Proc. of the 24th IEEE Real-Time Systems Symposium, Dec. 2003

Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary? *

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

Abstract

The Pfair algorithms are optimal for independent periodic real-time tasks executing on a multiple-resource system, however, they incur a high scheduling overhead by making scheduling decisions in every *time unit to enforce proportional progress for each task. In this paper, we will propose a novel scheduling algorithm, boundary fair (BF), which makes scheduling decisions and enforces fairness to tasks* only *at period boundaries. The BF algorithm is also* optimal *in the sense that it achieves 100% system utilization. Moreover, by making scheduling decisions at period boundaries, BF effectively reduces the number of scheduling points. Theoretically, the BF algorithm has the same complexity as that of the Pfair algorithms. But, in practice, it could reduce the number of scheduling points dramatically (e.g., upto 75% in our experiments) and thus reduce the overall scheduling overhead, which is especially important for on-line scheduling.*

1. Introduction

The multiple-resource *periodic scheduling problem* was first addressed by Liu in 1969 [9]. It concerns allocating m identical resources to n periodic tasks, where a task $T_i = (c_i, p_i)$ is characterized by two parameters: a resource requirement c_i and a period p_i. A *feasible periodic schedule* will allocate exactly c_i time units of a resource 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 resource can only be allocated to one task and a task can only occupy one resource for any time unit.

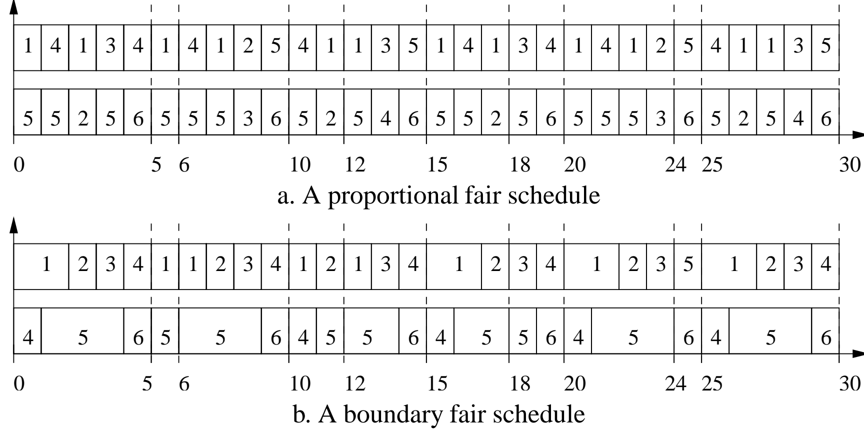
Proportional fair (Pfair) scheduling, first proposed by Baruah *et al.* [4], is the well-known optimal scheduling method for scheduling periodic tasks on multiple resources, which explicitly requires tasks to make proportional progress; that is, at any time t, the accumulated resource allocation for 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 T_i. While achieving full system utilization, the Pfair algorithms incur very high scheduling overhead by making scheduling decision at every time unit [1, 2, 4, 5, 8, 11].

Since a task can only miss its deadline at a period boundary, we propose in this paper a novel algorithm, *boundary fair (BF)*, which makes scheduling decisions *only* at period boundaries. That is, at any period boundary, the BF algorithm allocates resources to tasks for the time units between current boundary and next boundary. Similar to the Pfair algorithms, to prevent deadline misses, BF 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 BF algorithm is optimal in the sense that it can achieve 100% system utilization. Although it has the same complexity as that of the Pfair algorithms in theory, the BF algorithm could reduce the number of scheduling points dramatically in practice, and thus reduce the overall scheduling overhead, which is especially important for on-line scheduling. While the actual reduction depends on the task sets, from our experiments, the number of scheduling points is reduced upto 75% compared to that of the Pfair algorithms. Moreover, the overall time overhead to generate a feasible schedule for BF is much less than that of PD [5] (an efficient Pfair algorithm) when the number of tasks is less than 100.

