Unifying Fixed- and Dynamic-Priority Scheduling based on Priority Promotion and an Improved Ready Queue Management Technique

Risat Mahmud Pathan
Chalmers University of Technology, Sweden

***Abstract*—This paper proposes a new preemptive scheduling algorithm, called Fixed-Priority with Priority Promotion (FPP), for scheduling sporadic tasks on uni- and multiprocessor platform. In FPP scheduling, tasks are executed similar to traditional fixed-priority (FP) scheduling but the priority of some tasks may be promoted at fixed time interval (called, promotion point) relative to the release time of each job.**

A policy called Increase Priority at Deadline Difference (IPDD) to compute the promotion points and promoted priorities for each task is proposed. It is shown that when all tasks' priorities are governed under IPDD policy, then FPP scheduling essentially prioritizes jobs according to Earliest-Deadline-First (EDF) priority. It is known that inserting and removing jobs to and from the ready queue of traditional EDF scheduler is more complex and has higher overhead than that of FP scheduler. To avoid such problem in FPP scheduling, a simple data structure and efficient operations to insert and remove jobs to and from the ready queue are proposed.

Finally, an effective scheme to reduce overhead due to priority promotion is proposed: if a task set is not schedulable using traditional FP scheduling, then promotion points are assigned only to those tasks that need them to meet the deadlines; otherwise, tasks are assigned fixed priorities without any priority promotion and executed same as traditional FP scheduling. Empirical investigation shows the effectiveness of the proposed scheme in reducing overhead on uniprocessor and in accepting larger number of task sets in comparison to that of using state-of-the-art global schedulability tests for multiprocessors.

I. Introduction

The thirst to utilize increasingly more processing capacity of underlying hardware platform while meeting the deadlines of hard real-time sporadic tasks has resulted in the design of numerous scheduling algorithms. The preemptive dynamic-priority-based EDF scheduling is an optimal algorithm for uniprocessor: if there is an algorithm that can schedule a task set such that all the deadlines are met, then the task set is also schedulable using EDF scheduling [15]. In contrast, fixed-priority scheduling does not provide such a guarantee, even under the (optimal for uniprocessor) Deadline-Monotonic (DM) priority assignment [22].

For uniprocessor, although EDF can better utilize the processing capacity, many practical systems implement FP scheduling due to its efficient run-time support and low overhead in managing the ready queue. For multiprocessors, there is no evidence whether global fixed-priority (G−FP) scheduling dominates or is dominated by the global earliest-deadline first (G−EDF) scheduling: some task set may only be deemed schedulable using the state-of-the-art G−EDF test while others only using G−FP test [11]. The question this paper addresses is: **can we combine the schedulability and implementation benefits of both FP and EDF?**

In many real-time systems, e.g., avionics, spacecraft and automotive, it is important to efficiently use the processing resources due to size, weight and power constraints. Reducing overhead of task scheduling in such systems can cut cost for mass production of, for example, cars, trucks or aircraft. A theoretically "good" scheduling algorithm may not be used in practice if overhead of implementation (e.g., managing tasks in the ready queue) is large. This paper proposes an unifying approach to integrate the schedulability and implementation benefits of both FP and EDF scheduling.

A new preemptive scheduling algorithm, called Fixed Priority with Priority Promotion (FPP), is proposed in this paper. Under FPP scheduling, each task has a fixed priority that may undergo priority promotion at fixed time interval (called, *promotion points*) relative to the release time of each job. For example, consider task τ_i that has (initial) fixed priority p with two promotion points δ_1 and δ_2 at which the priority of the task is promoted to priority levels p_1 and p_2 such that $\delta_1 < \delta_2$ and $p_2 < p_1 < p$ (lower priority value implies higher fixed priority). If a job of task τ_i is released at time r_i, the priority of this job is p at time r_i and promoted to priority levels p_1 and p_2 at time $(r_i + \delta_1)$ and $(r_i + \delta_2)$, respectively. Other than priority promotion, FPP scheduling is same as traditional FP scheduling on uniprocessor and multiprocessors[1] platform while applicable to implicit-, constrained- or arbitrary-deadline sporadic tasks.

The FPP scheduler consists of a *dispatcher* and a *ready-queue manager*. The dispatcher at each time instant dispatches the highest-priority ready job if a processor is idle. If all the processors are busy, then a newly released job with higher priority can preempt a currently-executing relatively lower priority job. Active jobs that cannot be executed wait in the ready queue. The ready-queue manager inserts and removes jobs to and from the ready queue. The ready-queue manager also takes care of priority promotion of the jobs that are currently awaiting execution in the ready queue.

The effectiveness of FPP scheduling in meeting the deadlines of the tasks depends on the promotion points and promoted priorities of each task. A simple policy called Increase Priority at Deadline Difference (IPDD) to compute (offline) the promotion points and promoted priorities for

[1]In this paper, the term "FPP scheduling" in general applies to scheduling on uniprocessor and multiprocessors. For multiprocessors, FPP scheduling means global FP scheduling with priority promotion.

978-1-4799-8603-3/15/$31.00 ©2015 IEEE

each task is proposed. When all the tasks are assigned priorities based on IPDD policy, it will be shown that the FPP scheduling essentially prioritizes jobs of the tasks according to EDF priority. Executing jobs of the tasks in EDF order but using priority-promotion-based FPP scheduler is one of the major contributions in this paper.

Since jobs can be prioritized in EDF order, the management of jobs in the ready queue of FPP scheduler would suffer from the same overhead problems (as discussed by Buttazzo [10]) if it is implemented similar to that of traditional EDF scheduler. On the other hand, the ready queue management and run-time support for traditional FP scheduling is much simpler, which is the main reason for its popularity in many commercial real-time kernels. This paper proposes a simple data structure and constant-time, i.e., $O(1)$ operations for implementing the ready queue. The ready queue management using the proposed scheme has similar benefits as that of traditional FP scheduler, which is another major contribution of this paper.

The only source of additional overhead for managing the jobs in the ready queue of FPP scheduler in comparison to that of FP scheduler is the cost of priority promotion. To reduce such overhead due to priority promotion, a joint priority assignment and schedulability test, called FPP_Test, is proposed for FPP scheduling. The FPP_Test assigns traditional fixed priorities (with no promotion point) to some tasks while assigns priorities to other tasks (with promotion points) using IPDD policy. The FPP_Test thus combines the schedulability benefits of both fixed and dynamic (i.e., IPDD) priorities in addition to having the similar implementation benefits of traditional FP scheduler.

To measure the effectiveness of FPP scheduling in terms of reducing overhead for managing jobs in the ready queue in comparison to that of EDF scheduling, the execution of randomly generated task sets is simulated using both FPP and EDF scheduling. The ready queues are simulated using the proposed data structure (presented in Subsection IV-B1) for FPP scheduler and using a priority queue implemented as a binary min-heap (as is used in [17]) for EDF scheduler. The simulation result shows that ready queue management of FPP scheduler suffers significantly less overhead in comparison to that of EDF scheduler.

The FPP_Test is applicable to both uniprocessor and multiprocessor platform. On uniprocessor platform, any task set schedulable using the optimal preemptive EDF scheduling is also schedulable using FPP scheduling. Thus, FPP is also optimal for uniprocessor. On multiprocessor platform, it will be shown that the FPP_Test dominates both the state-of-the-art G-FP and G-EDF tests. Simulation result shows the effectiveness of FPP scheduling in determining higher percentage of schedulable task sets and in reducing the number of preemptions and migrations.

Related Work. The FPP algorithm is similar to the well-known *dual-priority* scheduling which was first proposed by Burns and Wellings [9] in 1993, and analyzed by Davis and Wellings [14], [13] considering shared resources, release jitter and for scheduling soft real-time tasks. In dual-priority scheduling, each task undergoes priority promotion only once. In contrast, a task in FPP scheduling may have more than one promotion point. The reason for having more than one promotion point is to have the power to prioritize jobs in EDF order to meet deadlines.

Gonzalez Harbour et al. [18] considered scheduling FP scheduling of periodic tasks where each task is divided into a collection of precedence-constrained subtasks such that each subtask has its own priority. It is shown using an example by Burns and Wellings [9] that a task set may be schedulable in dual-priority scheduling and may not be schedulable using the approach proposed by Gonzalez Harbour et al. [18]. After around one-and-half decade, Burns [8] presented an open problem at the RTSOPS seminar in ECRTS 2010: Is the utilization bound of dual-priority scheduling of implicit-deadline tasks on uniprocessor 100%? While Burns [8] solved this problem for task set having $n = 2$ tasks, the answer to this question for $n > 2$ is still unknown for dual-priority scheduling. This paper will show that the utilization bound of FPP scheduling of implicit-deadline tasks on uniprocessor is 100% for any n.

