Department of Computer Science, UTSA

Technical Report: CS-TR-2009-005

An Optimal Boundary-Fair Scheduling Algorithm for Multiprocessor Real-Time Systems[*]

Dakai Zhu, Xuan Qi
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, 78249
{dzhu,xqi}@cs.utsa.edu

Daniel Mossé and Rami Melhem
Department of Computer Science
Univresity of Pittsburgh
Pittsburgh, PA 15260
{mosse,melhem}@cs.pitt.edu

Abstract

Although the scheduling problem for multiprocessor real-time systems 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, we propose a novel optimal scheduling algorithm, namely *boundary fair (Bfair)*, which follows the same line of research as the well-known Pfair scheduling algorithms and can also achieve full system utilization. However, different from the Pfair algorithms that make scheduling decisions at *every* time unit to enforce proportional progress (i.e., *fairness*) for each task, Bfair makes scheduling decisions and enforces fairness to tasks *only* at tasks' period boundaries. The correctness of the Bfair algorithm to meet the deadlines of all tasks' instances

[*] A preliminary version of this paper appeared in IEEE RTSS 2003. This work is supported in part by NSF award CNS-0720651.

is formally proved. The 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 (by upto 94%) and the time overhead of Bfair 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 needs dramatically reduced number of context switches and task migrations, as low as 18% and 15%, respectively, when compared to those of Pfair schedules.

1 Introduction

For different (such as 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 [25]. Although the scheduling theory for uniprocessor systems has been well developed, such as the optimal EDF (earliest deadline first) and RM (rate-monotonic) scheduling algorithms [18], the scheduling for multiprocessor real-time systems is still an evolving research field and many problems remain open due to their intrinsic difficulties.

In general, there are two major approaches for scheduling real-time tasks in multiprocessor real-time systems: *partitioned* and *global* scheduling [10, 11]. With the emergence of multicore processors, where the shared cache architecture can significantly alleviate the task migration overhead for global scheduling, there is a reviving interest in global scheduling and many interesting results have been reported in recent years.

In this work, we focus on the problem of scheduling periodic real-time tasks on multiprocessor systems. It concerns allocating m identical processors to n periodic real-time tasks, where a task $T_i = (c_i, p_i)$ is characterized by two parameters: the worst case computation requirement c_i and a period p_i. A *feasible periodic schedule* will allocate 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 $k \in \{1, 2, 3, \ldots\}$ with the constraints that a processor can only be allocated to one task and a task can only occupy one processor for any time unit.

The *proportional fair (Pfair)* scheduling, first proposed by Baruah *et al.*, is the well-known optimal scheduling method for scheduling periodic tasks on multiple processors, which explicitly requires tasks to make proportional progress (i.e., *fairness*) [6]. That is, at any time t, the accumulated processor allocation for task T_i will be either $\lfloor t \cdot w_i \rfloor$ or $\lceil t \cdot w_i \rceil$, where $w_i = \frac{c_i}{p_i}$ is the weight of task T_i. Note that, the fairness is a more strict requirement than that of the original problem. Although the Pfair algorithm can achieve fully system utilization while ensuring all deadlines are met, it can incur quite high scheduling overhead by making scheduling decision at every time unit.

Observing the fact that a task can only miss its deadline at a period boundary, we propose in this paper a novel scheduling algorithm, *boundary fair (Bfair)*, which makes scheduling decisions *only* at tasks' period boundaries. Specifically, at any period boundary, the Bfair algorithm allocates processors to tasks for the time units between current boundary and the next boundary. Similar to the Pfair algorithm, to prevent deadline misses, Bfair ensures fairness for tasks at the boundaries; that is, for any period boundary time b_t, the difference between $b_t \cdot w_i$ and the number of time units allocated to task T_i is less than one time unit. The same as the Pfair algorithm, Bfair is also optimal and can achieve 100% system utilization.

We have formally proved the correctness of the Bfair algorithm on meeting the deadlines of all tasks' instances. Although the complexity of Bfair is the same as that of the Pfair algorithm at each scheduling point, it can reduce the number of scheduling points significantly, and thus reduce the overall scheduling overhead, which is especially important for on-line scheduling. Our simulation results show that, compared to that of the Pfair algorithm, the number of scheduling points can be reduced by upto 94%. Moreover, 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 [2]. Furthermore, by aggregating the time allocation of tasks for the time interval between consecutive period boundaries, the resulting Bfair schedule only needs dramatically reduced number of context switches and task migrations, as low as 18% and 15%, 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.

The remainder of this paper is organized as follows. The related work is discussed 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 uni-processor periodic real-time systems, for static and dynamic priority assignments, respectively [18], neither of them is optimal for multiprocessor real-time systems [11]. Traditionally, there are two major approaches for scheduling real-time tasks in multiprocessor real-time systems: *partitioned* and *global* scheduling [10, 11]. In partitioned scheduling, each task is assigned to a specific processor and processors can only execute tasks that are assigned to them. This approach simplifies the scheduling problem after partitioning, where different well-developed uniprocessor scheduling algorithms (e.g., RM and EDF [18]) can be applied to the subset of tasks on each individual processor. In [24], Oh and Baker studied the partitioned scheduling based 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. Later, a better bound of $(m + 1)(2^{1/(m+1)} - 1)$ for RMFF was shown in [20]. 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 upper bound on the individual task's utilization [21].

In global scheduling, on the other hand, all tasks are put into a shared single queue and all processors fetch the next ready task for execution from the global queue. That is, tasks may migrate and execute on different processors depending on their run-time behaviours. For global qscheduling, the utilizations of tasks play an important role. Suppose that α is the maximum utilization of every individual task. It has been shown that, for global EDF, a task set is schedulable on m processors if the total utilization does not exceed $m(1-\alpha)+\alpha$ [13]. For global RM scheduling, system utilization of $(m/2)(1 - \alpha) + \alpha$ can be guaranteed [5]. Andersson *et al.* also studied one scheduling algorithm named RM-US, where tasks with utilizations higher than some threshold θ have the highest priority [3]. For $\theta = \frac{1}{3}$, Baker showed that RM-US can guarantee a system utilization of $(m + 1)/3$ [5].

