Frame-Based Proportional Round-Robin

Arnab Sarkar, Partha P. Chakrabarti, *Senior Member*, *IEEE*, and
Rajeev Kumar, *Senior Member*, *IEEE*

Abstract—All known real-time proportional fair scheduling mechanisms either have high scheduling overheads ($O(lg\, n)$ per time-slot) or do not efficiently handle dynamic task sets. This paper presents *Frame-Based Proportional Round-Robin* ($FBPRR$), a real-time fair scheduler providing high and bounded proportional fairness accuracy and $O(1)$ scheduling overhead with the ability to efficiently handle a set of dynamic tasks. $FBPRR$ achieves this by applying the benefits of *Virtual-Time Round-Robin (VTRR)* scheduling mechanism within a frame-based scheduling approach. Simulation results show that the algorithm gains a speedup of 5 to 20 times (over $O(lg\, n)$ complexity schedulers) with fairly high fairness.

Index Terms—Proportional fairness, ERfair, virtual time, real time, $O(1)$ scheduling, round-robin.

1 Introduction

Fairness has been a desirable criterion of a schedule ever since concurrent execution of independently authored applications became possible in time shared systems. Various algorithms, primarily different flavors of round-robin, such as simple round-robin, weighted round-robin, and prioritized round-robin, have been developed. Fairness has gained even more importance in meeting today's scheduling requirements of coexisting, independently written, possibly misbehaving (those which attempt to use more CPU time than that which is allocated to it) real-time applications with different timeliness constraints. An interesting example of such applications is provided by multimedia systems because they manage continuous media (audio and video streams), characterized by implicit temporal semantics, for implementing video conference, telepresence, video on demand, and other similar services.

Systems running such applications not only demand meeting deadlines, but also proportionate progress of all the running tasks with time, leading to the development of a proportional fair class of schedulers. Consider a set of tasks $\{T_1, T_2, \ldots, T_n\}$, with each task T_i having a computation requirement of e_i time units, required to be completed within a period of p_i time units from the start of the task. Proportional fair schedulers need to manage their task allocation and preemption in such a way that not only are all task deadlines met, but also each task is executed at a consistent rate proportional to its task weight $\frac{e_i}{p_i}$. More formally, let the start time of a task T_i be s_i. Then, proportional fairness guarantees the following for every task T_i: *At the end of any time slot t, $s_i \leq t \leq s_i + p_i$, at least $\frac{e_i}{p_i} * (t - s_i)$ of the total execution requirement of e_i must be completed*. Obviously, for such a criterion to be guaranteed, we must have

$$\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1. \quad (1)$$

Also, since we usually consider discrete timelines, appropriate integral values must be considered while examining fairness.

This sort of equitable resource management has attracted considerable interest among the research community in the last two decades [6], [7], [11], [12]. Typically, these algorithms divide the tasks into equal-sized subtasks. At every time slot, an appropriate subtask from the set of runnable tasks is scheduled to ensure fairness. Research has progressed in two parallel streams primarily differing in their definition of $weight$. The first stream of work which includes schedulers such as Weighted Fair Queuing (WFQ) (1990) [5], Lottery Scheduler (1995) [14], Earliest Eligible Virtual Deadline First (EEVDF) (1996) [13], Virtual-Time Round-Robin (VTRR) (2001) [8], Stratified Round Robin (2003) [10], Group Ratio Round Robin (2004) [9], etc., define the weight w_i of a task T_i as:

$$w_i = \frac{sh_i}{\sum_{T_j \in A} sh_j},$$

where sh_i denotes the relative share of the resource that T_i should receive and A denotes the set of all the active tasks in the system. Although this stream has produced even $O(1)$ (amortized) time algorithms like $VTRR$, Stratified Round-Robin, etc., their general drawback is that the share of each task needs to be adjusted whenever the total summation of shares of the active tasks in the system change. This goes together with the problem of ascertaining that the new share satisfies the task's timing constraints. For example, if the summation of shares in the system increases (e.g., because new tasks are created), then a real-time task's share must increase by a proportional amount. The second stream includes three *Proportionate-fair (Pfair)* scheduling mechanisms (PF (1993) [3], PD (1995) [4], and PD^2 (2004) [2]) and their work-conserving variant *Early Release Fair (ERfair)* (2000) [1]. They define the weight of a task T_i as: $w_i = \frac{ei}{p_i}$, as mentioned earlier. Typically, these algorithms determine the scheduling bandwidth (earliest and latest

• *The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology (IIT), Kharagpur, WB 721 302, India. E-mail: {arnab, ppchak, rkumar}@cse.iitkgp.ernet.in.*

Manuscript received 17 May 2005; revised 23 Nov. 2005; accepted 22 Mar. 2006; published online 20 July 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0163-0505.

slots) of the next subtask for each task in the system. At every time slot, an appropriate subtask from the set of runnable tasks is chosen. In almost all cases, priority queues are used to select the next subtask, leading to $O(lg\, n)$ overheads per time-slot. This turns out to be a reasonably expensive overhead for ensuring fairness, especially in real-time systems where time is at a premium. Several variants have been proposed to reduce scheduling complexity [15]. However, in general, in both the streams of work, obtaining an algorithm faster than $O(lg\, n)$ providing optimum achievable fairness has remained elusive.

Thus, the primary objective of this work is the development of an $O(1)$ highly accurate *proportional fair* real-time scheduler that efficiently handles *dynamic* task sets.

In this paper, we present a scheme called *Frame-Based Proportional Round-Robin* ($FBPRR$) that meets all these objectives. The idea is to define a frame/window of a certain specific size (consisting of a certain number of time slots) and to allocate shares (of time slots) to each task in proportion to their weights $\frac{e_i}{p_i}$ within the frame. These shares are executed in VTRR [8] fashion within the frame, thus providing high proportional fairness accuracy within the frame. After execution inside a frame, each task is put in an appropriate future frame such that the ERfairness [1] of the system remains preserved at frame boundaries. Here, we have assumed that the smallest weight that a task can have is bounded. For example, it is hardly possible in practice to find tasks having weight less than say about 0.001, but still having real-time requirements. Experimental results using this scheme show that a speedup of 5 to 20 times can be obtained (over $O(lg\, n)$ complexity schedulers) with high fairness accuracy.