Organization. The rest of the paper is organized as follows. Section II presents the task model. The IPDD policy and important lemmas of this policy are presented in Section III. The dispatcher and ready queue manager of FPP scheduler are presented in Section IV. Technique to reduce the total number of promotion points, particularly, the FPP_Test is proposed in Section V. The FPP_Test is applied to both uni- and multiprocessors considering constrained-deadline tasks and experimental results are presented in Section VI and Section VII, respectively. Finally, Section VIII concludes this paper.

II. Task Model

This paper considers scheduling a collection of n sporadic tasks in set $\Gamma = \{\tau_1, \ldots \tau_n\}$. Each task τ_i is characterized by a triple (C_i, D_i, T_i), where C_i represents the worst-case execution time (WCET), D_i is the relative deadline, and T_i is the minimum inter-arrival time of the jobs or instances of task τ_i. Successive arrivals of the instances (called *jobs*) of task τ_i are separated by at least T_i time units. Each job of task τ_i after its release requires at most C_i units of execution time before its relative deadline. The *release time* and *absolute deadline* of job J_a of task τ_i are respectively denoted by r_a and d_a such that $d_a = r_a + D_i$.

A job is called *active* if it is released but has not completed its execution. An active job may be *in execution* or *awaiting* execution in the ready queue at any time instant. The FPP scheduling is applicable to implicit-, constrained- and arbitrary-deadline tasks. This paper assumes that lower priority value implies higher priority levels; i.e., 1 and n are the highest and lowest priority levels, respectively.

III. Priority Promotion Policy: IPDD

The IPDD priority-promotion policy requires n distinct fixed-priority levels to determine the promotion points and promoted priorities of n tasks. Tasks are indexed in deadline-monotonic order, i.e., if $j < i$ for any two tasks τ_j and τ_i, then $D_j \leq D_i$. Therefore, there are $(i-1)$ tasks (i.e., $\tau_1, \tau_2, \ldots \tau_{i-1}$) that have their relative deadlines no larger than that of task τ_i. The IPDD policy computes the promotion points and promoted priorities for each task $\tau_i \in \Gamma$ as follows:

- Task τ_i has i different priority levels: starting from priority level i to the highest priority level 1. Task τ_i's initial priority i is promoted $(i-1)$ times. Figure 1 depicts IPDD priority-promotion policy for task τ_i.
- Each job of task τ_i when released has (initial) priority level i, which is promoted to priority level $(i-1)$ at the first promotion point; then promoted to priority level $(i-2)$ at the second promotion point; and continuing in this manner, finally, promoted to the (highest) priority level 1 at the last, i.e., $(i-1)^{th}$ promotion point.
- The promotion points apply to each job of task τ_i. Each promotion point of a job is a fixed time interval from the release time of the job. The κ^{th} promotion point is equal to the (relative) deadline difference of tasks τ_i and $\tau_{i-\kappa}$, which is equal to $(D_i - D_{i-\kappa})$. The priority of each job of τ_i is promoted to priority level $(i-\kappa)$ at the κ^{th} promotion point which is $(D_i - D_{i-\kappa})$ time units later than its release time, for $\kappa = 1, 2, \ldots (i-1)$. Since $D_i \geq D_{i-1} \ldots \geq D_1$, we have $(D_i - D_{i-1}) \leq (D_i - D_{i-2}) \ldots \leq (D_i - D_1)$, which implies that priority of task τ_i is non-decreasing.
- If any two tasks τ_i and τ_j, where $i < j$, have the same relative deadline, then task τ_j's initial priority is i (not j) since $D_j - D_i = 0$ and the promotion points of τ_j are computed the same way that are computed for τ_i.

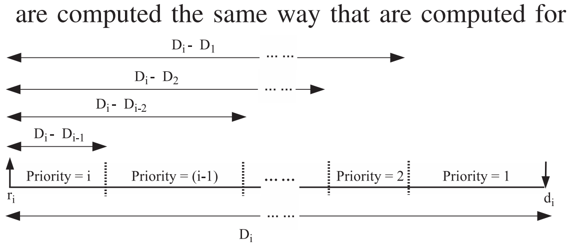

Figure 1. IPDD priority-promotion policy for task τ_i. Consider that an arbitrary job of task τ_i is released at time r_i and has deadline at $d_i = r_i + D_i$. The dotted vertical lines are the promotion points. The κ^{th} promotion points is $(D_i - D_{i-\kappa})$ time units later than time r_i for $\kappa = 1, 2 \ldots (i-1)$.

According to IPDD policy, i priority levels are common (shared) for any two tasks τ_i and τ_j whenever $i < j$. Due to sharing of priority levels, the number of *distinct* fixed-priority levels required to assign priorities to all the n tasks is at most n. Example 1 demonstrates the IPDD policy using an example of three tasks.

Example 1. *Consider (C_i, D_i, T_i) for three tasks $\tau_1 \equiv (1, 2, 4)$, $\tau_2 \equiv (4, 7, 8)$ and $\tau_3 \equiv (3, 10, 16)$. In IPDD policy, each job of task τ_1 starts with priority level 1 which is never promoted since there is no other task with smaller relative deadline than D_1. If a job of task τ_1 is released at time r_1, then the priority of this job remains at priority level 1 during $[r_1, r_1 + D_1)$.*

Since there is one other task (i.e., τ_1) with smaller relative deadline than D_2, each job of τ_2 starts with priority level 2, which is promoted exactly once at time $r_2 + (D_2 - D_1) = r_2 + (7-2) = r_2 + 5$, where r_2 is the release time of an arbitrary job of τ_2. The priority of the job remains at priority level 2 and 1 respectively during $[r_2, r_2 + 5)$ and $[r_2 + 5, r_2 + D_2) = [r_2 + 5, r_2 + 7)$.

Since there are two other tasks (i.e., τ_1 and τ_2) with smaller relative deadlines than D_3, each job of τ_3 starts with priority level 3, which is promoted twice — first at time $r_3 + (D_3 - D_2) = r_3 + (10-7) = r_3 + 3$ and second at time $r_3 + (D_3 - D_1) = r_3 + (10-2) = r_3 + 8$, where r_3 is the release time of an arbitrary job of τ_3. The priority of the job is at priority level 3, 2 and 1 respectively during $[r_3, r_3 + 3)$, $[r_3 + 3, r_3 + 8)$ and $[r_3 + 8, r_3 + D_3) = [r_3 + 8, r_3 + 10)$. The FPP schedule of this task set is given in Figure 2 (the scheduler is formally presented in subsection IV-A).

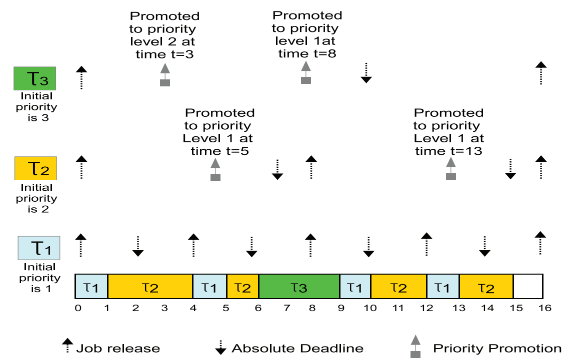

Figure 2. FPP schedule of three tasks in Example 1 where each task is assigned priorities using IPDD policy. Note that task τ_3's priority is promoted to priority level 1 at time $t = 8$ and is not preempted by the third job of task τ_1 that is released at time $t = 8$ although both jobs have the same priority. This is because a newly released job cannot preempt a currently executing job if both have the same priority in FPP scheduling.

The remainder of this section presents important Lemmas regarding the properties of IPDD policy and will be used to show that FPP scheduling generates EDF schedule.