More recently, based on the concept of portion tasks, Andersson *et al.* proposed the EKG algorithm aiming to reducing the number of preemptions [4]. EKG assigns some tasks to processors as in conventional partitioned scheduling and splits a task into two portions if necessary, where the portion tasks are assigned to adjacent processors. The worst case system utilization bound can be achieved by EKG is 66%. Following the same line of research, Kato *et al.* also studied a few scheduling algorithms, which are different in how to handling portion tasks and thus achieve different system utilizations [15, 16, 17]. Assuming that the scheduler can be invoked at *any* time instance (i.e., continuous time domain) and processors can be allocated in arbitrary small share, the T-L plane based scheduling algorithms have been studied, which can achieve full system utilization and guarantee all the timing constraints of tasks [9, 12]. However, due to the limitation in real systems, processors are generally allocated to tasks in discrete time qunta (e.g., 10ms in Linux).

Based on the discrete time model, the *proportional fair (Pfair)* scheduling algorithm has been proposed for periodic real-time tasks, which is optimal in terms of achieving full system utilization [6]. The basic idea of Pfair is to enforce proportional progress (i.e., *fairness*) for each task at 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 [7]. A simplified PD algorithm, PD^2, uses two less parameters than PD to compare the priorities of tasks [2]. 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].

The supertask approach was first proposed to support non-migratory tasks in [23]: 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. However, it has been proved that Pfair schedules with such mapping constraints do exist [19]. 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 [14]. While this technique ensures that the supertask's non-migratory component tasks meet their deadlines, some system utilization is sacrificed.

The work reported in this paper is different from all existing results. For periodic tasks whose timing parameters are represented by integers, the proposed novel Bfair scheduling algorithm can achieve full system utilization. Different from the Pfair algorithms, which make scheduling decisions at every time unit, the Bfair algorithm makes scheduling decisions only at tasks period boundaries, which can significantly reduce the scheduling overhead, as shown in this paper.

3 System Models and Problem Formulation

In this section, we first define the boundary fairness as well as related notations. Then, 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.

The system considered consists of m identical processors and n periodic real-time tasks, $\{T_1, \ldots, T_n\}$, where each task $T_i = (c_i, p_i)$ is characterized by its worst case computation requirement c_i and its period p_i. c_i and p_i are integer multiples of a system unit time. The deadline for each task instance is the task's next period boundary. 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 $w_i < 1$ (notice that actually $0 < w_i \leq 1$; if $w_j = 1$, we can dedicate one processor to task T_j and consider a smaller problem with the remaining tasks and the remaining processors). We also assume that the system utilization $U = m$, the number of available processors. If $m - 1 < U < m$, we can add one dummy task $T_{n+1} = (c, p)$ such that $\sum_{i=1}^{n+1} w_i = m$. If $U \leq m - 1$, we can just use $\lceil U \rceil$ processors [23].

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 constraints [7]:

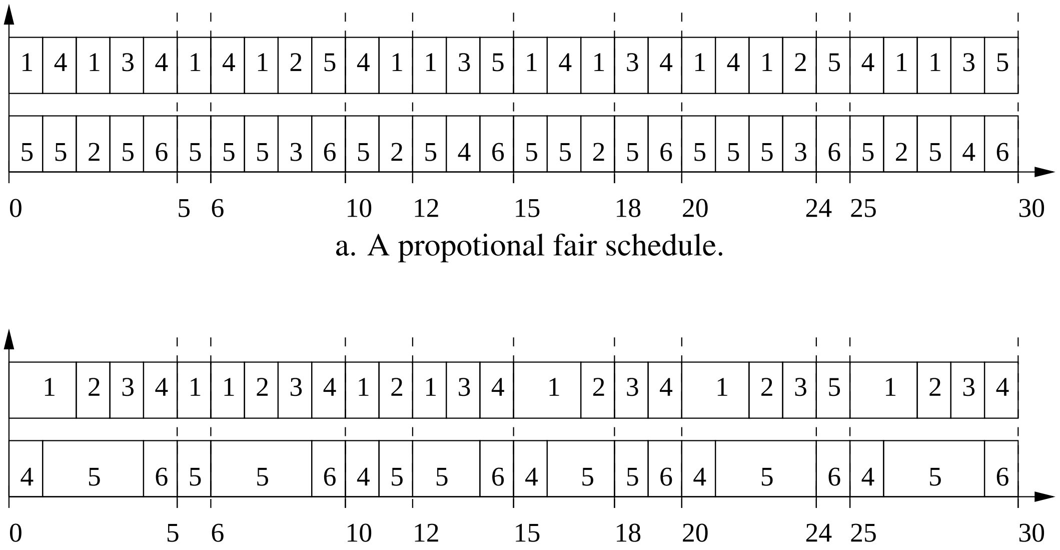
- *C1:* A processor can only be allocated to one task at any time, that is, processors can not be shared concurrently;
- *C2:* A task can only be allocated at most one processor at any time, that is, tasks are not parallel and thus cannot occupy more than one processor at any time.

Assume that the least common multiple of all tasks' period is LCM and the first instance of each task is available at time 0. 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}. Define *allocation error* for task T_i at boundary time b_k as the difference between $b_k \cdot w_i$ and the time units allocated to T_i before b_k. A periodic 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.

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

Proof If $U \leq m$, a proportional fair (Pfair) schedule is known to exist for the multiprocessor real-time scheduling problem [6]. From the definitions, we know that any Pfair schedule is also a boundary fair schedule (a Pfair schedule also conforms to the allocation error requirements at boundaries). That is, a boundary fair schedule exists if $U \leq m$.

◇



a. A propotional fair schedule.

b. A boundary fair schedule.

Figure 1: Different fair schedules for the 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 $LCM = 30$. Figure 1a shows one proportional fair schedule generated by the Pfair scheduling algorithm [6], where the dotted lines are tasks' period boundaries. Note that this schedule is also a boundary fair schedule.