There are several contributions of this work. First, we introduce the concept of *boundary fairness* in the periodic scheduling problem, which is not as fair as Pfair (at any time) but *fair enough* (only at period boundaries) to get a feasible schedule. Second, we propose a BF scheduling algorithm and prove its correctness to generate a feasible schedule. Finally, the proposed algorithm is also optimal in the sense of achieving 100% system utilization.

* This work has been supported by the Defense Advanced Research Projects Agency through the PARTS project (Contract F33615-00-C-1736).



a. A proportional fair schedule

b. A boundary fair schedule

Figure 1. Different fair schedules for the example; the dot lines are period boundaries

The remainder of this paper is organized as follows. Section 2 defines the boundary fairness and related concepts and gives a motivating example. Section 3 presents a BF algorithm and its complexity analysis. The correctness of the BF algorithm is presented in Section 4. Experimental results are reported in Section 5. Closely related work is discussed in Section 6 and Section 7 gives out our conclusion.

2. Preliminaries

In this section, we formally state the multiple-resource periodic scheduling problem, and define the boundary fairness as well as related notations. An example is also presented to illustrate the idea of boundary fairness.

The system consists of m identical resources and n periodic tasks, $\{T_1, \ldots, T_n\}$, where each task $T_i = (c_i, p_i)$ is characterized by its resource 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 resource to T_j and consider the remaining tasks on the remaining resources). We also assume that the system utilization $U = m$, the number of resources available[1].

The *multiple-resource periodic scheduling problem* is to construct a *periodic schedule* for the above tasks, which allocates exactly c_i time units of a resource 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 [5]:

- *C1:* A resource can only be allocated to one task at any time, that is, resources can not be shared concurrently;
- *C2:* A task can only be allocated at most one resource at any time, that is, tasks are not parallel and thus cannot occupy more than one resource 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 multiple-resource periodic scheduling problem, if the system utilization, U, is no bigger than m, the number of resources, a boundary fair schedule exists.*

Proof If $U \leq m$, a proportional fair (Pfair) schedule is known to exist for the multiple-resource periodic scheduling problem [4]. 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 $U \leq m$.

$\diamondsuit$

To illustrate the idea of boundary fairness, we first consider an example task set that has 6 tasks: $T_1 = (2, 5)$,

1 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 $\lceil U \rceil$ resources [11].

$T_2 = (3, 15)$, $T_3 = (3, 15)$, $T_4 = (2, 6)$, $T_5 = (20, 30)$, $T_6 = (6, 30)$. Here, $\sum_{i=1}^{6} w_i = 2$ and $LCM = 30$. Figure 1a shows one proportional fair schedule generated by PF [4], where the dotted lines are the period boundaries. Note that this schedule is also a boundary fair schedule.

From Figure 1a we see that there is an excessive number of scheduling points as well as context switches due to the requirement of proportional progress (fairness) for each task at *any* time. Consider the *schedule section* between two consecutive 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 [10] and pack tasks within this section on two resources *sequentially* (consecutively fill resources with tasks one by one), after T_1, T_2, T_3 are packed on the first resource, there is one time unit left and part of T_4's allocation (one time unit) is packed on the first resource; the rest of T_4's allocation (another time unit) is packed on the second resource 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.

Considering 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 resources to tasks for time units $[b_k, b_{k+1})$ *properly*. The details are discussed in the next section.

3. A Boundary Fair (BF) Algorithm

The BF algorithm has the following high-level structure: at each boundary time, it allocates resources to tasks for time units between the current and next 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 resource 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 BF algorithm, we give some definitions. We say that the *remaining work* for task T_i after allocating $[b_k, b_{k+1})$ is the same as the allocation error (see Section 2) 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 resources in $[b_k, b_{k+1})$, we get $RW_i^{k+1} = PW_i^{k+1} - o_i^{k+1}$.