Lemma 1. *If the priority of job J_a is promoted to priority level ℓ at time t_a, then $(d_a - t_a) = D_\ell$.*

Proof: Assume that job J_a is a job of task τ_i. Therefore, $d_a = r_a + D_i$. According to IPDD policy, $t_a = r_a + (D_i - D_\ell)$. Consequently, $(d_a - t_a) = D_\ell$. ∎

Lemma 2. *If job J_a has higher priority than another job J_b at time t according to IPDD policy, then*

1) *the deadline d_a is smaller than the deadline d_b, and*
2) *J_a's priority never becomes smaller than that of J_b.*

Proof: Consider that J_a and J_b have priorities ν and ℓ at time t where $\nu < \ell$. We will show that (1) $d_a < d_b$, and (2) J_a's priority is never becomes smaller than that of J_b.

Since J_b's priority is ℓ at time t, its priority will ultimately be promoted to (higher) priority level ν according to

IPDD policy. Let J_b's priority will be promoted to priority ν at time t_b where $t < t_b$. On the other hand, J_a's priority is already at priority ν at time t. Let J_a's priority was set to priority ν at time t_a where $t_a \leq t$. Therefore, $t_a < t_b$. From Lemma 1, it follows that $(d_a - t_a) = D_\nu$ and $(d_b - t_b) = D_\nu$. Since $t_a < t_b$, it follows that $d_a < d_b$ (part (1) is proved).

It follows from IPDD policy that the priorities of J_a and J_b are set to priority level κ, for $\kappa = \nu, (\nu - 1), \ldots 1$, respectively at time $(t_a + D_\nu - D_\kappa)$ and $(t_b + D_\nu - D_\kappa)$. Since $t_a < t_b$, we have $(t_a + D_\nu - D_\kappa) < (t_b + D_\nu - D_\kappa)$ for $\kappa = \nu, \ldots 1$. Therefore, priority of J_a is promoted to higher priority level earlier than that of J_b. Any job having priority κ remains at priority level κ for duration of $(D_\kappa - D_{\kappa-1})$ time units in IPDD policy. Therefore, J_a's priority is never smaller than that of J_b (part (2) is proved). ∎

Lemma 3. *Consider that job J_a has priority ℓ at time t according to IPDD promotion policy. If a new job J_b of task τ_ℓ is released at time t, then the $d_a \leq d_b$.*

Proof: According to IPDD policy, job J_b of task τ_ℓ has priority ℓ at time t since it is released at time t. Since J_b is released at time t, we have $(d_b - t) = D_\ell$.

Job J_a's priority is already at priority ℓ at time t. Let J_a's priority was set to priority level ℓ at time t_a where $t_a \leq t$. From Lemma 1, we have $(d_a - t_a) = D_\ell$. Since $t_a \leq t$, we have $(d_a - t) \leq D_\ell$. From $(d_b - t) = D_\ell$ and $(d_a - t) \leq D_\ell$, its follows that $d_a \leq d_b$. ∎

Lemma 4. *Consider set $\mathcal{J}$ of active jobs. If all the jobs in $\mathcal{J}$ have same priority ℓ at some time instant, then the job with the earliest deadline is promoted to priority level ν no later than that of any other job in $\mathcal{J}$, where $\nu < \ell$.*

Proof: Consider any two jobs J_a and J_b in $\mathcal{J}$. Without loss of generality assume that $d_a \leq d_b$. Let J_a and J_b are promoted to higher-priority level ν at time t_a and t_b, respectively. We will show that $t_a \leq t_b$, which implies that J_a with deadline no later than that of J_b is promoted to priority level ν no later than that of job J_b.

It follows from Lemma 1 that $(d_a - t_a) = D_\nu$ and $(d_b - t_b) = D_\nu$. Since $d_a \leq d_b$, we have $t_a \leq t_b$. ∎

The dispatcher and the ready queue manager of FPP scheduler are presented in section IV. Section V presents techniques to reduce the number of promotion points by not assigning priorities to all the tasks using IPDD policy (i.e., some tasks have no promotion point).

IV. Dispatcher and Ready Queue Manager

The dispatcher of the FPP scheduler determines which active job to execute while the ready-queue manager is responsible for managing the ready jobs in the ready queue.

A. The Dispatcher

The dispatcher of FPP scheduler considering global multiprocessor scheduling on m identical processors is presented below. When $m = 1$, this dispatcher applies to uniprocessor. In addition, some important events related to the operations performed by the ready-queue manager are also highlighted below. The FPP dispatcher at each time t works as follows.

- At most m highest-priority jobs at time t are dispatched for execution. If t is the promotion point for a currently-executing job, then its priority is promoted[2] at time t.
- If all the m processors are busy and a new job J_{new} with priority *higher* than that of the currently-executing lowest-priority job J_{exe_low} is released at time t, then J_{new} starts execution by preempting J_{exe_low}. The preempted job J_{exe_low} is inserted in the ready queue. This (insertion) event managed by the ready-queue manager is called the "rel_prmt" event.
- If all the m processors are busy and a new job J_{new} with priority *not* higher than that of the currently-executing lowest-priority job J_{exe_low} is released at time t, then J_{new} does not preempt J_{exe_low}. And, J_{new} is inserted in the ready queue. This (insertion) event managed by the ready-queue manager is called the "rel_no_prmt" event. Note that if J_{new} has the same priority as that of J_{exe_low} (ties in priority ordering), then J_{new} does not preempt J_{exe_low}.
- If some processor becomes idle while the ready queue is not empty, then the job having the highest priority from the ready queue is removed and dispatched for execution on the idle processor. The ready-queue manager performs this removal and this event is called the "idle_remv" event.

Example 1 presents the FPP schedule for three tasks. Theorem 1 proves that FPP scheduling executes jobs in EDF order if all tasks have priorities based on IPDD.

Theorem 1. *If tasks are given priorities based on the IPDD policy, then the jobs of the tasks are executed in EDF order by the FPP scheduler at each time instant t.*

Proof: If the number of active jobs is no more than m, then each active job is executing at time t on separate processor. The claim of this theorem holds trivially.

Now consider the case when the number of active jobs is exactly m at time t. If a new job J_{new} arrives at time t (i.e., number of active job becomes larger than m) such that the priority of J_{new} is not higher than that of the currently-executing lowest-priority job J_{exe_low}, then J_{new} is inserted in the ready queue. If J_{new}'s priority is smaller than that of J_{exe_low}, then from Lemma 2 it follows that

[2]For example, to perform the promotion, a special task can be designed whose only job is to promote the priority of the application tasks, as pointed by Burns [9] for dual-priority scheduling. In addition, all the priority promotions of a currently-executing job may be postponed until a new job is released. This is because the execution of a currently-executing job may be interfered only if a new job is released. When a new job is released at time t, the priority of the currently-executing job is determined considering the last priority promotion at or immediately before t. This can avoid unnecessary overhead due to priority promotion of the executing tasks.

the absolute deadline of job J_{new} is larger than that of job J_{exe_low}. If J_{new}'s priority is equal to J_{exe_low}, then it follows from Lemma 3 that the deadline of job J_{new} is not smaller than that of job J_{exe_low}. Similarly, since J_{exe_low} is the currently-executing lowest-priority job, its deadline is not smaller than any other currently-executing job. Therefore, job J_{new} with EDF priority not higher than any of the currently-executing job is inserted in the ready queue.

On the other hand, if J_{new} has higher priority than that of job J_{exe_low}, then J_{new} preempts the execution of J_{exe_low} and J_{exe_low} is inserted in the ready queue. According to Lemma 2, job J_{new} has earlier deadline than that of job J_{exe_low}. Therefore, job J_{exe_low} having a relatively lower EDF priority is inserted in the ready queue.

According to Lemma 2, if job J_a is prioritized by the dispatcher over another job J_b, then job J_b will never have higher priority (even if its priority might be promoted) than job J_a at another (future) time instant. Therefore, no job that is inserted in the ready queue can preempt the jobs that are in execution.

Whenever some processor becomes idle at time t while the ready queue is not empty, the highest-priority job from the ready queue is removed and dispatched for execution. However, due to priority promotion, there may be multiple jobs waiting at the highest priority level in the ready queue. It will be shown in subsection IV-B that the ready-queue manager (when handling the `idle_remv` event) removes the job with shortest deadline (i.e., highest-priority EDF job) from the ready queue. Therefore, jobs are executed in EDF priority order in FPP scheduling in all cases. ■

Now we concentrate on the ready queue manager, in particular, the events it has to manage. In addition to the `rel_prmt`, `rel_no_prmt` and `idle_remv` events, the ready queue manager needs to handle another event. If some job's priority is to be promoted while that job is awaiting execution in the ready queue, the ready-queue manager needs to manage this promotion. This event managed by the ready-queue manager is called "`pri_prom`" event. Therefore, the ready-queue manager needs to handle four different events: `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom`. If multiple events occur at the same time, they are managed by the ready-queue manager in any order.

