FPCL and FPZL Schedulability Analysis

Robert I. Davis  
Real-Time Systems Research Group, Department of Computer Science, University of York, YO10 5DD, York (UK)  
rob.davis@cs.york.ac.uk

Shinpei Kato  
Department of Electrical and Computer Engineering Carnegie Mellon University, Pittsburgh, PA 15213 USA. shinpei@ece.cmu.edu

Abstract

This paper presents the Fixed Priority until Critical Laxity (FPCL) and Fixed Priority until Zero Laxity (FPZL) and scheduling algorithms for multiprocessor real-time systems. FPZL is similar to global fixed priority pre-emptive scheduling; however, whenever a task reaches a state of zero laxity it is given the highest priority. FPCL is a variant of FPZL that introduces no additional scheduling points beyond those present with fixed priority scheduling. FPZL and FPCL are minimally dynamic algorithms, in that the priority of a job can change at most once during its execution, bounding the number of pre-emptions.

Polynomial time and pseudo-polynomial time sufficient schedulability tests are derived which are applicable to both FPCL and FPZL. These tests are then improved by computing upper bounds on the amount of execution that each task can perform in the zero-laxity / critical-laxity state. An empirical evaluation shows that FPCL and FPZL are highly effective, with a significantly larger number of tasksets deemed schedulable by the tests derived in this paper, than by state-of-the-art schedulability tests for EDZL scheduling.

Extended version

This paper builds on the paper “FPZL Schedulability Analysis” [34] published in the proceedings of RTAS 2011. This paper extends the analysis given in [34] to cover the FPCL scheduling algorithm as well as FPZL. The analysis for FPCL is a superset of that provided for FPZL in [34] and reduces to it if the critical laxity parameter of each task is set to zero. In this paper, the experimental evaluation given in [34] has been extended to include schedulability tests and simulation of FPCL as well as FPZL. We also report on a prototype implementation of FPCL and FPZL in the Linux kernel, running on an Intel Core 2 Quad processor (Q9650).

1. Introduction

Approaches to multiprocessor real-time scheduling, can be categorized into two broad classes: partitioned and global. Partitioned approaches allocate each task to a single processor, dividing the multiprocessor scheduling problem into one of task allocation (bin-packing) followed by uniprocessor scheduling. In contrast, global approaches allow tasks to migrate from one processor to another at runtime. Each approach has its distinct advantages and disadvantages. Partitioned scheduling typically has lower overheads in accessing and manipulating run-queues [23], while global scheduling has advantages in spare capacity sharing, and is more appropriate for use in open systems, as there is no need to run load balancing / task allocation algorithms when the set of tasks changes. Recent work [16] suggests that global scheduling research should focus on a small to medium number of processors (e.g. 2 to 12 processors) as global scheduling techniques may be most appropriate for clusters of processing cores that share a cache. In such cases, the cost of cache related migration delays were found not to differ substantially from the cost of cache related pre-emption delays [17].

In this paper, we focus on global scheduling techniques with the aim of increasing effectiveness, in terms of the number of tasksets that can be guaranteed schedulable, without compromising efficiency, in terms of overheads caused by pre-emption and migration.

In this paper, we present minimally dynamic global scheduling algorithms for real-time multiprocessor systems called Fixed Priority until Critical Laxity (FPCL) and FPZL (Fixed Priority until Zero Laxity). FPZL is based on global fixed priority pre-emptive scheduling, which for brevity we refer to as global FP scheduling. Under FPZL, jobs are scheduled according to the fixed priority of their associated task, until a situation is reached where the remaining execution time of a job is equal to the time to its deadline, and that job would not be scheduled to execute on the basis of its fixed priority. Such a job has zero laxity and will miss its deadline unless it executes continually until completion. FPZL gives such zero-laxity jobs the highest priority. The schedules produced by FPZL and global FP scheduling are identical until the latter fails to execute a task with zero laxity. Such a task will subsequently miss its deadline. Hence FPZL dominates global FP scheduling, in the sense that all priority ordered tasksets that are schedulable according to global FP scheduling are also schedulable according to FPZL. FPZL is closely related to EDZL [43], [28], [9], [27], [50], [51], and [30] which applies the same zero-laxity rule to global EDF scheduling.

FPCL is a variant of FPZL that reduces the number of scheduling points with respect to FPZL. Like global fixed priority pre-emptive scheduling, FPCL re-schedules only at task release and completion events. FPCL is closely related to EDCL [45], which is a similar variant of EDZL.

1.1. Related work

For a comprehensive survey of multiprocessor real-time scheduling, including early work on utilisation-based...
schedulability tests for global FP, and global EDF scheduling of periodic tasksets with implicit deadlines, the interested reader is referred to [33].

During the last ten years, sophisticated schedulability tests have been developed for global FP, and global EDF scheduling of sporadic tasksets with constrained and arbitrary deadlines. These tests rely on the analysis of response times and processor load rather than utilisation.

In 2000, Andersson and Jonsson [1] provided a simple response time test applicable to tasksets with constrained-deadlines scheduled using global FP scheduling.

In 2003, Baker [6] developed a fundamental schedulability test strategy, based on considering the minimum amount of interference in a given interval that is necessary to cause a deadline to be missed, and then taking the contra-positive of this to form a sufficient schedulability test. This basic strategy underpins an extensive thread of subsequent research into schedulability tests for global EDF [10], [21], [15], [14], global FP [13], [22], [7], [37], and EDZL scheduling [28].

Baker's work was subsequently built upon by Bertogna et al. [19] in 2005, (see also Bertogna et al. [22]). They developed sufficient schedulability tests for: (i) any work conserving algorithm, (ii) global EDF, and (iii) global FP scheduling based on bounding the maximum workload in a given interval. In 2007, Bertogna and Cirinei [20] adapted this approach to iteratively compute an upper bound on the response time of each task, using the upper bound response times of other tasks to limit the amount of interference considered. In 2009, Guan et al. [39] extended the response time analysis of Bertogna and Cirinei [20] for global FP scheduling, using ideas from [12].

In 2009 and 2010, Davis and Burns [31], [32] showed that priority assignment is fundamental to the effectiveness of global FP scheduling. They proved that Audsley's optimal priority assignment algorithm [3], [4] is applicable to some of the sufficient tests developed for global FP scheduling, including the simple response time test of Andersson and Jonsson [1] and the deadline-based test of Bertogna et al. [22], but not to others such as the response time tests of Bertogna and Cirinei [20], and Guan et al. [39].

In 1989, Leung [44] considered global Least Laxity First (LLF), referred to in [44] as the Slack Time algorithm. Leung showed that global LLF dominates global EDF, and that determining exact schedulability under LLF, global EDF or global FP is a hard problem (co-NP-hard) for $m > 1$ (more than one processor).

The Earliest Deadline first until Zero Laxity (EDZL) algorithm was introduced in 1994 by Lee [43], who showed that EDZL dominates global EDF scheduling, and is sub-optimal for two processors (see also Cho et al. [30], Park et al. [51]). Here, sub-optimal is used to mean that EDZL can “schedule any feasible set of ready tasks”. This weak form of optimality is appropriate for online scheduling algorithms, which cannot take account of future arrival times. In 2006, Piao et al. [50] showed that EDZL is also completion time predictable A simpler proof of predictability was given by Cirinei and Baker [28] in 2007, who also developed a sufficient schedulability test for EDZL based on the fundamental strategy of Baker [6].

In 2008, Baker et al. [9] gave an iterative sufficient test for EDZL based on the approach taken by Bertogna et al. [22] for work conserving algorithms and global EDF. This test reduces the over-estimation of carry-in interference, a feature of the previous tests, by iteratively calculating a lower bound on the slack for each task. The empirical evaluation in [9] shows that this iterative test for EDZL outperforms other tests for EDZL given in [28] and, as expected, similar tests for global EDF.

Also in 2008, Kato and Yamasaki [45], introduced EDCL, effectively a variant of EDZL, which increases job priority on the basis of critical-laxity at the release or completion time of a job. This has the effect of reducing the maximum number of context switches to two per job, the same as EDF, at the expense of slightly inferior schedulability, when compared to EDZL. Kato and Yamasaki [45] also corrected a minor flaw in the polynomial time schedulability test for EDZL given in [28].

In 2009, Kato et al. [53], [47] presented research on RMZL (RMZL and FPZL are names for essentially the same scheduling algorithm). Papers [53] and [47] were initially published in Japanese, with an English language version of [53] subsequently made available in May 2010 as a technical report [48]. Independently, Davis and Burns developed schedulability analysis for FPZL [34], initially published as a technical report in April 2010 [35]. The analysis given for FPZL in [34] is applicable to constrained-deadline tasksets with no restrictions on the priority ordering which may be used; whereas the analysis given for RMZL in [48] is limited to implicit-deadline tasksets with task priorities assigned in Rate Monotonic priority order. As well as being more generally applicable, the FPZL analysis dominates, and significantly outperforms the RMZL schedulability test; see [34] for a detailed discussion and empirical comparison.

1.2. Intuition and motivation

The research described in this paper is motivated by the need to close the large gap that currently exists between the best known approaches to global multiprocessor real-time scheduling for sporadic tasksets with constrained deadlines and what may be possible as indicated by feasibility / infeasibility tests.

Dynamic priority scheduling has the potential to schedule many more tasksets than fixed task or fixed job priority algorithms. However, this theoretical advantage must be balanced against the increased overheads that dynamic changes in priority can bring via a significant increase in the number of pre-emption / migrations.

For example, the LLREF scheduling algorithm [29], which is optimal for periodic tasksets with implicit deadlines, and the LRE-TL scheduling algorithm [38] which
is optimal for sporadic tasksets with implicit deadlines, divide the timeline into intervals that start and end at task releases/deadlines (referred to as TL-planes in [29]). In each interval, LLREF and LRE-TL ensure that each active task \( \tau_i \) executes for at least \( U_i t \), where \( U_i \) is the task’s utilisation, and \( t \) is the length of the time interval. Hence every task can in the worst-case execute in every interval between task deadlines, resulting in \( n-1 \) pre-emptions per job release, where \( n \) is the number of tasks. In systems with a large number of tasks, this level of pre-emptions leads to prohibitively high overheads.