The paper is organized as follows: In the next section, we introduce some terminology and definitions that will be required in the later sections. We present the FBPRR algorithm along with the fundamental results on fairness and complexity in Section 3. Experimental results are presented in Section 4. We conclude in Section 5.

2 Terminology and Definitions

2.1 Notations

- t: Time; represents the tth time slot.
- n: Total number of tasks.
- T: The set of tasks. Symbolically,

$$T = \{T_1, T_2, T_3, \ldots, T_n\},$$

  where T_i is the ith task.
- T_i^j: jth subtask of T_i.
- s_i: Starting time of T_i. This is equivalent to its arrival time.
- e_i: Execution requirement of T_i (in number of time slots).
- p_i: Period within which the execution of T_i must complete.
- re_i: Currently remaining execution requirement of T_i.
- rp_i: Currently remaining period of T_i; the number of time slots remaining within which T_i must execute so that its deadline is not violated.
- G: Frame size (in number of time slots). The size of a frame is a design parameter and is appropriately chosen.
- ctu: Summation of weights of all the currently active tasks in the system. Its value is updated whenever a new task arrives or an existing task departs.
- f_i: Denotes the ith frame since the start of the schedule.
- sum_shr_i: The sum of the shares of all tasks running in the ith frame.
- naf_i: Next allotted frame for task T_i.
- nas_i: Next allotted share for task T_i.
- $count_i$: The remaining unexecuted shares (execution requirement) of task T_i in the current frame.
- ift: Intraframe time; the number of time slots that have passed in the current frame.
- fst: Starting time of the current frame.
- FL: An array (of size G) of linked lists (buckets).
- FA: A sequence; each element in the sequence points to a distinct array of type FL.
- L_i: A sorted queue of tasks that are to be executed in frame i.

2.2 Definitions

$lag(T_i, t)$: The difference between the amount of time actually allocated to a task and the amount of time that would be allocated to it in an ideal system with a scheduling quantum approaching zero. Formally, the lag is defined as follows:

$$lag(T_i, t) = \frac{e_i}{p_i} * (t - s_i) - (e_i - re_i). \tag{2}$$

Early-Release Fairness (ERfairness): A schedule is early-release fair ($ERfair$) iff:

$$(\forall T_i, t :: lag(T_i, t) < 1). \tag{3}$$

That is, the amount of underallocation associated with each task must always be less than one quantum.

naf_i: The next allotted frame for task T_i. naf_i is calculated when T_i completes execution in a frame and has to be allotted a future frame in which it will execute next. It gives the number of frames that T_i can skip execution but still avoid underallocation. Thus,

$$naf_i = \left\lfloor \frac{rp_i * ctu}{re_i * G} \right\rfloor. \tag{4}$$

nas_i: The next alloted share for task T_i. nas_i is calculated when T_i completes execution in a frame and determines the share of T_i in the next frame in which it will execute. This is given by the difference between the number of time slots of execution which T_i must complete by the end of its next alloted frame and the number of time slots of execution which T_i has already completed. Thus,

$$nas_i = \left\lceil \frac{e_i}{p_i}((naf_i + 2)G + (fst - s_i)) \right\rceil - (e_i - re_i). \tag{5}$$

vt_i: The *virtual time* of a task T_i inside a frame is a measure of the degree to which it has currently received its proportional allocation relative to other tasks inside a

frame. The virtual time of the ith task in a given frame frame is defined as:

$$vt_i = \frac{nas_i - count_i}{nas_i}. \quad (6)$$

vft_i: The *virtual finish time* is defined as the virtual time a task would have after executing for one time slot. At the beginning of a frame:

$$vft_i = \frac{1}{nas_i}. \quad (7)$$

Each time after T_i executes in a time slot, its vft is incremented by $\frac{1}{nas_i}$.

qvt_i: The *queue virtual time* is a measure of what a task's vft should be if it has received exactly its proportional share allocation. At the beginning of the ith frame:

$$qvt_i = \frac{1}{G}. \quad (8)$$

After each time slot within a frame, qvt is incremented by $\frac{1}{G}$.

3 The FBPRR Strategy

The FBPRR scheduling strategy may be conceptualized by the following three steps:

1. *Initialization:* Given a set of tasks, the FBPRR algorithm starts by defining a frame of a certain size, G, and finding the share (the number of time slots that will be allotted in a frame) of each task within the frame.
2. *Intraframe Virtual-Time Round-Robin (VTRR)-based scheduling:* Within a frame, each task is executed in VTRR fashion. At the beginning of each frame, a sorted list of the tasks that are to be run in the frame is formed. The scheduler schedules each task starting from the beginning of this list for one time quantum in round-robin manner. The next task (T_i) encountered in the sorted list in round-robin sequence is selected for execution only if at least one of the following two conditions are satisfied:
   a. The current remaining share of T_i is greater than the current remaining share the task being served presently.
   b. Execution of T_i will not result in its over-allocation by more than 1 time-slot. This condition is verified by the inequality
   $$vft_i - qvt < \frac{1}{nas_i},$$
   where vft_i is the current virtual finish time of T_i, qvt is the current queue virtual time in the frame, and nas_i is the share allotted to T_i in this frame.

   If none of these conditions is satisfied, the remaining tasks in the list are skipped and scheduling gets reinitiated again from the beginning of the list.
3. *Handling frame transition and new task arrival:* After a task completes execution within a frame, it is rescheduled for execution in an appropriate future frame (using (4)) with a proper share (calculated using (5)) so that the ERfairness of the system is maintained. When a new task arrives, its execution frame and share value are determined based on its weight ($\frac{e_i}{p_i}$) and inserted in an appropriate frame.