From Figure 1a we can see that there is an excessive number of scheduling points, context switches as well as task migrations due to the requirement of proportional progress (fairness) for each task at *any* time. 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 [22] 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 first processor, there is one time unit left and part of T_4's allocation (one time unit) is packed on the first processor; the rest of T_4's allocation (another time unit) is packed on the second processor followed by T_5 and T_6. Thus, we can schedule $[0, 5)$ as shown in Figure 1b. Continuing the above process for other schedule sections until LCM, we can get a boundary fair schedule as shown in Figure 1b.

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 schedule (which is actually a Bfair schedule) can dramatically reduce number of context switches and task migrations. Compared to the Pfair schedule in Figure 1a, which has 52 context switches and 18 task migrations within one LCM, the Bfair schedule shown in Figure 1b requires only 45 context switches and 9 task migrations. 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 the deadline misses can only happen at the end of a task's period, we propose a novel scheduling algorithm: at *any* boundary time point b_k ($k = 0, \ldots, f-1$), we allocate processors to tasks for time units $[b_k, b_{k+1})$ *properly*. The details are discussed in the next section.

4 A Boundary Fair (Bfair) Algorithm

The Bfair 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 processor slots after allocating the mandatory time units for every task, a dynamic priority is assigned to all *eligible* (as defined later) tasks and a few tasks with the highest priorities will get one *optional* time unit each.

Before formally presenting the Bfair algorithm, we give some definitions. We say that the *remain-*

ing work for task T_i after allocating $[b_k, b_{k+1})$ is the same as the allocation error (see Section 3) of T_i at b_{k+1} and denoted as RW_i^{k+1}. The mandatory integer units needed for T_i while allocating $[b_k, b_{k+1})$ is defined 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 from $[b_{k-1}, b_k)$ 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 T_i gets one 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 in $[b_k, b_{k+1})$, we get $RW_i^{k+1} = PW_i^{k+1} - o_i^{k+1}$.

Similar to the notations used in [6], 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 $RW_i^{k+1} > 0$. Furthermore, we define task T_i to be *pre-behind* at b_{k+1} if $PW_i^{k+1} > 0$.

Algorithm 1 The Bfair algorithm at b_k

```
for ($T_1, \dots, T_n$) do
    /*allocate mandatory units for $T_i$ */
    $m_i^{k+1} = max\{0, \lfloor RW_i^k + (b_{k+1} - b_k) \cdot w_i \rfloor\}$;
    $PW_i^{k+1} = RW_i^k + (b_{k+1} - b_k) \cdot w_i - m_i^{k+1}$;
end for
$RU = m \cdot (b_{k+1} - b_k) - \sum m_i^{k-1}$;
/*allocate optional processor-time units if any*/
/*Pick up the $RU$ highest priority tasks*/
$SelectedTaskSet$ = TaskSelection($RU$);
for ($T_i \in SelectedTaskSet$) do
    $o_i^{k+1} = 1$; /*allocate an optional unit for $T_i$*/
end for
for ($T_1, \dots, T_n$) do
    $RW_i^{k+1} = PW_i^{k+1} - o_i^{k+1}$;
end for
GenerateSchedule($b_k, b_{k+1}$);
```

The Bfair algorithm is presented in Algorithm 1, where RU is the *remaining units* after allocating tasks' mandatory units. It is used to determine how many optional units need to be allocated. Initially, $RW_i^0 = 0$ ($i = 1, \dots, n$).

First, the algorithm allocates mandatory units for each task T_i in the first FOR loop. Next, if there are time units left (i.e., $RU > 0$), the function of $TaskSelection(RU)$ will return the RU highest priority *eligible* tasks[1] and each of them will get one optional unit. After allocating all time units, RW_i^{k+1} are updated in the second FOR loop and the schedule for section $[b_k, b_{k+1})$ is generated by

[1] A task T_i is eligible for an optional unit if $PW_i^{k+1} > 0$ (it is pre-behind) and $m_i^{k+1} < b_{k+1} - b_k$ (it is not fully allocated).

function $GenerateSchedule()$, which sequentially packs tasks to processors following the idea of McNaughton's algorithm [22] (see Figure 1b in Section 3).

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

```
/*For task $T_i$ and $T_j$, assume that $i < j$;*/
$s = 1$;
while ($\alpha_{k+s}(T_i) = \alpha_{k+s}(T_j) = '+'$) do
    $s = s + 1$;
end while
if ($\alpha_{k+s}(T_i) > \alpha_{k+s}(T_j)$) then
    return ($T_i > T_j$);
else if ($\alpha_{k+s}(T_i) < \alpha_{k+s}(T_j)$) then
    return ($T_i < T_j$);
else if ($\alpha_{k+s}(T_i) = \alpha_{k+s}(T_j) = '0'$) then
    return ($T_i > T_j$);
else if ($UF_i^{k+1} > UF_j^{k+1}$) then
    return ($T_i < T_j$);
else
    return ($T_i > T_j$);
end if
```

To determine tasks' priorities when allocating optional units, following the idea in [6], a *characteristic string* of task T_i at **boundary time** b_k is a finite string over $\{-, 0, +\}$ and is defined as:

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

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 $\alpha_{k+s}(T_i) \neq '+'$. Then, if $\alpha_{k+s}(T_i) = '-'$, the *urgency factor* is defined as $UF_i^k = \frac{1-(b_{k+s} \cdot w_i - \lfloor b_{k+s} \cdot w_i \rfloor)}{w_i}$, which is the minimal time needed for a task to collect enough work demand to receive one unit allocation and become punctual after b_{k+s}. Finally, the priority for task T_i at time b_k is defined as a tuple $\eta_i^k = \{\alpha(T_i, k), UF_i^k\}$.

