k-2}$, there is $m_i^{k-1} = o_i^{k-1} = 0$. Moreover, $\forall T_i \in AS_{high}^{k-1} \cup PWBS^{k-1}$, we have $m_i^{k-1} + o_i^{k-1} \leq b_{k-1} - b_{k-2}$. Recall that, $\exists T_y \in PWBS^{k-1}$, such that $m_y^{k-1} < b_{k-1} - b_{k-2}$ and

$o_y^{k-1} = 0$. Therefore,

$$\sum_{T_i \in \Phi^{k-1}} (u_i \cdot (b_{k-1} - b_{k-2})) = (b_{k-1} - b_{k-2}) \cdot \sum_{T_i \in \Phi^{k-1}} u_i$$
$$\leq \sum_{T_i \in \Phi^{k-1}} (m_i^{k-1} + o_i^{k-1}) < (v_1 + v_2) \cdot (b_{k-1} - b_{k-2})$$
$$\times \sum_{T_i \in \Phi^{k-1}} u_i < (v_1 + v_2).$$

After the allocation of the interval $[b_{k-1}, b_k)$, we have

$$\begin{aligned}\sum_{T_i \in \Phi^{k-1}} RW_i^k &= \sum_{T_i \in \Phi^{k-1}} (RW_i^{k-1} + u_i \cdot (b_k - b_{k-1}) - m_i^k - o_i^k) \\ &\leq \sum_{T_i \in \Phi^{k-1}} (RW_i^{k-1} + u_i \cdot (b_k - b_{k-1})) \\ &\quad - \sum_{T_j \in PWBS^{k-1} \cup AS_{high}^{k-1}} (m_j^k + o_j^k) \\ &< \sum_{T_i \in \Phi^{k-1}} (RW_i^{k-1}) + (v_1 + v_2) \cdot (b_k - b_{k-1}) \\ &\quad - (v_1 + v_2) \cdot (b_k - b_{k-1}) \\ &= \sum_{T_i \in \Phi^{k-1}} (RW_i^{k-1}) \leq 0.\end{aligned}$$

Note that, for other tasks that are not in Φ^{k-1} at the boundary time b_{k-1}, they can be either punctual or ahead at the boundary time b_k after allocating both mandatory and optional units for the interval of $[b_{k-1}, b_k)$. That is, for such tasks, we have $RW^k \leq 0$. Thus, for all the tasks, we will have $\sum RW^k \leq 0$, which contradicts with the assumption that $\sum RW^k \geq 1$. Therefore, we have that $PWBS^{k-2} \neq \emptyset$.

That is, both $PWAS^{k-2}$ and $PWBS^{k-2}$ are not empty. The same as before, depending on whether or not the tasks in $PWBS^{k-2}$ are eligible to compete for the optional units during the interval of $[b_{k-1}, b_{k-2})$, we will get either:

(i) $\forall T_x \in PWAS^{k-2}$, $T_x \in PAS^{k-2}$; or
(ii) $\forall T_y \in PWBS^{k-2}$, $m_y^{k-2} = b_{k-2} - b_{k-3}$.

If (b) is true, that is, $\forall T_y \in PWBS^{k-1}$, $m_y^{k-1} = b_{k-1} - b_{k-2}$. We can define the two task sets for b_{k-2} as follows:

$$PWAS^{k-2} = \{T_x | (T_x \in AS^{k-2}) \, AND \, (\eta_x^{k-3} < \eta_y^{k-3}, \forall T_y \in PWBS^{k-2})\};$$
$$PWBS^{k-2} = PWBS^{k-1} \neq \emptyset.$$

Notice that $\forall T_y \in PWBS^{k-2} = PWBS^{k-1}$, there is $T_y \in BS^{k-2}$ and $\alpha_{k-1}(T_y) =$ '+' (Property 1); that is, $PWBS^{k-2} \subseteq BS^{k-2}$.

Suppose that $PWAS^{k-2}$ is empty, we will have $\forall T_x \in AS^{k-2}$ and $\forall T_y \in PWBS^{k-2}$, there is $\eta_x^{k-3} \geq \eta_y^{k-3}$. Note that $\alpha_{k-2}(T_y) = \alpha_{k-1}(T_y) =$ '+', thus $\alpha_{k-2}(T_x) = \alpha_{k-1}(T_x) =$ '+'. We will have $\forall T_x \in AS^{k-2}$, $T_x \in AS^{k-1}$ (otherwise, $T_x \in BS^{k-1}$; since $\alpha_{k-1}(T_x) =$ '+', there will be $m_x^k = b_k - b_{k-1} \Rightarrow T_x \in PWBS^{k-1} = PWBS^{k-2} \subseteq BS^{k-2}$, a contradiction), and also $T_x \in AS^k$ (otherwise, $T_x \in BS^k = PWBS^{k-1} = PWBS^{k-2} \subseteq BS^{k-2}$, a contradiction), so we have $AS^{k-2} \subseteq AS^k$. Note that, $o_x^{k-1} = o_x^k = 1$. From Property 1, we have:

$$\begin{aligned}-\sum_{T_i \in AS^{k-2}} RW_i^{k-2} &< -\sum_{T_i \in AS^{k-2}} RW_i^{k-1} \\ &< -\sum_{T_i \in AS^{k-2}} RW_i^k \leq -\sum_{T_i \in AS^k} RW_i^k \\ &< \sum_{T_i \in BS^k = PWBS^{k-1}} RW_i^k \\ &< \sum_{T_i \in PWBS^{k-1} = PWBS^{k-2}} RW_i^{k-1} \\ &< \sum_{T_i \in PWBS^{k-2}} RW_i^{k-2} \leq \sum_{T_i \in BS^{k-2}} RW_i^{k-2}.\end{aligned}$$

That is, $\sum_i RW_i^{k-2} > 0$, a contradiction.

Thus, we will have that both $PWAS^{k-2}$ and $PWBS^{k-2}$ are not empty. The same as before, we will get either:

(i) $\forall T_x \in PWAS^{k-2}$, $T_x \in PAS^{k-2}$; or
(ii) $\forall T_y \in PWBS^{k-2}$, $m_y^{k-2} = b_{k-2} - b_{k-3}$.

Continue the above steps to the boundary time b_{k-w}, where $\forall T_i$, $RW_i^{k-w} = 0$ (note that $\forall T_i$, $RW_i^0 = 0$). At that boundary we will have two non-empty sets *PWBS* and *PWAS*, which is a contradiction.

That is, when allocating $[b_{k-1}, b_k)$, we have $|ES^k| \geq (b_k - b_{k-1}) \cdot h - \sum_i m_i^k$. □

References

[1] J.H. Anderson, A. Srinivasan, Early-release fair scheduling, in: Proc. of the 12th Euromicro Conference on Real-Time Systems, Jun. 2000, pp. 35–43.
[2] J.H. Anderson, A. Srinivasan, Pfair scheduling: Beyond periodic task systems, in: Proc. of the 7th Int'l Workshop on Real-Time Computing Systems and Applications, Dec. 2000, pp. 297–306.
[3] J.H. Anderson, A. Srinivasan, Mixed pfair/erfair scheduling of asynchronous periodic tasks, in: Proc. of the 13th Euromicro Conference on Real-Time Systems, Jun. 2001, pp. 76–85.
[4] B. Andersson, S. Baruah, J. Jonsson, Static-priority scheduling on multiprocessors, in: Proc. of the 22th IEEE Real-Time Systems Symposium, Dec. 2001, pp. 193–202.
[5] B. Andersson, K. Bletsas, S. Baruah, Scheduling arbitrary-deadline sporadic task systems on on multiprocessors, in: Proceedings of the 2008 IEEE Real-Time Systems Symposium, 2008, pp. 385–394.
[6] B. Andersson, J. Jonsson, The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%, in: Proc. of the 15th Euromicro Conference on Real-Time Systems, 2003, pp. 33–40.
[7] B. Andersson, E. Tovar, Multiprocessor scheduling with few preemptions, in: Proc. of the IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications, Sept. 2006, pp. 322–334.
[8] T.P. Baker, Multiprocessor edf and deadline monotonic schedulability analysis, in: Proc. of The 24th IEEE Real-Time Systems Symposium, Dec. 2003, pp. 120–129.
[9] S. Baruah, Techniques for multiprocessor global schedulability analysis, in: Proc. of the IEEE Real-Time Systems Symposium, Dec. 2007, pp. 119–128.
[10] S. Baruah, T. Baker, Schedulability analysis of global edf, Real-Time Syst. 38 (3) (2008) 223–235.
[11] S. Baruah, N. Fisher, Component-based design in multiprocessor real-time systems, in: Proceedings of the 6th IEEE International Conference on Embedded Systems and Software, May 2009, pp. 209–214.
[12] S.K. Baruah, N.K. Cohen, C.G. Plaxton, D.A. Varel, Proportionate progress: a notion of fairness in resource allocation, Algorithmica 15 (6) (1996) 600–625.
[13] S.K. Baruah, J. Gehrke, C.G. Plaxton, Fast scheduling of periodic tasks on multiple resources, in: Proc. of The International Parallel Processing Symposium, Apr. 1995, pp. 280–288.
[14] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. System Sci. 7 (1973) 448–461.
[15] H. Cho, B. Ravindran, E.D. Jensen, An optimal real-time scheduling algorithm for multiprocessors, in: Proc. of the IEEE Real-Time Systems Symposium, Dec. 2006, pp. 101–110.
[16] H. Cho, B. Ravindran, E.D. Jensen, *T–L* plane-based real-time scheduling for homogeneous multiprocessors, J. Parallel Distrib. Comput. 70 (3) (2010) 225–236.
[17] V.N. Darera, Bounds for scheduling in non-identical uniform multiprocessor systems. Master's thesis, Indian Institute of Science, India, 2006.
[18] R.I. Davis, A. Burns, A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Technical Report YCS-2009-443, University of York, Department of Computer Science, 2009.
[19] M.L. Dertouzos, A.K. Mok, Multiprocessor on-line scheduling of hard-real-time tasks, IEEE Trans. Softw. Eng. 15 (12) (1989) 1497–1505.
[20] U.C. Devi, J.H. Anderson, Tardiness bounds under global edf scheduling on a multiprocessor, in: Proc. of the 26th IEEE Real-Time Systems Symposium, Dec. 2005, pp. 330–341.
[21] S.K. Dhall, C.L. Liu, On a real-time scheduling problem, Oper. Res. 26 (1) (1978) 127–140.
[22] N. Fisher, J. Goossens, S. Baruah, Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible, Real-Time Syst. 45 (Jun.) (2010) 26–71.