Minimally dynamic scheduling algorithms, such as FPZL and FPCL (and EDZL and EDCL) offer a potential solution to this problem. Note, by minimally dynamic, we mean that the priority of a job changes at most once during its execution, hence bounding the number of pre-emptions / migrations to at most two per job release. By comparison, global FP and global EDF scheduling incur at most one pre-emption / migration per job release.

We note that semi-partitioned scheduling algorithms [2], [40], [46], where a small number of tasks are permitted to migrate from one processor to another, offer an alternative approach to achieving enhanced schedulability without excessive overheads, based on partitioned rather than global scheduling.

### 1.3. Organisation

The remainder of the paper is organised as follows: Section 2 describes the terminology, notation and system model used. Section 3 describes sufficient tests for global FP scheduling. These tests are used in Section 4 to derive polynomial time and pseudo-polynomial time sufficient schedulability tests for FPCL. These schedulability tests are a generalisation of the tests given in [34] for FPZL. The more general tests given in this paper for FPCL are also applicable to FPZL when the critical laxity parameters are set to zero. Section 4 also shows how the schedulability tests for FPCL can be improved by bounding the amount of execution that each task can perform in the critical-laxity state. Section 5 discusses the implementation of FPCL and FPZL, assuming as a starting point an event driven global FP scheduler. Section 6 provides a brief discussion of priority assignment for FPCL and FPZL. Section 7 presents an empirical investigation into the effectiveness of FPCL and FPZL and their schedulability tests. Section 8 describes a prototype implementation of FPCL and FPZL and illustrates the effectiveness of the algorithms running on a multicore processor. Finally, Section 9 concludes with a summary and suggestions for future research.

### 2. System model, terminology and notation

In this paper, we are interested in FPCL and FPZL scheduling of an application on a homogeneous multiprocessor system comprising \( m \) identical processors. The application or taskset is assumed to comprise a static set of \( n \) tasks \( \{ \tau_1...\tau_n \} \), where each task \( \tau_i \) is assigned a unique priority \( i \), from 1 to \( n \) (where \( n \) is the lowest priority).

Tasks are assumed to comply with the sporadic task model. In this model, tasks give rise to a potentially infinite sequence of jobs. Each job of a task may arrive at any time once a minimum inter-arrival time has elapsed since the arrival of the previous job of the same task.

Each task \( \tau_i \) is characterised by its relative deadline \( D_i \), worst-case execution time \( C_i \), and minimum inter-arrival time or period \( T_i \). The utilisation \( U_i \) of each task is given by \( C_i/T_i \). A task’s worst-case response time \( R_i \) is defined as the longest time from a job of the task arriving to it completing execution.

It is assumed unless otherwise stated that all tasks have constrained deadlines (\( D_i \leq T_i \)). The tasks are assumed to be independent and so cannot be blocked from executing by another task other than due to contention for the processors. Further, it is assumed that once a task starts to execute it will not voluntarily suspend itself.

Intra-task parallelism is not permitted; hence, at any given time, each job may execute on at most one processor. As a result of pre-emption and subsequent resumption, a job may migrate from one processor to another. The cost of pre-emption, migration, and the run-time operation of the scheduler is assumed to be either negligible, or subsumed into the worst-case execution time of each task.

Under global FP scheduling, at any given time, the \( m \) highest-priority ready jobs are executed.

Under FPZL scheduling, if a job reaches zero laxity then it is given the highest priority and will execute until completion. The laxity of a job is given by the elapsed time to its deadline less its remaining execution time. Under FPZL, at any given time, at most \( m \) tasks may be in the zero-laxity state without a deadline being missed.

Schedulability analysis for FPZL may identify certain tasks as being able to enter the zero-laxity state. We refer to these tasks as zero-laxity tasks. An upper bound on the maximum amount of execution that a job of task \( \tau_i \) can perform in the zero-laxity state is denoted by \( Z_{i,b} \).

FPCL scheduling uses the concept of critical laxity [45]. The laxity of a task is said to be critical, if at a scheduling point (corresponding to task release or completion), the laxity of the task is such that it will inevitably miss its next deadline, if it is not scheduled to execute before the next scheduling point. Schedulability analysis for FPCL may identify certain tasks as being able to enter the critical-laxity state. We refer to these tasks as critical-laxity tasks. An upper bound on the maximum amount of execution that a job of task \( \tau_i \) can perform in the critical-laxity state is denoted by \( K_{i,b} \). Further, the critical laxity parameter \( X_i \) for task \( \tau_i \) is defined as the longest time that can elapse between one scheduling point and the next, while task \( \tau_i \) is ready but not executing.

The following notation is used to refer to subsets of tasks: \( hp(i) \) is the set of tasks with priorities higher than \( i \), \( lpz(i) \) is the set of zero-laxity tasks with initial priorities lower than \( i \), and similarly \( lpcl(i) \) is the set of critical-laxity
tasks with initial priorities lower than $i$.

Finally, when discussing the schedulability of a given task $\tau_k$, we use the term interference to refer to the execution of other tasks, at a priority higher than $k$, that can potentially delay the completion of task $\tau_k$.

3. Schedulability tests for global FP

In this section, we recapitulate two sufficient schedulability tests for global FP scheduling of sporadic task sets. These tests are described in more detail in [32].

3.1. Deadline Analysis for global FP

In [22], Bertogna et al. developed a polynomial time sufficient schedulability test for global FP scheduling based on the approach of Baker [6]. They showed that if task $\tau_k$ is schedulable in an interval of length $L$, then an upper bound on the interference over the interval due to a higher priority task $\tau_i$ with a carry-in job is given by the following equation. (Note a carry-in job is a job that is released strictly prior to the start of the interval, and causes interference within that interval).

$$I_i^D(L, C_k) = \min(W_i^D(L), L - C_k + 1)$$ (1)

where $W_i^D(L)$ is an upper bound on the workload of task $\tau_i$ in an interval of length $L$, given by:

$$W_i^D(L) = N_i^D(L)C_i + \min(C_i, L + D_i - C_i - N_i^D(L)T_i)$$ (2)

and $N_i^D(L)$ is the maximum number of jobs of task $\tau_i$ that contribute all of their execution time in the interval:

$$N_i^D(L) = \left[\frac{L + D_i - C_i}{T_i}\right]$$ (3)

Bertogna et al. [22] used Equation (1), with $D_k$ as the length of the interval, and strategy of Baker [6] to form a schedulability test for each task $\tau_k$:

**DA test for global FP scheduling:** A sporadic taskset is schedulable, if for every task $\tau_k$ in the taskset, the inequality given by Equation (4) holds:

$$D_k \geq C_k + \frac{1}{m} \sum_{i=1}^{m} I_i^D(D_k, C_k)$$ (4)

where $h_p(k)$ is the set of tasks with priorities higher than $k$. Note we have re-written Equation (4) in a different form from that presented in [22] for ease of comparison with the response time schedulability test given in [20].

Guarn et al. [39] showed that if task $\tau_k$ is schedulable in an interval of length $L$, then an upper bound on the interference in that interval due to a higher priority task $\tau_i$ without a carry in job is given by:

$$I_i^{NC}(L, C_k) = \min(W_i^{NC}(L), L - C_k + 1)$$ (5)

where:

$$W_i^{NC}(L) = N_i^{NC}(L)C_i + \min(C_i, L - N_i^{NC}(L)T_i)$$ (6)

and

$$N_i^{NC}(L) = \left\lfloor L/T_i \right\rfloor$$ (7)

The difference between the two interference terms given by Equations (1) and (5) is:

$$I_i^{DIFF-D}(L, C_k) = I_i^D(L, C_k) - I_i^{NC}(L, C_k)$$ (8)

Davis and Burns [32] showed that the worst-case scenario for global FP scheduling occurs when there are at most $m-1$ carry-in jobs. Thus, the approach of Guarn et al. [39], can be used to form an improved version of the DA test as follows:

**DA-LC test for global FP scheduling:** A sporadic taskset is schedulable, if for every task $\tau_k$ in the taskset, the inequality given by Equation (9) holds:

$$D_k \geq C_k + \frac{1}{m} \left( \sum_{i=1}^{m} I_i^{NC}(D_k, C_k) + \sum_{i=MD(k,m-1)} I_i^{DIFF-D}(D_k, C_k) \right)$$ (9)

where $MD(k, m-1)$ is the subset of the $\min(k, m-1)$ tasks with the largest values of $I_i^{DIFF-D}(D_k, C_k)$ from the set of tasks $h_p(k)$.

We note that the DA-LC test reduces to the DA test if the $I_i^{DIFF-D}(D_k, C_k)$ term is included for all of the higher priority tasks, rather than just those with the $m-1$ largest values, hence the DA-LC test dominates the DA test.

3.2. Response Time Analysis for global FP

Bertogna and Cirinei [20] extended the basic approach used in the DA test to iteratively compute an upper bound response time $R_k^{UB}$ for each task, using the upper bound response times of higher priority tasks to limit the amount of interference considered. This approach applies the same logic as [22], while recognising that the latest time that a task can execute is when it completes with its worst-case response time rather than at its deadline.

In [20], Bertogna and Cirinei showed that if task $\tau_k$ is schedulable in an interval of length $L$, then an upper bound on the interference in that interval due to a higher priority task $\tau_i$ with a carry-in job is given by:

$$I_i^R(L, C_k) = \min(W_i^R(L), L - C_k + 1)$$ (10)

where $W_i^R(L)$ is an upper bound on the workload of task $\tau_i$ in an interval of length $L$, taking into account the upper bound response time of task $\tau_i$:

$$W_i^R(L) = N_i^R(L)C_i + \min(C_i, L + R_i^{UB} - C_i - N_i^R(L)T_i)$$ (11)

and $N_i^R(L)$ is given by:

$$N_i^R(L) = \left[\frac{L + R_i^{UB} - C_i}{T_i}\right]$$ (12)

The response time test of Bertogna and Cirinei [20] may be expressed as follows:

**RTA test for global FP scheduling (Theorem 7 in [20]):** A sporadic taskset is schedulable, if for every task $\tau_k$ in the taskset, the upper bound response time $R_k^{UB}$ computed via the fixed point iteration given in Equation...
(13) is less than or equal to the task’s deadline:

$$R_k^{UB} \leftarrow C_k + \frac{1}{m \min_{i \in \text{chp}(k)}} \sum_{i \in \text{chp}(k)} R_i^{UB} C_i$$

Iteration starts with $R_k^{UB} = C_k$, and continues until the value of $R_k^{UB}$ converges or until $R_k^{UB} > D_k$, in which case task $\tau_k$ is unschedulable.

