Real-Time Syst (2012) 48:750–788
DOI 10.1007/s11241-012-9149-x

FPSL, FPCL and FPZL schedulability analysis

Robert I. Davis · Shinpei Kato

Published online: 24 March 2012

Abstract This paper presents the Fixed Priority until Static Laxity (FPSL), Fixed Priority until Critical Laxity (FPCL) and Fixed Priority until Zero Laxity (FPZL) 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. FPSL and FPCL are variants of FPZL that introduce no additional scheduling points beyond those present with fixed priority scheduling. FPSL, FPCL and FPZL 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 for these algorithms. The tests are then improved by computing upper bounds on the amount of execution that each task can perform at the highest priority. An empirical evaluation shows that FPSL, FPCL, and FPZL are highly effective, with

This paper builds on the paper "FPZL Schedulability Analysis" by Davis and Burns (2011b) published in the proceedings of RTAS 2011. This paper extends Davis and Burns (2011b) analysis to cover the FPSL and FPCL scheduling algorithms as well as FPZL. The analysis given in this paper is a superset of that provided for FPZL and reduces to it if the laxity threshold of each task is set to zero. In this paper, the experimental evaluation has been extended to include schedulability tests for FPSL (which also apply to FPCL) and simulation of FPCL as well as FPZL. We also report on a prototype implementation of FPCL and FPZL in a Linux kernel, running on an Intel Core 2 Quad processor (Q9650).

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

S. Kato
Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail: shinpei@ece.cmu.edu

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.

Keywords Real-time · Schedulability analysis · Fixed priority · Zero laxity · Critical laxity · FPSL · FPCL · FPZL · Global scheduling · Multiprocessor

1 Introduction

Approaches to multiprocessor real-time scheduling, can be categorised 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 run-time. Each approach has its distinct advantages and disadvantages. Partitioned scheduling typically has lower overheads in accessing and manipulating run-queues (Brandenburg et al. 2008), 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 by Bastoni et al. (2010a) 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 (Bastoni et al. 2010b).

In this paper, we focus on global scheduling techniques with the aim of increasing their effectiveness, in terms of the number of tasksets that can be guaranteed schedulable, without compromising efficiency, in terms of the overheads caused by pre-emption and migration. We present minimally dynamic global scheduling algorithms for real-time multiprocessor systems called Fixed Priority until Static Laxity (FPSL), Fixed Priority until Critical Laxity (FPCL) and FPZL (Fixed Priority until Zero Laxity). We consider the use of these algorithms to schedule sporadic tasks with constrained deadlines.

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 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 (Lee 1994; Cho et al. 2002; Park et al. 2005; Baker and Cirinei 2006; Piao et al. 2006; Cirinei and Baker 2007; Chao et al. 2008) which applies the same priority promotion rule to global EDF scheduling.

FPSL and FPCL are variants of FPZL that reduce the number of scheduling points with respect to FPZL. Like global fixed priority pre-emptive scheduling, FPSL and FPCL re-schedule only at job release and completion events. FPCL is closely related to EDCL (Kato and Yamasaki 2008), 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 Davis and Burns (2011b).

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.

Andersson and Jonsson (2000) provided a simple response time test applicable to tasksets with constrained-deadlines scheduled using global FP scheduling.

Baker (2003) 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 (Baker and Baruah 2009; Bertogna 2007; Baruah and Baker 2009; Baruah et al. 2009), global FP (Baruah and Fisher 2008; Bertogna et al. 2009; Baker 2006; Fisher and Baruah 2006), and EDZL scheduling (Cirinei and Baker 2007).

Baker's work was subsequently built upon by Bertogna et al. (2005, 2009). 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. Bertogna and Cirinei (2007) 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. Guan et al. (2009) extended the response time analysis of Bertogna and Cirinei (2007) for global FP scheduling, using ideas from Baruah (2007).

Davis and Burns (2009, 2010a) showed that priority assignment is fundamental to the effectiveness of global FP scheduling. They proved that the optimal priority assignment algorithm of Audsley (1991, 2001) is applicable to some of the sufficient tests developed for global FP scheduling, including the simple response time test of Andersson and Jonsson (2000) and the deadline-based test of Bertogna et al. (2009), but not to others such as the response time tests of Bertogna and Cirinei (2007), and Guan et al. (2009).

Leung (1989) considered global Least Laxity First (LLF), referred to in that paper 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 by Lee (1994), who showed that EDZL dominates global EDF scheduling, and is sub-

optimal for two processors (see also Cho et al. 2002; Park et al. 2005). 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. Piao et al. (2006) showed that EDZL is also completion time predictable A simpler proof of predictability was given by Cirinei and Baker (2007), who also developed a sufficient schedulability test for EDZL based on the fundamental strategy of Baker (2003).