The priority comparison function $Compare(T_i, T_j)$, which is used by $TaskSelection()$, compares two eligible tasks' priorities and is shown in Algorithm 2. First, the characteristic strings of the tasks' are compared, then 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} (the $WHILE$ condition in Algorithm 2). If there is a difference, the task with higher character (here, we have $'-' < '0' < '+'$) has higher priority; if both of them equal $'0'$, the task with smaller identifier has higher priority; if both of them equal

$'-'$, the urgency factors are compared and the task with smaller urgency factor has higher priority; when there still a tie, the task with the smaller identifier has higher priority.

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 corresponding tasks in the $WHILE$ loop (line 3 in Algorithm 2); this is because after the end of a period, the character for a task is no longer equal $'+'$. So, the complexity for $Compare()$ is $O(p_{max})$. Using any linear-comparison selection algorithm [8], $TaskSelection()$ from line 9 of Algorithm 1 needs to make $O(n)$ calls to the function $Compare()$ to decide which RU-subset of all 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. Thus, the overall complexity of the Bfair algorithm is $O(n \cdot p_{max})$, as in the Pfair algorithm [6].

4.2 Constant-Time Priority Comparison

To improve the efficiency of the priority comparison function, following the idea in [7], we have designed a constant time comparison algorithm to compare two eligible tasks' priorities. Specifically, when $\alpha_{k+1}(T_i) = \alpha_{k+1}(T_j) =' +'$ $(i < j)$, instead of looking forward to future boundaries, we can compare the priorities for two counter tasks CT_i and CT_j, which have the weight of $1 - w_i$ and $1 - w_j$, respectively. Since the characters for CT_i and CT_j would be $'-'$ at b_{k+1}, we will need to calculate the urgency factors for them. If CT_i's urgency factor is less than that of CT_j, T_j has the higher priority; otherwise, T_i has the higher priority. It can be proved that the above process will return the same result as that of $Compare()$ for any two eligible tasks. Thus, using the constant priority comparison function and any linear-comparison selection algorithm (e.g., [8]), the complexity of our BF algorithm will be $O(n)$, which is comparable to that of the PD algorithm, $O(min\{n, m \lg n\})$ [7].

Table 1: The execution of the Bfair algorithm for the example

$time$	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

4.3 Sample execution of the Bfair algorithm

Before we present the proof of correctness for our Bfair algorithm, we illustrate the execution of Bfair for the example in page 7. There are 10 boundary time points within $[0, 30)$. The parameters used by our algorithm are calculated as shown in Table 4.3, 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 $(b_1 - b_0) \cdot m = (5 - 0) \cdot 2 = 10$, 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 Figure 1b (see Section 3).

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 Figure 1b, which is also generated from the proportional fair schedule as explained on Page 7. However, compared to the Pfair scheduling algorithm that has 30 scheduling points [6], there are only 10 scheduling points for the Bfair algorithm. Furthermore, as mentioned earlier, the schedule generated by the Bfair algorithm (Figure 1b) will also incur much less context switches and task migrations than the schedule generated by the Pfair algorithm (Figure 1a).

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 3):
  $\sum_{i=1}^{n} m_i^{k+1} \leq (b_{k+1} - b_k) \cdot m$;
- There are enough eligible tasks to claim the remaining units if any (see Lemma 4):
  the number of eligible tasks $\geq (b_{k+1} - b_k) \cdot m - \sum_{i=1}^{n} m_i^{k+1}$;

- 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 our 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 $RW_i^k = PW_i^k - o_i^k$. For convenience, we define some notations. Three task sets are defined as in [6]:

$AS^k = \{T_i | RW_i^k < 0\}$: 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 optional units allocation. 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 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 $(m_i^k < b_k - b_{k-1})\}$;

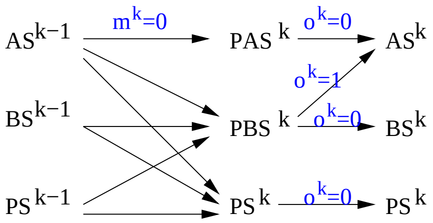


Figure 2: Task transitions from b_{k-1} to b_k.

From these definitions, we can get task transitions between b_{k-1} and b_k as shown in Figure 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.

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 $m_i^k + o_i^k = b_k - b_{k-1}$, $0 < RW_i^k < RW_i^{k-1}$.

$\diamondsuit$

Below we give a proof sketch of Lemma 2, which will be used by Lemmas 3 and 4. For task $T_x \in PAS^k \subseteq AS^k$, from Properties 1c and 1e, 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 Figure 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 2.

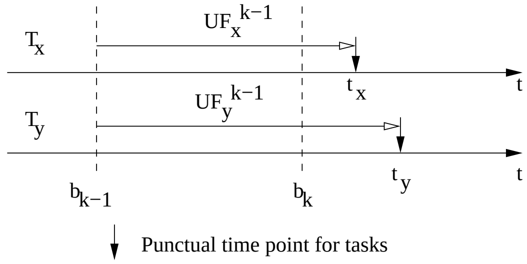


Figure 3: Urgency factors for different tasks.

Lemma 2 *If* $\exists T_x \in PAS^k$, $o_x^{k-1} = 1$, *and* $\forall T_y \in ES^{k-1}$ *(*$x \neq y$*), then* $\alpha_{k-1}(T_y) =' -'$ *and* $UF_y^{k-1} \geq UF_x^{k-1} > b_k - b_{k-1}$.

$\diamondsuit$

To prove that the Bfair algorithm correctly generates a boundary fair schedule, first we show that two conditions are always satisfied during allocating $[b_{k-1}, b_k)$: (1) the summation of all tasks' mandatory integer units is at most equal to the available time units on the m 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 the fact that there is at least one boundary (i.e., b_0) in which every task is punctual. This is formally proved in the following two lemmas.

Lemma 3 *If* $\sum_i w_i = m$ *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* $\sum_i m_i^k \leq (b_k - b_{k-1}) \cdot m$.