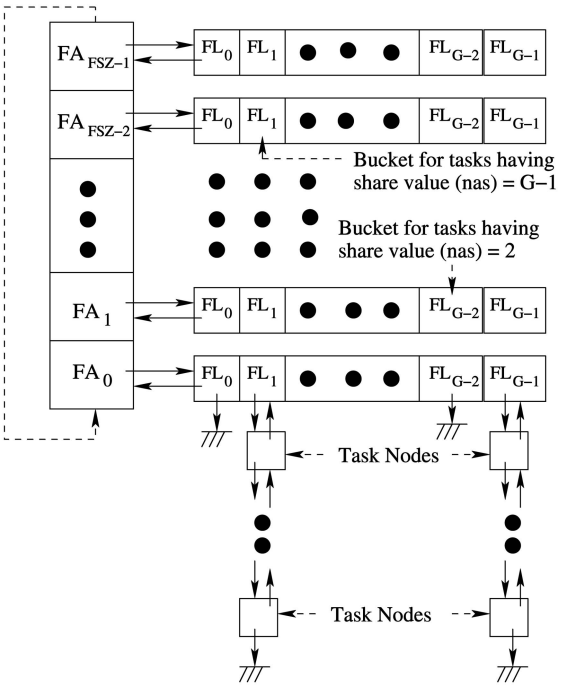

Fig. 1. The principal data structure: FA forms the array of arrays FL of linked lists. Each array FL in FA corresponds to a frame. Each linked list FL_i forms the bucket of tasks with share value $G - i$. FSZ, the size of FA, defines the sliding window of the maximum number of frames that may be accessed simultaneously. To maintain this sliding window, FA has been implemented as a circular array.

3.1 Detailed Algorithm

3.1.1 Data Structures

The algorithm primarily uses two data structures, namely, an array of tasks and an array FA of arrays FL of linked lists. The array of tasks stores information (such as e_i, p_i, etc.) about each task T_i. The array of arrays, FA, manages all the runnable tasks. Each array FL in FA corresponds to a frame. Each linked list FL_i forms the bucket of tasks with share value $G - i$. The nodes corresponding to each task in FL_i contain information including re_i, rp_i, $count_i$, and vft_i.

3.1.2 Size of Array FA

The size FSZ of array FA is determined by the maximum number of frames that may ever be required to be accessed simultaneously. This number is obtained from the lower bound ($1/k$) of the weights of tasks in the system. FSZ is defined as: $FSZ = \lceil \frac{k}{G} \rceil + 1$. FSZ thus defines the sliding window of the maximum number of frames that may be accessed simultaneously. To maintain this sliding window, FQ has been implemented as a circular array. Fig. 1 gives a pictorial representation of the principal data structure used in the algorithm.

The FBPRR algorithm consists of three functions. The main function, *Algorithm FBPRR*, which carries out the overall scheduling calls two other functions, namely,

Function Initialize (FA) which initializes various parameters of the scheduler at the start of scheduling, and *Function Schedule (L_i)* which is called at the beginning of each frame to schedule the tasks within the frame in VTRR fashion.

Algorithm 1 Algorithm FBPRR
Initialize (FA). {Defined in Algorithm 2}
Label1: Select the next nonempty frame FA_i.
if all frames are empty **then**
exit.
end if
Form sorted list L_i of tasks in FA_i.
Schedule (L_i). {Defined in Algorithm 3}
goto Label1.

Algorithm 2 *Function Initialize (FA)*
{For each task T_i, calculate naf_i, and nas_i. Initialize re_i, rp_i,
$count_i$ and vft_i. Create a new list node for T_i and insert it at the tail of list FL_{nas_i} in frame FA_{naf_i}.}
$ctu \leftarrow 0$.
for each active task T_j in T **do**
$ctu \leftarrow ctu + \frac{e_j}{p_j}$.
end for
for each active task T_i in T **do**
Calculate naf_i. {Using (4)}
$re_i \leftarrow e_i$; $rp_i \leftarrow p_i$.
$nas_i \leftarrow \frac{e_i}{p_i}(naf_i + 1)G$.
$count_i \leftarrow nas_i$.
$vft_i \leftarrow \frac{1}{nas_i}$.
$sum\_shr_{naf_i} \leftarrow sum\_shr_{naf_i} + nas_i$.
end for
Create a new list node α_i for T_i.
Insert α_i at FL_{G-naf_i} in FA_{naf_i+1}.

Algorithm 3 *Function Schedule (L_i)*
Point j to the beginning of queue L_i
$qvt_i \leftarrow \frac{1}{G}$.
while L_i is not empty **do**
Execute task pointed to by j. {Let this task be T_k}
Decrement $count_k$.
Decrement re_k.
Increment qvt_i by $\frac{1}{G}$ and vft_k by $\frac{1}{nas_k}$.
if $re_k = 0$ {Task T_k has completed execution} **then**
$ctu \leftarrow ctu - \frac{e_k}{p_k}$.
Remove α_k from L_i.
end if
if $count_k = 0$ {The share of T_k has exhausted} **then**
Remove α_k from L_i.
$rp_k \leftarrow s_k + p_k - fst - ift$.
Calculate naf_k and nas_k {Using (4) and (5), respectively}.
$count_k \leftarrow nas_k$ and $vft_k \leftarrow \frac{1}{nas_k}$.
$sum\_shr_{naf_k} \leftarrow sum\_shr_{naf_k} + nas_k$.
Insert α_k at the tail of FL_{nas_k} in the frame FA_{i+naf_k+1}.
end if
if a new task T_m has arrived **then**
$ctu \leftarrow ctu + \frac{e_m}{p_m}$.
Create a new list node α_m.
Calculate naf_m, nas_m, $count_m$, and vft_m.
Insert α_m at an appropriate frame based on its naf value.
end if
if ($count_k < count_{k+1}$) or ($vft_k - qvt_i < \frac{1}{nas_k}$) {$j$ points to T_k} **then**
Point j to next element of queue. {j now points to T_{k+1}}
else
Point j to the beginning of the queue. {j now points to T_1}
end if
end while

3.2 Sorting

To execute in VTRR fashion, the tasks need to be sorted (in nonincreasing order of share values) when a frame starts. As the share values of a task can range between 1 and G, we use a counting sort technique to order the tasks in $O(G)$ or $O(n)$ (since the size of G is proportional to the task set size n) time.