Baker et al. (2008) gave an iterative sufficient test for EDZL based on the approach taken by Bertogna (2007) and Bertogna et al. (2009) 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 by Baker et al. (2008) shows that this iterative test for EDZL outperforms other tests for EDZL (Cirinei and Baker 2007) and as expected, similar tests for global EDF.

Kato and Yamasaki (2008), introduced EDCL; a variant of EDZL, which increases job priority on the basis of 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 global EDF, at the expense of slightly inferior schedulability, when compared to EDZL. Kato and Yamasaki (2008) also corrected a minor flaw in the polynomial time schedulability test for EDZL in Cirinei and Baker (2007).

Takeda et al. (2009) and Kato and Yamasaki (2009b) presented research on RMZL (RMZL and FPZL are names for essentially the same scheduling algorithm). These papers were initially published in Japanese, with an English language version of Takeda et al. (2009) subsequently made available in May 2010 as a technical report (Kato et al. 2010). Independently, Davis and Burns (2011a) developed schedulability analysis for FPZL, initially published as a technical report in April 2010 (Davis and Burns 2010b). The analysis given for FPZL by Davis and Burns (2011a) is applicable to constrained-deadline tasksets with no restrictions on the priority ordering which may be used; whereas the analysis given for RMZL by Kato et al. (2010) 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 Davis and Burns (2011a) 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-emptions / migrations.

For example, the LLREF scheduling algorithm (Cho et al. 2006), which is optimal for periodic tasksets with implicit deadlines, and the LRE-TL scheduling algorithm

(Funk and Nadadur 2009) which is optimal for sporadic tasksets with implicit deadlines, divide the timeline into intervals that start and end at task releases and deadlines (referred to as TL-planes by Cho et al. 2006). In each interval, LLREF and LRE-TL ensure that each active task τ_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 FPSL, FPCL, and FPZL (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.

1.3 Organisation

The remainder of the paper is organised as follows: Sect. 2 describes the terminology, notation and system model used. Section 3 describes sufficient tests for global FP scheduling. These tests are used in Sect. 4 to derive polynomial time and pseudo-polynomial time sufficient schedulability tests for FPSL. These schedulability tests are a generalisation of the tests given by Davis and Burns (2011a) for FPZL, and also hold for FPCL. Section 4 also shows how the schedulability tests for FPSL can be improved by bounding the amount of execution that each task can perform once its priority has been promoted to the highest level. 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 on priority assignment. Section 7 presents an empirical investigation into the effectiveness of FPSL and FPZL and their associated 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, Sect. 9 concludes with a summary and suggestions for future research.

2 System model, terminology and notation

In this paper, we are interested in global FP, FPSL, 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 $(\tau_1 \ldots \tau_n)$, where each task τ_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 τ_i is characterised by its relative *deadline* D_i, *worst-case execution time* C_i $(C_i \leq D_i)$, 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.

Job parallelism, sometimes referred to as 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.

2.1 Global FP, FPZL, FPCL and FPSL scheduling algorithms

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

Under FPZL scheduling, if the laxity of a job reaches zero 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.

FPCL scheduling uses the concept of *critical laxity* (Kato and Yamasaki 2008) which can be described as follows: at each scheduling point, corresponding to job release or completion, if there are more than m ready jobs, then the laxity of each ready job is evaluated with respect to the maximum time that could potentially elapse until the next scheduling point. If at the next scheduling point, a jobs's laxity could be negative, then it is classified as being critical-laxity and has its priority promoted to the highest level immediately. The criterion used by FPCL to promote the priority of a job therefore depends on the dynamic properties of other jobs, for example their remaining execution times.

FPSL scheduling is similar to FPCL; however, it uses a static *laxity threshold.* If at a scheduling point, corresponding to job release or completion, the laxity of a job is less than the pre-computed laxity threshold for its task then the priority of the job is promoted to the highest level. Unlike FPCL, this criterion is independent of the dynamic properties of other jobs. The laxity threshold X_i for task τ_i is defined as the longest time that can elapse between one scheduling point and the next, while a job of task τ_i is ready but not executing.

Certain tasks may have their priority promoted as a result of the operation of the FPSL, FPCL or FPZL scheduling algorithms. In the remainder of this paper, we generically (i.e. independent of the algorithm used) refer to these tasks as *critical-laxity tasks*, as they are promoted to the highest priority level when the scheduling algorithm deems that their laxity has become critical to ensuring that deadlines are met. An upper bound on the maximum amount of execution that a job of task τ_i can perform in the *critical-laxity state*, i.e. at the highest priority, is denoted by K_i^{UB} and referred to as the task's *critical-laxity execution time*.

Under FPSL, FPCL, or FPZL at any given time, at most m tasks may be in the critical-laxity state without a deadline being missed.

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

Finally, when discussing the schedulability of a given task τ_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 τ_k.

3 Schedulability tests for global FP

In this section, we recapitulate two sufficient schedulability tests for global FP scheduling of sporadic tasksets. These tests are described in more detail by Davis and Burns (2010a).

