Real-Time Scheduling on Uni- and Multiprocessors Based on Priority Promotions*

Risat Mahmud Pathan

Chalmers University of Technology
412 96, Göteborg, Sweden
http://orcid.org/0000-0002-9902-7558
risat@chalmers.se

Abstract

This paper addresses the problem of real-time scheduling of a set of sporadic tasks on uni- and multiprocessor platform based on priority promotion. A new preemptive scheduling algorithm, called Fixed-Priority with Priority Promotion (FPP), is proposed. In FPP scheduling, tasks are executed similar to traditional fixed-priority (FP) scheduling but the priority of some tasks are 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. FPP scheduling prioritizes jobs according to Earliest-Deadline-First (EDF) priority when all tasks' priorities follow IPDD policy.

It is known that managing (i.e., inserting and removing) jobs in the ready queue of traditional EDF scheduler is more complex than that of FP scheduler. To avoid such problem in FPP scheduling, a simple data structure and efficient operations to manage jobs in the ready queue are proposed. In addition, techniques for implementing priority promotions with and without the use of a hardware timer are proposed.

Finally, an effective scheme to reduce the average number of priority promotions 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 traditional fixed priorities without any priority promotion. 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.

1 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 [17]. In contrast, fixed-priority scheduling does not provide such a guarantee, even under the (optimal for uniprocessor) Deadline-Monotonic (DM) priority assignment [22].

* This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA and by the ARTEMIS Joint Undertaking under grant agreement no. 621429 for EMC2 project. This article is based on our earlier work in [24].

© Risat Mahmud Pathan; licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE) Leibniz Transactions on Embedded Systems, Vol. 3, Issue 1, Article No. 2, pp. 02:1–02:29
Leibniz Transactions on Embedded Systems Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
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 [15]. 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 the 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 intervals (called, promotion points) relative to the release time of each job. For example, consider task $T_i$ that has (initial) fixed priority $p$ with two promotion points $\delta_1$ and $\delta_2$ 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 $T_i$ 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. After a task’s priority is promoted, its priority remains at this promoted priority until either (i) the task completes execution, or (ii) another promotion point is reached at which the task’s priority is again promoted. As will be evident later, two or more jobs may have the same fixed priorities due to priority promotion. The FPP scheduler has a special tie-breaking policy in such case: a newly released job cannot preempt a currently-executing job if both of these jobs have the same priority. Other than priority promotion, FPP scheduling is same as traditional FP scheduling on uniprocessor and multiprocessors 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 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. Recall that a job with shorter absolute deadline has smaller priority in EDF scheduling. We say that job $J_a$ has higher EDF priority than job $J_b$ if the absolute deadline of $J_a$ is shorter than that of job $J_b$. Executing jobs of the tasks in EDF order but using priority-promotion-based FPP scheduler is one of the major contributions in this paper.

---

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.
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 [12]) 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. Such priority assignment is effective for task set that is neither schedulable using pure FP scheduling nor using pure EDF scheduling. This result is very important for scheduling on multiprocessors since neither FP nor EDF scheduling is optimal for multiprocessor scheduling which is in contrast to scheduling on uniprocessor for which EDF is the optimal algorithm. 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 4.2.1) for FPP scheduler and using a priority queue implemented as a binary min-heap (as is used in [10]) 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.

Finally, techniques to implement priority promotion with and without using hardware timer are proposed. We tackle the challenge of using one hardware timer to implement multiple priority promotions that are due at some later time. In addition, we also address the problem of implementing priority promotions without using a timer (i.e., based on pure software approach). In such software-based approach, a technique called delayed promotion is used: some jobs’ priority promotions are delayed until it is necessary to ensure specific property of the underlying schedule.

Related Work. The FPP algorithm is similar to the well-known dual-priority scheduling which was first proposed by Burns and Wellings [11] in 1993, and analyzed by Davis and Wellings [13, 16] 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 [11] 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 [9] 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 [9] 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 2 presents the task model. The IPDD policy and important lemmas of this policy are presented in Section 3. The dispatcher and ready queue manager of FPP scheduler are presented in Section 4. Technique to reduce the total number of promotion points, particularly, the FPP_Test is proposed in Section 5. The FPP_Test is applied to both uni- and multiprocessors considering constrained-deadline tasks and experimental results are presented in Section 6 and Section 7, respectively. Techniques to implement priority promotions for FPP scheduling are proposed in Section 8. Finally, Section 9 concludes this paper.

2 Task Model

This paper considers scheduling a collection of \( n \) sporadic tasks in set \( \Gamma = \{\tau_1, \ldots, \tau_n\} \). Each task \( \tau_i \) 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 \( \tau_i \). Successive arrivals of the instances (called jobs) of task \( \tau_i \) are separated by at least \( T_i \) time units. Each job of task \( \tau_i \) 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 \( \tau_i \) 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.

3 Priority Promotion Policy: IPDD

The effectiveness of FPP scheduling depends on the promotion points and promoted priorities for each task. In this section, the IPDD priority-promotion policy that can prioritize jobs of the tasks in EDF order while executing using priority-promotion-based FPP scheduling is presented.

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 \( \tau_j \) and \( \tau_i \), 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 \( \tau_i \). The IPDD policy computes the promotion points and promoted priorities for each task \( \tau_i \in \Gamma \) as follows:

- Task \( \tau_i \) has \( i \) different priority levels: starting from priority level \( i \) to the highest priority level 1. Task \( \tau_i \)'s initial priority \( i \) is promoted \((i - 1)\) times. At each of the \((i - 1)\) promotion points, the priority is promoted by one priority level. Figure 1 depicts IPDD priority-promotion policy for task \( \tau_i \).

- Each job of task \( \tau_i \) 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., \((i - 1)\text{th}\) promotion point.

- The promotion points apply to each job of task \(\tau_i\). Each promotion point of a job is a fixed time interval from the release time of the job. The \(\kappa\text{th}\) promotion point is equal to the (relative) deadline difference of tasks \(\tau_i\) and \(\tau_{i-\kappa}\), which is equal to \((D_i - D_{i-\kappa})\). The priority of each job of \(\tau_i\) is promoted to priority level \((i - \kappa)\) at the \(\kappa\text{th}\) promotion point which is \((D_{i-\kappa} - D_{i-\kappa + 1})\) 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 \((D_i - D_{i-1}) \leq (D_i - D_{i-2}) \ldots \leq D_1\), which implies that priority of task \(\tau_i\) is non-decreasing.