In each frame, there is a bucket corresponding to each share value between 1 and G. Initially, and after a task finishes execution within a frame, its naf and nas values (along with other attributes) are calculated to find the next frame and share value. A task is always placed in the appropriate bucket based on its share in the frame. A counter max_shr corresponding to the nafth frame keeps track of the maximum share value encountered until now. max_shr is updated if the current nas value is higher. At the beginning of a frame, a linear scan of the buckets starting from FL_{G-max_shr} to FL_{G-1} sorts the tasks into one sorted queue. As, on an average, $max\_shr << G$, the actual scanning overhead is low.

3.3 Frame Size Adjustment

Due to the use of integral values (using floor/ceiling functions) in the definitions of nas and naf, the sum of shares (sum_shr) of all tasks in a frame can possibly become higher than G (thus making the working frame size larger than G). This happens more when the system is heavily loaded. In such a situation, algorithm $FBPRR$ selects $sum\_shr - G$ tasks starting from the task having the highest share value and reduces their share by 1. However, this reduction in share value does not cause the ERfairness criterion to be violated at frame boundaries as the lags of all tasks still remain less than 1 at the frame boundary. Similarly, when the system is lightly loaded, sum_shr can be less than G. Then, the algorithm keeps executing tasks in the frame (even if their allotted shares have been exhausted) until the sum of the shares executed in the frame becomes G. The system still remains $ERfair$ because overallocation does not affect ERfairness and frame size is never increased beyond G.

3.4 Examples

3.4.1 Example 1

We consider four tasks, T_1, T_2, T_3, T_4, having weights 3/5, 1/5, 3/25, 2/25. Let the execution time required by each task be 18 time slots. So, $e_1 = e_2 = e_3 = e_4 = 18$. Hence, $p_1 = 30$, $p_2 = 90$, $p_3 = 150$, $p_4 = 225$. Let the frame size G be 10 and $ctu = 1.0$. The initial naf and nas values of the tasks T_1, T_2,

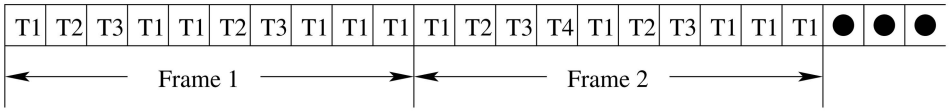

Fig. 2. Example 1: A partial FBPRR schedule.

T_3, and T_4 are 0, 0, 0, 1 and 6, 2, 2, 1, respectively. Therefore, tasks T_1, T_2, and T_3 get scheduled to execute in the first frame, while T_4 gets scheduled in the second frame. After execution in the first frame, $re_1 = 12$, $rp_1 = 20$, $re_2 = 16$, $rp_2 = 80$, $re_3 = 16$, and $rp_3 = 140$. The naf values of T_1, T_2, and T_3 will be 0. The corresponding nas values will be 6, 2, and 2. sum_shr_2 becomes 11. So, T_1 will execute for one less time slot due to the frame size adjustment policy described above. Fig. 2 depicts this scenario. The sequence of executions within the frames as obtained in Fig. 2 is due to the VTRR scheduling mechanism used for scheduling tasks inside the frames.

Next, we take another example to illustrate FBPRR's intraframe scheduling policy.

3.4.2 Example 2

Consider three tasks, T_1, T_2, and T_3. These tasks have been considered to execute in a particular frame and have initial nas values $6, 6, 2$. Let the frame size be 14. So, their initial $vfts$ are $\frac{1}{6}$, $\frac{1}{6}$, $\frac{1}{2}$ and initial qvt is $\frac{1}{14}$. Fig. 3 shows the sequence of executions of the subtasks in this frame. T_1, the first member of the sorted list, gets scheduled in the first time slot. T_2, the next member get executed in the second time slot because its current count value (6) is greater than that of T_1 (5). In the third time slot, T_3 gets scheduled because it satisfies the condition:

$$vft - qvt < \frac{1}{nas} \qquad \left(\frac{1}{2} - \frac{3}{14} < \frac{1}{2}\right).$$

T_1 and T_2 again get scheduled in the fourth and fifth time slots. In the sixth time slot, T_1 gets scheduled instead of T_3 since it cannot satisfy the condition:

$$vft_3 - qvt < \frac{1}{nas_3} \qquad \left(1 - \frac{6}{14} <> \frac{1}{2}\right).$$

The rest of the execution sequence is obvious and can be easily interpreted from the above discussion.

3.5 Analysis of the Algorithm

Lemma 1. *A task T_i of weight $\frac{e_i}{p_i}$ currently having remaining execution requirement re_i time slots and remaining period rp_i time slots will not suffer underallocation at the end of its next frame of execution provided it executes next in the $(naf_i + 1)$th frame after the current frame with a share nas_i and ctu is less than or equal to 1 (that is, the system is not overloaded).*

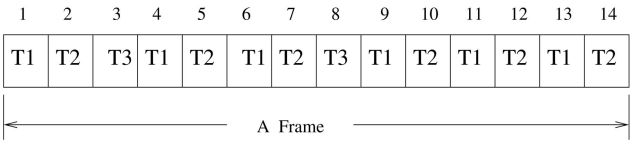

Fig. 3. Example 2: FBPRR's execution sequence within a frame.

Proof. (By step-by-step deduction):

1. T_i will not be underallocated after the execution of its next subtask if it gets scheduled at or before the next $\lfloor \frac{rp_i}{re_i} \rfloor$ time slots.
2. Thus, considering a frame size of 1, if T_i gets scheduled within the next $\lfloor \frac{rp_i}{re_i} \rfloor$ frames, it will not get underallocated after executing in its next frame.
3. Now, considering any frame size G, T_i will avoid underallocation after execution in its next frame if it gets scheduled within the next $\lfloor \frac{rp_i}{re_i * G} \rfloor$ frames.
4. Because $\lfloor \frac{rp_i}{re_i * G} \rfloor$ is always greater than or equal to $\lfloor \frac{rp_i * ctu}{re_i * G} \rfloor$ (as $ctu \leq 1$), T_i cannot become underallocated if it gets scheduled within the next naf_i $(= \lfloor \frac{rp_i * ctu}{re_i * G} \rfloor)$ frames.