A note on ready-queue manager for FP scheduler. The ready-queue manager of traditional FP scheduling also needs to manage the `rel_prmt`, `rel_no_prmt` and `idle_remv` events. The ready-queue for FP scheduler can be implemented as an array of length n where task control blocks (TCBs) of the ready tasks are stored. The κ^{th} position of the array stores the TCB of the ready task that has current priority κ for $\kappa = 1, \ldots n$.

If `rel_prmt` or `rel_no_prmt` event occurs, then a job (particularly, J_{exe_low} or J_{new}) is inserted in the ready queue. If a job of task τ_κ is to be inserted in the ready-queue, then the priority of τ_κ is used to index the ready-queue array position at which the TCB of τ_κ is stored. Therefore, insertion is done in constant time.

If an `idle_remv` event occurs, then the highest-priority job from the ready queue is removed and dispatched for execution. Finding the highest-priority job from the ready-queue can be performed in constant time as follows. A bitmap array $B[n \ldots 1]$ of the ready-queue array is maintained. Initially, all the elements in bitmap B are zero to specify that there is no job awaiting execution at any priority level in the ready-queue array. When a job with priority κ is inserted in to the ready-queue array, the κ^{th} bit of the bitmap is set (i.e., $B[\kappa] = 1$) to specify that there is a job awaiting execution in the ready queue at priority level κ. Determining the highest-priority job from the ready queue is to find the position of the least set bit in the bitmap $B[n \ldots 1]$, which can be performed in constant time using, for example, deBruijn sequence [21], if not supported as a machine-level instruction [25]. Once the position is known, the TCB of the job is removed and corresponding job is dispatched and we set $B[\kappa] = 0$.

A note on ready-queue manager for EDF scheduler. Implementing EDF requires to keep track of all absolute deadlines and perform a dynamic mapping between absolute deadlines and priorities. If the number of possible absolute deadlines for all the active tasks is larger than the the total number of distinct priority levels, managing the ready tasks is complex in EDF scheduling. If the number of active tasks is smaller than the number of different priority levels, updating the ready queue has higher overhead in comparison to that of FP scheduling, as is discussed by Buttazzo [10]. In the worst case, all the ready jobs may need to be remapped to new priority levels, which increases the overhead of ready queue management. The complexity and overhead in managing ready queue of EDF scheduler make it less popular in commercial kernel although EDF always performs better in terms of schedulability on uniprocessor.

Theorem 1 shows that the FPP scheduling executes jobs similar to EDF when tasks are given priorities using IPDD policy. However, managing jobs in the ready queue of FPP scheduler, if implemented similar to that of known for EDF, will have the same overhead problems that EDF suffers. In this paper, a new ready-queue management scheme for FPP scheduler is proposed. In particular, a data structure for the ready queue and constant-time operations to manage each of the `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom` events is proposed.

In FPP scheduling, if the priority of a currently-executing job is promoted, then such promotion does not remap any job in the ready queue. However, if the priority of a job *residing* in the ready queue is promoted, then such promotion (as will be evident shortly) remaps that job to a new position in the ready queue data structure and thus incurs overhead.

Inspired by the discussion of Buttazzo [10], the *overhead model* this paper considers is the sum of total number of times each of the jobs in the ready queue is remapped to some other position. It will be empirically shown, based on this overhead model, that `FPP` scheduler has significantly lower overhead than that of `EDF`. Such low overhead of `FPP` scheduler shall make it popular in practice.

B. The Ready Queue Manager.

This subsection presents the data structure of the ready queue and operations to handle the events `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom`.

1) Data-Structure for the Ready Queue: Due to priority promotion and sharing of priority levels in `FPP` scheduling, multiple active jobs may have the same priority at the same time instant. Therefore, the ready queue may need to store more than one job at the same priority level. An array of total n linked lists are used to implement the ready queue of the `FPP` scheduler. The κ^{th} linked list at any time instant stores all the TCBs of the ready jobs that have priority level κ at that time instant.

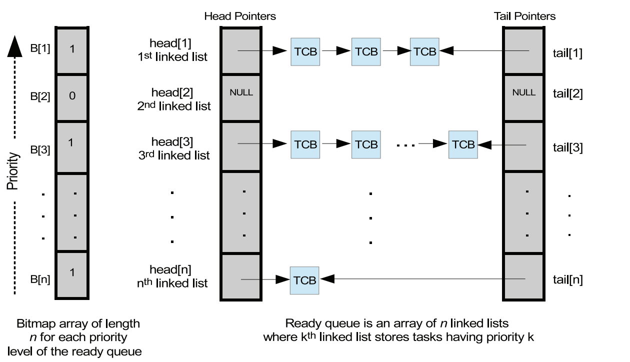

Figure 3. Proposed data structure of the ready queue for `FPP` scheduling.

The κ^{th} linked list has two pointers: `head[`κ`]` and `tail[`κ`]` that respectively point the first and last TCB in the κ^{th} linked list. This ready queue data structure along with the bitmap is depicted in Figure 3. The purpose of the bitmap $B[n \ldots 1]$ is to perform efficient searching to find the highest priority job from the ready queue.

2) Operations by the Ready Queue Manager: The ready-queue manager updates the ready queue whenever `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom` event occurs. The jobs stored in the ready queue at any time instant will satisfy the following two properties:

- **P1**: All jobs stored in the ν^{th} linked list of the ready queue have higher `EDF` priorities than any other job stored in the ℓ^{th} linked list where $\nu < \ell$.
- **P2**: All jobs in the ℓ^{th} linked list are stored in order of non-increasing `EDF` priority, i.e., the `head[`ℓ`]` and `tail[`ℓ`]` respectively points the highest and lowest priority `EDF` job in the ℓ^{th} linked list for $\ell = 1, \ldots n$.

Assume that these two properties hold at time t_0 (such t_0 exists at least for the case when the system starts, i.e., when there is no job in the ready queue). Consider that some event (`rel_prmt`, `rel_no_prmt`, `idle_remv` or `pri_prom`) occurs at time t such that there is no other event after t_0 and before t. We will show that how properties P1 and P2 continue to hold after ready queue is updated to handle the event that occurs at time t. Given the structure of the ready-queue in subsection IV-B1, operations on the ready queue for managing events `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom` are presented below.

Event `rel_prmt`. This event occurs if a newly released job J_{new} starts executing by preempting currently-executing lowest-priority job J_{exe_low}. The TCB of job J_{exe_low} is inserted in the ready queue. If priority of job J_{exe_low} is ℓ when preempted by J_{new}, then the TCB of job J_{exe_low} is **inserted at the front** of the ℓ^{th} linked list of the ready queue. The insertion at the front is done in constant time using the `head[`ℓ`]` pointer. If the bit $B[\ell] = 0$, then we set $B[\ell] = 1$ to specify that there is a TCB awaiting execution at priority level ℓ.

Since job J_{exe_low} has priority ℓ at time t and was in execution, the $1^{st}, 2^{nd}, \ldots (\ell - 1)^{th}$ linked lists of the ready queue at time t are empty. It follows from the proof of Theorem 1 that any job in the ready queue at time t neither has higher priority nor has earlier deadline than the currently-executing lowest-priority job J_{exe_low}. Therefore, inserting job J_{exe_low} at the front of the ℓ^{th} linked list at time t guarantees that P1 and P2 continues to hold. ■

Event `rel_no_prmt`. This event occurs if a newly released job J_{new} cannot preempt currently-executing lowest-priority job J_{exe_low} and J_{new} is inserted in the ready queue. If priority of J_{new} is ℓ at time t, then the TCB of J_{new} is **inserted at the end** of the ℓ^{th} linked list. The insertion at the end is done in constant time using the `tail[`ℓ`]` pointer. If bit $B[\ell] = 0$, then we set $B[\ell] = 1$.