[23] K. Funaoka, S. Kato, N. Yamasaki, Work-conserving optimal real-time scheduling on multiprocessors, in: Proc. of the 20th Euromicro Conference on Real-Time Systems, Jul. 2008, pp. 13–22.
[24] S. Funk, V. Nelis, J. Goossens, D. Milojevic, G. Nelissen, On the design of an optimal multiprocessor real-time scheduling algorithm under practical considerations (extended version). Technical Report http://arxiv.org/abs/1001.4115, Jan 2010.
[25] J. Goossens, S. Funk, S. Baruah, Priority-driven scheduling of periodic task systems on multiprocessors, Real-Time Syst. 25 (2–3) (2003) 187–205.
[26] N. Guan, M. Stigge, W. Yi, G. Yu, Fixed-priority multiprocessor scheduling with liu and layland's utilization bound, in: Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, Apr. 2010, pp. 165–174.
[27] N. Guan, W. Yi, Z. Gu, Q. Deng, G. Yu, New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms, in: Proc. of the 2008 IEEE Real-Time Systems Symposium, Dec. 2008, pp. 137–146.
[28] P. Holman, J.H. Anderson, Guaranteeing pfair supertasks by reweighting, in: Proc. of the 22nd IEEE Real-Time Systems Symposium, Dec. 2001, pp. 203–212.
[29] J. Jun, M.L. Sichitiu, Fairness and qos in multihop wireless networks, in: Proc. of the IEEE 58th Vehicular Technology Conference, 2003, pp. 2936–2940.
[30] S. Kato, K. Funaoka, N. Yamasaki, Semi-partitioned scheduling of sporadic task systems on multiprocessors, in: Proc. of the 21th Euromicro Conference on Real-Time Systems, Jul. 2009, pp. 249–258.
[31] S. Kato, N. Yamasaki, Real-time scheduling with task splitting on multiprocessors, in: Proc. of the IEEE In'l Conference on Embedded and Real-Time Computing Systems and Applications, Aug. 2007, pp. 441–450.
[32] S. Kato, N. Yamasaki, Portioned edf-based scheduling on multiprocessors, in: Proc. of the 8th ACM international conference on Embedded software, Oct. 2008, pp. 139–148.
[33] S. Kato, N. Yamasaki, Semi-partitioned fixed-priority scheduling on multiprocessors, in: Proc. of the 15th IEEE Symposium on Real-Time and Embedded Technology and Applications, 2009, pp. 23–32.
[34] G. Levin, S. Funk, C. Sadowski, I. Pye, S. Brandt, Dp-fair: a simple model for understanding optimal multiprocessor scheduling, in: Proc. of the Euromicro Conference on Real-Time Systems, Jul. 2010, pp. 3–13.
[35] C.L. Liu, James W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, J. ACM 20 (1) (1973) 46–61.
[36] D. Liu, Y.-H. Lee, Pfair scheduling of periodic tasks with allocation constraints on multiple processors, in: Proc. of the 18th Int'l Parallel and Distributed Processing Symposium, Apr. 2004, pp. 119–126.
[37] J.M. Lopez, J.L. Diaz, D.F. Garcia, Minimum and maximum utilization bounds for multiprocessor rm scheduling, in: The Proc. of the 13th Euromicro Conference on Real-Time Systems, Jun. 2001, pp. 67–75.
[38] J.M. Lopez, M. Garcia, J.L. Diaz, D.F. Garcia, Worst-case utilization bound for edf scheduling on real-time multiprocessor systems, in: The Proc. of the 12th Euromicro Conference on Real-Time Systems, Jun. 2000, pp. 25–33.
[39] R. McNaughton, Scheduling with deadlines and loss functions, Manage. Sci. 6 (1959) 1–12.
[40] M. Moir, S. Ramamurthy, Pfair scheduling of fixed and migrating tasks on multiple resources, in: Proc. of the 20th IEEE Real-Time Systems Symposium, Dec. 1999, pp. 294–303.
[41] D.I. Oh, T.P. Baker, Utilization bound for n-processor rate monotone scheduling with stable processor assignment, Real-Time Syst. 15 (2) (1998) 183–193.
[42] L. Sha, T. Abdelzaher, K.-E. Arzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, A.K. Mok, Real time scheduling theory: a historical perspective, Real-Time Syst. 28 (2–3) (2004) 101–155.
[43] I. Shin, A. Easwaran, I. Lee, Hierarchical scheduling framework for virtual clustering of multiprocessors, in: Proc. of the 20th Euromicro Conference on Real-Time Systems, Jul. 2008, pp. 181–190.
[44] A. Srinivasan, J.H. Anderson, Optimal rate-based scheduling on multiprocessors, J. Comput. Syst. Sci. 72 (Sept.) (2006) 1094–1117.
[45] D. Zhu, D. Mossé, R. Melhem, Periodic multiple resource scheduling problem: how much fairness is necessary, in: Proc. of The 24th IEEE Real-Time Systems Symposium, Dec. 2003, pp. 142–151.
[46] D. Zhu, X. Qi, D. Mossé, R. Melhem, An optimal boundary-fair scheduling algorithm for multiprocessor real-time systems. Technical Report CS-TR-2009-005, Dept. of Computer Science, Univ. of Texas at San Antonio, Jun. 2009.