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

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

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

$$\begin{aligned} PWAS^{k-1} &= \{T_x|T_x \in PAS^k\}; \\ PWBS^{k-1} &= \{T_y|(T_y \in BS^{k-1}) AND (m_y^k \geq 1)\}; \end{aligned}$$

Notice that, both $PWAS^{k-1}$ and $PWBS^{k-1}$ are not empty. Otherwise, if $PWAS^{k-1} = \emptyset$, we have $\sum_i PW_i^k \geq 0$, which contradicts $\sum_i PW_i^k \leq -1$. 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:

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

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:

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

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

So, both $PWAS^{k-1}$ and $PWBS^{k-1}$ are not empty. From Lemma 2 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 2, 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:

$$\begin{aligned} PWAS^{k-2} &= \{T_x | T_x \in PWAS^{k-1}\} \neq \emptyset; \\ PWBS^{k-2} &= \{T_y | (T_y \in BS^{k-2}) AND (\sum_{l=k-1}^{k} (m_y^l + o_y^l) > 0)\}; \end{aligned}$$

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 Properties 1a and 1d, $T_y \in PBS^k$ 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:

$$\begin{aligned} & \sum_{T_y \in PBS^k} PW_i^k > \sum_{T_y \in BS^{k-2}} PW_i^k \\ > & \sum_{T_y \in BS^{k-2}} RW_i^{k-1} > \sum_{T_y \in BS^{k-2}} RW_i^{k-2} \\ = & -\sum_{T_x \in AS^{k-2}} RW_i^{k-2} \geq -\sum_{T_x \in PAS^{k-1}} RW_i^{k-2} \\ \geq & -\sum_{T_x \in PAS^k} RW_i^{k-2} > -\sum_{T_x \in PAS^k} PW_i^k \end{aligned}$$

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:

$$\begin{aligned} PWAS^{k-2} &= \{T_x | (T_x \in AS^{k-2}) AND (\exists T_y \in PWBS^{k-2}, \\ &\qquad \alpha(T_x, k-3) < \alpha(T_y, k-3))\}; \\ PWBS^{k-2} &= PWBS^{k-1} \neq \emptyset; \end{aligned}$$

If $PWAS^{k-2} = \emptyset$, then $\forall T_x \in AS^{k-2}$ and $\forall T_y \in PWBS^{k-2}$, $\alpha(T_x, k-3) \geq \alpha(T_y, k-3)$. Thus $\alpha_{k-2}(T_x) = \alpha_{k-2}(T_y) ='+'$. Since $m_y^k \geq 1$, whatever the value of $\alpha_{k-1}(T_x)$ is, we will have $T_x \in (PBS^k \cup PS^k)$. Notice that $PAS^k \neq \emptyset$, 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

$$\begin{aligned} & \sum_{T_i \in BS^{k-1}} RW_i^{k-1} = -\sum_{T_i \in AS^{k-1}} RW_i^{k-1} \\ = \ & \sum_{T_i \in PWBS^{k-1}} RW_i^{k-1} + \sum_{T_i \in (BS^{k-1} - PWBS^{k-1})} RW_i^{k-1} \\ \geq \ & -\sum_{T_i \in AS^{k-2}} RW_i^{k-1} - \sum_{T_i \in PWAS^{k-1} = PAS^k} RW_i^{k-1} \end{aligned}$$

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

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

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 $\sum_i m_i^k \leq (b_k - b_{k-1}) \cdot m$.

$\diamondsuit$

Lemma 4 *If $\sum_i w_i = m$ 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 $|ES^k| \geq (b_k - b_{k-1}) \cdot m - \sum_i m_i^k$; that is, the number of eligible tasks is no less than the number of remaining units (RU) to be allocated.*

Proof The proof is similar to that for Lemma 3. If $|ES^k| < (b_k - b_{k-1}) \cdot m - \sum m_i^k$, we will show that there are two non-empty sets associated with each of the previous boundaries, contradicting the fact that every task is punctual at the most recent boundary.

Suppose $|ES^k| < (b_k - b_{k-1}) \cdot m - \sum_i m_i^k$. 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 $BS^k \subseteq BS^{k-1}$. Since

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

thus $BS^k \neq \emptyset$. Define:

$$\begin{aligned} PWAS^{k-1} = \{T_x | (T_x \in AS^{k-1}) AND (\exists T_y \in PWBS^{k-1}, \\ \alpha(T_x, k-2) < \alpha(T_y, k-2))\}; \\ PWBS^{k-1} = BS^k \neq \emptyset \end{aligned}$$

If $PWAS^{k-1} = \emptyset$, we will have $\forall T_x \in AS^{k-1}$ and $\forall T_y \in PWBS^{k-1}$, $\alpha(T_x, k-2) \geq \alpha(T_y, k-2)$. From Property 1, $\alpha_{k-1}(T_y) =' +'$, there will be $\alpha_{k-1}(T_x) =' +'$. Since $\forall T_x \in AS^{k-1}$, there is $m_x^k < b_k - b_{k-1}$, then $o_x^k = 1$ and $T_x \in AS^k$. Thus, $AS^{k-1} \subseteq AS^k$. Since,

$$- \sum_{T_i \in AS^{k-1}} RW_i^{k-1} < - \sum_{T_i \in AS^{k-1}} RW_i^k$$

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

then we will have $\sum_i RW_i^{k-1} > 0$, which contradicts with $\sum_i RW_i^{k-1} = 0$.

So, both $PWAS^{k-1}$ and $PWBS^{k-1}$ are not empty.