3.1 Deadline analysis for global FP

Bertogna et al. (2009) developed a polynomial time sufficient schedulability test for global FP scheduling based on the approach of Baker (2003). They showed that if task τ_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 τ_i with a carry-in job is given by the following equation.[1] In global FP scheduling, a *carry-in job* is defined as a job that is released strictly prior to the start of the interval, and causes interference within that interval.

$$I_i^D(L, C_k) = \min\left(W_i^D(L), L - C_k + 1\right) \tag{1}$$

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

$$W_i^D(L) = N_i^D(L)C_i + \min\left(C_i, L + D_i - C_i - N_i^D(L)T_i\right) \tag{2}$$

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

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

Bertogna et al. (2009) used (1), with D_k as the length of the interval, and the strategy of Baker (2003) to form a schedulability test for each task τ_k:

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

$$D_k \geq C_k + \left\lfloor \frac{1}{m} \sum_{\forall i \in hp(k)} I_i^D(D_k, C_k) \right\rfloor \tag{4}$$

[1] Note we adopt the approach to time representation used by Bertogna et al. (2009). Time is represented by non-negative integer values, with each time value t viewed as representing the whole of the interval $[t, t+1)$. This enables mathematical induction on clock ticks and avoids confusion with respect to end points of execution.

where $hp(k)$ is the set of tasks with priorities higher than k. Note we have re-written (4) in a different form from that presented by Bertogna et al. (2009) for ease of comparison with the response time schedulability test given by Bertogna and Cirinei (2007).

Guan et al. (2009) showed that if task τ_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 τ_i *without* a carry in job is given by:

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

where:

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

and

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

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

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

Davis and Burns (2010a) 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 Guan et al. (2009) 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 τ_k in the taskset, the inequality given by* (9) *holds*:

$$D_k \geq C_k + \left\lfloor \frac{1}{m} \left(\sum_{\forall i \in hp(k)} I_i^{NC}(D_k, C_k) + \sum_{i \in MD(k, m-1)} I_i^{DIFF-D}(D_k, C_k) \right) \right\rfloor \tag{9}$$

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 $hp(k)$.

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 (2007) 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 Bertogna and Cirinei (2007), 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.

Bertogna and Cirinei (2007) showed that if task τ_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 τ_i with a carry-in job is given by:

$$I_i^{R}(L, C_k) = \min\left(W_i^{R}(L), L - C_k + 1\right) \tag{10}$$

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

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

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

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

The response time test of Bertogna and Cirinei (2007) may be expressed as follows:

RTA test for global FP scheduling (Theorem 7 from Bertogna and Cirinei 2007): *A sporadic taskset is schedulable, if for every task τ_k in the taskset, the upper bound response time R_k^{UB} computed via the fixed point iteration given by (13) is less than or equal to the task's deadline*:

$$R_k^{UB} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{\forall i \in hp(k)} I_i^R\left(R_k^{UB}, C_k\right) \right\rfloor \tag{13}$$

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 τ_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 $I_i^R(R_k^{UB})$.

Guan et al. (2009) 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. (2009) showed that if task τ_i does not have a carry-in job, then the interference term is given by (5). The difference between the two interference terms ((10) and (5)) is then given by:

$$I_i^{DIFF-R}(L, C_k) = I_i^R(L, C_k) - I_i^{NC}(L, C_k) \tag{14}$$

Using this result, Guan et al. (2009) improved upon the response time test of Bertogna and Cirinei (2007).

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

$$R_k^{UB} \leftarrow C_k + \left\lfloor \frac{1}{m} \left(\sum_{\forall i \in hp(k)} I_i^{NC}\left(R_k^{UB}, C_k\right) + \sum_{i \in MR(k,m-1)} I_i^{DIFF-R}\left(R_k^{UB}, C_k\right) \right) \right\rfloor \tag{15}$$

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 (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 τ_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 FPSL

In this section, we derive polynomial time and pseudo-polynomial time sufficient schedulability tests for FPSL and show that they apply directly to FPZL. (In Sect. 5 we show that these schedulability tests are also applicable to FPCL).

The tests derived in this section 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 state.

With FPZL, each job of a critical-laxity task has its priority promoted when its laxity reaches zero. In contrast, under FPSL, a job of a critical-laxity task τ_i has its priority promoted when a scheduling point occurs and the laxity of the job is less than or equal to the task's *laxity threshold* X_i. For FPSL to operate correctly, the laxity threshold for each critical-laxity task must be set to a value such that jobs of the task are guaranteed to have their priority promoted before their laxity $x_i(t)$ becomes negative, despite the fact that priority promotion can only take place at scheduling points corresponding to the release or completion of some job. Note that only jobs of tasks classified by the schedulability analysis as critical-laxity tasks can have their priority promoted in this way by FPSL.