Since P1 holds at time t_0 and job J_{new} has priority ℓ at time t, it follows that all the jobs in the κ^{th} linked list at time t have higher and lower `EDF` priorities than that of job J_{new} where $\kappa < \ell$ and $\kappa > \ell$, respectively. According to Lemma 3, all the jobs in the ℓ^{th} linked list at time t have their absolute deadlines no later than that of job J_{new} since job J_{new} is released at time t. Therefore, inserting J_{new} at the end of the ℓ^{th} linked list at time t guarantees that P1 and P2 continues to hold. ■

Event `idle_remv`. This event occurs when some processor becomes idle while the ready-queue is not empty. In such case, the highest-priority job from the ready queue is removed and dispatched for execution. The highest-priority job is in the *lowest-indexed non-empty* linked list. The lowest-indexed non-empty linked list (i.e., non-empty linked-list at the highest priority level) is found in constant time using bitmap B based on the same technique used to find the highest-priority ready task in traditional `FP` scheduling, for example, using deBruijn sequence [21].

Assume that the ℓ^{th} linked list of the ready queue is the lowest-indexed non-empty linked list. Note that there may be multiple jobs awaiting execution in the ℓ^{th} linked list.

The job from the **front** of the ℓ^{th} linked list is **removed** and dispatched for execution. The removal from the front is done in constant time using `head[`ℓ`]` pointer. If `head[`ℓ`]` becomes NULL after this removal (i.e., the ℓ^{th} linked list becomes empty), then we set the $B[\ell] = 0$ to specify that there is no TCB awaiting execution at priority level ℓ.

Since property P2 holds at time t_0, the job from the **front** of the lowest-indexed non-empty linked list has the highest `EDF` priority at time t. And, after the removal of this job the remaining jobs in the ready queue also satisfy P1 and P2 since removal a job cannot violate P1 or P2. ■

Event `pri_prom`. This event occurs at time t when the priority of some job in the ready queue is to be promoted. If the priority of a job from the ℓ^{th} linked list is to be promoted to priority level ν, then this job is removed from the ℓ^{th} linked list and inserted to the ν^{th} linked list.

Since P2 holds at time t_0, the job at the front of the ℓ^{th} linked list has deadline no later than any other jobs in ℓ^{th} linked list. According to Lemma 4, given a set of jobs having the same priority ℓ at time t_0, the priority of the job with earliest deadline will be promoted to priority level ν no later than any other job in that set. Consequently, the priority of the job at the front of the ℓ^{th} linked list is to be promoted to priority level ν. Let job J_a is the job at the front of the ℓ^{th} linked list.

The TCB of job J_a is **removed from the front** of the ℓ^{th} linked list and **inserted at the end** of the ν linked list. The removal and insertion can be done in constant time using the `head[`ℓ`]` and `tail[`ν`]` pointers, respectively. Finally, if the ℓ^{th} linked list becomes empty after this removal, then we set the $B[\ell] = 0$. If $B[\nu] = 0$, then we set the $B[\nu] = 1$ to specify that the ν^{th} linked list is now not empty.

Since the promoted priority of job J_a is ν at time t, all jobs in the κ^{th} linked list, where $\kappa > \nu$, have absolute deadline larger than that of job J_a at time t according to Lemma 2. Since P1 holds at time t_0 and job J_a is in the ℓ^{th} linked list at time t_0, it follows that all the jobs in the ν^{th} linked list have smaller absolute deadlines than that of job J_a. Therefore, inserting J_a at the end of the ν^{th} linked list ensures that P1 and P2 continues to hold. ■

In summary, when all the tasks are given priorities based on `IPDD` policy, the `FPP` scheduler executes jobs in `EDF` priority order. Therefore, existing `EDF` schedulability tests for uniprocessor and multiprocessors (i.e., `G-EDF` test) can be used to determine whether `FPP` scheduling can guarantee the schedulability of the tasks that are given priorities using `IPDD` policy.

While the ready queue management of `FPP` scheduler has similar implementation benefits of `FP` scheduler (i.e., each event can be handled in constant time), the only source of additional overhead in comparison to `FP` scheduler is the cost of priority promotion. However, the number of promotion points can be reduced by assigning some tasks of a task set traditional fixed priorities with no promotion point. To this end, a technique to reduce the total number of promotion points and a new schedulability test called `FPP_Test` for `FPP` scheduling are proposed.

V. `FPP_TEST` TO REDUCE NUMBER OF PROMOTIONS

In this section, a schedulability test called `FPP_Test` to determine whether a task set is schedulable in `FPP` scheduling is proposed. The `FPP_Test` also determines the priorities of the tasks where a subset of the tasks is assigned traditional fixed priorities (without any priority promotion) while other tasks are assigned priorities (with priority promotion) based on the `IPDD` policy.

To determine which tasks can be assigned fixed priorities with no promotion point, an important feature of the state-of-the-art `FP` schedulability test is exploited. For uniprocessor and multiprocessor (global) `FP` scheduling, the corresponding state-of-the-art schedulability tests are of *iterative* nature: the schedulability of each task τ_i is tested separately. For example, the well-known response time analysis (RTA) for `FP` scheduling on uniprocessor [1] and multiprocessors [24] is of iterative nature: the response time R_i of each task τ_i is computed. The crucial observation is that when determining the schedulability of τ_i using an iterative `FP` schedulability test, the worst-case interference computation due to the higher-priority tasks in set $hp(i)$ does not assume that the jobs of the tasks in $hp(i)$ are also scheduled using `FP` scheduling; rather, it only assumes that jobs of the tasks in $hp(i)$ have higher priorities and cause maximum interference on τ_i. Consequently, Corollary 1 holds.

> **Corollary 1:** If task τ_i is deemed to be schedulable at priority level ℓ using an iterative test where $hp(i)$ is assumed to be the set of higher priority tasks, then the schedulability of τ_i is preserved when it is assigned traditional fixed-priority level ℓ regardless whether the jobs of the tasks in $hp(i)$ are scheduled using dynamic or fixed priority.

It follows from Corollary 1 that if some task τ_i is deemed schedulable using an iterative test at fixed-priority level ℓ, then task τ_i does not need to have any promotion point and the tasks in $hp(i)$ may be assigned fixed or `IPDD` (i.e., essentially dynamic) priorities in `FPP` scheduling. The `FPP_Test` is designed based on this observation.

The `FPP_Test` (presented in Figure 4) requires two schedulability tests to determine the schedulability of a task set in `FPP` scheduling. These two tests, denoted by T_{fp} and T_{edf} in Figure 4, are not "real" schedulability tests. Depending on the task model (e.g., implicit-, constrained- or arbitrary-deadline) and processor platform (uniprocessor or multiprocessors), we will plug in the state-of-the-art iterative `FP` test and `EDF` test respectively in place of T_{fp} and T_{edf}. In section VI–VII, the actual tests used in place of T_{fp} and T_{edf} are presented respectively for uniprocessor and multiprocessor platform.

Algorithm: `FPP_Test` (Task Set Γ)

1. **For** priority level $k = n$ to $k = 1$
2.   **For** each priority-unassigned task $\tau_i \in \Gamma$
3.   **If** τ_i is schedulable at priority level k using
4.   test T_{fp} with all other priority-unassigned
5.   tasks assumed to have higher priorities
6.   **Then**
7.   assign τ_i to priority `k`
8.   break (continue outer loop)
9.   **End If**
10.   **End For**
11.   **If** all the priority-unassigned tasks pass T_{edf}, **Then**
12.   Compute promotion points only for the
13   priority-unassigned tasks using `IPDD` policy
        //***Comment:*** *`IPDD` policy uses (higher) priority levels*
        // $k, (k-1) \dots 1$ *for these priority-unassigned tasks*
14.   **Return** True
15.   **Else**
16.   **Return** False
17.   **End If**
18. **End For**
19. **Return** True

Figure 4. Improved priority promotion policy for `FPP` scheduling.

The `FPP_Test` in Figure 4 takes as input a task set Γ and returns "true" if the task set is deemed to be `FPP` schedulable; otherwise, it returns "false". The `FPP_Test` also determines the tasks that are given fixed priorities and the tasks that are given priorities based on `IPDD` policy.

Initially, all the tasks in set Γ are "priority-unassigned" in Figure 4, i.e., no task has any priority. Based on Audsley's OPA algorithm [2], the `FPP_Test` starts assigning traditional fixed priorities to the tasks starting from the lowest priority level. For each priority level k in line 1, some priority-unassigned task is searched using the inner loop in line 2-10 to assign it the fixed-priority level k. Whether or not a (priority-unassigned) task, say task τ_i, can be assigned priority level k is determined in line 3–5 by applying the iterative `FP` test T_{fp} and assuming higher priorities for all other (priority-unassigned) tasks. If such a task τ_i is found in line 3–5, then task τ_i is assigned the traditional fixed-priority level k in line 7 and the priority assignment for next (higher) priority level starts by jumping from line 8 to line 1. If the outer loop in line 1–18 terminates after assigning fixed priorities to all the tasks in Γ in line 7, then the algorithm returns "true" in line 19. And, `FPP` schedules all tasks similar to `FP` scheduling without any priority promotion.