The same as in Lemma 3, we will get either

(a) $\forall T_x \in PWAS^{k-1}, T_x \in PAS^{k-1}$; or

(b) $\forall T_y \in PWBS^{k-1}, m_y^{k-1} = b_{k-1} - b_{k-2}$;

If (a) is true, that is, $\forall T_x \in PWAS^{k-1}, T_x \in PAS^{k-1} \subseteq AS^{k-2}$. Define:

$$
\begin{aligned}
PWAS^{k-2} &= PWAS^{k-1} \neq \emptyset; \\
PWBS^{k-2} &= \{T_y | (T_y \in BS^{k-2}) AND (\sum_{l=k-1}^{k} (m_y^l + o_y^l) > 0)\};
\end{aligned}
$$

Notice that, $PWBS^{k-2} \neq \emptyset$; otherwise, $\forall T_y \in BS^{k-2}$, $m_y^k = 0$ and $T_y \in BS^k$, which contradicts with $\forall T_i \in BS^k, m_i^k = b_k - b_{k-1} > 0$. That is, 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) $\forall T_y \in PWBS^{k-2}, m_y^{k-2} = b_{k-2} - b_{k-3}$;

If (b) is true, that is, $\forall T_y \in PWBS^{k-1}, m_y^{k-1} = b_{k-1} - b_{k-2}$. Define:

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

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, $PWBS^{k-2} \subseteq BS^{k-2}$. If $PWAS^{k-2}$ is empty, we will have $\forall T_x \in AS^{k-2}$ and $\forall T_y \in PWBS^{k-2}, \alpha(T_x, k-3) \geq \alpha(T_y, k-3)$; notice that $\alpha_{k-2}(T_y) = \alpha_{k-1}(T_y) =' +'$, then $\alpha_{k-2}(T_x) = \alpha_{k-1}(T_x) =' +'$. 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, $T_x \in BS^k$ and $m_x^k < b_k - b_{k-1}$, 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 $|ES^k| \geq (b_k - b_{k-1}) \cdot m - \sum_i m_i^k$.

$\diamondsuit$

The proof for Lemma 4 is similar to that for Lemma 3 and is omitted. From Lemma 3 and 4, we can get the following theorem.

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 Lemma 3 and 4, the following two conditions are satisfied:
(1) $\sum_i m_i^k \leq (b_k - b_{k-1}) \cdot m$; and
(2) $|ES^k| \geq (b_k - b_{k-1}) \cdot m - \sum_i m_i^k$;

After allocating m_i^k, task T_i will belong to one of the sets in the middle column of Figure 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,

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

Since $\sum_i (m_i^k + o_i^k) = (b_k - b_{k-1}) \cdot m$, 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.

◇

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 our Bfair algorithm (denoted by BF) on reducing the number of scheduling points as well as the overall scheduling overhead. For comparison, we also implemented three Pfair algorithms: the original algorithm PF [6], an improved algorithm PD [7] and the most efficient algorithm PD^2 [2].

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

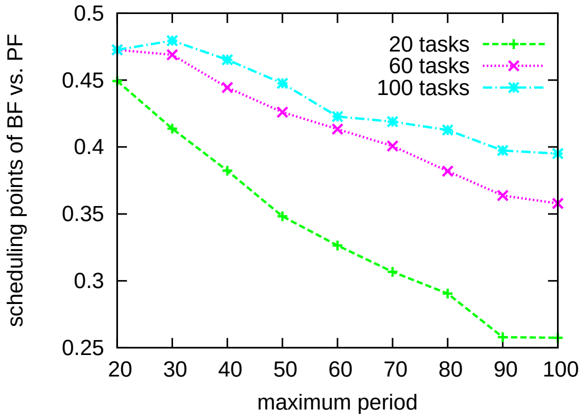


Figure 4: The number of scheduling points with varying p_{max}; $p_{min} = 10$.

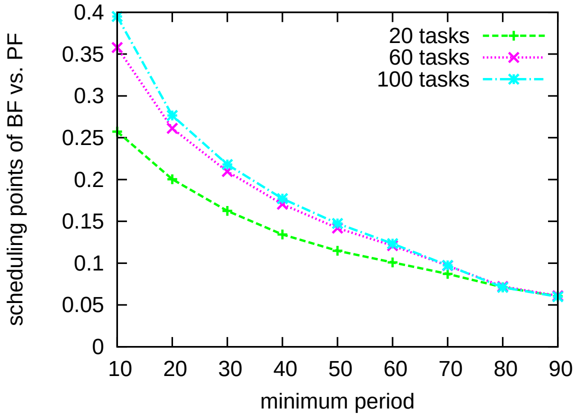


Figure 5: The number of scheduling points with varying p_{min}; $p_{max} = 100$.

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, with fixed minimum task period $p_{min} = 10$, we vary the maximum period p_{max} from 20 to 100 and show the normalized number of scheduling points for the Bfair algorithm with that of Pfair (PF) algorithms as the baseline. Note that, the scheduling points for Bfair are the period boundaries of all tasks which are independent of tasks' computation requirements, while the the number of scheduling points for Pfair (PF) algorithms will be LCM. The results are shown in Figure 4.

From the figure, we can see that the number of scheduling points of our 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 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 $p_{max} = 100$, Figure 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 upto 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 our 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 server with two Intel quad-core 2.4GHz processors and 16 GB memory. As mentioned before, to select k highest priority tasks from n tasks, the task selection function can be implemented in $O(n)$ [8]. However, in the implementation of the algorithms, a simple linear search technique with a complexity of $O(k \cdot n)$ is used.

Moroever, the original Pfair (PF) algorithm is implemented as described in [6]. For the PD algorithm, we implement its constant time priority comparison function. To limit the search space of the task selection function, tasks are first divided into 7 priority categories [7] with the complexity of $O(n)$; then the tasks are selected from high priority category to low priority category; if not all tasks in a category can be selected, the $O(k \cdot n)$ task selection function is used within that category. In this way, we effectively reduced the number of priority comparison needed by the PD algorithm. While the complexity of implemented PD algorithm is still $O(m \cdot n)$, where m is the number of processors, the results (see below) are almost linear with the number of tasks. For the PD^2 algorithm, the window of tasks is generated online. The group deadline of each window is computed

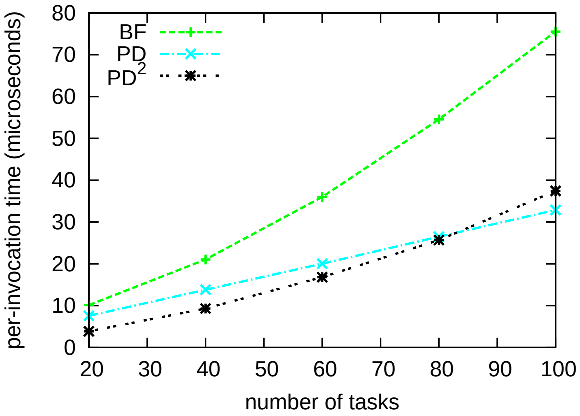


Figure 6: Execution time (in microseconds) at each scheduling point.

with a constant time approach [2].