Smaller values for the laxity threshold of a task are beneficial in terms of the impact that task has on the schedulability of other tasks. This is because a smaller laxity threshold implies that jobs of the task will spend less time executing in the critical-laxity state (i.e. at the highest priority). We therefore aim to set the laxity threshold X_i of each critical-laxity task τ_i to the smallest possible value such that all jobs of the task are guaranteed to have their priority promoted before their laxity becomes negative, despite the fact that priority promotion can only take place at scheduling points given by the release or completion of some job.

As a job of a critical-laxity task τ_i can have its priority promoted on its own release, then an upper bound on the laxity threshold of the task is given by:

$$D_i - C_i \tag{16}$$

If $X_i \geq D_i - C_i$, then each job of the task enters the critical-laxity state as soon as it is released.

Further, the maximum time that can occur between scheduling points while a job of task τ_i is ready but not executing, and therefore its laxity is reducing, is bounded by the maximum time for which other tasks can execute in preference to task τ_i before one of them completes. This is given by $MC(m, hp(i), lpcl(i))$ where $MC(m, hp(i), lpcl(i))$ returns the mth 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. Hence a further upper bound on the laxity threshold of task τ_i is given by:

$$MC\big(m, hp(i), lpcl(i)\big) \tag{17}$$

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

$$X_i = \min\big(D_i - C_i, MC\big(m, hp(i), lpcl(i)\big)\big) \tag{18}$$

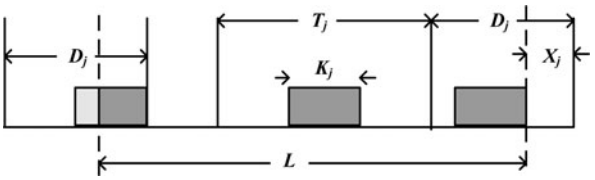


Fig. 1 Interference due to a lower priority critical laxity task

This value for the laxity threshold ensures that jobs of task τ_i are guaranteed to have their priority promoted before their laxity $x_i(t)$ becomes negative.

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 Sect. 4.3.

We now derive polynomial time and pseudo-polynomial time sufficient schedulability tests for FPSL. These schedulability tests are a generalization of the tests given for FPZL by Davis and Burns (2011a). With FPZL, there are additional scheduling points whenever the laxity of a task reaches zero. Hence with FPZL, the laxity threshold X_i of each task τ_i is effectively set to zero. Setting the laxity threshold of all tasks to zero in a schedulability test for FPSL provides an equivalent schedulability test for FPZL.

4.1 Deadline analysis for FPSL

Schedulability under FPSL is similar to that under FPZL:

1. Up to m tasks may be deemed unschedulable without priority promotion according to analysis of their response times; and yet, due to priority promotion no jobs will 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 τ_k, in an interval of length L, that could potentially be caused by a lower priority task τ_j executing for at most K_j in the critical-laxity state.

Figure 1 illustrates the worst-case scenario. This occurs when the last job of τ_j in the interval starts executing in the critical-laxity state as early as possible, and completes at the end of the interval, at a time X_j prior to its deadline. Further, each previous job of task τ_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 τ_j in the interval later.

An upper bound on the amount of workload due to task τ_j in the critical-laxity state, in an interval of length L is given by:

$$W_j^{CL}(L) = \begin{cases} \min\left(L, K_j^{UB}\right) & L \leq T_j - X_j \\ K_j^{UB} + N_j^{CL}(L) K_j^{UB} & \\ \quad + \min\left(K_j^{UB}, L - T_j + X_j - N_j^{CL}(L) T_j\right) & L > T_j - X_j \end{cases} \tag{19}$$

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

$$N_j^{CL}(L) = \lfloor (L - T_j + X_j)/T_j \rfloor \tag{20}$$

and K_j^{UB} ($\leq C_j$) is an upper bound on the amount of execution that any job of task τ_j can perform in the critical-laxity state.

If task τ_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 τ_j executing in the critical-laxity state is given by:

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

With FPZL, FPCL and FPSL, critical-laxity tasks may have the priority of their jobs promoted to the highest priority level, because of this we need to refine the definition of a *carry-in* job previously used in global FP scheduling.

In the context of FPZL, FPCL, and FPSL scheduling, a carry-in job is defined as a job that is released prior to the start of the interval and starts to execute at a higher priority than the task of interest τ_k, strictly before the start of the interval. (Note this definition also holds for global FP scheduling where jobs do not change priority).

Now if we pessimistically assume that jobs of lower priority critical-laxity tasks can have their priority promoted as early as possible, with as much execution time remaining as possible, independent of the execution pattern of other tasks, then their execution at the highest priority can be modelled as if it were simply the execution of a high priority task under global FP scheduling. Hence, with these pessimistic assumptions, the proof that the worst-case scenario occurs when there are at most $m - 1$ carry-in jobs given by Davis and Burns (2010a) also holds for FPZL, FPCL, and FPSL.