- If any two tasks \(\tau_i\) and \(\tau_j\), where \(i < j\), have the same relative deadline, then task \(\tau_j\)'s initial priority is \(i\) (not \(j\)) since \(D_j - D_i = 0\) and the promotion points of \(\tau_j\) are computed the same way that are computed for \(\tau_i\).

Each task \(\tau_i\) has \(i\) priority levels \(i, (i - 1), \ldots 1\) with total \((i - 1)\) promotion points. And, each task \(\tau_j\) has \(j\) priority levels \(j, (j - 1), \ldots i, (i - 1), \ldots 1\) with total \((j - 1)\) promotion points where \(i < j\). In other words, according to IPDD policy, \(i\) priority levels are common (shared) for any two tasks \(\tau_i\) and \(\tau_j\) 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 \(n\). As will be evident later, if two or more newly released jobs have the same initial priority, the FPP scheduler breaks the tie arbitrarily. If a newly released job has the same priority as that of a currently-executing job, the new job does not preempt the executing job in FPP scheduling. Example 1 demonstrates the IPDD policy using an example of three tasks.

**Example 1.** Consider \((C_1, D_1, T_i)\) 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 \(\tau_1\) 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 \(\tau_1\) 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., \(\tau_1\)) with smaller relative deadline than \(D_2\), each job of \(\tau_2\) 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 \(\tau_2\). 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., \(\tau_1\) and \(\tau_2\)) with smaller relative deadlines than \(D_3\), each job of \(\tau_3\) starts with priority level 3, which is promoted twice – first at time \(r_3 + (D_3 - D_2) = (3 + 16 - 8) = 11\) before promotion point; and continuing in this manner, finally, promoted to the (highest) priority level 1 at the last, i.e., \((i - 1)\text{th}\) promotion point.
Figure 2 FPP schedule of three tasks in Example 1 where each task is assigned priorities using IPDD policy. Note that task $\tau_3$’s priority is promoted to priority level 1 at time $t = 8$ and is not preempted by the third job of task $\tau_1$ 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 2. If the priority of job $J_a$ is promoted to priority level $\ell$ at time $t_a$, then $(d_a - t_a) = D_\ell$.

Proof. Assume that job $J_a$ is a job of task $\tau_i$. Therefore, $d_a = r_a + (D_i - D_\ell)$. Consequently, $(d_a - t_a) = D_\ell$.

Lemma 3. 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 $\nu$ and $\ell$ 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 $J_b$. 

$r_3 + (10 - 7) = r_3 + 3$ 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 $\tau_3$. 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 (assuming strictly periodic release for all tasks staring from time 0) is given in Figure 2 (the scheduler is formally presented in Subsection 4.1).
Since \( J_b \)’s priority is \( \ell \) at time \( t \), its priority will ultimately be promoted to (higher) priority level \( \nu \) according to IPDD policy. Let \( J_b \)’s priority will be promoted to priority \( \nu \) at time \( t_b \) where \( t < t_b \). On the other hand, \( J_a \)’s priority is already at priority \( \nu \) at time \( t \). Let \( J_a \)’s priority be set to priority \( \nu \) at time \( t_a \) where \( t_a \leq t \). Therefore, \( t_a < t_b \). From Lemma 2, it follows that 

\[
(d_a - t_a) = D_\nu \text{ and } (d_b - t_b) = D_\nu.
\]

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 \( \kappa \), for \( \kappa = \nu, (\nu - 1), \ldots, 1 \), respectively at time \( (t_a + D_\nu - D_a) \) and \( (t_b + D_\nu - D_a) \). Since \( t_a < t_b \), we have \((t_a + D_\nu - D_a) < (t_b + D_\nu - D_a)\) 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 \( \kappa \) remains at priority level \( \kappa \) 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).

\[\blacktriangleleft\]

**Lemma 4.** Consider that job \( J_a \) has priority \( \ell \) at time \( t \) according to IPDD promotion policy. If a new job \( J_b \) of task \( \tau_t \) is released at time \( t \), then the \( d_a \leq d_b \).

**Proof.** According to IPDD policy, job \( J_b \) of task \( \tau_t \) has priority \( \ell \) 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 \( \ell \) at time \( t \). Let \( J_a \)’s priority be set to priority level \( \ell \) at time \( t_a \) where \( t_a \leq t \). From Lemma 2, 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 \( d_a \leq d_b \).

\[\blacktriangleleft\]

**Lemma 5.** Consider set \( F \) of active jobs. If all the jobs in \( F \) have same priority \( \ell \) at some time instant, then the job with the earliest deadline is promoted to priority level \( \nu \) no later than that of any other job in \( F \), where \( \nu < \ell \).

**Proof.** Consider any two jobs \( J_a \) and \( J_b \) in \( F \). Without loss of generality assume that \( d_a \leq d_b \). Let \( J_a \) and \( J_b \) are promoted to higher-priority level \( \nu \) 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 \( \nu \) no later than that of job \( J_b \). It follows from Lemma 2 that \((d_a - t_a) = D_\nu \) and \((d_b - t_b) = D_\nu \). Since \( d_a \leq d_b \), we have \( t_a \leq t_b \).

\[\blacktriangleleft\]

It will be shown based on Lemmas 2–5 that the FPP scheduling generates the EDF schedule of the tasks when priorities to all the tasks are given using IPDD policy. The dispatcher and the ready queue manager of FPP scheduler are presented in Section 4. Then Section 5 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).

### 4 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.

#### 4.1 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 \( J_{\text{new}} \) with priority higher than that of the currently-executing lowest-priority job \( J_{\text{exe}_\text{low}} \) is released at time \( t \), then \( J_{\text{new}} \) starts execution by preempting \( J_{\text{exe}_\text{low}} \). The preempted job \( J_{\text{exe}_\text{low}} \) is inserted in the ready queue. This (insertion) event managed by the ready-queue manager is called the “\text{rel_prmt}” event.

- If all the \( m \) processors are busy and a new job \( J_{\text{new}} \) with priority not higher than that of the currently-executing lowest-priority job \( J_{\text{exe}_\text{low}} \) is released at time \( t \), then \( J_{\text{new}} \) does not preempt \( J_{\text{exe}_\text{low}} \). And, \( J_{\text{new}} \) is inserted in the ready queue. This (insertion) event managed by the ready-queue manager is called the “\text{rel_no_prmt}” event. Note that if \( J_{\text{new}} \) has the same priority as that of \( J_{\text{exe}_\text{low}} \) (ties in priority ordering), then \( J_{\text{new}} \) does not preempt \( J_{\text{exe}_\text{low}} \).

- 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 “\text{idle_remv}” event.