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. We implement the constant time priority comparison function as described in Section 4 and the implemented Bfair algorithm has an overall complexity of $O(n^2)$. Note that at most $n-1$ optional units need to be allocated per-invocation for the Bfair algorithm.

Figure 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 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 differ-

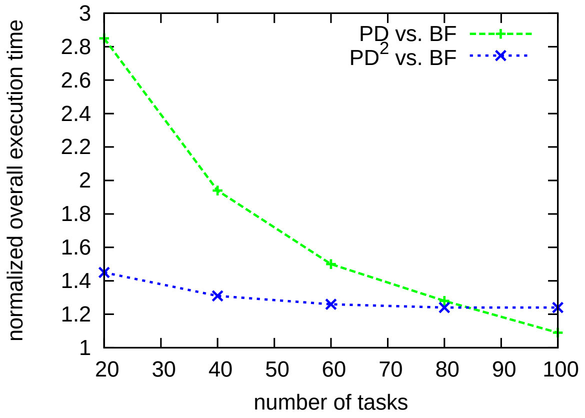


Figure 7: Normalized overall execution time to generate the schedule within LCM.

ent 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. Figure 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 execution time for such task sets is further shown in Figures 8 and 9. 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 as shown in Figures 10 and 11.

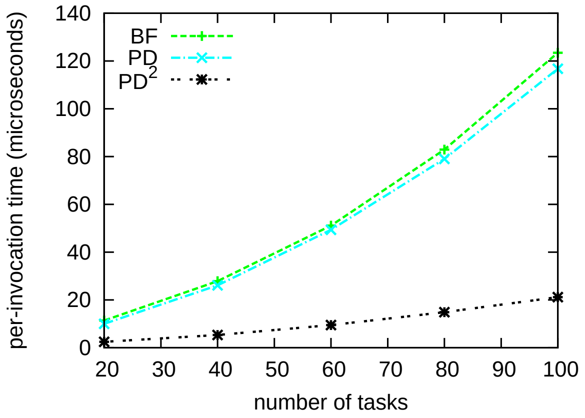


Figure 8: Execution time (in microseconds) at each scheduling point for light tasks.

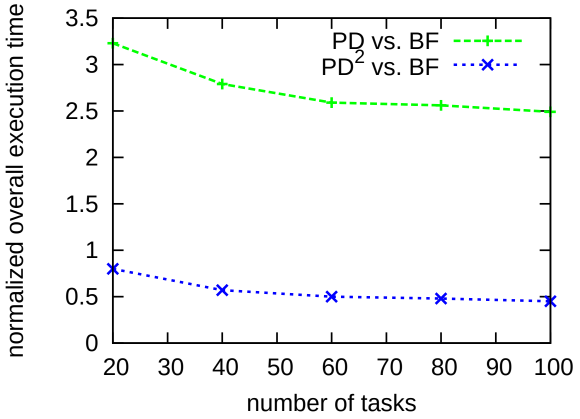


Figure 9: Normalized overall execution time to generate the schedule within LCM for light tasks.

6.3 Number of Context Switches and Task Migrations

In addition to less run-time overhead, as shown in the example on page 3, by aggregating the time allocation of tasks together for the time interval between consecutive period boundaries, the schedule

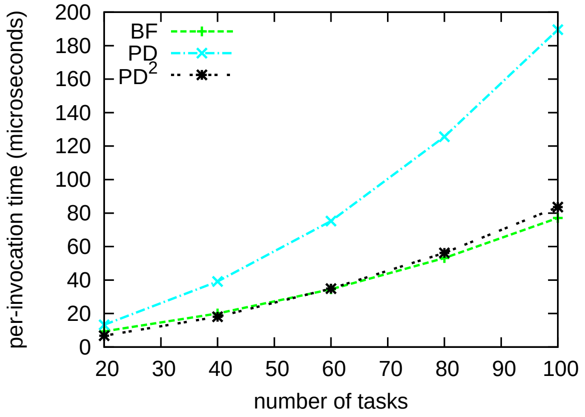


Figure 10: Execution time (in microseconds) at each scheduling point for heavy tasks.

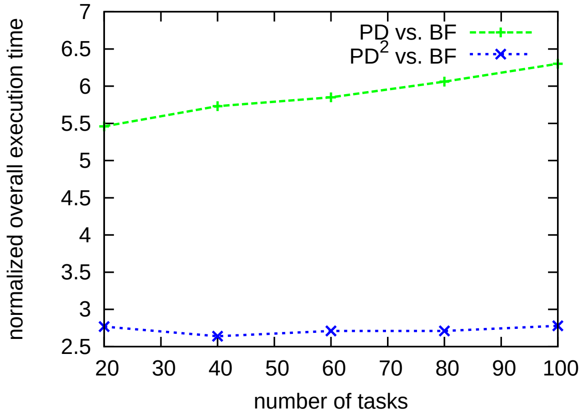


Figure 11: Normalized overall execution time to generate the schedule within LCM for heavy tasks.