Algorithm 1 The BF 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 = m · (b_{k+1} − b_k) − Σ m_i^{k−1};
7: /*allocate optional time-resource units if any*/
8: /*Pick up the RU highest priority tasks*/
9: SelectedTaskSet = TaskSelection(RU);
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_1, ..., T_n) do
14:    RW_i^{k+1} = PW_i^{k+1} − o_i^{k+1};
15: end for
16: GenerateSchedule(b_k, b_{k+1});
```

Similar to [4], 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 $PW_i^{k+1} > 0$.

The BF 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, $RW_i^0 = 0$ ($i = 1, \ldots, n$).

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[2] 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 function $GenerateSchedule()$, which sequentially packs tasks to resources following the idea of McNaughton's algorithm [10] (see Figure 1b and Section 2).

To determine tasks' priorities when allocating optional units, following the idea in [4], 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 $Compare(T_i, T_j)$, used by $TaskSelection()$, compares two eligible tasks'

2 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).

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

```
1: /*For task T_i and T_j, assume that 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
```

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.

Complexity of the BF 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 (e.g., [6]), $TaskSelection()$ 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 resources. Thus, the overall complexity of the BF algorithm is $O(n \cdot p_{max})$, as in the PF algorithm [4].

Constant-Time Priority Comparison

To improve the efficiency of the priority comparison function, following the idea in [5], we can design a constant time comparison algorithm to compare two eligible tasks' priorities. That is, 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. Due to the space limitation, we will not present the algorithm and the proof here. Thus, using the constant priority comparison function and any linear-comparison selection algorithm (e.g., [6]), the complexity of our BF algorithm will be $O(n)$, which is comparable to that of the PD algorithm, $O(min\{n, m \lg n\})$ [5].

Sample execution of the BF algorithm

Before we present the proof of correctness for our BF algorithm, we illustrate the execution of BF for the example in page 3. There are 10 boundary time points within [0, 30). The parameters used by our algorithm are calculated as shown in Table 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 resources sequentially as shown in Figure 1b (see Section 2).

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 BF algorithm is shown in Figure 1b, which is also generated from proportional fair schedule as explained in Page 3. However, there are only 10 scheduling points for

$time$	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

Table 1. The execution of the BF algorithm for the example

the BF algorithm and 30 for the PF algorithm [4]. Furthermore, the schedule generated by BF (Figure 1b) will also incur less context switches than the schedule generated by PF (Figure 1a).

From this example, we can see that when allocating the resources in $[b_k, b_{k+1})$:

- 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 resources 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.

4. Analysis of the BF 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 [4]:

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

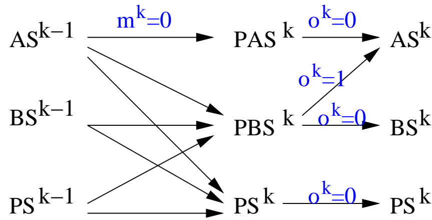


Figure 2. Task transitions from b_{k-1} to b_k.

From these definitions, we can get task transitions between b_{k-1} and b_k as shown by 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 BF 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 $m_i^k + o_i^k = b_k - b_{k-1}$, $0 < RW_i^k < RW_i^{k-1}$.

$\diamond$

Below we give a proof sketch of Lemma 2, which will be used by Lemmas 3 and 4; the formal proof is omitted for the sake of brevity. 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.

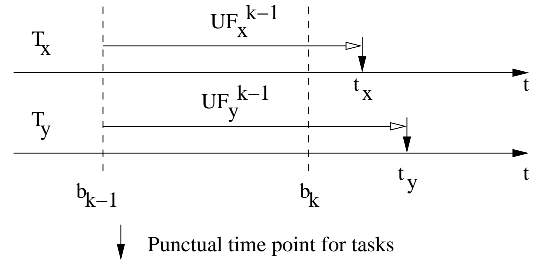


Figure 3. Urgency factors for different tasks.

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 $UF_y^{k-1} \geq UF_x^{k-1} > b_k - b_{k-1}$.* $\diamond$

To prove that the BF 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 resources; (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 bound-

ary (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 resources 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 BF 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:

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

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:

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

If $PWAS^{k-2} = \emptyset$, then $\forall T_x \in AS^{k-2}$ and $\forall T_y \in PWBS^{k-2}$, $\alpha(T_x, k-3) \geq \alpha(T_y, k-3)$. 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

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

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

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

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:

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

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

◇

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 resources 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.* ◇

The proof for Lemma 4 is similar to that for Lemma 3 and is omitted here due to space limitation. 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,

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

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 BF 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 BF algorithm generates a feasible schedule. Moreover, BF is optimal in the sense that it utilizes 100% of each of the resources in a system.

5. Evaluation

In this section, we will experimentally demonstrate the performance of our BF algorithm on reducing the number of scheduling points as well as the overall scheduling overhead compared with that of the PD algorithm. Note that PD [5] is more efficient than PF [4] and both of them make scheduling decisions at every time unit. Another algorithm, PD^2, improves on PD by using a simplified priority comparison function which has two less parameters [3].

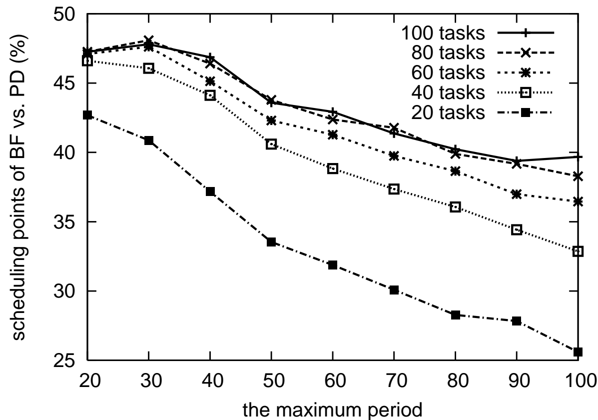


Figure 4. The number of scheduling points of BF vs. PD

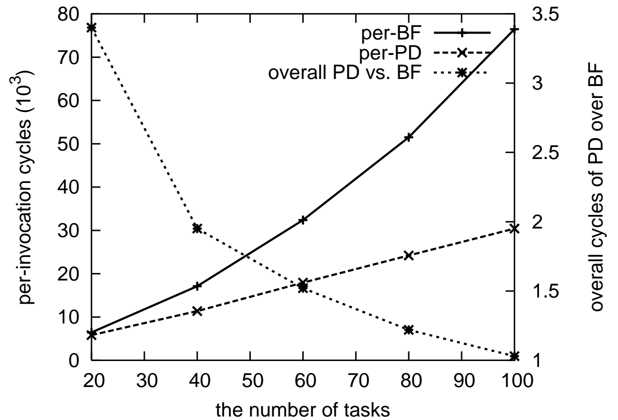


Figure 5. The per-invocation and overall execution time (in cycles) of BF and PD

However, both priority comparison functions have the same complexity of $O(1)$ and we do not expect PD^2 can improve on PD too much, especially when both of them are implemented in $O(n)$ where PD can effectively limit the search space as discussed in this paper while PD^2 cannot.

In our experiments, each task set contains 20 to 100 tasks and the period for a task is uniformly distributed within the minimum and maximum periods considered. The minimum task period is set as 10. For each data point, we averaged the results for 100 randomly generated task sets. First, we vary the maximum period from 20 to 100 and show the reduction of scheduling points of BF compared with PD. The scheduling points for BF are the period boundaries of all tasks which are independent of tasks' resource requirement, while there are LCM scheduling points for PD. The results are shown in Figure 4; these results are conservative since we discarded all task sets with $LCM > 2^{32}$.

From Figure 4, we can see that the number of scheduling points of our BF algorithm varies from 25% to 48% of the PD algorithm. For a fixed 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 BF. For a task set with fixed number of tasks, the periods of tasks are more separated with higher maximum period and thus there are less period boundaries and thus scheduling points for BF. Even when there are 100 tasks in a task set, BF has only 48% scheduling points compared with PD. For applications in real-time systems, tasks are more harmonic than randomly generated task sets and more scheduling points reduction is expected. For example, in a harmonic task set, the number of scheduling points for BF is $LCM/min_i(p_i)$. In the future, we will characterize this issue more accurately.

Next, we compare the run-time overhead of our BF algorithm with that of the PD algorithm by running both algorithms on the SimpleScalar micro-architecture simulator [7]. As mentioned before, to select k highest priority tasks from n tasks, the task selection function can be implemented in $O(n)$ [6]. However, our implementation uses a simple search technique with a complexity of $O(k \cdot n)$.

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 [5] 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 resources, the results (see below) are almost linear with the number of tasks.

Similarly, for BF, eligible tasks are first divided into 3 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 3 and the implemented BF algorithm has an overall complexity of $O(n^2)$. Note that at most $n-1$ optional units need to be allocated per-invocation for BF.