Dakai Zhu received the BE in Computer Science and Engineering from Xi'an Jiaotong University in 1996, the ME degree in Computer Science and Technology from Tsinghua University in 1999, and the PhD degree in Computer Science from the University of Pittsburgh in 2004. He joined the University of Texas at San Antonio as an Assistant Professor in 2005. His research interests include real-time systems, power aware computing and fault-tolerant systems. He has served on program committees (PCs) for several major IEEE and ACM-sponsored real-time conferences (e.g., RTAS and RTSS). He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2010. He is a member of the IEEE and the IEEE Computer Society.

Xuan Qi received the B.S. degree in computer science from Beijing University of Posts and Telecommunications in 2005. He is now a PhD candidate in Computer Science Department, University of Texas at San Antonio. His research interests include real-time systems, parallel systems, and high performance computing. His current research focuses on energy-efficient scheduling algorithms for multiprocessor/multicore real-time systems with reliability requirements.

Daniel Mossé received the BS degree in mathematics from the University of Brasilia in 1986 and the MS and PhD degrees in computer science from the University of Maryland in 1990 and 1993, respectively. He has been a professor at the University of Pittsburgh since 1992. His research interests include fault-tolerant and real-time systems, as well as networking. The current major thrust of his research is real-time systems, power management issues, and networks (wireless and security). Typically funded by the US National Science Foundation and the US Defense Advanced Research Projects Agency, his projects combine theoretical results and actual implementations. He was an associate editor of the IEEE Transactions on Computers and is currently on the editorial board of the Kluwer Journal of Real-Time Systems. He has served on program committees (PCs) and as a PC chair for most major IEEE and ACM-sponsored real-time conferences. He is a member of the IEEE and the IEEE Computer Society.

Rami Melhem received the BE degree in electrical engineering from Cairo University in 1976, the MA degree in mathematics and the MS degree in computer science from the University of Pittsburgh in 1981, and the PhD degree in computer science from the University of Pittsburgh in 1983. He was an assistant professor at Purdue University prior to joining the faculty of the University of Pittsburgh in 1986, where he is currently a professor of computer science and electrical engineering and the chair of the Computer Science Department. His research interests include real-time and fault-tolerant systems, optical interconnection networks, high- performance computing, and parallel computer architectures. He was on the editorial board of the IEEE Transactions on Computers and the IEEE Transactions on Parallel and Distributed Systems. He is the editor for the Kluwer/Plenum book series on computer science and is on the editorial board of the IEEE Computer Architecture Letters and the Journal of Parallel and Distributed Computing. He has served on the program committees of numerous conferences and workshops and was the general chair of the Third International Conference on Massively Parallel Processing Using Optical Interconnections (MPPOI Õ96). He serves on the advisory boards of the IEEE Technical Committees on Parallel Processing and on Computer Architecture. He is a fellow of the IEEE and a member of the IEEE Computer Society and the ACM.