We now show that the interference from task τ_j executing in the critical-laxity state can be maximised without it being necessary to consider τ_j as having a carry-in job. The scenario that maximises interference within an interval is shown in Fig. 1. In this scenario, task τ_j has a job that is released prior to the start of the interval; however, without any reduction in interference in the interval, the first job of task τ_j can be assumed to have had its priority promoted at the earliest at the start of the interval, but not before. Hence the first job of τ_j need not be in the critical-laxity state prior to the release of the problem job at the start of the interval. Task τ_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-D} terms—see (8)). (Effectively, the first job of τ_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 τ_i. In this case, the maximum interference with a carry-in job occurs when the first job of τ_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 Fig. 2 below.

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 (1). Similarly, execution in the critical-laxity state cannot increase the amount of interference from a higher priority task

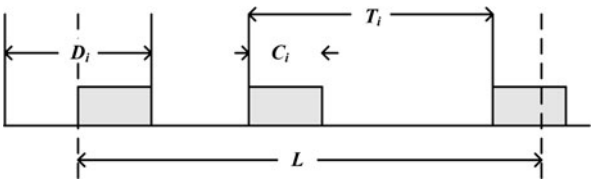


Fig. 2 Interference due to a higher priority critical laxity task

with no carry-in job, given by (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 FPSL, each task τ_k is therefore schedulable without requiring priority promotion if the following inequality holds.

$$D_k \geq C_k + \left\lfloor \frac{1}{m} \left(\sum_{\forall i \in hp(k)} I_i^{NC}(D_k, C_k) + \sum_{i \in MD(k,m-1)} I_i^{DIFF-D}(D_k, C_k) + \sum_{\forall j \in lpcl(k)} I_j^{CL}(D_k, C_k) \right) \right\rfloor \quad (22)$$

where $I_i^{NC}(D_k, C_k)$ is given by (5), $I_i^{DIFF-D}(D_k, C_k)$ is given by (8), $I_j^{CL}(D_k, C_k)$ is given by (21), and $lpcl(k)$ is the set of critical-laxity tasks with lower priorities than k.

If the inequality in (22) does not hold, then the task is a critical-laxity task. Under FPSL, 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 (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 FPSL is a polynomial time test requiring $O(n^2)$ operations, assuming that 'Compute K_k^{UB}' takes linear time.

The schedulability test for FPSL given in Algorithm 1 reduces to the equivalent DA-LC schedulability test for FPZL by setting the laxity threshold X_j for every task equal to zero in (18), (19) and (20).

As (19) is monotonically non-decreasing in X_j, then, for any interval length L, (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 FPSL, which in turn dominates the DA-LC test for global FP scheduling.

4.2 Response time analysis for FPSL

In this section, we provide a response time test for FPSL. This sufficient schedulability test is a generalization of the equivalent test for FPZL given by Davis and Burns (2011a). It reduces to that test for FPZL by setting the laxity threshold X_i for each task τ_i to zero.

Algorithm 1 DA-LC schedulability test for FPSL

```
1   countCL = 0
2   for (each priority level k, lowest first) {
3       Determine schedulability of τ_k according to (22)
4       if (τ_k is not schedulable without priority promotion) {
5           mark τ_k as a 'critical laxity' task
6           countCL = countCL + 1
7           Calc X_k according to (18)
8           Compute K_k^UB
9       }
10  }
11  if (countCL > m)
12      return unschedulable
13  else
14      return schedulable
```

The response time test for FPSL builds on the work of Bertogna and Cirinei (2007) and Guan et al. (2009) (i.e. (14)). It computes an upper bound R_k^{UB} on the response time of each task τ_k. If task τ_k is schedulable under FPSL 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 τ_j executing in the critical-laxity state can be obtained by substituting R_k^{UB} for the length of the interval in (21).

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

$$R_k^{UB} \leftarrow C_k + \left\lfloor \frac{1}{m} \left(\sum_{\forall i \in hp(k)} I_i^{NC}\left(R_k^{UB}, C_k\right) + \sum_{i \in MR(k,m-1)} I_i^{DIFF-R}\left(R_k^{UB}, C_k\right) + \sum_{\forall j \in lpcl(k)} I_j^{CL}\left(R_k^{UB}, C_k\right) \right) \right\rfloor \tag{23}$$

where $I_i^{NC}(R_k^{UB}, C_k)$ is given by (5), $I_i^{DIFF-R}(R_k^{UB}, C_k)$ is given by (14), $I_j^{CL}(D_k, C_k)$ is given by (21), and $lpcl(k)$ 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 τ_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 FPSL at most m tasks may be critical-laxity tasks without a deadline being missed.

Using (23), we can construct a sufficient schedulability test for FPSL 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.