In summary, the FPP dispatcher works similar to traditional global FP scheduler with one additional feature: jobs may undergo priority promotion. If the total number of active jobs is not more than \( m \), then all active jobs are in execution and the ready queue is empty. If the total number of active jobs is more than \( m \), then all the processors are busy and some active jobs are in the ready queue. Example 1 presents the FPP schedule for three tasks. Theorem 6 proves that FPP scheduling executes jobs in EDF order if all tasks have priorities based on IPDD.

**Theorem 6.** 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 \( J_{\text{new}} \) arrives at time \( t \) (i.e., number of active job becomes larger than \( m \)) such that the priority of \( J_{\text{new}} \) is not higher than that of the currently-executing lowest-priority job \( J_{\text{exe}_\text{low}} \), then \( J_{\text{new}} \) is inserted in the ready queue. If \( J_{\text{new}} \)’s priority is smaller than that of \( J_{\text{exe}_\text{low}} \), then from Lemma 3 it follows that the absolute deadline of job \( J_{\text{new}} \) is larger than that of job \( J_{\text{exe}_\text{low}} \). If \( J_{\text{new}} \)’s priority is equal to \( J_{\text{exe}_\text{low}} \), then it follows from Lemma 4 that the deadline of job \( J_{\text{new}} \) is not smaller than that of job \( J_{\text{exe}_\text{low}} \). Similarly, since \( J_{\text{exe}_\text{low}} \) is the currently-executing lowest-priority job, its deadline is not smaller than any other currently-executing job. Therefore, job \( J_{\text{new}} \) with EDF priority not higher than any of the currently-executing job is inserted in the ready queue.

On the other hand, if \( J_{\text{new}} \) has higher priority than that of job \( J_{\text{exe}_\text{low}} \), then \( J_{\text{new}} \) preempts the execution of \( J_{\text{exe}_\text{low}} \) and \( J_{\text{exe}_\text{low}} \) is inserted in the ready queue. According to Lemma 3, job \( J_{\text{new}} \) has earlier deadline than that of job \( J_{\text{exe}_\text{low}} \). Therefore, job \( J_{\text{exe}_\text{low}} \) having a relatively lower EDF priority is inserted in the ready queue.

\(^2\) 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 [11] 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. Section 8 will present different hardware and software-based techniques to implement priority promotion.
According to Lemma 3, 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. Consequently, if job $J_b$ is inserted in the ready queue because it cannot be prioritized by the dispatcher over another job $J_a$, then job $J_b$ from the ready queue (even if its priority might be promoted) cannot preempt the execution of job $J_a$ at some other (later) time. 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 4.2 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 $\kappa$th position of the array stores the TCB of the ready task that has current priority $\kappa$ 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 $\tau_{\kappa}$ is to be inserted in the ready-queue, then the priority of $\tau_{\kappa}$ is used to index the ready-queue array position at which the TCB of $\tau_{\kappa}$ 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 $B[n\ldots1]$ 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. When a job with priority $\kappa$ is inserted in to the ready-queue array, the $\kappa$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 $\kappa$. Determining the highest-priority job from the ready queue is to find the position of the least set bit in the bitmap $B[n\ldots1]$, which can be performed in constant time using, for example, deBruijn sequence [21], if not supported as a machine-level instruction [27]. Once the position is known, the TCB of the job is removed and corresponding job is dispatched and we set $B[\kappa] = 0$. An interesting discussion how the highest-priority task from the ready queue can be removed efficiently for FP scheduling can be found in [27].

In FP scheduling at most one element (i.e., job) is stored at each priority level in the ready queue. If the number of supported priority levels is smaller than the number of fixed-priority tasks, then the ready queue may be split into $n$ different priority levels as is discussed by Buttazzo in [12]. In such case, the a FIFO queue is maintained at each priority level.
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 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 [12].

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 when overheads are not considered.

Theorem 6 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 [12], 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.

4.2 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.

4.2.1 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. This is because two jobs of two different tasks $\tau_i$ and $\tau_j$ shares $i$ priority levels where $i < j$. 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 $\kappa^{th}$ linked list at any time instant stores all the TCBs of the ready jobs that have priority level $\kappa$ at that time instant.

The $\kappa^{th}$ linked list has two pointers: $\text{head}[\kappa]$ and $\text{tail}[\kappa]$ that respectively point the first and last TCB in the $\kappa^{th}$ 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.

4.2.2 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:
Figure 3 Proposed data structure of the ready queue for FPP scheduling.

- **P1**: All jobs stored in the $\nu^{th}$ linked list of the ready queue have higher EDF priorities than any other job stored in the $\ell^{th}$ linked list where $\nu < \ell$.
- **P2**: All jobs in the $\ell^{th}$ linked list are stored in order of non-increasing EDF priority, i.e., the head[$\ell$] and tail[$\ell$] respectively points the highest and lowest priority EDF job in the $\ell^{th}$ 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 $t$. Maintaining properties P1 and P2 are very important to ensure that operations on the ready queue can be done in constant time. Given the structure of the ready-queue in Subsection 4.2.1, 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 $\ell$ when preempted by $J_{new}$, then the TCB of job $J_{exe\_low}$ is inserted at the front of the $\ell^{th}$ linked list of the ready queue. The insertion at the front is done in constant time using the head[$\ell$] pointer. We set $B[\ell] = 1$ to specify that there is a TCB awaiting execution at priority level $\ell$.

Since job $J_{exe\_low}$ has priority $\ell$ at time $t$ and was in execution, the $1^{st}$, $2^{nd}$, ..., $(\ell - 1)^{th}$ linked lists of the ready queue at time $t$ are empty. It follows from the proof of Theorem 6 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 $\ell^{th}$ 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 \( \ell \) at time \( t \), then the TCB of \( J_{new} \) is inserted at the end of the \( \ell \)th linked list. The insertion at the end is done in constant time using the \( \text{tail}[\ell] \) pointer. We set \( B[\ell] = 1 \) to specify that there is a TCB awaiting execution at priority level \( \ell \).

Since P1 holds at time \( t_0 \) and job \( J_{new} \) has priority \( \ell \) at time \( t \), it follows that all the jobs in the \( \kappa \)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 4, all the jobs in the \( \ell \)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 \( \ell \)th linked list at time \( t \) guarantees that P1 and P2 continue 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 \( \ell \)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 \( \ell \)th linked list. The job from the front of the \( \ell \)th linked list is removed and dispatched for execution. The removal from the front is done in constant time using \( \text{head}[\ell] \) pointer. If \( \text{head}[\ell] \) becomes NULL after this removal (i.e., the \( \ell \)th linked list becomes empty), then we set the \( B[\ell] = 0 \) to specify that there is no TCB awaiting execution at priority level \( \ell \).