   Now, we show that, if T_i executes in the $(naf_i + 1)$th frame after the current frame, its correct share should be nas_i.
5. Let us assume that T_i has completed execution of its share within a frame after ift time slots have passed within the frame.
6. Hence, the number of time slots that have elapsed since its arrival is given by: $fst + ift - s_i$.
7. The number of time slots after which the frame will end is $G - ift$.
8. If T_i executes next in the $(naf_i + 1)$th frame, the number of time slots that will elapse between the end of the current frame and the end of the $(naf_i + 1)$th frame is $(naf_i + 1)G$.
9. Therefore, the number of time slots between the arrival of T_i and the $(naf_i + 1)$th frame's completion is given by:

$$\begin{aligned}&(fst + ift - s_i) + (G - ift) + (naf_i + 1)G\\ &= (naf_i + 1)G + G + (fst - s_i)\\ &= (naf_i + 2)G + (fst - s_i).\end{aligned}$$

10. Hence, to avoid underallocation after executing in the $(naf_i + 1)$th frame, T_i must complete execution of: $\lceil \frac{e_i}{p_i}((naf_i + 2)G + (fst - s_i)) \rceil$ time slots of its total execution requirement of e_i time slots.
11. T_i has actually already completed $e_i - re_i$ time slots of execution.
12. Therefore, to avoid underallocation after executing in the $(naf_i + 1)$th frame, T_i must execute with a share:

$$nas_i = \left\lceil \frac{e_i}{p_i}((naf_i + 2)G + (fst - s_i)) \right\rceil - (e_i - re_i).$$

Hence, if T_i executes in the $(naf_i + 1)$th frame after the current frame with a share nas_i and $ctu < 1$, then T_i will not get underallocated after execution in the frame. □

Theorem 1. *Algorithm FBPRR satisfies ERfairness at frame boundaries.*

Proof. (By induction):

1. At $t = 0$, all tasks have a lag of 0; the hypothesis is trivially true.
   At each frame boundary, the naf and nas values for all tasks which executed in the previous frame are calculated giving the appropriate frame and share values for the tasks such that they do not get underallocated.
2. We assume the truth of the hypothesis after the ith frame, that is, at $t = iG$, where i is an integer.
3. We have to establish the truth of the hypothesis at $t = (i + 1)G$. All tasks scheduled to execute in the $(i + 1)$th frame may have either come from the ith frame or from some earlier frame according to the naf value that was calculated initially (if it is executing for the first time) or after the exhaustion of its share in the frame where it last executed. Now, by Lemma 1, no task (whether it executes in the $(i + 1)$th frame or gets scheduled for execution in a later frame) executing with its corresponding share of nas can get underallocated at the $(i + 1)$th frame's completion.

Thus, FBPRR is ERfair at frame boundaries. So, FBPRR satisfies bounded fairness property. □

Theorem 1 establishes that FBPRR is ERfair at frame boundaries. While it is not possible to guarantee that FBPRR is fair at each time slot within a frame, we present a theorem that limits the overallocation of tasks. While limits to overallocation do not guarantee fairness, it helps control the underallocation in many cases.

Theorem 2. *Within a fully loaded frame, no task T_j will ever be overallocated by more than one time-slot unless T_j is the currently heaviest task, that is, the task with the highest remaining execution requirement within the frame.*

Proof.

1. Let $TS = \{ts_1, ts_2, \ldots, ts_k\}$ be the subset of tasks that are to be run in the current frame.
2. Let $nas_1, nas_2, \ldots, nas_k$ be their corresponding share values.
3. Now, $\sum_{j=1}^{k} nas_j = G$, where G is the frame size.
4. So, each task ts_j must execute for nas_j time-slots within a period of G time-slots.
5. Thus, the initial weight of each task $ts_j = \frac{nas_j}{G}$.
6. At any instant, $(ift - 1)$ within the frame ts_j has executed for $(t_j - 1)$ time-slots.
7. Hence, to ensure that ts_j will not be overallocated by more than one time-slot due to its execution in the next time-slot, the following must hold:

$$t_j - \frac{nas_j}{G} * ift < 1.$$

8. Dividing the above expression by nas_j throughout,

$$\frac{t_j}{nas_j} - \frac{ift}{G} < \frac{1}{nas_j}.$$

9. Now, by definition, $\frac{t_j}{nas_j} = vft_j$ and $\frac{ift}{G} = qvt_j$. Hence,

$$vft_j - qvt_j < \frac{1}{nas_j}.$$

10. By the algorithm, the next task in the sorted queue is executed only when the above expression is true. If it is false, the tasks in the remaining portion of the queue are skipped (because, for all these tasks, the above expression will implicitly be false) and we start scheduling from the beginning of the queue again, thus executing the currently heaviest task. □

Theorem 3. *Algorithm FBPRR has a scheduling complexity of $O(1)$.*

Proof. Let us analyze the complexity of each step of algorithm FBPRR.

1. The first line contains function *Initialize()*. Initialization takes $O(n)$ time, but is done only once, at the beginning of scheduling. So, scheduling complexity is not affected by this function.
2. Selection of the next nonempty FL list before the start of each frame can be done within a constant number of steps in the worst case because the size of FA is fixed.
3. Sorting takes $O(n)$ time in the worst case (we have used a counting sort technique. In most cases, however, as $max\_shr << G$, the actual sorting overhead becomes very low). This is done at the start of each frame. As each frame is of $O(n)$ size, the effective overhead of sorting on the scheduling complexity at each time-slot is $O(1)$.
4. Function *Schedule()* is called to schedule the subtasks within a frame. Let us analyze the *while loop* at the third step of this function. Due to the VTRR strategy, a task can be selected for execution in $O(1)$ time. Putting a task in a future frame after its execution in the current frame and task removal can also be done in constant time. Insertion of a new task into an appropriate frame can be done in amortized constant time. Hence, the function *Schedule()* can schedule a frame of size G in $O(G)$ time. So, the scheduling overhead at each time-slot is $O(1)$.