Figure 5 shows the execution time (in cycles) of each scheduling instance. We fix the maximum period as 100 and the resource requirement of a task (c_i) is uniformly distributed between 1 and its period (p_i); thus, on average, the number of resources is around half of the number of tasks. From Figure 5 we can see that BF uses more cycles per-invocation than PD (note that, the *actual* per-invocation cycles of the implemented PD algorithm is almost linear with the number of tasks) and BF becomes worse as the number of tasks increases. However, as shown through the dotted line (which corresponds to the Y-axis on the **right** side of

the figure), when the number of tasks is less than 100, BF uses much less time to generate the whole schedule than PD because of less scheduling points. For applications with very large number of tasks and fewer number of resources, the PD (especially, PD^2) algorithm with a more efficient implementation (with complexity of $O(m \lg n)$) may outperform BF by using less time to generate a schedule. However, realistic systems will have fewer than 100 tasks and, furthermore, the schedule generated by BF incurs less context switches (see Figure 1 in Section 2).

6. Closely Related Work

Although much work has been done on multiple-resource scheduling, we will focus on the related work about Pfair scheduling. The first optimal solution for the general periodic scheduling problem of multiple resources, PF [4], makes scheduling decisions at every time unit and explicitly requires all tasks to make proportional progress. 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 [5]. 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 $O(min\{n, m \lg n\})$. A variant of Pfair scheduling, early-release scheduling, was 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 resources [2].

The supertask approach [11] was first proposed to support non-migratory tasks: tasks bound to a specific resource 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. To solve this problem, [8] reconsiders this approach furnishing it with a *reweighting* technique, which inflates a supertask's weight to ensure that its component tasks meet their deadlines if the supertask is scheduled in a Pfair manner. While this technique ensures that the supertask's non-migratory component tasks meet their deadlines, some system utilization is sacrificed.

7. Conclusion

In this paper, we present a *novel* scheduling algorithm for multi-resource systems. Unlike its predecessor, the Pfair scheduling, which makes scheduling decisions at every time unit to ensure proportional progress for all tasks at *any* time, our algorithm, boundary fair (BF) scheduling, *only* makes scheduling decisions and maintains fairness for tasks at the period boundaries, which effectively reduces the number of scheduling points as well as context switches compared to that of the Pfair algorithms. Although the acutal reduction of the scheduling points depends on the task sets, from our experiments, the number of scheduling points in the BF algorithm is as little as 25% of that in Pfair algorithms. The complexity of BF is the same as that of the PF algorithm [4] and a more efficient implementation with a constant time priority comparison function achieves comparable complexity to that of the PD algorithm [5]. However, by reducing the number of scheduling points, BF reduces the overall scheduling overhead, which is especially important for on-line scheduling. The correctness of the BF algorithm to generate a boundary fair schedule is presented and an example is used to illustrate how BF algorithm works.

References

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

[2] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proc. of the* 7th *International Workshop on Real-Time Computing Systems and Applications*, Dec. 2000.

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

[4] 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.

[5] 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.

[6] 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.

[7] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison, Jun. 1997.

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

[9] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary 37-60*, pages 28 – 37, Nov. 1969.

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

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