We note that using the RTA test, task schedulability needs to be determined in priority order, highest priority first, as upper bounds on the response times of higher priority tasks are required for computation of the interference term $I_i^R(R_k^{UB} C_k)$.

In [39], Guan et al. showed that at most $m-1$ higher priority tasks with carry-in jobs may contribute interference in the worst-case, and used this result to improve the RTA test as follows:

Guan et al. [39] showed that if task $\tau_i$ does not have a carry-in job, then the interference term is given by Equation (5). The difference between the two interference terms (Equation (10) and Equation (5)) is then given by:

$$I_i^{DIFF-R}(L_c, C_k) = I_i^R(L_c, C_k) - I_i^{NC}(L_c, C_k)$$

(14)

Using this result, Guan et al. [39] improved upon the response time test of Bertogna and Cirinei [20].

**RTA-LC test for global FP scheduling:** A sporadic taskset is schedulable, if for every task $\tau_k$ in the taskset, the upper bound response time $R_k^{UB}$ computed via the fixed point iteration given in Equation (15) is less than or equal to the task’s deadline:

$$R_k^{UB} \leftarrow C_k + \frac{1}{m} \left( \sum_{i \in \text{chp}(k)} I_i^{NC}(R_k^{UB} C_k) + \sum_{i \min_{i \in \text{chp}(k)}} I_i^{DIFF-R}(R_k^{UB} C_k) \right)$$

(15)

where $MR(k, m-1)$ is the subset of the $\min(k, m-1)$ tasks with the largest values of $I_i^{DIFF-R}(R_k^{UB} C_k)$, given by Equation (14), from the set of tasks $hp(k)$. Iteration starts with $R_k^{UB} = C_k$, and continues until the value of $R_k^{UB}$ converges or until $R_k^{UB} > D_k$, in which case task $\tau_k$ is unschedulable.

We note that the RTA-LC test reduces to the RTA test if the $I_i^{DIFF-R}(R_k^{UB} C_k)$ term is included for all of the higher priority tasks, rather than just those with the $m-1$ largest values, hence the RTA-LC test dominates the RTA test. Both the RTA and RTA-LC tests for global FP scheduling are pseudo-polynomial in complexity.

**4. Schedulability tests for FPCL (and FPZL)**

In this section, we derive polynomial time and pseudo-polynomial time sufficient schedulability tests for FPCL. These tests are applicable to sporadic tasksets with constrained deadlines, and are independent of the priority assignment policy used. They are based on the tests described in the previous section for global FP scheduling. We also show how the schedulability tests can be improved by computing a bound on the maximum amount of execution in the critical-laxity (or zero-laxity) state.

The FPCL scheduling algorithm is related to EDCL [45], but uses a somewhat different approach to determining when priority promotion should take place.

With global FP scheduling (and global EDF scheduling), re-scheduling is only required at task release or completion, as these are the only times at which the $m$ highest priority tasks can change. With FPZL (and EDZL) scheduling, the priority of a task also changes when the task reaches zero-laxity, and so additional scheduling points are required. This requires additional support in the operating system for rescheduling at zero-laxity events. Such support adds to both scheduler complexity and overheads.

In [45] Kato and Yamasaki proposed an alternative method of priority promotion, based on the concept of critical-laxity (CL) which provides most of the benefits of zero-laxity priority promotion, while ensuring that rescheduling takes place only on task release or completion.

The concept of critical laxity is as follows: at each scheduling point, corresponding to task release or completion, if there are more than $m$ ready tasks, then the laxity of each ready task is evaluated with respect to the maximum time that could potentially elapse until the next scheduling point. If at the next scheduling point, a task’s laxity could be negative, then it is classified as being a critical-laxity task (CL-task) and its priority promoted to the highest level immediately.

With FPZL, each zero-laxity task has its priority promoted when its laxity reaches zero. In contrast, under FPCL, each critical-laxity task $\tau_i$ has its priority promoted when a scheduling point occurs and the laxity of the task is less than or equal to the value of its critical-laxity parameter $X_i$. (Note that only tasks classified by the schedulability analysis as critical-laxity tasks can have their priority promoted in this way by FPCL).

With FPCL, the critical laxity parameter $X_i$ of each critical-laxity task $\tau_i$ is set to the smallest value such that the task is guaranteed to have its priority promoted before its laxity $x_i(t)$ becomes negative, despite the fact that priority promotion can only take place at scheduling points given by task release or completion.

As a job of a critical-laxity task $\tau_i$ can have its priority promoted at the earliest on its release, then:

$$X_i \leq D_i - C_i$$

(16)

Further, the maximum time between scheduling points during which other tasks can execute in preference to task $\tau_i$ is constrained by:

$$X_i \leq MC(m, hp(i), lpcl(i))$$

(17)

where $MC(m, hp(i), lpcl(i))$ returns the $m$th longest time that any job of a higher priority task can execute, or any job of a critical-laxity lower priority task can execute in the critical-laxity state.

Combining Equations (16) and (17), we have:

$$X_i = \min(D_i - C_i, MC(m, hp(i), lpcl(i)))$$

(18)

Note that the value of $X_i$ depends on which lower priority tasks are critical-laxity tasks, and on their execution times $K_j$ in the critical-laxity state. The set of critical-
laxity tasks can be determined by schedulability analysis. For now, we assume that $K_j = C_j$ for all critical-laxity tasks. We return to this point in Section 4.3.

We note that if any task in the taskset releases jobs strictly periodically, then that task’s period can also be used as a constraint on the critical laxity parameters for all tasks.

We now derive polynomial time and pseudo-polynomial time sufficient schedulability tests for FPCL. These schedulability tests are a generalization of the tests given for FPZL in [34]. With FPZL, there are additional scheduling points whenever the laxity of a task reaches zero. Hence with FPZL, the critical laxity parameter $X_i$ of each task $\tau_i$ is zero. Setting the critical laxity parameter of all tasks to zero in a schedulability test for FPCL provides an equivalent schedulability test for FPZL.

4.1. Deadline Analysis for FPCL (and FPZL)

Schedulability under FPCL is similar to that under FPZL:

1. Up to $m$ tasks may be deemed unschedulable according to analysis of their response times; and yet, due to the critical-laxity rule, the tasks will not miss their deadlines.
2. Critical-laxity tasks have an additional impact on the schedulability of other tasks.

We now derive the maximum interference on a higher priority task $\tau_j$, in an interval of length $L$, that could potentially be caused by a lower priority task $\tau_k$ executing for at most $K_j$ in the critical laxity state.

![Figure 1: Interference in an interval](image)

Figure 1 illustrates the worst-case scenario. This occurs when the last job of $\tau_j$ in the interval starts executing in the critical-laxity state as early as possible, completes at the end of the interval, at a time $X_j$ prior to its deadline. Further, each previous job of task $\tau_j$ is assumed to be released $T_j$ prior to the subsequent job, and to execute in the critical-laxity state as late as possible, thus completing at its deadline. We return to the precise behaviour of the first job of $\tau_j$ in the interval later.

An upper bound on the amount of critical-laxity workload due to task $\tau_j$ in an interval of length $L$ is given by:

$$W_j^{CL}(L) = \begin{cases} \min(L,K_j^{UB}) & L \leq T_j - X_j \\ K_j^{UB} + N_j^{CL}(L)K_j^{UB} + \min(K_j^{UB},L - T_j + X_j - N_j^{CL}(L)T_j) & L > T_j - X_j \end{cases}$$

where $N_j^{CL}(L)$ is the number of jobs of task $\tau_j$ that contribute all of their critical-laxity execution in the interval.

$N_j^{CL}(L) = \left\lfloor \frac{(L - T_j + X_j)}{T_j} \right\rfloor$ (20) and $K_j^{UB} (\leq C_j)$ is an upper bound on the amount of execution that any job of task $\tau_j$ can perform in the critical-laxity state.

If task $\tau_k$ is schedulable in an interval of length $L$, then an upper bound on the interference in that interval due to a lower priority task $\tau_j$ executing in the critical-laxity state is given by:

$$I_j^{CL}(L,C_k) = \min(W_j^{CL}(L),L - C_k + 1)$$ (21)

We now show that the interference from task $\tau_j$ executing in the critical-laxity state can be maximised without $\tau_j$ being considered as having a carry-in job. The scenario that maximises interference within an interval is shown in Figure 1. In this scenario, task $\tau_j$ appears to have a carry-in job; however, without any reduction in interference in the interval, the first job of task $\tau_j$ can be assumed to have become critical at the earliest at the start of the interval, but not before. Hence the first job of $\tau_j$ need not be in the critical-laxity state prior to the release of the problem job at the start of the interval. Task $\tau_j$ does not therefore need to be considered when determining the $m$-1 tasks that contribute the largest amounts of additional carry-in interference (i.e. the $I_i^{DIFF}$ terms – see Equation (8)). (Effectively, the first job of $\tau_j$ is only released at a priority higher than $k$ at or after the release of the problem job, thus it does not qualify as causing ‘carry-in’ interference).

We now consider the interference from a higher priority critical-laxity task $\tau_i$. In this case, the maximum interference with a carry-in job occurs when the first job of $\tau_i$ in the interval starts executing at the start of the interval, and completes at its deadline, with all subsequent jobs executing as early as possible, see Figure 2 below.