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 \( \ell \)th linked list is to be promoted to priority level \( \nu \), then this job is removed from the \( \ell \)th linked list and inserted to the \( \nu \)th linked list.

Since P2 holds at time \( t_0 \), the job at the front of the \( \ell \)th linked list has deadline no later than any other jobs in \( \ell \)th linked list. According to Lemma 5, given a set of jobs having the same priority \( \ell \) at time \( t_0 \), the priority of the job with earliest deadline will be promoted to priority level \( \nu \) no later than any other job in that set. Consequently, the priority of the job at the front of the \( \ell \)th linked list is to be promoted to priority level \( \nu \). Let job \( J_a \) is the job at the front of the \( \ell \)th linked list.

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

Since the promoted priority of job \( J_a \) is \( \nu \) at time \( t \), all jobs in the \( \kappa \)th linked list, where \( \kappa > \nu \), have absolute deadline larger than that of job \( J_a \) at time \( t \) according to Lemma 3. Since P1 holds at time \( t_0 \) and job \( J_a \) is in the \( \ell \)th linked list at time \( t_0 \), it follows that all the jobs in the \( \nu \)th linked list have smaller absolute deadlines than that of job \( J_a \). Therefore, inserting \( J_a \) at the end of the \( \nu \)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.
If a currently-executing job's priority is promoted, then such promotion does not need to reorder the jobs in the ready queue (i.e., no ready queue management overhead is incurred). In contrast, if the priority of a job residing in the ready queue is promoted, then such promotion repositions the job to a higher-priority position in the ready queue data structure and thus incurs overhead. 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 in next section.

5 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 \( \tau_i \) is tested separately. For example, the well-known response time analysis (RTA) for FP scheduling on uniprocessor [2] and multiprocessors [26] is of iterative nature: the response time \( R_i \) of each task \( \tau_i \) is computed. The crucial observation is that when determining the schedulability of \( \tau_i \) using an iterative FP schedulability test for uniprocessor [2] or for multiprocessors [25], 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 \( \tau_i \). Consequently, Corollary 7 holds.

\[ \text{Corollary 7. If task } \tau_i \text{ is deemed to be schedulable at priority level } \ell \text{ using an iterative test where } hp(i) \text{ is assumed to be the set of higher priority tasks, then the schedulability of } \tau_i \text{ is preserved when it is assigned traditional fixed-priority level } \ell \text{ regardless whether the jobs of the tasks in } hp(i) \text{ are scheduled using dynamic or fixed priority.} \]

It follows from Corollary 1 that if some task \( \tau_i \) is deemed schedulable using an iterative test at fixed-priority level \( \ell \), then task \( \tau_i \) 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 Sections 6–7, the actual tests used in place of \( T_{fp} \) and \( T_{edf} \) are presented respectively for uniprocessor and multiprocessor platform.

The FPP_Test in Figure 4 takes as input a task set \( \Gamma \) 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.
Algorithm: FPP_Test (Task Set $\Gamma$)

1. For priority level $k = n$ to $k = 1$
2. For each priority-unassigned task $\tau \in \Gamma$
3. If $\tau_i$ 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 $\tau_i$ 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 priority-unassigned tasks using IPDD policy
// Comment: IPDD policy uses (higher) priority levels
// $k,(k-1)\ldots 1$ for these priority-unassigned tasks
13. Return True
14. Else
15. Return False
16. End If
17. End For
18. Return True

Figure 4 Improved priority promotion policy for FPP scheduling.

Initially, all the tasks in set $\Gamma$ are “priority-unassigned” in Figure 4, i.e., no task has any priority. Based on Audsley’s OPA algorithm [1], 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 $\tau_i$, 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 $\tau_i$ is found in line 3–5, then task $\tau_i$ 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 $\Gamma$ 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 $\Gamma$ 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 4), 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 4.2.2 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., \((n - q + 1), \ldots n\)) 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., \((n - q), \ldots 1\)) priority levels are dispatched for execution. Properties P1 and P2 (defined in Section 4.2.2) 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.

6 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. [2] is considered in place of \(T_{fp}\) in Figure 4 to determine whether a (priority-unassigned) task \(\tau_i\) 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 [30], 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 [17], 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 [14] 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 \(\tau_i\) in the task set are generated as follows:

- The minimum inter-arrival time \(T_i\) of each task \(\tau_i\) is generated from the uniform random distribution within the range \([10ms, 1000ms]\).
The WCET of task $\tau_i$ is set to $C_i = u_i \cdot T_i$.

The relative deadline $D_i$ of task $\tau_i$ 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 \{0.025$m$, 0.05$m$, ..., 0.975$m$ $m$\} 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 4.2.2) 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 [10] or binomial min-heap [8], 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 $n$.

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. $\text{EDF}_{av}$ and $\text{FPP}_{av}$ 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:

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

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 for 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.

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., $O(\log n)$) 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.

The reduction in promotion point at each higher utilization level for task set with larger cardinality is much higher than that of task set with smaller cardinality. For example, the average reduction in promotion points at $U = 0.8$ is around 80\% and 40\% for task sets with cardinality $n = 40$ and $n = 10$, respectively. This is because, when the number of tasks in a task set for a given utilization level is fewer, the utilization of individual task is relatively larger and the execution time of individual task tends to be larger. As execution time a individual task increases, jobs in the ready queue stay longer and may undergo a relatively larger number of priority promotions. And, more priority promotions cause higher number of times the jobs in the ready queue change their positions.

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.

\[^3\] 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.
Real-Time Scheduling on Uni- and Multiprocessors based on Priority Promotions

7 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 [26] recently proposed an iterative G-FP test that is shown to perform better than any other iterative test proposed earlier. This G-FP test [26] is used in place of $T_{fp}$ in Figure 4 to determine if a priority-unassigned task $\tau_i$ can be assigned fixed priority level $k$. For G-EDF scheduling, Bertogna and Baruah [6] proposed a step-by-step approach to apply different G-EDF schedulability tests proposed by other researchers. This G-EDF test in [6] 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.

There is no evidence regarding whether the G-FP test in [26] dominates or is dominated by the G-EDF test in [6]. It is not difficult to see that if a task set is deemed to be schedulable using G-FP test [26] or G-EDF test [6], 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 [26] and G-EDF test [6]. 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.

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 and Figure 10).