If no task can be assigned the current priority level k (i.e., the test in line 3–5 is false for all the priority-unassigned tasks), then the inner loop in line 2–10 terminates. In such case, there exist some priority-unassigned tasks. The schedulability of all these priority-unassigned tasks are tested in line 11 by applying test T_{edf}. If these priority-unassigned tasks pass T_{edf}, then the promotion points *only* for these priority-unassigned tasks are computed in line 12-13 based on the `IPDD` policy and the algorithm returns "true" in line 14. Otherwise, the algorithm returns "false" in line 16. Note that the tasks assigned priorities using the `IPDD` policy in line 12-13 have higher priorities than any task that is given traditional fixed priority in line 7.

The `FPP_Test` guarantees schedulability of Γ using `FPP` scheduling if it returns "true". Assume that when the algorithm returns true, there are q tasks that are assigned traditional fixed priorities in line 7 and the remaining $(n-q)$ tasks are given priorities based on `IPDD` policy in line 12-13 for some q, $0 \leq q \leq n$. Each of the q tasks that is given traditional fixed priority in line 7 is schedulable in `FPP` scheduling based on **Corollary 1**. The schedulability of the $(n-q)$ tasks that are given `IPDD` priorities is not affected by the q tasks because these q tasks are given lower (traditional) fixed priorities. Since the $(n-q)$ tasks, having priorities based on `IPDD` policy, are essentially scheduled in `EDF` order by the `FPP` scheduler (proved in Section IV), satisfying the T_{edf} test in line 11 guarantees that these $(n-q)$ tasks are also schedulable in `FPP` scheduling. If a task set is schedulable using traditional `FP` scheduling, then no promotion point is assigned to any task using the `FPP_Test` and all tasks are executed similar to traditional `FP` scheduling using `FPP` scheduler.

The ready queue management scheme of subsection IV-B2 still applies when priorities are assigned using `FPP_Test`. This is because the ready jobs of the q tasks having traditional fixed priorities are (i) stored in the linked lists corresponding to the q lower (i.e., $(n-q+1), \dots n$) priority levels, (ii) never promoted to a higher priority level since they have no promotion point, and (iii) dispatched for execution only after all the jobs that are given the $(n-q)$ higher $(i.e., (n-q), \dots 1)$ priority levels are dispatched for execution. Properties P1 and P2 (defined in Section IV-B2) do not necessarily need to hold for the tasks that are assigned traditional fixed priorities but always hold for the tasks assigned priorities using `IPDD` policy.

VI. `FPP_TEST` FOR UNIPROCESSOR

In this section, the `FPP_Test` in Figure 4 is applied for determining schedulability of constrained-deadline tasks on uniprocessor. The response-time test for uniprocessor `FP` scheduling proposed by Audsley et al. [1] is considered in place of T_{fp} in Figure 4 to determine whether a (priority-unassigned) task τ_i can be assigned fixed priority level k. Note that this response-time test (which is an exact test) combined with Audsley's OPA algorithm in Figure 4 guarantees optimal fixed-priority assignment. And, the quick processor demand analysis (QPA), which is an exact `EDF` test, proposed by Zhang and Burns [26], is considered in place of T_{edf} in line 11 to determine whether all the priority-unassigned tasks are schedulable using `EDF`. The QPA test is an efficient implementation of the processor demand analysis proposed by Baruah et al. [4].

If a task set is `EDF` schedulable, then the QPA test in line 11 will also be satisfied when not all the tasks are assigned fixed priorities in line 7. Consequently, any task set that is schedulable using optimal EDF scheduling on uniprocessor also satisfies the FPP_Test. Since preemptive EDF is optimal [15], the FPP scheduling where priorities are assigned using the FPP_Test is also an optimal scheduling algorithm for constrained-deadline tasks on uniprocessor. For implicit-deadline task sets, the utilization bound of FPP algorithm is thus 100% because the utilization bound of EDF for such task system is 100% [23].

While the performance of FPP scheduling in terms optimality is same as EDF scheduling, it is not straightforward to see whether the overhead for managing jobs in the ready queue of FPP scheduler is lower or higher than that of EDF scheduler. Simulation using randomly generated task sets is conducted to measure such overhead.

Task set generation for uni- and multiprocessors. Each of the experiments is characterized by a pair (m, n) where m is the number of processors and n is the cardinality of a task set. For experiments on uniprocessor, we use $m = 1$. The UUnifast-Discard algorithm [12] is used to generate n utilization values of a task set. This algorithm takes as input the number of tasks n and total utilization U of the n tasks. And, it generates n utilizations $\{u_1, u_2, \ldots u_n\}$ of the n tasks such that the total utilization of these n tasks is U. Once a set of n utilizations $\{u_1, u_2, \ldots u_n\}$ of a task set is generated, the other parameters of each task τ_i in the task set are generated as follows:

- The minimum inter-arrival time T_i of each task τ_i is generated from the uniform random distribution within the range $[10ms, 1000ms]$.
- The WCET of task τ_i is set to $C_i = u_i \cdot T_i$.
- The relative deadline D_i of task τ_i is generated from the uniform random distribution within the range $[C_i, T_i]$ for constrained deadline tasks; otherwise D_i is set to T_i for implicit-deadline tasks.

Task sets are randomly generated at 40 different utilization levels $\{0.025m, 0.05m, \ldots 0.975m, m\}$ for each experiment (m, n). A total of 1000 task sets at each of the 40 utilization levels are generated. Each of the 1000 task sets generated at a particular utilization level, say U, has cardinality n and total utilization equal to U.

Sources of Overhead. Insertion/removal of jobs to/from the ready queue of FPP scheduler (as discussed in subsection IV-B2) can be done in constant time. However, jobs that are in the FPP ready queue may need to change their position (i.e., upgraded to higher-priority linked list) due to pri_prom events. On the other hand, if the ready queue of EDF scheduler is implemented as binary min-heap [17] or binomial min-heap [6], then each insertion/removal of a job to/from the ready queue of EDF scheduler may need to reorder the remaining jobs in the ready queue in order to satisfy the *min-heap* property. Our objective is to compare such overhead in terms of total number of times different jobs in the ready queue change their position to handle pri_prom event (in FPP scheduling) and to maintain min-heap property (in EDF scheduling). The EDF ready queue is simulated using a binary min-heap where the ready job with the shortest absolute deadline is stored in the root. And, the FPP ready queue is simulated using the proposed data structure in Figure 3.

Experiments (Uniprocessor). The randomly-generated task sets that are (exclusively) schedulable using FPP/EDF (i.e., satisfy FPP_Test) and *not* schedulable using traditional FP scheduling are considered to compare overheads between FPP and EDF. Figure 5 presents the number of such task sets for each utilization level for different n.

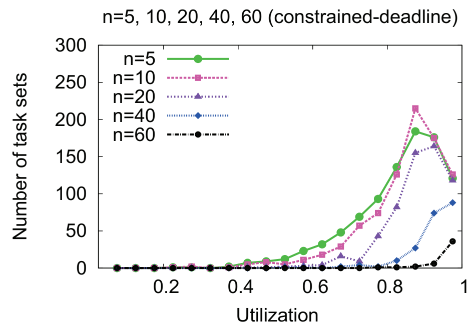

Figure 5. Number of task sets (out of 1000 task sets) that are schedulable using FPP/EDF scheduling but not schedulable using FP scheduling.

For each such task set, the execution is simulated using both FPP and EDF scheduling. The ready jobs that need to await execution are stored in the corresponding ready queue and reordered when necessary. Since it is not computationally feasible to consider all possible release offsets and inter-arrival separations of sporadic tasks exhaustively in simulation, all release offsets are set to zero and all tasks are released periodically. The simulation is run for L time units where $L = min\{lcm(T_1, T_2, \ldots T_n), 10^8\}$ to avoid simulation for very large hyperperiod.