![Figure 2](image)

Figure 2

We observe that this is effectively the same scenario that leads to the worst-case interference from a higher priority task which does not enter the critical-laxity state but completes at its deadline, and is given by Equation (1). Similarly, critical-laxity execution cannot increase the amount of interference from a higher priority task with no carry-in job, given by Equation (5). This is an important observation. It means that when calculating interference from higher priority tasks, we do not need to know if they are critical-laxity tasks.

Under FPCL, each task $\tau_k$ is therefore schedulable without requiring priority promotion if the following inequality holds.
\[ D_k \geq C_k + \left( \sum_{i \in \text{high}(k)} I_{i}^{NC}(D_k, C_k) + \frac{1}{m} \sum_{i \in \text{medium}(k,m-1)} I_{i}^{\text{DIFF}-D}(D_k, C_k) + \sum_{j \in \text{low}(k)} I_{j}^{DC}(D_k, C_k) \right) \]  

(22)

where \( I_{i}^{NC}(D_k, C_k) \) is given by Equation (5), \( I_{i}^{\text{DIFF}-D}(D_k, C_k) \) is given by Equation (8), \( I_{j}^{CL}(D_k, C_k) \) is given by Equation (21), and \( \text{lpcl}(k) \) is the set of critical-laxity tasks with lower priorities than \( k \).

If the inequality in Equation (22) does not hold, then the task is a critical-laxity task. Under FPCL, at most \( m \) tasks can be critical-laxity tasks without a deadline being missed.

We note that the critical-laxity status of each task is unknown until its schedulability is checked via Equation (22), hence task schedulability needs to be checked in priority order, lowest priority first.

Algorithm 1 presents the DA-LC schedulability test for FPCL. Note, for now we make the pessimistic assumption that a critical-laxity task completes all of its execution in the critical-laxity state, hence line 9, ‘Compute \( K_k^{UB} \),’ can be assumed to set \( K_k^{UB} = C_k \).

The DA-LC schedulability test for FPCL is a polynomial time test requiring \( O(n^2) \) operations, assuming that ‘Compute \( K_k^{UB} \),’ takes linear time.

```
1 countCL = 0
2 for (each priority level k, lowest first) {
3     Determine schedulability of \( \tau_k \) according to Equation (22)
4     if ( \( \tau_k \) is not schedulable) {
5         mark \( \tau_k \) as a ‘critical-laxity’ task
6         countCL = countCL + 1
7         Calc \( X_k \) according to Equation (18)
8         Compute \( K_k^{UB} \)
9     }
10 }
11 if (countCL > m) return unschedulable
12 else return schedulable
```

Algorithm 1: DA-LC schedulability test for FPCL (and FPZL)

The schedulability test for FPCL given in Algorithm 1 reduces to the equivalent DA-LC schedulability test for FPZL by simply setting the critical laxity parameters \( X_j = 0 \) for all tasks in Equations (18), (19) and (20). In this case, the tasks classified by the test as critical-laxity tasks are in fact zero-laxity tasks.

As Equation (19) is monotonically non-decreasing in \( X_j \), then, for any interval length \( L \), Equation (21) yields interference \( I_{j}^{CL}(L,C_k) \) that is no greater when all \( X_j = 0 \), than it does for positive \( X_j \). Thus the DA-LC test for FPZL dominates the DA-LC test for FPCL, which in turn dominates the DA-LC test for global FP scheduling.

4.2. Response Time Analysis for FPCL (and FPZL)

In this section, we provide a response time test for FPCL. This sufficient schedulability test is a generalization of the equivalent test for FPZL given in [34]. It reduces to that test for FPZL by setting the critical laxity parameter \( X_j \) for each task \( \tau_j \) to zero.

The response time test for FPCL builds on the work of Bertogna and Cirinei [20] and Guan et al. [39] (Equation (14)). It computes an upper bound \( R_k^{UB} \) on the response time of each task \( \tau_k \). If task \( \tau_k \) is schedulable under FPCL with a response time bounded by \( R_k^{UB} \), then an upper bound on the interference in an interval of length \( R_k^{UB} \) due to a lower priority task \( \tau_j \) executing in the critical-laxity state can be obtained by substituting \( R_k^{UB} \) for the length of the interval in Equation (21).

An upper bound on the worst-case response time of a task \( \tau_k \), that is schedulable under FPCL without requiring priority promotion, can be found using the fixed point iteration given by Equation (23).

\[ R_k^{UB} \leftarrow C_k + \frac{1}{m} \sum_{i \in \text{high}(k,m-1)} I_{i}^{\text{DIFF}-R}(R_k^{UB}, C_k) + \sum_{j \in \text{lpcl}(k)} I_{j}^{DC}(R_k^{UB}, C_k) \]  

(23)

where \( I_{i}^{NC}(R_k^{UB}, C_k) \) is given by Equation (5), \( I_{i}^{\text{DIFF}-R}(R_k^{UB}, C_k) \) is given by Equation (14), \( I_{j}^{CL}(D_k, C_k) \) is given by Equation (21), and \( \text{lpcl}(k) \) is the set of critical-laxity tasks with lower priorities than \( k \).

Iteration starts with \( R_k^{UB} = C_k \), and continues until the value of \( R_k^{UB} \) converges in which case \( \tau_k \) is schedulable, or until \( R_k^{UB} > D_k \). If \( R_k^{UB} > D_k \), then the task is a critical-laxity task. Recall that under FPCL, at most \( m \) tasks may be critical-laxity tasks without a deadline being missed.

Using Equation (23), we can construct a sufficient schedulability test for FPCL based on upper bound response times; however, this requires an iterative approach that computes the upper bound response times of tasks in priority order, highest priority first, but then backtracks (re-starts) whenever a critical-laxity task is identified. This backtracking approach is necessary due to the dependency of higher priority task response times on which lower priority tasks are critical-laxity tasks and the dependency of lower priority task schedulability (critical-laxity status) on the response times of higher priority tasks.

Under FPCL, the interference term (Equation (21)) due to each lower priority critical-laxity task \( \tau_j \) depends via the parameter \( X_j \) (see Equation (18)) on the tasks with priorities lower than \( j \) that are also critical-laxity tasks. The interference term due to each task \( \tau_j \) is monotonically non-decreasing in \( X_j \), and \( X_j \) is monotonically non-decreasing as additional critical-laxity tasks are added to the set \( \text{lpcl}(j) \), hence interference can only increase as further critical-laxity
tasks are identified. This dependency implies that once a task is identified as a critical laxity task, the critical-laxity parameters and upper bound response times of all higher priority tasks must be re-calculated.

Algorithm 2 presents the RTA-LC schedulability test for FPCL. Algorithm 2 initially assumes that there are no critical-laxity tasks and starts computing task response times in priority order, highest priority first (lines 6 and 7). Then, whenever a task \( \tau_k \) is encountered where Equation (23) results in a value of \( R_k^{UB} > D_k \), the task is marked as a critical-laxity task and its upper bound response time is set to its deadline (lines 8 and 9). We note that provided that the taskset is schedulable under FPCL, then this is the correct upper bound response time, as the critical-laxity rule will prevent the task from actually missing its deadline.

```
1 countCL = 0
2 Initialize all \( R_k^{UB} = C_k \), \( X_k = 0 \) and \( K_k^{UB} = 0 \)
3 repeat = true
4 while (repeat) {
5     repeat = false
6     for (each priority level k, highest first) {
7         Determine \( R_k^{UB} \) according to Eq. (23)
8         if (\( R_k^{UB} > D_k \)) {
9             Calc \( X_k \) according to Equation (18)
10            Compute \( K_k^{UB} \)
11            if (\( \tau_k \) not marked as a CL task) {
12                \( \tau_k \) as a CL task
13                repeat = true
14            }
15            countCL = countCL + 1
16            if (countCL > m) {
17                repeat = false
18                break (exit for loop)
19            }
20        }
21        if (\( R_k^{UB} \) or \( K_k^{UB} \) differ from prev. values)
22            repeat = true
23    }
24    if (countCL > m)
25        return unschedulable
26    else
27        return schedulable
28 }
```

Algorithm 2: RTA-LC schedulability test for FPCL (and FPZL)

The discovery of a critical-laxity task effectively invalidates the upper bound response times calculated for all higher priority tasks, and also the critical-laxity parameters \( (X_j, ...) \) for all higher priority critical-laxity tasks. These values could be too small, and therefore need to be recalculated (line 14). However, if more than \( m \) critical-laxity tasks have been found, then the critical-laxity rule cannot prevent all deadline misses and the taskset is deemed unschedulable. In this case, the algorithm can exit immediately (lines 16-18).

We note that lines 21-22 are not required when a simple fixed value of \( K_k^{UB} = C_k \) is used for the critical-laxity execution time of task \( \tau_k \). However, when the computed value of \( K_k^{UB} \) depends on the response times of higher priority tasks then this additional convergence check is required. We return to this point in Section 4.3.

We note that the upper bound response time for a task \( \tau_i \) is monotonically non-decreasing in the amount of critical-laxity execution time of each of the tasks with lower priority than \( i \). Hence, the calculation of \( R_i^{UB} \) can be made more efficient on subsequent iterations of the ‘while’ loop (line 4) by using as an initial value, the value of \( R_i^{UB} \) computed on the previous iteration.

The ‘while’ loop (lines 4-25) continues to iterate until either \( m+1 \) critical laxity tasks are found, in which case the taskset is unschedulable under FPCL, or there are \( m \) or fewer critical-laxity tasks and the upper bound response times and critical-laxity parameters \( (X_j, ...) \) have been re-calculated since the final critical-laxity task was found. In this case, the taskset is schedulable.

Under the assumption that ‘Compute \( K_k^{UB} \)’, sets \( K_k^{UB} = C_k \), lines 21-22 are not required, and so the ‘while’ loop (line 4 to 25) only repeats when ‘repeat’ is set to true on line 14. This can only happen at most \( m \) times, as a result of finding a critical laxity task, before the taskset is declared unschedulable. Hence the maximum number of times that a response time can be computed (line 7) is \( O(mn) \). By comparison, the RTA-LC test for global FP scheduling requires \( O(n) \) such response time calculations.