Algorithm 2 RTA-LC schedulability test for FPSL

```
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 (23)
8              if (R_k^UB > D_k) {
9                    R_k^UB = D_k
10                   Calc X_k according to (18)
11                   Compute K_k^UB
12                   if (τ_k not marked as a CL task) {
13                         mark τ_k as a CL task
14                         repeat = true
15                         countCL = countCL + 1
16                         if(countCL > m) {
17                               repeat = false
18                               break (exit for loop)
19                         }
20                   }
21             }
22             [if (R_k^UB or K_k^UB differ from prev. values)
23                   repeat = true]
24       }
25 }
26 if (countCL > m)
27       return unschedulable
28 else
29       return schedulable
```

Under FPSL, the interference term (21) due to each lower priority critical-laxity task τ_j depends via the parameter X_j (see (18)) on the tasks with priorities lower than j that are also critical-laxity tasks. The interference term due to each task τ_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 $lpcl(j)$, 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 thresholds and upper bound response times of all higher priority tasks must be re-calculated.

Algorithm 2 presents the RTA-LC schedulability test for FPSL. 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 τ_k is encountered where (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 FPSL, then this

is the correct upper bound response time, as priority promotion will prevent the task from actually missing its deadline.

The discovery of a critical-laxity task effectively invalidates the upper bound response times calculated for all higher priority tasks, and also the laxity thresholds (X_j) for all higher priority critical-laxity tasks. These values could be too small, and therefore need to be re-calculated (line 14). However, if more than m critical-laxity tasks have been found, then priority promotion 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 22–23 are not required when a simple fixed value of $K_k^{UB} = C_k$ is used for the critical-laxity execution time of task τ_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 Sect. 4.3.

We note that the upper bound response time for a task τ_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 deemed unschedulable, or there are m or fewer critical-laxity tasks and the upper bound response times and laxity thresholds (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 $K_k^{UB} = C_k$, the RTA-LC schedulability test for FPSL requires $O(mn)$ response time calculations (i.e. (23)), each of which is pseudo-polynomial in complexity. This can be seen by noting that when 'Compute K_k^{UB}' sets $K_k^{UB} = C_k$, lines 22–23 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 FPSL given in Algorithm 2 reduces to the equivalent RTA-LC schedulability test for FPZL by simply setting the laxity threshold X_j for every task equal to zero in (18), (19) and (20).

As (19) is monotonically non-decreasing in X_j, then, for any interval length L, (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 RTA-LC test for FPZL dominates the RTA-LC test for FPSL, which in turn dominates the RTA-LC test for global FP scheduling.

4.3 Bounding critical-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 FPSL. 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 τ_k given by (22) and (23) are DC-Sustainable. A schedulability test for task τ_k is referred to as *DC-Sustainable* if it is sustainable (Baruah and Burns 2006) 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 τ_k is *DC-Sustainable* if the following two conditions hold:

Condition 1 If task τ_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 τ_k schedulable for all deadline and execution time pairs $D'_k = D_k - w, C'_k = C_k - w$ where $v \leq w \leq C_k$.

Condition 2 If task τ_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 τ_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 laxity thresholds* (X_j) *and a fixed set of critical-laxity tasks*, (22) *is a DC-Sustainable schedulability test for task* τ_k.

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

$$D'_k - C'_k \geq + \left\lfloor \frac{1}{m} \left(\sum_{\forall i \in hp(k)} I_i^{NC}\left(D'_k, C'_k\right) + \sum_{i \in MD(k,m-1)} I_i^{DIFF-D}\left(D'_k, C'_k\right) + \sum_{\forall j \in lpcl(k)} I_j^{CL}\left(D'_k, C'_k\right) \right) \right\rfloor \tag{24}$$

Consider the behaviour of (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 (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 $D'_k - C'_k + 1$. This expression is unchanged by varying w. The RHS of (24) is therefore monotonically non-increasing with respect to increasing values of w.

In the case of Condition 1, as the LHS of (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 (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 (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 (24) does not hold for $w = v$, then it cannot hold for any value of $w: 0 \le w \le v$. □

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

$$R_k^{q+1} \leftarrow C'_k + \left\lfloor \frac{1}{m} \left(\sum_{\forall i \in hp(k)} I_i^{NC}\left(R_k^q, C'_k\right) + \sum_{i \in MR(k,m-1)} I_i^{DIFF-R}\left(R_k^q, C'_k\right) + \sum_{\forall j \in lpcl(k)} I_j^{CL}\left(R_k^q, C'_k\right) \right) \right\rfloor \tag{25}$$

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

Let $R_k^{UB}(D, C)$ be the response time upper bound given by (25) for task $\tau_k(D, C)$ with deadline D and execution time C. Similarly, let $R_k^{UB}(D + x, C + x)$ be the response time upper bound given by (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 laxity thresholds (X_j) and a fixed set of critical-laxity tasks, if $\tau_k(D, C)$ is schedulable according to (25) then $R_k^{UB}(C + x) \ge R_k^{UB}(C) + x$. Further, if $\tau_k(D, C)$ is not schedulable according to (25) then neither is $\tau_k(D + x, C + x)$.*

Proof Let $R_k^q(C)$ be the value computed by the qth iteration of (25) for task $\tau_k(D, C)$. Similarly, let $R_k^q(C + x)$ be the value computed by the qth iteration of (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, $R_k^q(C + x) \ge R_k^q(C) + x$.