Overhead Metric. For each utilization level, the sum of total number of times each of the jobs of a task set change their position in the ready queue is computed and then the average over all task sets is determined for both FPP and EDF scheduling. $\texttt{EDF}_{av}$ and $\texttt{FPP}_{av}$ denote the average number of times the jobs of a task set change their positions in EDF and FPP ready queue, respectively. The improvement of managing jobs in the ready queue of FPP scheduler in comparison to EDF scheduler at each utilization level is:

$$\texttt{Improvement} = \frac{\texttt{EDF}_{av} - \texttt{FPP}_{av}}{max\{\texttt{EDF}_{av}, \texttt{FPP}_{av}\}} \times 100\%$$

The value of Improvement ranges in $[-100\%, +100\%]$. For example, Improvement = -50% implies that the ready queue of EDF scheduler on average can reduce 50% overhead of managing jobs in the ready queue of FPP scheduler. And, Improvement = $+60\%$ implies that ready queue of FPP scheduler on average can reduce 60% overhead of managing jobs in the ready queue of EDF scheduler.

Empirical Results (Uniprocessor). The results of a series of simulations using different $n \in \{5, 10, 20, 40, 60\}$ are presented in Figure 6 where the x-axis is the utilization level U and the y-axis represents Improvement. The Improvement is non-negative in almost all the utilization levels[3]. The improvement of FPP scheduler over EDF scheduler is significant in most cases, i.e., FPP incurs noticeably less overhead (in terms of number of times jobs in the ready queue are remapped to new positions) than that of EDF.

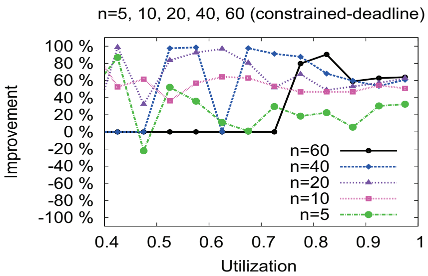

Figure 6. Improvement of FPP over EDF scheduling.

The "positive" improvement of FPP is due to two main reasons. First, the proposed data structure for FPP ready queue enables a job to be inserted/removed to/from the ready queue in constant time without causing other existing jobs in the ready queue to change their position. In contrast, each insertion/deletion to/from the ready queue of EDF scheduler may cause multiple (i.e., $O(logn)$) jobs to change their positions to maintain the min-heap property. Second, the FPP_Test test is effective in reducing the number of promotion points. This is verified by observing (the outcome of FPP_Test on random task sets) that it is almost always the case where some tasks for the majority of the task sets are given traditional fixed priorities with no promotion point.

Observing the significant reduction in overhead, it is expected that if the ready queue of EDF scheduler is implemented using some other data structure, the benefit of proposed ready-queue management scheme for FPP scheduler will still be realized. To verify this, experiment using other data structure is needed and is left as a future work.

VII. FPP_TEST FOR MULTIPROCESSORS

In this section, the FPP_Test in Figure 4 is applied to determine schedulability of constrained-deadline tasks scheduled on multiprocessors. For G-FP scheduling, Pathan and Jonsson [24] recently proposed an iterative G-FP test that is shown to perform better than any other iterative test proposed earlier. This G-FP test [24] is used in place of T_{fp} in Figure 4 to determine if a priority-unassigned task τ_i can be assigned fixed priority level k. For G-EDF scheduling, Bertogna and Baruah [5] proposed a step-by-step approach to apply different G-EDF schedulability tests proposed by other researchers. This G-EDF test in [5] is used in place of T_{edf} in line 11 of Figure 4 to determine if all the priority-unassigned tasks are schedulable on m processors using G-EDF scheduling.

[3]The value of Improvement at relatively lower utilization levels (e.g., when $U < 0.4$) is caused by very few task sets (Figure 5 presents the number of such task sets) and such outliers can be ignored.

There is no evidence regarding whether the G-FP test in [24] dominates or is dominated by the G-EDF test in [5]. It is not difficult to see that if a task set is deemed to be schedulable using G-FP test [24] or G-EDF test [5], then that task set also passes the FPP_Test for multiprocessors. In other words, the FPP_Test dominates the state-of-the-art G-FP and G-EDF tests. To measure the improvement of FPP_Test over G-FP test and G-EDF test, experiments using randomly generated task sets are conducted.

Empirical Results. For each experiment (m, n), random task sets are generated using the approach presented earlier. The schedulability of each of the 1000 task sets generated at each utilization level is determined based on FPP_Test, G-FP test [24] and G-EDF test [5]. The acceptance ratio for each test at each utilization level is computed. The acceptance ratio of a schedulability test is the percentage of task sets deemed schedulable at a given utilization level.

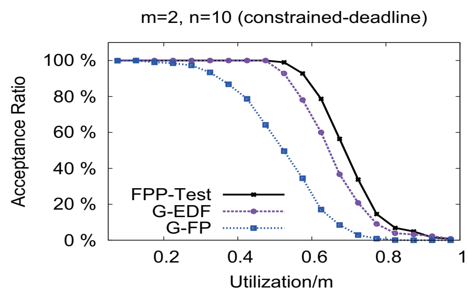

Figure 7. Acceptance ratio of FPP_Test, G-FP [24] and G-EDF [5].

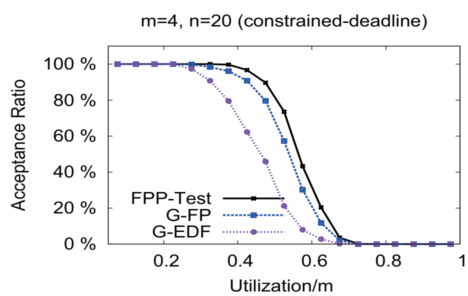

Figure 8. Acceptance ratio of FPP_Test, G-FP [24] and G-EDF [5].

A series of experiments for different (m, n), where $m \in \{2, 4, 8\}$ and $n \in \{3m, 5m, 10m\}$, are conducted. The result of two experiments with parameters $(m = 2, n = 10)$ and $(m = 4, n = 20)$ are presented in Figure 7 and Figure 8. The x-axis represents the system utilization U/m for utilization level U and the y-axis represents the acceptance ratio.

The performance of G-EDF test is better than G-FP test when $m = 2$ and $n = 10$ in Figure 7. This behavior is reversed in Figure 8. This shows neither G-FP nor G-EDF test empirically performs better than the other. The FPP_Test does not only theoretically dominate but also empirically performs better than both G-FP and G-EDF tests. The performance of FPP_Test test using implicit-deadline tasks is significantly better (see Figure 9).

The difference in acceptance ratios among the tests are more pronounced at higher utilization level since task sets

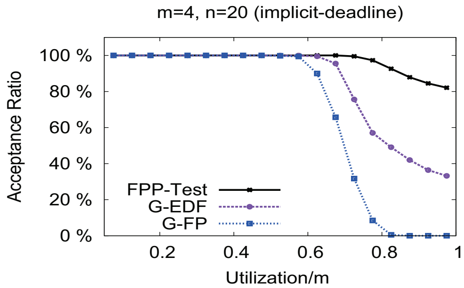

Figure 9. Acceptance ratio of `FPP_Test`, `G-FP` [24] and `G-EDF` [5].

with large total utilization are difficult to schedule. The `FPP_Test` has the ability to accept higher percentage of task sets in comparison to that of `G-FP` and `G-EDF` tests by exploiting the benefits of both fixed and dynamic priority.

A. Preemptions and Migrations

To investigate whether `FPP` scheduling incurs higher or lower number of preemptions and migrations in comparison to `G-EDF`, simulations are conducted. Execution of randomly-generated task sets that are not `G-FP` schedulable but schedulable using *both* `FPP` and `G-EDF` are simulated for `FPP` and `G-EDF` scheduling. For each utilization level $U \in \{0.025m, 0.05m, \ldots m\}$, the average number of preemptions and migrations that a task set suffers is computed for both `FPP` and `G-EDF` scheduling.

$\texttt{GEDF}_{avpr}$ and $\texttt{FPP}_{avpr}$ denote the average number of preemptions that a task set suffers in `G-EDF` and `FPP` scheduling, respectively. Similarly, $\texttt{GEDF}_{avmg}$ and $\texttt{FPP}_{avmg}$ denote the average number of migrations that a task set suffers in `G-EDF` and `FPP` scheduling, respectively. The improvement by `FPP` in reducing preemptions and migrations in comparison to `G-EDF` at each utilization level is:

$$\texttt{Improvement(prmt)} = \frac{\texttt{GEDF}_{avpr} - \texttt{FPP}_{avpr}}{max\{\texttt{GEDF}_{avpr}, \texttt{FPP}_{avpr}\}} \times 100\%$$

$$\texttt{Improvement(migr)} = \frac{\texttt{GEDF}_{avmg} - \texttt{FPP}_{avmg}}{max\{\texttt{GEDF}_{avmg}, \texttt{FPP}_{avmg}\}} \times 100\%$$

The value of improvement ranges in $[-100\%, +100\%]$. For example, `Improvement(prmt)` $= +50\%$ implies that `FPP` on average can reduce 50% preemptions that occur in `G-EDF` scheduling. The results of simulations for different $n \in \{10, 20\}$ and $m \in \{2, 4\}$ are presented in Figure 10 and Figure 11, where the x-axis is the system utilization U/m and y-axis represents `Improvement(prmt)` and `Improvement(migr)`, respectively.