Hence, the algorithm has a scheduling complexity of $O(1)$. □

4 Experiments and Results

In this section, we experimentally evaluate the performance of algorithm $FBPRR$ and compare it against the $ERfair$ algorithm. For the purpose of comparison, we have

implemented the fastest form of the $ERfair$ scheduler by avoiding the implementation of the tie-breaking rules. The evaluation methodology is based on simulation experiments using randomly generated task sets.

4.1 Experimental Setup

The experimentation framework used is as follows: The data sets consist of randomly generated hypothetical periodic tasks whose execution periods (p_i) and weights ($\frac{e_i}{p_i}$) have been taken from normal distributions. The task weights in a data set may either be generated from a single distribution or can be generated from two separate distributions in which a certain percentage of the tasks is generated from one distribution and the rest is generated from another distribution. The latter simulates cases where the task weights of the system are skewed in nature. For example, there may be situations in practice where the system consists of a few heavy tasks along with many lightweight tasks.

Given the total number of tasks to be generated (n) and the summation of weights of the n tasks (U), two different types of task weight distributions have been considered for the evaluation of the $FBPRR$ algorithm, as listed below:

- *Task weight distribution type 1:* All tasks are generated from a single distribution with standard deviation $(\sigma) = 0.1$ and mean $(\mu) = \frac{U}{2}$.
- *Task weight distribution type 2:* Tasks are generated from two separate distributions; 10 percent of the tasks cumulatively weighing h (values of h considered were 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, and 0.7) are generated from a distribution with $\sigma = 0.1$ and $\mu = h/2$; the remaining 90 percent of the tasks cumulatively weighing $U - h$ are generated from a distribution with $\sigma = 0.1$ and $\mu = \frac{U-h}{2}$.

The summation of weights of the tasks in each of the generated task distributions is not constant. Making the summation of weights constant helps in the evaluation and comparison of the algorithms. Therefore, the weights have been scaled uniformly to make the cumulative weight of each distribution constant and equal to U.

All the task periods have also been generated from a normal distribution having $\sigma = 3,500$ and $\mu = 4,000$. For each of these distribution types, different types of data sets have been generated by setting different values for the following parameters:

1. *Task set size n*: Sizes considered were 5, 10, 25, 50, and 100 tasks.
2. *Workload*: Three different workloads were considered; we have considered cases when the processor is 90 percent, 95 percent, or 100 percent loaded. Results for lower workloads on the processor have not been included here because we found that the scheduler always shows higher fairness and similar speedups under lightly loaded conditions. The performance results under heavy loads are more important for the evaluation and comparison the algorithm's characteristics.
3. *Frame size G*: For each combination of the above parameters, measurements have been taken for six different frame sizes (values n, $2n$, $5n$, $10n$, $15n$, $20n$). Each value is a multiple of the task set size n.

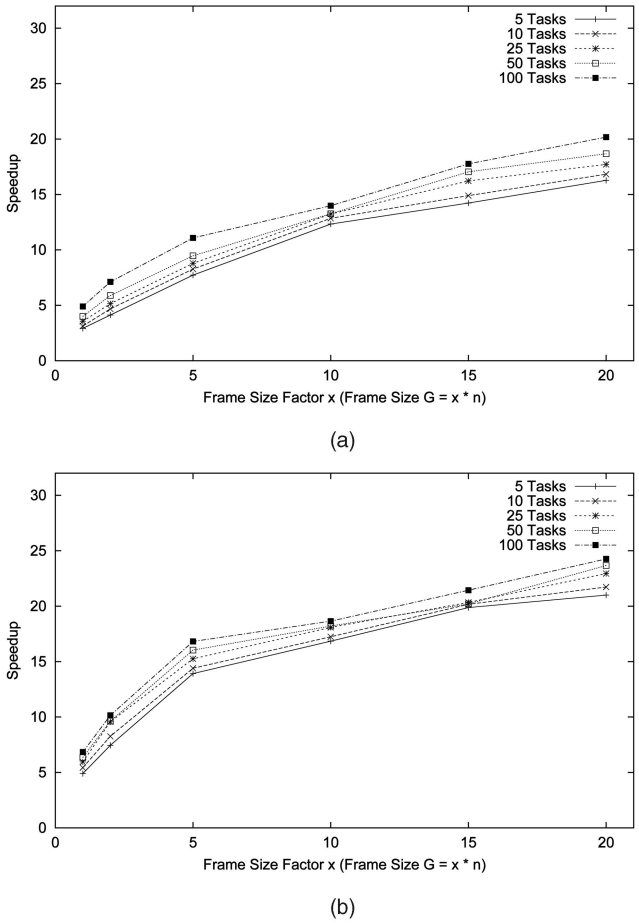

Fig. 4. Speed Up: FBPRR over ERfair. (a) Task distribution type 1. (b) Task distribution type 2, $h = 0.5$.

During experimentation, no slack has been provided between the periods of two consecutive instances of a task. This has been done to keep the total load on the system constant throughout the schedule. The schedule length has been taken to be $500,000$ time slots.

4.2 Results

Time Measurements. For each of the developed schedulers, we have measured the average execution times for both the new and the existing (ERfair) algorithms running them on 50 different instances of each data set type. Using these average execution times, the speedup achieved by the new algorithm over ERfair has been calculated. Fig. 4a shows the "speedup" plots obtained for task weight distribution type 1. Fig. 4b shows the "speedup" plots for $h = 0.5$ for task weight distribution type 2 when $U = 1.0$.

Fairness Measurements. Any quantum-based fair scheduling algorithm (such as PF [3], PD [4], PD^2 [2], etc.) approximates the *ideal fluid schedule*, which requires that there should be no underallocation or overallocation of any task within a task set at any instant of time. This stringent requirement gets relaxed when the early-release criterion is followed as we are no longer bothered about overallocation of each task at each time instant.