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

### 7.1 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.

\(\text{GEDF}_{\text{avpr}}\) and \(\text{FPP}_{\text{avpr}}\) denote the average number of preemptions that a task set suffers in G-EDF and FPP scheduling, respectively. Similarly, \(\text{GEDF}_{\text{avmg}}\) and \(\text{FPP}_{\text{avmg}}\) 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:

\[
\text{Improvement(prmt)} = \frac{\text{GEDF}_{\text{avpr}} - \text{FPP}_{\text{avpr}}}{\max\{\text{GEDF}_{\text{avpr}}, \text{FPP}_{\text{avpr}}\}} \times 100\%
\]

\[
\text{Improvement(migr)} = \frac{\text{GEDF}_{\text{avmg}} - \text{FPP}_{\text{avmg}}}{\max\{\text{GEDF}_{\text{avmg}}, \text{FPP}_{\text{avmg}}\}} \times 100\%
\]

The value of improvement ranges in \([-100\%, +100\%]\). For example, \(\text{Improvement(prmt)} = +50\%\) 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 11 and Figure 12, where the x-axis is the system utilization \(U/m\) and y-axis represents Improvement(prmt) and Improvement(migr), respectively.

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 11 and Figure 12.

### 7.2 Other Implementation Issues

Brandenburg and Anderson identified six different major sources of overhead in implementing G-EDF algorithm using a Linux extension, called LITMUSRT, 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 4.2) 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 4.2.2) continue to hold regardless of the 
order these (nearly concurrent) events are processed.

Another issue to implement FPP scheduling is the mechanism used to implement priority 
promotion. One way to implement such priority promotion is by using hardware timers to handle 
pri_prom events: when a programmed timer expires, the handler can promote the priority and 
repositions 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. In next section, different techniques to implement priority promotions 
are proposed. 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 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 [30], 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. [28] can be 
made OPA-compatible using approach used by Davis and Burns for the DA-LC test in [14]. 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.
8 Techniques to Implement Priority Promotion

This section presents different techniques to implement priority promotion, i.e., how event pri_prom is implemented and is handled. First, four different hardware timer-based approaches are presented to implement priority promotion (Subsection 8.1) along with a discussion about advantage and disadvantage of each alternative. Second, a software-based approach that does not rely on any support of hardware timer is presented (Subsection 8.1). A detailed implementation of priority promotion using each of the suggested approaches is left as a future work.

In FPP scheduling, the priorities of the currently-executing jobs need to be promoted so that preemption decision can be taken when a (new) job is released while all the cores are busy. In addition, priorities of the jobs stored in the ready queue need to be promoted to ensure that properties P1 and P2 (also restated below for better readability of this section) always hold.

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

Based on the operations on the ready queue (please see Subsection 4.2.2) an important observation for job \( J_k \) of task \( \tau_k \) is presented below:

**Observation 8.** The TCB of job \( J_k \) is never stored in the (lower-priority) \( \ell \)th link lists where \( \ell = (k+1), (k+2), \ldots , n \). This is because the priority of the job is \( k \) or higher (i.e., \( (k-1), (k-2), \ldots , 1 \)). When the TCB of job \( J_k \) is in the \( \ell \)th linked list of the ready queue at time \( t \) where \( 1 \leq \ell \leq k \), its absolute deadline is not larger than \( D_\ell \) relative to \( t \).

Maintaining both P1 and P2 at each time instant is the key to efficiently (in constant-time) perform the insertion and removal operations to and from the ready queue, which is same as FP scheduler in terms of time complexity (please see Subsection 4.2.2). However, unlike FP scheduling, priority promotions cause repositioning of jobs in the ready queue. Overhead related to such repositioning depends on how priority promotion (i.e., event pri_prom) is implemented and handled. In this section, we present techniques to implement priority promotions (i) based on hardware-based approach using timers (Subsection 8.1), and (ii) software-based approach with no timer (Subsection 8.2). In the remainder of this section, we consider that all the tasks are assigned priorities based on IPDD policy and need priority promotion.

According to IPDD priority-promotion policy, the initial priority of a job of task \( \tau_i \) is \( i \). This initial priority \( i \) is promoted to priority levels \( (i-1), (i-2), \ldots , 1 \) at offsets \( (D_i - D_{i-1}), (D_i - D_{i-2}), \ldots , (D_i - D_1) \) relative to the release time of the job. The difference in time between the \((k-1)\)th and \( \kappa \)th promotion times is denoted as \( \theta_k^\kappa \) and is given as follows for \( \kappa = 1, 2, \ldots , (i-1)\):

\[
\theta_k^\kappa = D_1 - D_{i-1} - D_{i-k}
\]

(1)

If a job of task \( \tau_i \) is released at time \( r_i \), the first promotion point is at \( (r_i + \theta_1^1) \) which is \( \theta_1^1 \) time units later than its release time. The second priority-promotion point is \( \theta_2^1 \) time units later than the 1st priority promotion point. In general, the \( \kappa \)th priority-promotion point is \( \theta_k^\kappa \) time units later than the \((k-1)\)th priority-promotion point.

8.1 Hardware Timer-Based Priority Promotion

Most computer platforms have a clock that increments a counter and can be programmed to generate an interrupt when the counter reaches a certain expiration count called the expiration
time. The combination of a clock and an expiration time is called a timer. In this subsection, we present different alternatives to implement priority promotion based on such hardware timers.

**Alternative 1.** Consider a platform where the number of hardware timers is sufficient such that one timer can be used for each active job. For such a platform, a one-shot timer can be programmed at each promotion time of an active job such that the timer expires at next priority-promotion time. Based on this principle, when a job of task \( \tau_i \) is released at time \( r_i \), a hardware timer is programmed to expire after \( \theta_i^1 \) time units. Remember that the initial priority \( i \) has to be promoted to priority level \( (i - 1) \) at time \( (r_i + \theta_i^1) \).

If a job completes its execution before the timer is expired, then the timer is disabled to avoid unnecessary interrupt at expiration. Otherwise, when the timer generates an interrupt at time \( (r_i + \theta_i^1) \), the priority of the job is promoted to priority level \( (i - 1) \) and the timer is programmed again to expire after \( \theta_i^2 \) time units at which priority is promoted to \( (i - 2) \) and so on. Following this approach, the timer is programmed to expire after \( \theta_i^k \) time units whenever priority of the job is set to priority level \( (i - (\kappa - 1)) \) for \( \kappa = 1, 2, \ldots (i - 1) \). After the priority of the job is set to (highest) priority level 1, the timer is not more programmed since there is no more priority promotion of this job according to IPDD policy.

While this approach is simple, it may not work for platform where the number of active jobs is larger than the number of available hardware timer. In such case, a queue of (future) timed events related to (future) priority promotions needs to be implemented using, for example, a single hardware timer. In the remainder of this subsection, different alternatives to manage a queue of (priority-promotion related) timed events using a single hardware timer are presented.