The schedulability test for FPCL given in Algorithm 2 reduces to the equivalent RTA-LC schedulability test for FPZL by simply setting the critical laxity parameters \( X_j = 0 \) for all tasks in Equations (18), (19) and (20). In this case, the tasks classified by the test as critical-laxity tasks are in fact zero-laxity tasks.

As Equation (19) is monotonically non-decreasing in \( X_j \), then, for any interval length \( L \), Equation (21) yields interference \( I_{k,j}^{CL} (L,C_i) \) that is no greater when all \( X_j = 0 \), than it does for positive \( X_j \). Thus the RTA-LC test for FPZL dominates the RTA-LC test for FPCL, which in turn dominates the RTA-LC test for global FP scheduling.

4.3. Bounding critical laxity (zero-laxity) execution time

So far, we have made the potentially pessimistic assumption that a task that can reach the critical-laxity state does so without having started to execute. Hence, we used an upper bound on the critical-laxity execution time of \( K_k^{UB} = C_k \). In this section, we derive a more effective upper bound and use this bound to improve the schedulability tests for FPCL and FPZL. The analysis we provide is for the general case of FPCL. This analysis also applies to FPZL when all \( X_j \) are set to zero.

First, we introduce the concept of DC-Sustainability and prove that the schedulability tests for task \( \tau_k \) given by
Equations (22) and (23) are DC-Sustainable. A schedulability test for task $\tau_k$ is referred to as DC-Sustainable if it is sustainable [11] with respect to simultaneous and equal changes in both the execution time and the deadline of the task. Below we give a formal definition of DC-Sustainability.

**Definition:** A schedulability test $S$ for a task $\tau_k$ is DC-Sustainable if the following two conditions hold:

**Condition 1:** If task $\tau_k$ is deemed schedulable by test $S$ with some paired deadline and execution time values $D'_k = D_k - v$, $C'_k = C_k - v$ where $0 \leq v \leq C_k$ then test $S$ is guaranteed to deem task $\tau_k$ schedulable for all deadline and execution time pairs $D'_k = D_k - w$, $C'_k = C_k - w$ where $0 \leq w \leq C_k$.

**Condition 2:** If task $\tau_k$ is deemed unschedulable by test $S$ with some paired deadline and execution time values $D'_k = D_k - v$, $C'_k = C_k - v$ where $0 \leq v \leq C_k$ then test $S$ is guaranteed to deem task $\tau_k$ unschedulable for all deadline and execution time pairs $D'_k = D_k - w$, $C'_k = C_k - w$ where $0 \leq w \leq v$.

**Theorem 1:** Given a fixed set of critical-laxity parameters $(X_j)$ and a fixed set of critical-laxity tasks, Equation (22) is a DC-Sustainable schedulability test for task $\tau_k$.

**Proof:** We can re-write Equation (22) as follows:

$$D'_k - C'_k \geq \left[ \frac{1}{m} \left( \sum_{\forall \text{ch}(k)} I^NC(D_i', C_i') + \sum_{i \in \text{MDR}, m=1} I^{\text{DIFF}}_i(D_i', C_i') + \sum_{\forall \text{apcl}(k)} I^{\text{CL}}_i(D_i', C_i') \right) \right]$$

(24)

Consider the behaviour of Equation (24) for paired deadline and execution time values $D'_k = D_k - w$, $C'_k = C_k - w$ as $w$ takes different values in the range $0 \leq w \leq C_k$. The RHS of Equation (24) gives an upper bound on the interference from higher priority tasks and lower priority tasks executing in the critical-laxity state in an interval of length $D'_k = D_k - w$. By inspecting the component Equations (1), (2), (3), (5), (6), (7), (8), (19), (20), and (21) it can be seen that this interference is monotonically non-decreasing with respect to the length of the interval $D'_k$. We must however also consider the dependence of component Equations (5) and (21) on $C'_k$, which also varies with $w$. $C'_k$ appears in the second term in the min function of each of these equations in the expression $D'_k - C'_k \leq 1$. This expression is unchanged by varying $w$. The RHS of Equations (24) is therefore monotonically non-increasing with respect to increasing values of $w$.

In the case of Condition 1, as the LHS of Equation (24) is unchanged and the RHS is monotonically non-increasing for increasing values of $w$: $0 \leq w \leq C_k$ then it follows that, given that Equation (24) holds for $w=v$, it must also hold for all values of $w$: $v \leq w \leq C_k$.

In the case of Condition 2 as the LHS of Equation (24) is unchanged and the RHS is monotonically non-decreasing for decreasing values of $w$: $0 \leq w \leq C_k$ then it follows that, given that Equation (24) does not hold for $w=v$, then it cannot hold for any value of $w$: $0 \leq w \leq v$.

We now prove that Equation (23) is also a DC-Sustainable schedulability test for task $\tau_k$, given a fixed set of critical laxity parameters $(X_j)$ and a fixed set of critical-laxity tasks. Below, we re-write Equation (23), using the variable $q$ to indicate the fixed point iteration.

$$R'^{q+1}_k \leftarrow C'_k + \frac{1}{m} \left[ \sum_{\forall \text{apcl}(k)} I^NC(R^q_k, C'_k) + \sum_{i \in \text{MDR}, m=1} I^{\text{DIFF}}_i(R^q_k, C'_k) + \sum_{\forall \text{apcl}(k)} I^{\text{CL}}_i(R^q_k, C'_k) \right]$$

(25)

Recall that iteration begins with $R^0_k = C'_k$ (the execution time of task $\tau_k$), and ends when either $R^q_k = C'_k$ or when $R^{q+1}_k > D'_k$, in which case task $\tau_k$ is unschedulable.

Let $R^{UB}_k(D, C)$ be the response time upper bound given by Equation (25) for task $\tau_k(D, C)$ with deadline $D$ and execution time $C$. Similarly, let $R^{UB}_k(D+x, C+x)$ be the response time upper bound given by Equation (25) for task $\tau_k(D+x, C+x)$ with deadline $D+x$ and execution time $C+x$.

**Lemma 1:** Given a fixed set of critical-laxity parameters $(X_j)$ and a fixed set of critical-laxity tasks, if $\tau_k(D, C)$ is schedulable according to Equation (25) then $R^{UB}_k(D+x, C+x) \geq R^{UB}_k(D, C) + x$. Further, if $\tau_k(D, C)$ is not schedulable according to Equation (25) then neither is $\tau_k(D+x, C+x)$.

**Proof:** Let $R^q_k(D, C)$ be the value computed by the $q$th iteration of Equation (25) for task $\tau_k(D, C)$. Similarly, let $R^q_k(D+x, C+x)$ be the value computed by the $q$th iteration of Equation (25) for task $\tau_k(D+x, C+x)$.

We prove the Lemma by induction, showing that on every iteration $q$ until convergence or the deadline of $\tau_k(D, C)$ is exceeded, $R^q_k(D+x, C+x) \geq R^q_k(D, C) + x$.

Initial condition: in each case iteration starts with an initial value corresponding to the execution time of $\tau_k$, hence $R^0_k(C) = C$ and $R^0_k(C+x) = C+x$, so $R^0_k(C+x) \geq R^0_k(C) + x$.

Inductive step: assume that $R^q_k(C+x) \geq R^q_k(C) + x$, and consider the values computed for $R^{q+1}_k(D, C)$ and $R^{q+1}_k(D+x, C+x)$ on iteration $q+1$. The floor function (second term on the RHS of Equation (25)) contains three summation terms; together, these terms give an upper bound on the interference from higher priority tasks and lower priority tasks executing in the critical-laxity state in an interval of length $R^q_k$. Inspection of the component Equations (5), (6), (7), (10), (11), (12), (14), (19), (20), and (21)) shows that this interference term is no smaller for input values $R^q_k(C+x) \geq R^q_k(C) + x$, and $C'_k = C+x$ (the latter is used in Equations (10) and (21)) than it is for input values
$R^U_k (C)$ and $C'_k = C$, hence once the value of $C'_k$ is added (first term on the RHS of Equation (25)), we have $R^U_{k+1} (C + x) \geq R^U_k (C) + x$.

We note that if the fixed point iteration for $\tau_k (D, C)$ converges on $R^U_k (D, C) = R^U_{k+1} (C)$, then the smallest possible value of $R^U_k (D + x, C + x)$ is $R^U_{k+1} (C) + x$. Further, if $\tau_k (D, C)$ is unschedulable, then it follows that $R^U_{k+1} (C) > D$ which implies that $R^U_{k+1} (C + x) > D + x$ and therefore $\tau_k (D + x, C + x)$ must also be unschedulable $\square$

**Theorem 2:** Given a fixed set of critical-laxity parameters ($X_j$) and a fixed set of critical-laxity tasks, Equation (25), and hence Equation (23) is a DC-Sustainable schedulability test.

**Proof:** We can choose an execution time of $C'_j = 0$ and a deadline of $D'_k = D_k - C_k$ for task $\tau_k$. With these parameters, $\tau_k$ is deemed schedulable by Equation (25). We then consider all possible deadline and execution time pairs $D'_k = D_k - w, C_k = C_k - w$ for $w$ from 1 to $C_k$ (recall that execution times are represented by non-negative integers). Let $v$ be the smallest value of $w$, if any, for which $\tau_k$ is unschedulable. Lemma 1 tells us that for all larger values of $w$, $\tau_k$ will also be unschedulable. Proof that Conditions 1 and 2 in the definition of DC-Sustainability hold follows directly from the observation that task schedulability is monotonically decreasing with respect to increasing values of $w$ $\square$

We now show how a bound on the critical-laxity execution time of each critical-laxity task can be derived. Let us assume that we are using the DA-LC schedulability test (Algorithm 1) or the RTA-LC schedulability test (Algorithm 2) for FPCL, and that task $\tau_k$ has been identified as a critical-laxity task by Equation (22) or Equation (23). We know that task $\tau_k$ cannot be guaranteed to complete all of its execution within its deadline, without entering the critical-laxity state. However, if we can show that $\tau_k$ is guaranteed to complete $C'_j = C_j - v$ units of execution time by an effective deadline of $D'_k = D_k - X_k - v - 1$, then that proves that the task’s laxity is at least $X_k + 1$ at $D'_k$, and so it can execute for at most $v$ units of time in the critical-laxity state.