TABLE 1
Fairness of FBPRR (Distribution Type 1)

n	U	G					
		n	$2n$	$5n$	$10n$	$15n$	$20n$
25	0.9	0.0246	0.0004	0.0000	0.0001	0.0003	0.0007
	0.95	0.0390	0.0011	0.0003	0.0004	0.0006	0.0014
	1.0	0.1636	0.0098	0.0385	0.0944	0.1635	0.2948
50	0.9	0.0192	0.0001	0.0000	0.0002	0.0005	0.0023
	0.95	0.0365	0.0003	0.0005	0.0009	0.0014	0.0089
	1.0	0.1067	0.0106	0.0655	0.1736	0.2203	0.3572
100	0.9	0.0148	0.0001	0.0000	0.0003	0.0010	0.0035
	0.95	0.0288	0.0005	0.0012	0.0044	0.0537	0.1800
	1.0	0.0678	0.0121	0.0839	0.2022	0.3505	0.4282

n: Task set size; ***U***: Sum of task weights ($\sum \frac{e_i}{p_i}$); ***G***: Frame size.
G=kn: Speedup when G is k times n.

TABLE 2
Fairness of FBPRR (Distribution Type 2; $h = 0.5$)

n	U	G					
		n	$2n$	$5n$	$10n$	$15n$	$20n$
25	0.9	0.0030	0.0015	0.0022	0.0028	0.0039	0.0044
	0.95	0.0188	0.0112	0.0051	0.0053	0.0079	0.0086
	1.0	0.1921	0.1266	0.1535	0.1748	0.2189	0.3296
50	0.9	0.0058	0.0053	0.0007	0.0036	0.0051	0.0094
	0.95	0.0160	0.0143	0.0057	0.0087	0.0119	0.0429
	1.0	0.1318	0.0711	0.1458	0.2163	0.2999	0.4090
100	0.9	0.0071	0.0061	0.0053	0.0082	0.0103	0.0316
	0.95	0.0160	0.0090	0.0212	0.0254	0.0788	0.2373
	1.0	0.1534	0.0869	0.1968	0.2745	0.4745	0.6409

n: Task set size; ***U***: Sum of task weights ($\sum \frac{e_i}{p_i}$); ***G***: Frame size.
G=kn: Speedup when G is k times n.

In order to determine the degree of fairness achieved, we have defined a measure called $average\ miss$. It is based on the lag of each task at each instant of time. We define a term called $miss$ as follows:

$$miss = \begin{cases} lag & \text{if } lag > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

Thus, if, at a given time slot, a task has $lag = 3$, it is considered to have suffered three $misses$ at that time slot. We determine the $miss$ values for each task at each quantum of time. Using these $miss$ values, the $average\ miss$ over the entire schedule length is found out. This is given by:

$$avg\_miss = \frac{\sum miss}{tot\_tslot * n}. \tag{10}$$

Here, tot_tslot represents the total number of time slots in the schedule and n represents the task set size.

The value of the avg_miss gives a measure of the number of misses per time slot per task. Thus, if the avg_miss value of a schedule is 0.0016, it means that there will be 0.0016 misses per time slot per task or 16 misses every $10,000$ time slots. Since ERfair is an optimal algorithm, it suffers no miss at any time slot. Due to this optimal behavior, its fairness value (avg_miss) is 0 for all the different types of data sets mentioned earlier. Table 1 summarizes the fairness results of the FBPRR algorithm for task distribution type 1 for task set sizes 25, 50, and 100 and various frame sizes. Table 2 shows the fairness results for the same set of parameters for $h = 0.5$ of task weight distribution type 2. In Fig. 5, we present the speedup and fairness plots corresponding to various values of h for task weight distribution type 2 for three different task set sizes (25, 50, and 100) and frame size $10n$ (where n stands for the task set size) in a fully loaded system. The nature of the distribution is similar for other values of G.

It may be interesting to note here that, although pure round-robin and VTRR (as adopted by us) schedulers provide higher speedups compared to ERfair, their fairness distortions are also high. Experiments have revealed that applying a pure VTRR scheduler provides a speedup of 28 times over ERfair with a fairness value (avg_miss) of 1.9 for a data set from distribution type 2 ($h = 0.5$) consisting of 100 tasks in a fully loaded processor. The speedup of a pure

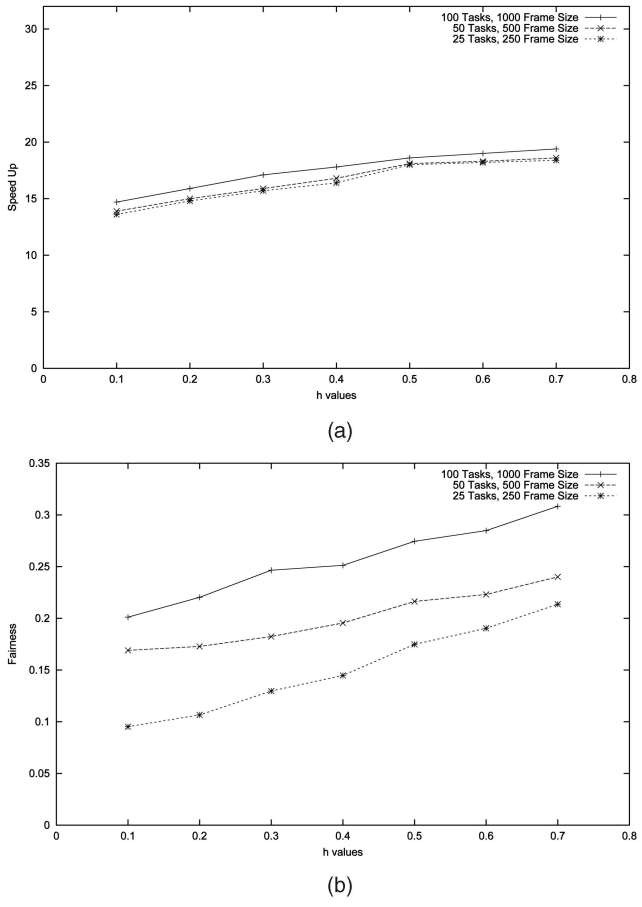

Fig. 5. (a) Speedup and (b) fairness values for task weight distribution type 2 for various values of h. Plots correspond to task set sizes 25, 50, and 100 and frame size $10n$, where n is the task set size when $U = 1.0$.

round-robin scheduler for the same set of parameters is even higher, being 31 times that of ERfair, although (expectedly) it provides a very poor fairness value of 5.8. This compares with speedups in the range of 10 to 18 and fairness values in the range of 0.09 to 0.27 when frame sizes are between $2n$ and $10n$.