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 $p_{min} = 10$, Figure 12 first shows the normalized number of context switches for the schedule of BF when the one of PF is used as the baseline with varying p_{max}. 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 Figure 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 Figure 13. From these results, we can see that, even with $p_{min} = 10$ and $p_{max} = 100$, there are only

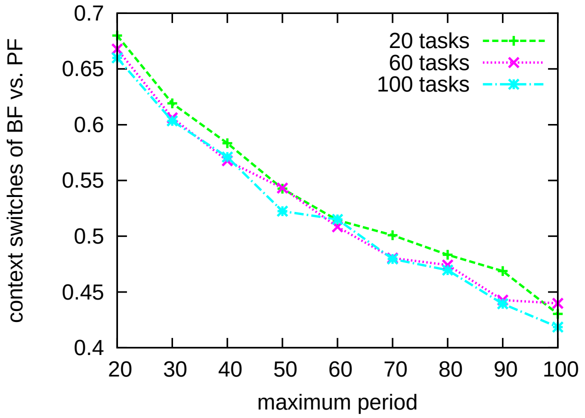


Figure 12: Normalized number of context switches with varying p_{max}; $p_{min} = 10$;

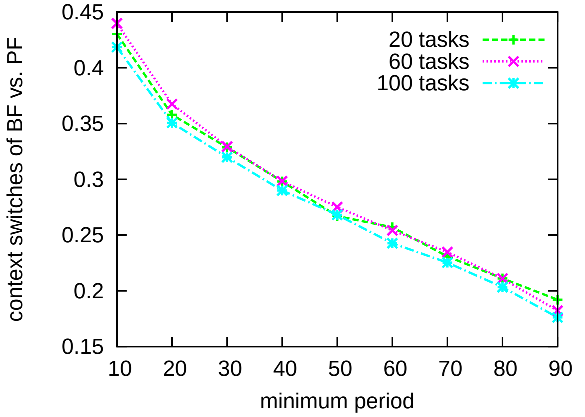


Figure 13: Normalized number of context switches with varying p_{min}; $p_{max} = 10$;

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, upto 82% of context siwtches can be saved, which will be very helpful to reduce the run-time overhead of real-time systems.

As another metric, Figures 14 and 15 further show the normalized number of task migrations required for the resluting Bfair schedule when compared to that of the Pfair schedule. With the same reasonings as those for context switches, we can see that the number of task migrations is also significantly reduced, upto 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.

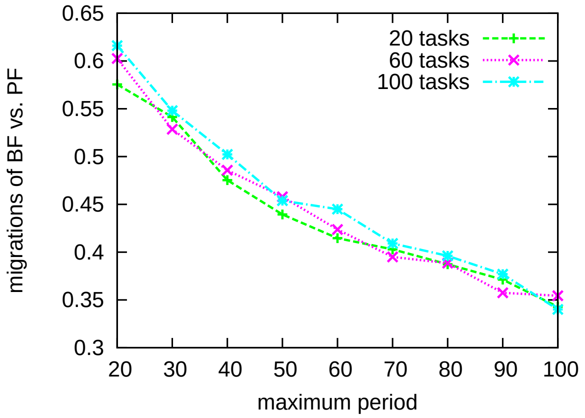


Figure 14: Normalized number of task migrations with varying p_{max}; $p_{min} = 10$;

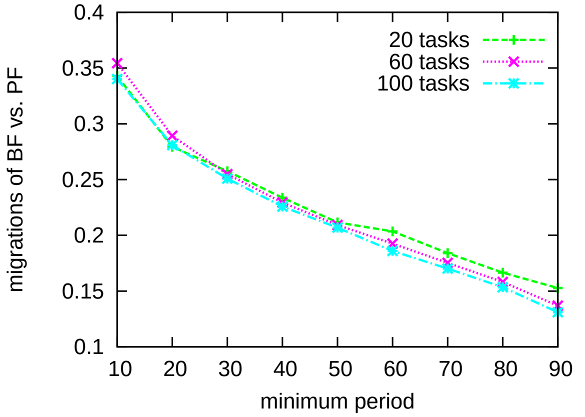


Figure 15: Normalized number of task migrations with varying p_{min}; $p_{max} = 100$.

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 [6], 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 (upto 94%) and the time overhead of Bfair for each scheduling point is comparable to that of the most efficient Pfair algorithm, PD^2 [2]. 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 only needs as low as 18% and 15%, respectively, when compared to those of Pfair schedules.

References

[1] J.H. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proc. of the* 12^{th} *Euromicro Conference on Real-Time Systems*, Jun. 2000.

[2] J.H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Proc. of the* 13^th *Euromicro Conference on Real-Time Systems*, Jun 2001.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proc. of The* 22^{th} *IEEE Real-Time Systems Symposium*, pages 193–202, Dec. 2001.

[4] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2006.

[5] T.P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Proc. of The* 24^{th} *IEEE Real-Time Systems Symposium*, pages 120–129, Dec. 2003.

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[7] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of The International Parallel Processing Symposium*, Apr. 1995.

[8] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[9] H. Cho, B. Ravindran, and E.G. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 101–110, 2006.

[10] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, 15(12):1497–1505, 1989.

[11] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operation Research*, 26(1):127–140, 1978.

[12] K. Funaoka, S. Kato, and N. Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[13] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[14] P. Holman and J.H. Anderson. Guaranteeing pfair supertasks by reweighting. In *Proc. of the* 22^{nd} *IEEE Real-Time Systems Symposium*, Dec. 2001.

[15] S. Kato, K. Funaoka, and N. Yamasaki. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proceedings of the 21th Euromicro Conference on Real-Time Systems*, 2009.

[16] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[17] S. Kato and N. Yamasaki. Portioned edf-based scheduling on multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 139–148, 2008.

[18] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):46–61, 1973.

[19] D. Liu and Y.-H. Lee. Pfair scheduling of periodic tasks with allocation constraints on multiple processors. In *Proceedings of the 18th Int'l Parallel and Distributed Processing Symposium*, pages 119–126, 2004.

[20] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Minimum and maximum utilization bounds for multiprocessor rm scheduling. In *The Proc. of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 67–75, 2001.

[21] J.M. Lopez, M. Garcia, J.L. Diaz, and 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 (ECRTS)*, pages 25–33, 2000.

[22] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[23] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating tasks on multiple resources. In *Proc. of the* 20^{th} *IEEE Real-Time Systems Symposium*, Dec. 1999.

[24] D.I. Oh and T.P. Baker. Utilization bound for n-processor rate monotone scheduling with stable processor assignment. *Real-Time Systems*, 15(2):183–193, 1998.

[25] L. Sha, , T. Abdelzaher, K.-E. Arzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A.K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.