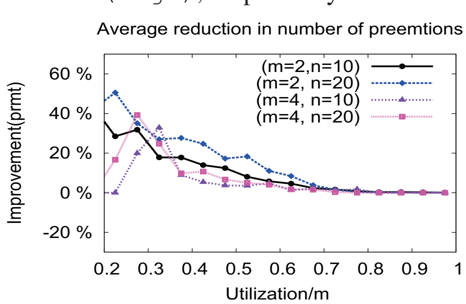

Figure 10. `Improvement(prmt)` of `FPP` over `G-EDF` scheduling.

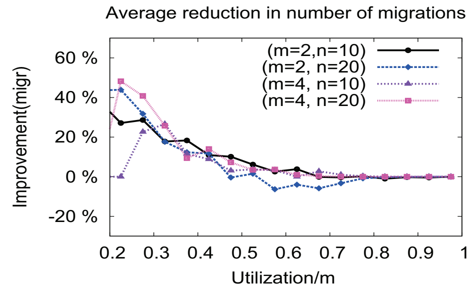

Figure 11. `Improvement(migr)` of `FPP` over `G-EDF` scheduling.

The value of `Improvement` for both preemptions and migrations is non-negative in almost all the utilization levels. `FPP` scheduling can significantly reduce the number of preemptions and migrations that are incurred in `G-EDF` scheduling. Since it is more difficult to schedule task sets at higher utilization levels, the improvement decreases as utilization level increases in Figure 10 and Figure 11.

B. Other Implementation Issues

Brandenburg and Anderson identified six different major sources of overhead in implementing `G-EDF` algorithm using a Linux extension, called LITMUS^{RT}, that allows different multiprocessors algorithm to be implemented as plugin components [7]. By conducting similar experiments for `FPP` scheduling, the execution time of each task can be inflated to account such overhead in the corresponding schedulability analysis. Although this paper does not implement `FPP` scheduling algorithm in an RTOS, two important implementation issues of `FPP` scheduler warrant further discussion: sharing the ready queue and managing timers for multiple `pri_prom` events.

If scheduling decisions are handled on multiple processors, for example, arrival of different jobs are handled concurrently on different processors (similar to [7]), then the ready queue of `FPP` scheduler is a shared resource. This ready queue needs to be protected against concurrent updates using synchronization primitives. As a result, the operations on the ready queue of `FPP` scheduler can be done in constant time (as discussed in Section IV-B) plus any additional delay incurred by such synchronization primitives. Note that race condition in handling multiple events in `FPP` scheduling will not occur. This is because when multiple `rel_prmt`, `rel_no_prmt`, `idle_remv` and/or `pri_prom` events occur very close in time, then these events can be handled in any order and properties **P1** and **P2** (defined in Section IV-B2) continue to hold regardless of the order these (nearly concurrent) events are processed.

Another issue is how multiple timers can be managed to handle multiple `pri_prom` events. The `pri_prom` event of a job in the ready queue may be implemented using *one-shot* timer. When timer expires, the handler can reposition the ready job to the appropriate higher-priority linked list of the ready queue. If the number of hardware timers is not sufficient to implement all the `pri_prom` events, then a queue of timers needs to be managed. Holenderski et al. [19] proposed interesting technique for managing a queue of timers based on *event queue* that multiplex multiple (future) timed events on a single hardware timer. By implementing the queue of timers as a binary min heap (using expiration time as the key), searching the insertion position of a new timed event can be done efficiently. Given the effectiveness of FPP scheduling in reducing (i) the number of re-mappings of jobs in the ready queue, and (ii) the number of preemptions and migrations, I expect that FPP scheduling when implemented on real platform would show benefits over G-EDF scheduling.

Applicability of `FPP_Test` to Arbitrary-Deadline Tasks. The FPP_Test in Figure 4 can also be applied arbitrary-deadline tasks as follows. For uniprocessor platform, the iterative FP test proposed by Lehoczky [20] can be used as the T_{fp} test and the QPA test [26], which also applies to arbitrary-deadline tasks, can be used as the T_{edf} test. For multiprocessor platform, the iterative OPA-incompatible global FP test proposed for arbitrary-deadline tasks by Guan et al. [16] can be made OPA-compatible using approach used by Davis and Burns for the DA-LC test in [12]. This new test then can be used as the T_{fp} test in Figure 4. And, the G-EDF test proposed by Baruah and Baker [3] can be used as the T_{edf} test for arbitrary-deadline tasks.

VIII. Conclusion

The proposed FPP scheduling algorithm unifies the fixed- and dynamic-priority scheduling paradigms. For uniprocessor, the FPP scheduling is also optimal as EDF scheduling. For multiprocessors, it dominates the state-of-the-art G-FP and G-EDF tests for constrained-deadline tasks. Technique to reduce the number of promotion points is proposed so that overhead is low. The proposed data structure and operations for managing jobs in the ready queue of FPP scheduler have benefits similar to that of traditional FP scheduler. Simulation results show that overhead for managing jobs in the FPP ready queue is reduced significantly in comparison to that of EDF scheduler.

Acknowledgement

This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Info. Proc. Letters*, 79(1):39–44, 2001.

[3] S. Baruah and T. Baker. Global edf schedulability analysis of arbitrary sporadic task systems. In *Proc. of ECRTS*, 2008.

[4] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. of RTSS*, pages 182 –190, 1990.

[5] M. Bertogna and S. Baruah. Tests for global edf schedulability analysis. *J. Syst. Archit.*, 57(5):487–497, May 2011.

[6] B. Brandenburg, J. Calandrino, and J. Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *Proc. of RTSS*, 2008.

[7] B. B. Brandenburg and J. H. Anderson. On the Implementation of Global Real-Time Schedulers. In *Proc. of RTSS*, pages 214–224, 2009.

[8] A. Burns. Dual Priority Scheduling: Is the Processor Utilisation bound 100%? . In *Proc. of the 1st International Real-Time Scheduling Open Problems Seminar (RTSOPS), in conjunction with the ECRTS*, 2010.

[9] A. Burns and A. Wellings. Dual Priority Assignment: A Practical Method of Increasing Processor Utilisation . In *Proc. of the Fifth Euromicro Workshop on Real-time Systems*, 1993.

[10] G. C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[11] R. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

[12] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47:1–40, 2011.

[13] R. Davis and A. J. Welling. Dual Priority Scheduling. In *Proc. of RTSS*, 1995.

[14] R. I. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. *Technical Report YCS 230, Dept of Computer Science, University of York, UK*, 1994.

[15] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[16] N. Guan, M. Stigge, W. Yi, and G. Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. *Proc. of RTSS*, pages 387–397, 2009.

[17] M. Gutierrez, M. Aldea, A. Burns, and M. Gonzalez. A Deadline-Floor Inheritance Protocol for EDF Scheduled Embedded Real-Time Systems with Resource Sharing. *to appear in the IEEE Transactions on Computers*.

[18] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. In *Proc. of RTSS*, 1991.

[19] M. Holenderski, W. Cools, R. Bril, and J. Lukkien. Multiplexing real-time timed events. In *Proc. of ETFA*, 2009.

[20] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of RTSS*, 1990.

[21] C. Leiserson, H. Prokop, and K. H. Randall. using de bruijn sequences to index a 1 in a computer word. *MIT Technical Report*, 1998. http://supertech.csail.mit.edu/papers/debruijn.pdf.

[22] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[23] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[24] R. M. Pathan and J. Jonsson. Interference-aware fixed-priority schedulability analysis on multiprocessors. *Real-Time Systems*, 50(4):411–455, 2014.

[25] M. Short. Improved task management techniques for enforcing edf scheduling on recurring tasks. In *Proc. of RTAS*, 2010.

[26] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9):1250–1258, Sept 2009.