Due to the DC-Sustainability of the single task schedulability tests given by Equations (22) and (23), each of these equations can be used as the basis of a binary search to determine the smallest value of $v$ ($0 \leq v \leq C_k$) such that task $\tau_k$ is guaranteed to complete $C'_k = C_k - v$ units of execution time by a deadline $D'_k = D_k - X_k - v - 1$, thus computing an upper bound $K_{UB}^j = v$ on the amount of time that a job of task $\tau_k$ can spend executing in the critical-laxity state. The initial minimum value of $v$ for the search is $v = 0$, while the initial maximum value is $v = C_k$, which is deemed to result in schedulability, as it is equivalent to $\tau_k$ having zero execution time.

In the DA-LC test, a binary search based on Equation (22) can be used to ‘Compute $K_{UB}^j$’ (line 9 of Algorithm 1), for each critical-laxity task, improving the effectiveness of the test. As task schedulability is determined lowest priority first, no further iteration is required. At each priority level, task schedulability depends on static parameters of higher priority tasks, and on the critical-laxity status, parameters ($X_j$), and execution times ($K^{UB}_j$) of lower priority tasks which have already been computed.

In the RTA-LC test, a binary search based on Equation (23) can also be used to ‘Compute $K^{UB}_j$’, (line 11 of Algorithm 2) for each critical-laxity task. However, in this case, a further convergence check (lines 22-23) is required as the critical-laxity execution times computed by the binary searches are dependent on the response times of higher priority tasks, and vice-versa. We note that Algorithm 2 will either find more than $m$ critical-laxity tasks or converge on unchanging values for the response times, critical-laxity parameters, and critical-laxity execution times. Such convergence is guaranteed because:

(i) the response times of higher priority tasks are monotonically non-decreasing with respect to increases in the critical-laxity execution time of lower priority tasks, and similarly, the critical-laxity execution times of lower priority tasks computed by binary search are monotonically non-decreasing with respect to increases in the response times of higher priority tasks.

(ii) the critical-laxity parameter $X_j$ of a task $\tau_j$ is monotonically non-decreasing in the critical laxity execution times, and critical-laxity status of lower priority tasks.

(iii) the critical-laxity execution time $K_{UB}^j$ is monotonically non-decreasing with respect to the critical-laxity parameter $X_j$.

### 5. FPCL/FPZL and event-driven scheduling

In this section, we assume that the operating system already implements an event-driven global FP scheduler, we therefore discuss only the modifications required to support either FPCL or FPZL.

FPZL requires that when the laxity of a job reaches zero its priority is promoted to the highest level. The laxity of a job can reach zero at some intermediate point between task releases, and the completion of the currently running jobs. FPZL therefore requires support for additional zero-laxity timer events, typically handled via a timer interrupt from a fine-grained hardware timer-counter, with re-scheduling performed on those events, as well as at task release and completion. FPZL also requires the maintenance of a ‘task laxity’ queue of ready, but non-running tasks, ordered by increasing laxity. The laxity of the task at the head of this queue corresponds to the time to the next zero-laxity timer event. In a schedulable hard real-time system using FPZL, there are at most $m$ zero-laxity tasks and these tasks are known a priori, hence the ‘task laxity’ queue need only track the laxity of at most $m$ tasks. On expiry of a zero-laxity event, the scheduler runs and promotes the priority of the task at the head of the ‘task laxity’ queue to the highest.
FPCL requires no additional timer events / scheduling points, other than those provided by a standard global FP scheduler. However, at each scheduling point, the scheduler must first promote the priority of the jobs of critical-laxity tasks that have a laxity less than or equal to their critical-laxity parameter \( X_{j} \), before choosing the \( m \) highest priority tasks to run. As there are at most \( m \) critical-laxity tasks, this represents an additional overhead that is \( O(m) \).

FPCL reduces the number of scheduling points compared to FPZL. With FPCL, there are at most two context switches per task release (at release and completion), whereas with FPZL, there are at most three (at release, zero-laxity, and completion).

We note that the implementation of FPCL described above is highly efficient, but requires that the set of critical laxity tasks and their critical laxity parameters are known. This is only possible for tasksets that are deemed schedulable by one of the schedulability tests given in Section 4. However, it is possible to implement FPCL in a different way that does not require such information. While this alternative implementation is not as efficient as the one described above, it is more general and can be used with tasksets whose schedulability is unknown. This alternative implementation is therefore useful in exploring the effectiveness of FPCL, via simulations and experimental implementations, without the constraint that all of the tasksets examined must be deemed schedulable by an FPCL schedulability test.

The alternative implementation of FPCL is as follows: At each scheduling point (i.e. job release or completion) a set of at most \( m \) jobs are selected to run (the RUN set). The selection of the RUN set takes place according to the following steps:

1. As with a FP scheduler, the \( m \) highest priority ready jobs are initially selected as the RUN set. If there are no further ready jobs, then selection ends, otherwise it continues to step 2.
2. The maximum time \( Y \) to the next scheduling point is computed as the minimum remaining execution time of any job in the RUN set. The laxity of each ready job that is not in the RUN set is then computed on the basis that it will not start to run for a time \( Y \). If this laxity is negative, (i.e. the remaining execution time of the job + \( Y \) exceeds the time to the job’s deadline) then the job is marked as having critical laxity and is given the highest priority. If no critical-laxity jobs are found, then selection ends, otherwise it continues to step 3.
3. As a critical-laxity job has been found in step 2, the RUN set is re-evaluated such that it again contains the \( m \) highest priority jobs (at least one of which is now a critical-laxity job). If there are \( m \) or more\(^2\) critical laxity jobs, then selection ends, otherwise it continues from step 2.

We note that the above implementation may in the worst-case take up to \( m \) iterations of steps 2 and 3 to identify the critical-laxity jobs and so select which jobs to run. For relatively small numbers of processors (e.g. 2, 4, or 8), this approach results in a viable level of scheduling overheads as indicated by measurements of the prototype implementation described in Section 8.

Theorem 3: Any taskset that is deemed schedulable according to one of the schedulability tests for FPCL given in Section 4 is also schedulable under the alternative FPCL implementation described above.

Proof: To prove the theorem, we need only consider those tasks that are not identified as critical-laxity tasks by the schedulability test (we refer to such tasks as ordinary tasks), and show that (i) ordinary tasks remain schedulable under the alternative FPCL implementation, and (ii) that their jobs never become critical-laxity jobs. The remaining tasks (identified as critical-laxity tasks by the schedulability test) must then be trivially schedulable under the alternative FPCL implementation as it is able to guarantee the schedulability of up to \( m \) tasks via priority promotion.

Let \( r_{k} \) be any ordinary task that is part of a taskset that is deemed schedulable by the schedulability test. We make the initial assumption that no job of an ordinary task can have its priority promoted (i.e. become a critical-laxity job) under the alternative FPCL implementation. Under this assumption, we now show that the interference that ordinary task \( r_{k} \) is subject to due to the execution of some other task \( r_{i} \) cannot, when the alternative FPCL implementation is used, exceed that assumed by the schedulability test. This is trivially the case when \( r_{i} \) is also an ordinary task. If \( r_{i} \) is a critical-laxity task, then under the alternative FPCL implementation, jobs of \( r_{i} \) cannot enter the critical-laxity state earlier than assumed by the schedulability test due to the definition of the critical laxity parameter \( X_{i} \), see Equation (18). Hence, the interference due to \( r_{i} \) assumed by the schedulability test cannot be exceeded by that which occurs under the alternative FPCL implementation. Thus \( r_{k} \) remains schedulable under the alternative FPCL implementation without needing priority promotion.

Further, as the alternative FPCL implementation only promotes the priority of a job that would otherwise miss its deadline, no job of task \( r_{k} \) will have its priority promoted at run-time. As all jobs are initially released at their normal priorities, this is sufficient to show that our original assumption holds and hence that no job of an ordinary task can become a critical-laxity job under the alternative FPCL implementation. □

We note that Theorem 3 also applies with respect to the refined schedulability tests which make use of upper bounds on the amount of execution that can occur in the critical-

\(^2\) In the case of an unschedulable taskset, more than \( m \) jobs could become critical laxity jobs, in which case the RUN set arbitrarily contains the first \( m \) of them found. In this case some job is inevitably going to miss its deadline assuming that all jobs take their worst-case execution times.
laxity state. As priority promotion under the alternative FPCL implementation is guaranteed to take place no earlier than \( X_i \) prior to the deadline of each critical-laxity task \( \tau_i \), the time that each task \( \tau_i \) spends executing in the critical-laxity state under the alternative FPCL implementation is upper bounded by that assumed by the refined schedulability tests.

6. Priority assignment

In this section, we briefly discuss priority assignment for FPCL and FPZL. Davis and Burns [31], [32] showed that priority assignment is a key factor in global FP scheduling. As FPCL and FPZL are hybrids of global FP scheduling, we expect priority assignment to also be important for these scheduling algorithms.

The DA-LC and RTA-LC schedulability tests for FPCL and FPZL are independent of the priority ordering used. Hence they are compatible with heuristic priority assignment policies such as Deadline Monotonic Priority Ordering (DMPO) or DkC [31], [32]. When there are no critical-laxity (zero-laxity) tasks, FPCL (FPZL) reduces to global FP scheduling. In this case, Audsley’s Optimal Priority Assignment (OPA) algorithm [3], [4] provides the optimal priority assignment to use in conjunction with the DA-LC tests. However, when the OPA algorithm finds that there are no tasks that are schedulable at a particular priority level without recourse to the critical-laxity (zero-laxity) rule, then the following question arises: Which task should be assigned to that priority level? For the purposes of the empirical evaluation in Section 7, we answered this question via a simple heuristic. We computed the critical-laxity (zero-laxity) execution time for each unassigned task using a binary search, and assigned the task with the smallest proportion of its execution time in that state. The idea being that this is the task that would require the smallest percentage reduction in its execution time to be schedulable at that priority without recourse to the critical-laxity (zero-laxity) rule.