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

Inductive step: assume that $R_k^q(C + x) \ge R_k^q(C) + x$, and consider the values computed for $R_k^{q+1}(C + x)$ and $R_k^{q+1}(C)$ on iteration $q + 1$. The floor function (second term on the RHS of (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_k^q. 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 $R_k^q(C + x) \ge R_k^q(C) + x$, and $C'_k = C + x$ (the latter is used in (10) and (21)) than it is for input values $R_k^q(C)$ and $C'_k = C$, hence once the value of C'_k is added (first term on the RHS of (25)), we have $R_k^{q+1}(C + x) \ge R_k^{q+1}(C) + x$.

We note that if the fixed point iteration for $\tau_k(D, C)$ converges on $R_k^{UB}(D, C) = R_k^{q+1}(C)$, then the smallest possible value of $R_k^{UB}(D + x, C + x)$ is $R_k^{q+1}(C) + x$.

Further, if $\tau_k(D, C)$ is unschedulable, then it follows that $R_k^{q+1}(C) > D$ which implies that $R_k^{q+1}(C + x) > D + x$ and therefore $\tau_k(D + x, C + x)$ must also be unschedulable. □

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

Proof We can choose an execution time of $C'_k = 0$ and a deadline of $D'_k = D_k - C_k$ for task τ_k. With these parameters, τ_k is deemed schedulable by (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 τ_k is unschedulable. Lemma 1 tells us that for all larger values of w, τ_k will also be unschedulable. Proof that Conditions 1 and 2 in the definition of DC-Sustainability hold follow directly from the observation that task schedulability is monotonically decreasing with respect to increasing values of w. □

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 FPSL, and that task τ_k has been identified as a critical-laxity task by (22) or (23). We know that task τ_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 τ_k is guaranteed to complete $C'_k = C_k - v$ 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 (22) and (23), each of these equations can be used as the basis of a binary search to determine the smallest value of v $(0 \le v \le C_k)$ such that task τ_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 $K_k^{UB} = v$ on the amount of time that a job of task τ_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 τ_k having zero execution time.

In the DA-LC test, a binary search based on (22) can be used to 'Compute K_k^{UB}' (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, laxity thresholds (X_j), and critical-laxity execution times (K_j^{UB}) of lower priority tasks which have already been computed.

In the RTA-LC test, a binary search based on (23) can also be used to 'Compute K_k^{UB}' (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, laxity thresholds, 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 laxity threshold X_j of a task τ_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_j^{UB} is monotonically non-decreasing with respect to the laxity threshold X_j.

4.4 Applicability of the FPSL schedulability tests to FPCL

In this section, we show that the schedulability tests derived for FPSL also hold FPCL.

Theorem 3 *Any taskset that is deemed schedulable according to the sufficient schedulability tests given in Sects.* 4.1 *or* 4.2 (*i.e. the DA-LC or RTA-LC tests for FPSL given by Algorithm* 1 *or Algorithm* 2) *assuming that all of a critical-laxity task's execution is in the critical laxity state*, *is also schedulable under FPCL.*

Proof To prove the theorem, we need only consider those tasks that are *not* identified as critical-laxity tasks by the schedulability test for FPSL. We refer to such tasks as *ordinary tasks*. We show that the jobs of ordinary tasks never become critical-laxity jobs under FPCL and so remain schedulable. The remaining tasks, which have been identified as critical-laxity tasks by the schedulability test for FPSL, must then be trivially schedulable under FPCL as it is able to guarantee the schedulability of up to m tasks via priority promotion.

The proof is by contradiction. We assume that τ is a taskset that is schedulable according to the sufficient test for FPSL but is not schedulable under FPCL. Further, let J be the first job of an ordinary task τ_k from τ that becomes a critical-laxity job under FPCL. We note that if there is no such job, then all of the jobs of all of the ordinary tasks must always meet their deadlines under FPCL which suffices to prove the theorem.

As J is the first job of an ordinary task to be selected by the FPCL algorithm for priority promotion, then prior to the time at which FPCL promotes the priority of job J, no jobs of any ordinary task can be in the critical-laxity state. For FPCL to promote the priority of job J, it must therefore be the case that the total interference that J is subject to from jobs of higher priority ordinary tasks executing at their normal priorities and from jobs of critical-laxity tasks exceeds that considered by the FPSL schedulability test. If this were not the case, then J would be schedulable without priority promotion, and so the FPCL algorithm would not increase its priority.

Interference from ordinary tasks: The interference that J is subject to from jobs of any other ordinary task executing at their normal priority cannot exceed that considered by the schedulability test for FPSL, as the test uses an upper bound on such interference.