**Alternative 2.** A queue of (future) timed events can be maintained based on a timing wheel which is a sequential array of records [29]. Inspired from the discussion of Baruah et al. in [5], we assumed that time is represented using non-negative integers. The required number of records of the timing wheel depends on the maximum time difference between any two consecutive priority promotions. Based on Eq. (1), the maximum length between two consecutive priority promotions of a job of \( \tau_i \) is \( \max_{\kappa=1}^{\kappa=1} \{\theta_i^k\} \). Consequently, the maximum length of any two consecutive priority promotions for any task in set \( \{\tau_1, \tau_2, \ldots \tau_n\} \) is \( \max_{i=1}^{n} \max_{\kappa=1}^{\kappa=1} \{\theta_i^k\} \). For implementing a queue of priority-promotion related timed events, the required number of records of the timing wheel, denoted by \( L \), is given as follows in Eq. (2):

\[
L = 1 + \max_{i=1}^{n} \left\{ \max_{\kappa=1}^{\kappa=1} \{\theta_i^k\} \right\}
\]

At each position of the timing wheel, a linked list of pointers to jobs’ TCBs is stored. A variable to track current time, called \( CT \), is initially set to 0. This variable is incremented (\( mod \ L \)) at every system tick. In other words, \( CT \) is set to 1 after the first tick, set to 2 after the second tick, set to back to 1 after \( L \) ticks, and so on. Such a timing wheel is shown in Figure 13 and Figure 14.

When a job of task \( \tau_i \) is released, its next promotion time is \( \theta_i^1 \) time units later relative to current time. A pointer to the TCB of this job is stored in the linked list of the timing wheel at position \( (CT + \theta_i^1) \mod L \). When \( CT \) points to a location of the timing wheel at which the linked list is non-empty, the priorities of the jobs pointed by the elements of this linked list are promoted. After the priority of a job of task \( \tau_i \) at time \( CT \) is promoted to priority level \( (i - (\kappa - 1)) \) for some \( \kappa \) such that \( 1 \leq \kappa \leq (i - 1) \), a pointer to the TCB of this job is again stored in the linked list at position \( (CT + \theta_i^k) \mod L \) of the timing wheel for next (i.e., \( \kappa^{th} \)) priority promotion of this job.
According to Eq. (2), the value of $L$ is one larger than the maximum difference between any two consecutive promotion points. Therefore, $(CT + \theta_i^a) \mod L$ is never equal to $CT$. In other words, to distinguish between two (future) priority promotions separated by exactly $\max \{ \max \{\theta_i^a\}_{i=1}^n \}$ time units, the number of records (as is shown in Eq. (2)) of the timing wheel is one larger than the maximum difference between any two consecutive priority promotions.

Alternative 2 to manage multiple priority promotions requires the variable $CT$ to be incremented at every timer’s tick which may have high overhead. Next we propose Alternative 3 based on timing wheel but with the exception that variable $CT$ is not incremented at every system’s tick.

**Alternative 3.** Similar to Alternative 2, multiple (future) timed events related to priority promotions are stored in the linked lists of a timing wheel but without requiring to increment variable $CT$ at every system’s tick. A bitmap, denoted by $S[1, 2 \ldots L]$, of length $L$ corresponding to the timing wheel is maintained. If the linked list of the $a^{th}$ position of the timing wheel is empty, then $S[a] = 0$; otherwise, $S[a] = 1$. When all the linked lists of the timing wheel are empty, we set $CT$ to 0. The main idea is to program a one-shot timer that expires after a duration equal to the earliest time of occurrence of any (future) timed event stored in the timing wheel. The approach is described as follows:

- **(How an element is inserted in an empty timing wheel?)** Consider that there is no element in the timing wheel, i.e., $CT = 0$ and $\forall t; S[t] = 0$ at time $t$. Now consider that a new (future) timed event appears at time $t$ such that this event needs to occur after $a$ time units relative to time $t$. In other words, there is a future timed event that occurs after $a$ time units relative to $CT$. To implement priority promotion related to this event, a pointer to the TCB of the job is inserted in the $a^{th}$ linked list of the timing wheel. A one-shot timer is set to expire after $a$ time units. To remember the total duration for which this timer is programmed, we set Initial Value of the Timer as $IVT = a$. We also set $S[a] = 1$ to specify that the $a^{th}$ linked list of the timing wheel is non-empty.

- **(How an element is inserted in a non-empty timing wheel?)** Now consider that a new (future) timed event appears at time $t'$ such that this event needs to occur after $c$ time units relative to time $t'$ and the timing wheel is non-empty at time $t'$. Since the timing wheel is non-empty, the timer (that was last programmed) is running (not yet expired) at time $t'$.
Assume that at time $t'$ the remaining time to expire the timer is $b$. The total time elapsed between the time instant when the timer was last programmed and time $t'$ is $(IVT - b)$ since $IVT$ stores the total duration for which the time was last programmed. At time $t'$ the value of $CT$ is updated by setting it equal to the old value of $CT$ plus elapsed time from the time $CT$ was last set, i.e., $CT = CT + (IVT - b)$. In addition, $IVT$ is set to $IVT = b$ to reflect the fact that the timer expires after $b$ time units relative to the (updated) current time $CT$. After updating $CT$ and $IVT$, we consider where the new event that appears at time $t'$ is to be inserted in the timing wheel.

If $c < b$, then the new event has an earlier time of occurrence than that of any event stored in any linked list of the timing wheel. Remember that the one-shot timer was programmed to expire at time when the earliest event in the timing wheel will occur. Since $c < b$, the one-shot timer is reprogrammed at time $t'$ to expire after $c$ time units and we set $IVT = c$. On the other hand, if $c > b$, then the new event does not have an earlier occurrence time and the one-shot timer is not reprogrammed. After $CT$ and $IVT$ are updated at time $t'$, a pointer to the TCB of the job corresponding to the new event is inserted to the $(CT + c)^{th}$ linked list of the timing wheel. Note that the value $(CT + c)$ essentially means $(CT + c) \mod L$. If $(CT + c)^{th}$ linked list was empty before this insertion, we set $S[q] = 1$ where $q = (CT + c) \mod L$.

(How to deal with timer’s expiration?) The timer expires when no new event with earlier time of occurrence appears after $IVT$ was last set. When the one-shot timer expires after $IVT$ time units and generates an interrupt, the value of $CT$ is updated by setting $CT = (CT + IVT) \mod L$. And, the priorities of the jobs pointed by the TCB pointers stored in the $CT^{th}$ linked list of the timing wheel are promoted. If the next promotion time for such a job is a time units later, then the pointer to the TCB of that job is again inserted in the $(CT + a)^{th}$ linked list and we set $S[CT + a] = 1$. After processing each element of the $CT^{th}$ linked list in the timing wheel, all the TCB pointers are removed from the $CT^{th}$ linked list and we set $S[CT] = 0$.