7. Empirical investigation

In this section, we present the results of an empirical investigation, examining the effectiveness of the schedulability tests for FPCL and FPZL. We also conducted scheduling simulations which form necessary but not sufficient schedulability tests, thus providing upper bounds on the potential performance of the various scheduling algorithms.

7.1. Taskset parameter generation

The taskset parameters used in our experiments were randomly generated as follows:

- Task utilisation were generated using the UUnifast-Discard algorithm [31], giving an unbiased distribution of task utilisation. A discard limit of 1000 was used, but not needed.
- Task periods were generated according to a log-uniform distribution with a factor of 1000 difference between the minimum and maximum possible task period. This represents a spread of task periods from 1ms to 1 second, as found in most hard real-time applications.
- The log-uniform distribution was used as it generates an equal number of tasks in each time band (e.g. 1-10ms, 10-100ms etc.), thus providing reasonable correspondence with real systems.

In each experiment, the taskset utilisation (x-axis value) was varied from 0.025 to 0.975 times the number of processors in steps of 0.025. For each utilisation value, 1000 valid tasksets were generated and the schedulability of those tasksets determined using the various schedulability tests for different scheduling algorithms. The graphs plot the percentage of tasksets generated that were deemed schedulable in each case. Note the lines on all of the graphs appear in the order given in the legend. (The graphs are best viewed online in colour).

7.2. Scheduling simulation

We used a simulation of global FP, FPCL, FPZL, global EDF and EDZL scheduling to provide an upper bound on the potential performance of each scheduling algorithm, and hence to evaluate the quality of the schedulability tests. (Note the alternative implementation of FPCL described in Section 5 was used for the simulation. The more efficient implementation using pre-computed critical laxity parameters relies on the identification of critical-laxity tasks at the schedulability analysis stage, and so it was not possible to simulate its behaviour for tasksets that were not deemed schedulable by the analysis).

Our simulations ran for an interval of time equal to ten times the longest period of any task in the taskset. Each simulation started with synchronous release of the first job of each task, with subsequent jobs released as early as possible. Each job executed for its worst-case execution time. The simulation deemed a taskset schedulable by a given algorithm if it did not find a deadline miss during the time interval simulated, or any unavoidable deadline miss for any job that had execution time remaining at the end of the interval. Thus the simulation provides a necessary but not sufficient schedulability test. Any taskset failing the simulation, with a deadline miss, is guaranteed to be unschedulable, while tasksets that pass the simulation may or may not be schedulable.

We note that in the case of constrained-deadline sporadic tasksets, to the best of our knowledge, no tractable exact tests exist for any of the algorithms studied. Thus upper bounds on performance derived via simulation are one of the few ways in which the performance potential of each algorithm can be explored.
7.3. Schedulability test effectiveness

We investigated the performance of the FPCL and FPZL DA-LC, schedulability tests using Audsley’s OPA algorithm [3], [4] to assign priorities, and compared their performance to that of the equivalent test for global FP scheduling, and to schedulability tests for global EDF [22] (the “EDF-RTA” test) and EDZL scheduling [9] (the “EDZL-I test”). Also shown on the graphs are results for the necessary infeasibility test of Baker and Cirinei [8] (labelled “LOAD*”). This line gives the total number of tasksets at each utilisation level that we cannot be certain are infeasible (i.e. unschedulable by any algorithm). Further, the narrow lines on the graphs indicate an upper bound on the performance of each algorithm found via simulation. In the case of global FP, FPCL, and FPZL scheduling, these upper bounds assume Deadline minus Computation time Monotonic Priority Ordering (DCMPO) [31], which was found in the simulation studies to be significantly more effective than Deadline Monotonic Priority Ordering (DMPO). Note, it was not possible to simulate optimal priority assignment as simulation of all possible priority orderings is intractable.

Figures 3 to 5 below are for constrained-deadline tasksets. From these graphs, we can see that the EDF-RTA test for global EDF scheduling and the DA-LC test for global FP scheduling using DMPO have the lowest performance, with approximately 50% of the generated tasksets schedulable at a utilisation of 2.7 (=0.34m) and 2.8 (=0.35m) respectively, in the 8 processor case. The EDZL-I test performs significantly better with 50% of the tasksets schedulable at a utilisation of approx. 3.4 (=0.43m). Using optimal priority assignment significantly improves the performance of global FP scheduling, with 50% of the tasks schedulable at a utilisation of approximately 4.7 (=0.59m) according to the DA-LC test. The DA-LC test for FPZL, using Audsley’s OPA algorithm and a binary search to bound zero-laxity execution time (marked FPZL-LZ on the graph) has the highest performance, with 50% of tasks deemed schedulable at a utilisation of approx. 4.9 (=0.61m). As expected, this is slightly better than the DA-LC test for FPCL, again using Audsley’s OPA algorithm and a binary search to bound critical-laxity execution time (marked FPCL-LC on the graph), with 50% of tasks deemed schedulable at a utilisation of approx. 4.8 (=0.60m). Both FPCL and FPZL algorithms provide a modest improvement over global FP scheduling.

Our simulation results show that both global EDF and global FP scheduling with DMPO have relatively poor performance potential. This is because these algorithms typically favour executing tasks with short deadlines first. This has the effect of reducing the amount of available concurrency, in terms of the number of ready tasks, which makes the remaining tasks more difficult to schedule. By contrast, using DCMPO greatly improves the performance potential of global FP scheduling, particularly when there are a large number of processors and tasks. The simulation results show that EDZL, FPZL and FPCL (both with DCMPO priority ordering) have similar performance potential, which as the number of processors and tasks increases becomes close to the upper bound given by the LOAD* infeasibility test. As expected the performance of FPCL was marginally inferior to that of FPZL.

Figures 6 to 8 show the results of the same experiments, repeated for implicit-deadline tasksets. These graphs show that the performance of the schedulability tests for FPCL and FPZL significantly exceed that of the best known tests for global FP, global EDF and EDZL, with an increased gap between both FPCL and FPZL, and global FP scheduling using OPA, compared to the constrained deadline case. For example, in the 8 processor case, approximately 50% of the generated tasksets were schedulable at a utilisation of 6.1 (=0.76m) using FPCL (OPA) or FPZL (OPA), compared to 5.8 (=0.725m) for global FP scheduling using OPA, and 5 (=0.63) for EDZL(I). This increase in the relative performance of FPCL (and FPZL) is mainly due to the calculation of a less pessimistic bound on the amount of critical-laxity (zero-laxity) execution time having an increased effect compared to the constrained-deadline case. Further, the simulation results show that the performance potential of EDZL, FPZL and FPCL (with DCMPO) is very similar, with all three algorithms potentially able to schedule nearly all of the tasksets generated.
For implicit-deadline tasksets, we used our experimental results to obtain approximate values for the Optimality Degree (OD) [24] of each scheduling algorithm / schedulability test examined, over a domain corresponding to the tasksets generated in our experiments.

The Optimality Degree of a scheduling algorithm \( A \) combined with a schedulability test \( S \) is defined with respect to a domain of tasksets. It is given by the number of tasksets in the domain that are schedulable using algorithm \( A \) according to schedulability test \( S \), divided by the number of feasible tasksets in the domain. Hence an optimal algorithm supported by an exact schedulability test has OD = 1 for any domain.

For sporadic tasksets with implicit-deadlines, the utilisation bound for LRE-TL [38] is 100%, hence all of the implicit-deadline tasksets generated in our experiments are feasible (as their utilisation does not exceed \( m \)). For each of the algorithms / schedulability tests examined, an approximate value for the Optimality Degree can therefore be obtained by simply counting the total number of schedulable tasksets over the full range of utilisation values,
and dividing this number by the total number of tasksets generated. The Optimality Degree of each algorithm / schedulability test is given in Table 1 below, expressed as a percentage.

**Table 1: Approximate Optimality Degree**

<table>
<thead>
<tr>
<th>Algorithm / test</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPZL LZ DA-LC (OPA)</td>
<td>84.7%</td>
<td>79.4%</td>
<td>77.4%</td>
<td>76.6%</td>
</tr>
<tr>
<td>FPCL CL DA-LC (OPA)</td>
<td>83.7%</td>
<td>78.5%</td>
<td>76.7%</td>
<td>76.0%</td>
</tr>
<tr>
<td>FP DA-LC (OPA)</td>
<td>81.4%</td>
<td>75.7%</td>
<td>73.6%</td>
<td>73.2%</td>
</tr>
<tr>
<td>FP DA-LC (DMPO)</td>
<td>80.1%</td>
<td>70.0%</td>
<td>62.8%</td>
<td>57.2%</td>
</tr>
<tr>
<td>EDZL(I)</td>
<td>71.7%</td>
<td>66.2%</td>
<td>63.5%</td>
<td>62.3%</td>
</tr>
<tr>
<td>EDF (RTA)</td>
<td>74.2%</td>
<td>67.4%</td>
<td>62.5%</td>
<td>58.5%</td>
</tr>
</tbody>
</table>

Table 1 shows that the Optimality Degree for FPZL scheduling using the polynomial time DA-LC schedulability test derived in this paper, (with OPA priority assignment and zero-laxity execution time calculation) is 3-4% better than for global FP scheduling using OPA and an equivalent schedulability test, and 13% better than for EDZL, assuming the iterative schedulability test given in [9]. By comparison, FPCL scheduling has an Optimality Degree that is approx. 1% worse than FPZL and thus 2-3% better than global FP scheduling.

**8. Prototype implementation and experimental results**

In this section, we present our implementation of the FPCL, FPZL, and global FP scheduling algorithms in the Linux kernel 2.6.35, comparing practical implementation overheads of those algorithms in a real-world environment. Given our primary goal is to evaluate the effectiveness of priority promotion with different rules; we focused only on fixed-priority scheduling algorithms.