4.3 Discussion

From the results obtained in the previous subsection, we can make the following important observations and inferences:

$G = n$ is not a good choice of frame size since its speedup and fairness are dominated by $G = 2n$. Frame sizes in the range of $2n$ to $5n$ provide fairness close to ERfair with speedups in the range of 5 to 10 times that of ERfair at 100 percent workload. At lower workloads (90 percent and 95 percent loaded processor), the value of G may be increased to $20n$ to obtain higher speedups (in the range of 17 to 24 times) while still obtaining good values. From Fig. 5, it may be observed that FBPRR provides consistently stable speedups for various values of h in task distribution type 2. In fact, there is a slight increase in the speedups with increasing h values. This is due to the presence of larger sized tasks for higher values of h, which causes scheduling in a frame and sorting at frame boundaries to become faster. However, the higher skewness of tasks for larger h values results in a slight reduction in fairness.

5 Conclusions

In this paper, we presented a novel proportional fair scheduling algorithm. We proved that FBPRR has high and bounded proportional fairness accuracy, it guarantees $O(1)$ scheduling overhead, and it is able to work for a dynamic set of tasks. We have designed, implemented, and evaluated the FBPRR algorithm. The simulation results are promising.

References

[1] J. Anderson and A. Srinivasan, "Early-Release Fair Scheduling," *Proc. 12th Euromicro Conf. Real-Time Systems,* pp. 35-43, June 2000.

[2] J. Anderson and A. Srinivasan, "Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks," *J. Computer and System Sciences,* vol. 68, no. 1, pp. 157-204, Feb. 2004.

[3] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, "Proportionate Progress: A Notion of Fairness in Resource Allocation," *Algorithmica,* vol. 15, no. 6, pp. 600-625, 1996.

[4] S. Baruah, J. Gehrke, and C. Plaxton, "Fast Scheduling of Periodic Tasks on Multiple Resources," *Proc. Ninth Int'l Parallel Processing Symp.,* pp. 280-288, Apr. 1995.

[5] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Proc. ACM SIGCOMM '89,* pp. 1-12, Sept. 1989.

[6] K. Jeffay and S. Goddard, "A Theory of Rate-Based Execution," *Proc. IEEE Real-Time Systems Symp.,* pp. 304-314, 1999.

[7] K. Jeffay and S. Goddard, "Rate-Based Resource Allocation Models for Embedded Systems," *Lecture Notes in Computer Science,* vol. 2211, p. 204, 2001.

[8] J. Nieh, C. Vaill, and H. Zhong, "Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler," *Proc. General Track: 2002 USENIX Ann. Technical Conf.,* pp. 245-259, June 2001.

[9] J. Nieh, C. Vaill, and H. Zhong, "Group Ratio Round-Robin: An O(1) Proportional Share Scheduler," *Proc. General Track: 2004 USENIX Ann. Technical Conf.,* pp. 245-259, June 2004.

[10] S. Ramabhadran and J. Pasquale, "Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay," *Proc. ACM SIGCOMM,* pp. 239-249, 2003.

[11] J. Regehr, M. Jones, and J. Stankovic, "Operating System Support for Multimedia: The Programming Model Matters," Sept. 2000.

[12] A. Srinivasan, P. Holman, and J. Anderson, "The Case for Fair Multiprocessor Scheduling," *Proc. 11th Int'l Workshop Parallel and Distributed Real-Time Systems,* Apr. 2003.

[13] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *Proc. IEEE Real-Time Systems Symp.,* p. 288, Dec. 1996.

[14] C.A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," PhD Thesis No. MIT/LCS/TR-667, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, 1995.

[15] D. Zhu, D. Mossé, and R. Melhem, "Multiple-Resource Periodic Scheduling Problem: How Much Fairness Is Necessary?" *Proc. 24th IEEE Int'l Real-Time Systems Symp. (RTSS-03),* pp. 142-151, Dec. 2003.

Arnab Sarkar received the BSc degree in computer science in 2000 and the BTech degree in information technology in 2003 from the University of Calcutta, Kolkata, India. He is currently pursuing the MS degree in computer science and engineering at the Indian Institute of Technology (IIT), Kharagpur, India. Since July 2003, he has also been working as a research consultant with the Software Tools for Embedded Systems Group at IIT, Kharagpur. His current research interests include real-time scheduling, system software for embedded systems. and CAD for VLSI.

Partha P. Chakrabarti (M'89-SM'04) received the BTech and PhD degrees in computer science and engineering from the Indian Institute of Technology (IIT), Kharagpur, in 1985 and 1988, respectively. He joined the Department of Computer Science and Engineering, IIT, as a faculty member in 1988 and is currently a professor in the Computer Science and Engineering Department, where he currently holds the position of dean (Sponsored Research and Industrial Consultancy) and where he was the professor in charge of the state-of-the-art VLSI Design Laboratory. He has published more than 100 papers and collaborated with a number of world-class companies. His areas of interest include artificial intelligence, CAD for VLSI, and algorithm design. He received the President of India Gold Medal, the Swarnajayanti Fellowship, and the Shanti Swarup Bhatnagar Prize from the Government of India for his contributions. He is a senior member of the IEEE.

Rajeev Kumar (M'97-SM'03) received the MTech degree from the University of Roorkee (now the Indian Institute of Technology, Roorkee) in 1992 and the PhD degree from the University of Sheffield in 1997, both in computer science and engineering. He is an associate professor of computer science and engineering at the Indian Institute of Technology (IIT), Kharagpur. Prior to joining IIT, he worked for the Birla Institute of Technology & Science (BITS), Pilani and Defence Research & Development Organization (DRDO). His areas of interest include multimedia and embedded systems, programming languages and software engineering, and multiobjective combinatorial optimization. He is a senior member of the IEEE.