If the timing wheel is not empty after handling a timer’s expiration, then we have to program the timer for the next promotion event that will occur the earliest. To program the timer for the next earliest promotion time, the position of the first set bit of bitmap $S$ starting from position $CT$ is determined. This can be done based on techniques similar to finding the highest priority tasks from the ready queue of $FP$ scheduler. Let this position is $(CT + k) \mod L$, i.e., this position points to the $k^{th}$ linked list relative to the index of $CT^{th}$ linked list. If $k$ is not a valid index, then there is no element in the timing wheel and we set $CT = 0$. For a valid $k$, the jobs corresponding to the TCB pointers stored in the $(CT + k)^{th}$ linked list have the earliest promotion time. The timer is programmed to expire after $k$ time units and we set $IVT = k$.

Note that in the approach described above, variable $CT$ is not updated at every system’s tick. It is updated either when a new event is to be stored in the timing wheel and/or when the timer expires. One of the limitations with this approach is that if $L$ is too large, then a hierarchical bitmap needs to be maintained (as is suggested in [27]).

**Alternative 4.** This alternative to manage a queue of (future) timed events is based on the RELTEQ approach proposed in [19]. The main idea of RELTEQ is that events are stored in an ordered list based on the time of occurrences of the events relative to each other. The advantage of this approach is that no bitmap needs to be maintained. But the disadvantage is that linear search is needed to find the appropriate position for each new event. Please see details of RELTEQ in [19].
8.2 Software-Based Approach to Priority Promotion

In this subsection, we present software-based approach to show how priority promotion can be implemented without using a hardware timer. Under this scheme, the the priorities of the jobs that are in execution are never promoted. The priority of a job is promoted only if the job is in the ready queue. For such a job in the ready queue, priority promotions that are due at a time instant are delayed as long as properties P1 and P2 of the ready queue of FPP scheduler hold.

The jobs whose promotions are delayed are called colluding jobs. When any of the rel_prmt or rel_no_prmt event occurs (i.e., a new TCB has to be inserted in the ready queue due to the release of new job), we check if insertion of this new TCB in the ready queue according to the approach presented in Subsection 4.2.2 could violate property P1 or P2. Note that such violation may happen since priorities of the colluding jobs were not promoted when their promotions were due and the corresponding TCBs of colluding jobs were not repositioned in the right place in the ready queue. If we detect that such violation would occur, we fix the collusion by promoting some or all colluding jobs so that property P1 and P2 continue to hold after insertion of the new TCB.

A job \( J_i \) of task \( \tau_i \) is promoted according to IPDD priority-promotion policy in order to determine whether a newly released job \( J_k \) of task \( \tau_k \) needs to preempt the execution of \( J_i \) or not. In contrast, the priorities of the currently-executing jobs are never promoted in delayed preemption strategy. Whether a newly released job \( J_k \) preempts \( J_i \) or not can be determined by comparing their absolute deadlines since we want to execute jobs in EDF order. Therefore, we compare \( (r_i + D_i) \) and \( (r_k + D_k) \). If \( (r_i + D_i) \geq (r_k + D_k) \), then \( J_k \) preempts \( J_i \); otherwise, \( J_k \) does not preempt \( J_i \). In the remainder of this section, we present how the new TCB is inserted in the ready queue such that property P1 and P2 of the ready queue continue to hold after insertion of the new TCB.

Assume that property P1 and P2 hold at time \( t \) and a new job \( J_k \) of task \( \tau_k \) is released and all the cores are busy such that \( t < t' \). A new TCB (i.e., TCB of \( J_k \) or TCB of the preempted job \( J_i \)) is to be inserted in the ready queue at time \( t' \). Since \( t' \) is the earliest time at which a new TCB is to be inserted in the ready queue, property P1 and P2 continue to hold during the entire interval \([t, t')\) even if all promotions of the jobs in the ready queue during this interval are delayed. In delayed promotion strategy, the promotions of the jobs are delayed until a new job is released. Due to such delayed promotions, colluding jobs in the ready queue are not promoted (i.e., repositioned) to higher-priority linked lists of the ready queue. Consequently, property P1 and P2 may not hold at time \( t' \) if the new TCB is inserted without fixing the collusion. The challenge is to propose mechanism to ensure that after inserting the new job, property P1 and P2 continue to hold also at time \( t' \). There are two cases to consider:

- Case (i) – Job \( J_k \) preempts \( J_i \).
- Case (ii) – Job \( J_k \) does not preempt \( J_i \).

Case (i) – \( J_k \) preempts \( J_i \): In such case, a newly released job \( J_k \) preempts the currently-executing lowest EDF priority job \( J_i \) at time \( t' \). Since \( J_i \) was in execution just before \( J_k \) was released and because property P1 and P2 hold during \([t, t')\) during which no new job is released, the EDF priority of job \( J_i \) is larger than the EDF priority of any other job in the ready queue.

Job \( J_i \) is inserted at the front of the highest-priority non-empty linked list if the index of the highest-priority non-empty linked list of the ready queue is smaller than or equal to \( i \); otherwise, it is inserted as the first element in the \( i^{th} \) linked list of the ready queue. Such insertion can be done in constant time and ensures that Observation 8 still holds. It is easy to see that property P1 and P2 continue to hold at time \( t' \) after inserting the new TCB of \( J_k \) in the ready queue even though all due promotions during \([t, t')\) are delayed. In such case, the delayed promotions in \([t, t')\) are delayed further.
Algorithm: Fix_Collusion(Job \( J_k \))

// This algorithm is executed when a new job \( J_k \) cannot // preempt any job and needs to be inserted in // the ready queue (i.e., when rel_no_prmt event occurs)

1. \( NextList = k + 1 \)
2. \( s = \) Position of the first set bit of bitmap \( B[NextList ... n] \)
3. If \( s \) is a valid position of bitmap \( B \)
   4. While (the absolute deadline of the first element of the \( s^{th} \) linked list is smaller than the absolute deadline of \( J_k \))
      5. \( J = \) remove the first element from the \( s^{th} \) linked-list
      6. Insert \( J \) at the end of the \( k^{th} \) linked list
     7. If the \( s^{th} \) linked list is empty
        8. \( NextList = s + 1 \)
        9. Go to Step 2
     10. End If
     11. End While
12. End If
13. End While
14. Insert \( J_k \) at the end of the \( k^{th} \) linked list

Figure 15 Coalescing Priority Promotion to handle rel_no_prmt event.