**8.1. Prototype implementation**

We used the Linux kernel 2.6.35 as the underlying operating system for our implementation. The Linux kernel provides fixed-priority scheduling policies, also known as \texttt{SCHED_FIFO} and \texttt{SCHED_RR}. The \texttt{SCHED_FIFO} policy does not pre-empt tasks executing at the same priority level, whereas the \texttt{SCHED_RR} policy defines a \textit{time-slice} such that tasks at the same-priority are scheduled in a round-robin fashion. Since the tiebreaking rule among tasks at the same priority level does not affect schedulability for global FP-based scheduling algorithms, we implemented FPCL, FPZL, and global FP based on the \texttt{SCHED_FIFO} policy.

In our experience, even the tasks scheduled under the \texttt{SCHED_FIFO} policy may still be migrated on to different processors due to load balancing. To avoid such unexpected migrations, we force the \texttt{cpus_allowed} flag for each task to identify only the current processor so that the task is never migrated unless specifically required to do so by the CPU scheduler. We also modified the CPU scheduler to ensure that the tasks scheduled under the \texttt{SCHED_FIFO} policy are never pre-empted for any reason by background tasks assigned other scheduling policies.

We provide six system calls in our implementation. Figure 9 shows sample code for userspace tasks, where the \texttt{syscall_*} interfaces correspond to those system calls. A set of WCET, period, relative deadline, and priority parameters need to be set explicitly via the system calls. \texttt{syscall_run()} releases the first job of the task, and \texttt{syscall_wait_for_period()} generates a scheduling point for the Linux kernel. There is another interface, \texttt{syscall_wait_for_interval()}, to wait for a specific time interval if the task is not periodic. In fact, most Linux-based real-time operating systems [18], [26], [36], [49], [52] provide a similar set of programming interfaces.

```c
main(timeval C, timeval T, timeval D)
int prio, int nr_jobs,
{
    syscall_set_wcet(C);
    syscall_set_period(T);
    syscall_set_deadline(D);
    syscall_set_priority(prio);
    syscall_run();
    for (i = 0; i < nr_jobs; i++) {
        /* User’s code. */
        ...
        syscall_wait_for_period();
    }
}
```

**Figure 9: Sample code for user space tasks**

Our implementations of FPCL and global FP only dispatch new tasks in \texttt{syscall_run()} and \texttt{syscall_wait_for_period()}, since all context switches are aligned with the releases and completions of jobs. Hence, the CPU scheduler only needs to set a new value for the \texttt{cpus_allowed} mask for the dispatched task, and call the \textit{migration thread} supported by the Linux kernel, to migrate the task on to an appropriate processor. FPZL, on the other hand, is implemented in a somewhat more complex way. Under FPZL a task needs to be assigned the highest priority when the laxity of its job becomes zero. At
every scheduling point we therefore first determine if such a situation can occur before the next scheduling point. If so, we look ahead in the schedule to see when this will happen, and set up a high-resolution timer to invoke the scheduler at that time. The task dispatching procedure is the same as for FPCL and global FP.

8.2. Experimental results

We now compare our implementations of FPCL, FPZL, and global FP scheduling, using a 2.0 GHz Intel Core 2 Quad processor (Q9650) with 2 GBytes of main memory. Since our goal is to evaluate implementation overheads in scheduling, rather than evaluating basic performance (such as kernel response times and cache effects) we executed busy-loop tasks with exactly the same timing parameters as used in the simulations presented in Section 7. Each task uses the system calls presented in Section 8.1, and has the same structure of code illustrated in Figure 9.

We repeatedly measured the number of busy loops needed to correspond to the execution time of each task given by its WCET parameter, and used the minimum value obtained as the number of busy loops in the experiment, so as to minimize execution time overruns. We also measured the maximum execution time of a single scheduler invocation, and this execution time is included in the calculation of the laxity of a job.

The alternative implementation of the FPCL algorithm (described in Section 5) was used, so that tasksets could be executed that were not deemed schedulable by the (sufficient) schedulability tests. With a four core processor, this implementation required a maximum of 4 iterations. The maximum observed execution time of the scheduler was as follows (figures for a taskset of size 20): Linux scheduler only: 19.95uS, Linux scheduler + FPCL algorithm 26.12uS. This equates to an increase in the scheduler execution time of approx. 31%. This represents a moderate increase given that the baseline scheduler overheads are small.

Figure 10 and Figure 11 show the results of our experiments for constrained-deadline tasks, using the same basic taskset parameters as the simulations (Figure 3 and Figure 4 in Section 7.3). As the experiments with real hardware took considerably longer to run, we examined 100 tasksets at each utilization level, rather than 1000 as used in the simulations.

We note that the experimental results do not match the simulation results at high utilisation levels. In the experiments on the Q9650 processor, there are a larger percentage of tasksets that do not exhibit deadline misses at high utilisation levels compared to the scheduling simulations. This is due to the way in which the task execution times are approximated by the busy-wait loop variable. While this avoids execution time overruns, execution time under-runs mean that high priority, short period tasks typically do not generate their full utilisation over a long time period, and so lower priority, longer deadline tasks are less likely to miss their deadlines than would otherwise be the case. To characterise these differences, we measured the average-case utilisation of the tasksets. Excerpts from this data are shown in Table 2 for the experiments using 4 processors and 20 constrained-deadline tasks. Despite these difficulties, the experimental results given in Figure 10 and Figure 11 provide a means of comparing the three algorithms.

<table>
<thead>
<tr>
<th>Table 2: Average-case utilisation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expt.</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$m = 4, n = 20, D \leq T$</td>
</tr>
<tr>
<td>1.5</td>
</tr>
<tr>
<td>1.6</td>
</tr>
<tr>
<td>1.7</td>
</tr>
<tr>
<td>1.8</td>
</tr>
<tr>
<td>1.9</td>
</tr>
</tbody>
</table>

FPCL and FPZL successfully scheduled more task sets than global FP, as expected from the simulation results. While FPCL and FPZL were similarly competitive, FPZL
was very sensitive to the maximum cost estimation of a single scheduler invocation. If this estimation is optimistic, then FPZL causes many more deadline misses than FPCL. This happens because FPZL attempts to complete a zero-laxity job at its precise deadline; hence if execution times or scheduler invocation costs are under-estimated, the schedulability of FPZL is affected significantly. FPCL, on the other hand, is a more robust algorithm in this regard, because it typically tries to complete a critical-laxity job someway before the deadline.

9. Conclusions and future work

The motivation for our work was the desire to improve upon current state-of-the-art global scheduling methods for hard real-time systems in terms of practical techniques that enable the efficient use of processing capacity.

The intuition behind our work was that dynamic priority scheduling has the potential to schedule many more tasksets than fixed task or fixed job priority algorithms, and yet this theoretical advantage has to be tempered by the need to avoid prohibitively large overheads due to a high number of pre-emption. This led us to consider minimally dynamic scheduling algorithms which permit each job to change priority at most once during its execution. We introduced two such algorithms, based on global FP scheduling, called FPCL and FPZL. The number of context switches with FPZL is at most three per job for each zero-laxity task, and at most two per job for ordinary tasks. As there are at most \( m \) zero-laxity tasks, the increase in overheads compared to global FP scheduling is tightly bounded. With FPCL task priorities only change at task release and completion events, thus the number of context switches with FPCL is at most two per job.

The key contributions of this paper are as follows:

- The introduction of the FPCL and FPZL scheduling algorithms.
- The derivation of effective polynomial time and pseudo-polynomial time sufficient schedulability tests for FPCL and FPZL, based on similar tests for global FP scheduling.
- Improvements to these tests, bounding the amount of execution that may take place in the critical-laxity (zero-laxity) state.

The main conclusions that can be drawn from our empirical investigations are as follows:

- The zero-laxity rule employed by FPZL appears to have a large impact on taskset schedulability, compared to the performance of global FP scheduling, as shown by the simulation results. The performance potential of FPZL using DCMPO was found to be broadly similar to that of EDZL, and significantly better than that of global FP or global EDF scheduling.
- Using Audsley’s OPA algorithm to assign task priorities, the polynomial time schedulability tests for FPCL and FPZL result in a modest improvement over the equivalent test for global FP scheduling in the case of constrained-deadline task sets, with an increased improvement for implicit-deadline tasksets.
- The schedulability tests for FPCL and FPZL derived in this paper, and the best known schedulability tests for global FP scheduling, appear to significantly outperform tests for global EDF and EDZL. Even so, there remains a large gap between the sufficient schedulability tests for FPZL and what might be possible as shown by the simulation results.

Given the similarities between FPZL and EDZL, it is
interesting to consider why the schedulability tests for FPZL significantly outperform those for EDZL. All of these schedulability tests are sufficient, and so suffer from a degree of pessimism in terms of the computed interference. The advantage that the schedulability tests for FPZL have over those for EDZL is that this pessimism is restricted to tasks with higher priorities and lower priority zero-laxity tasks. With the schedulability tests for EDZL (and EDF), there is pessimism attributable to the calculation of interference from all other tasks. Further, the techniques derived in this paper, reduce the amount of interference considered due to tasks executing in the zero-laxity state, by bounding the amount of execution that takes place in that state. Nevertheless, the tests for FPZL have an additional element of pessimism compared to similar tests for global FP scheduling due to the inclusion of zero-laxity tasks in the interference term. This may account for the fact that the difference in performance between the schedulability tests for FPZL and global FP scheduling is not as large as the difference in the potential performance of the two algorithms as shown by simulation.

We implemented global FP, FPCL, and FPZL scheduling using the Linux kernel 2.6.35 as the underlying operating system. Our experimental implementation showed that both FCPL and FPZL can improve significantly upon the performance of global FP scheduling; however, FCPL is easier to implement and more robust that FPZL, when task execution times and scheduling overheads are subject to a small amount of uncertainty.

9.1. Acknowledgements

This work was funded by the EPSRC Tempo project (EP/G055548/1) and the EU funded ArtistDesign Network of Excellence. The authors would like to thank Alan Burns for his comments on an earlier draft of this paper.

References