Case (ii) – \( J_k \) does not preempt \( J_i \): In such case, a newly released job \( J_k \) does not preempt the currently-executing lowest EDF priority job \( J_i \) at time \( t' \). The TCB of job \( J_k \) is to be inserted in the ready queue. Based on the operations proposed in Subsection 4.2.2, this new TCB is inserted at the end of the \( k^{th} \) linked list since the priority of job \( J_k \) at time \( t' \) is \( k \). However, property P1 and P2 may not hold at time \( t' \) because promotions of the colluding jobs during \([t, t']\) are delayed and not placed in the right position in the ready queue before inserting this new TCB. The strategy to fix the collusion before inserting job \( J_k \) is as follows.

The absolute deadline of job \( J_k \) is \( D_k \) time units later than time \( t' \) because job \( J_k \) is released at time \( t' \). Since jobs in the \( k^{th} \) linked list of the ready queue at time \( t' \) have their absolute deadlines no later than \( D_k \) time units relative to \( t' \) for \( \ell = 1, 2, \ldots, k \), the EDF priorities of the jobs in these linked lists are higher than that of job of \( J_k \) (follows from Observation 8). If job \( J_k \) is inserted at the end of \( k^{th} \) linked list, property P1 and P2 continue to hold at time \( t' \) for the \( t^{1st}, 2^{nd}, \ldots, (k - 1)^{th}, k^{th} \) linked lists even if priority of the jobs in these linked lists are not promoted during \([t, t']\).

On the other hand, since priority promotions of the jobs in \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists are also delayed during \([t, t']\), the actual EDF priority of some of the colluding jobs in these linked lists may be higher than that of job \( J_k \) at time \( t' \). Such colluding jobs need to be promoted (i.e., need to be repositioned) to fix the collusion so that property P1 and P2 continue to hold after job \( J_k \) is inserted at the end of the \( k^{th} \) linked list. Although there may be many colluding jobs, we only need to promote the priority (i.e., reposition in the ready queue) of those colluding jobs from the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists whose EDF priority is larger than the EDF priority of \( J_k \) at time \( t' \) to fix the collusion.

Notice that property P1 and P2 hold for the jobs in the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists before inserting the new TCB at time \( t' \). Before inserting job \( J_k \) at the end of the \( k^{th} \) linked list, the higher EDF priority jobs from the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists are inserted (in decreasing EDF order) at the end of the \( k^{th} \) linked list. Finally, job \( J_k \) is inserted at the end of the \( k^{th} \) linked list. The following algorithm in Figure 15, called Fix_Collusion, implements this insertion.
Algorithm **Fix_Collusion** in Figure 15 selects those jobs from the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists whose EDF priorities are higher than that of job \(J_k\). These selected jobs are inserted in decreasing EDF order at the end of the \(k^{th}\) linked list, and finally, job \(J_k\) is inserted at the end of the \(k^{th}\) linked list. We will show that property P1 and P2 continues to hold after algorithm **Fix_Collusion** is executed at time \(t'\).

Line 1 initializes a variable \(NextList\) to \((k + 1)\) in order to start the search from the \((k + 1)^{th}\) linked list. However, the \((k + 1)^{th}\) linked list may be empty. The index of the first non-empty linked list among the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists is determined in line 2 based on the bitmap \(B[NextList, \ldots n]\). The index of the first set bit of the bitmap \(B[NextList, \ldots n]\) is stored in variable \(s\) in line 2.

If all of the \((n - k)\) lower priority linked lists are empty, then \(s\) has an invalid index. The condition in line 3 checks whether all of the \((n - k)\) lower priority linked lists are empty or not. If the condition in line 3 is false (i.e., there is no non-empty linked lists among the \((n - k)\) lower priority linked lists), then there is no TCB in the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists and the TCB of job \(J_k\) is inserted at the end of the \(k^{th}\) linked list in line 14. Since P1 and P2 hold for \(k\) higher priority linked lists before inserting \(J_k\) (i.e., jobs in these linked lists have deadline no larger than \(D_k\) relative to time \(t'\)) and since the deadline of \(J_k\) is \(D_k\) at time \(t'\), P1 and P2 hold after job \(J_k\) is inserted.

If condition in line 3 is true, then \(s\) is the index of the highest-priority non-empty linked lists among the \((k + 1)^{th}, (k + 2)^{th}, \ldots, (n - 1)^{th}, n^{th}\) linked lists. In such case, all the TCBs of the jobs having higher EDF priority than \(J_k\) (based on the condition of the while loop in line 4–5) are removed from the \(s^{th}\) linked list one-by-one in line 6 and inserted in non-increasing EDF priority order at the end of the \(k^{th}\) linked list in line 7.

Since the first job is removed from the \(s^{th}\) linked list each time condition in line 4–5 is true and because property P2 is satisfied (jobs in each list are in non-increasing EDF priority order), inserting the removed job \(J\) at the end of the \(k^{th}\) linked list ensures that jobs in the \(k^{th}\) linked list are in non-increasing EDF order.

We exit from the while loop in two cases: (i) some job’s EDF priority in the \(s^{th}\) linked list is lower than the EDF priority of \(J_k\) (i.e., condition in the while loop is false), or (ii) all the jobs from the \(s^{th}\) linked list are removed (i.e., condition in line 8 is true). In the first case, the algorithm exit from the loop and executes line 14. This is because there is no other jobs in the \((n - k)\) lower priority linked list having higher EDF priority. In the second case (when the condition in line 8 is true), the \(s^{th}\) list is empty and the \(NextList\) is set to \((s + 1)\) to select other higher EDF priority jobs from the remaining \((n - s)\) linked lists. In such case, the algorithm jumps to line 2 from line 10 and continues as described above. It is easy to see that number of times the while loop executes is no more than the number of delayed promotions in \([t, t']\). When the algorithm stops, property P1 and P2 hold at time \(t'\). Note that we need no timer to implement priority promotions based on software-based delayed promotion mechanism. Evaluating all these priority promotion schemes and the implementation of **FPP** scheduling on real platform is left as a future work.

### Conclusion

The proposed **FPP** scheduling algorithm shows how jobs can be executed in EDF order based on priority promotion. 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. A 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. Techniques to implement
priority promotions based on hardware timers or purely in software are proposed. Simulation results show that the overhead for managing jobs in the FPP ready queue is reduced significantly in comparison to that of an EDF scheduler.

---

References

21. Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de bruijn sequences to index a 1 in