Interference from critical-laxity tasks: Let τ_i be a critical-laxity task identified by the FPSL schedulability test. Recall that prior to J being selected by FPCL for priority promotion, no jobs of any ordinary task can be in the critical-laxity state. From the definition of the laxity threshold X_i, given by (18), this means that under FPCL, prior to J being selected for priority promotion, no job of τ_i can enter the critical-laxity state with more laxity than assumed by the schedulability test for FPSL. Hence, the interference that J is subject to due to jobs of τ_i cannot exceed that assumed by the FPSL schedulability test.

The total interference that J is subject to therefore cannot exceed that considered by the schedulability test for FPSL. As J is a job of an ordinary task, it must therefore be schedulable under FPCL without priority promotion, and so will not be selected for priority promotion by the FPCL scheduler. This contradicts the original assumption. As there is no such first job J of an ordinary task that becomes a critical-laxity job under FPCL, then taskset τ must also be schedulable under FPCL. □

We now extend Theorem 3 to the refined schedulability tests which make use of upper bounds on the amount of execution that can occur in the critical-laxity state. First, we prove the following Lemma.

Lemma 2 *Let τ be any taskset that is deemed schedulable according to the sufficient schedulability tests given in Sects. 4.1 or 4.2, (i.e. the DA-LC or RTA-LC tests for FPSL given by Algorithm 1 or Algorithm 2) using the upper bounds on the execution time in the critical-laxity state given in Sect. 4.3.*

Under the assumption that jobs of ordinary tasks do not enter the critical-laxity state, then no job of a critical-laxity task, belong to a taskset τ, enters the critical-laxity state under FPCL with more laxity or more remaining execution time than computed by the FPSL schedulability test.

Proof The proof is by contradiction. We assume that J is the first job of a critical-laxity task (τ_i) that enters the critical-laxity state under FPCL with more laxity ($> X_i$) or more remaining execution time ($> K_j^{UB}$) than computed by the schedulability test for FPSL.

As J is the first such job of a critical-laxity task, then from the assumption that jobs of ordinary tasks cannot become critical-laxity jobs under FPCL, and the definition of the laxity threshold X_i, given by (18), then priority promotion of job J by FPCL cannot take place when job J has more laxity than X_i.

The maximum possible execution time in the critical-laxity state is monotonically non-decreasing with respect to the laxity a job has when its priority is promoted. In the case of job J, a valid upper bound on its critical-laxity execution time is given by K_j^{UB}, the upper bound on the maximum critical-laxity execution time for a laxity of X_i computed by the FPSL schedulability test using the techniques described in Sect. 4.3. This is the case because; by the assumption in the Lemma none of the

ordinary tasks become critical-laxity tasks under FPCL, and the definition of job J ensures that prior to it entering the critical-laxity state all jobs of all other critical-laxity tasks comply with the assumptions of the FPSL schedulability test.

It follows that job J can only enter the critical-laxity state under FPCL with laxity and remaining execution time no greater than the values computed by the FPSL schedulability test. This contradicts the original assumption about job J, and so there can be no job of a critical-laxity task that enters the critical-laxity state under FPCL with more laxity or more remaining execution time than computed by the FPSL schedulability test. □

Theorem 4 *Any taskset τ that is deemed schedulable according to the sufficient schedulability tests given in Sects.* 4.1 *or* 4.2 (*i.e. the DA-LC or RTA-LC tests for FPSL*), *using the upper bounds on execution time in the critical-laxity state given in Sect.* 4.3, *is also schedulable under FPCL.*

Proof Proof follows the logic used in the proof of Theorem 3 to show that jobs of ordinary tasks cannot enter the critical-laxity state under FPCL. The only difference is that further consideration is needed regarding the interference from critical-laxity tasks (5^{th} paragraph of the proof) which is adapted as follows:

Interference from critical-laxity tasks: Let τ_i be a critical-laxity task identified by the FPSL schedulability test. By definition of job J, there can be no job of any ordinary task which has its priority promoted by FPCL before job J. Hence Lemma 2 applies, and there can also be no job of a critical-laxity task τ_i that enters the critical-laxity state with more laxity or more remaining execution time than computed by the FPSL schedulability test, prior to FPCL promoting the priority of job J. Hence, the interference that J is subject to due to jobs of τ_i cannot exceed that assumed by the FPSL schedulability test.

It then follows (6^{th} paragraph in the proof of Theorem 3), that there is no such first job J of an ordinary task that becomes a critical-laxity job under FPCL, and so taskset τ must also be schedulable under FPCL. □

Note, Theorems 3 and 4 do not claim that any taskset that is schedulable (i.e. according to some exact test) using FPSL with laxity thresholds X_i is also schedulable according to FPCL. Only that tasksets which are schedulable according to the sufficient schedulability tests for FPSL presented in this paper are also schedulable using FPCL.

5 Event-driven scheduling, FPZL, FPCL, and FPSL

In this section, we discuss the implementation of the FPZL, FPSL, and FPCL scheduling algorithms. We assume that the operating system already implements an event-driven global FP scheduler, we therefore discuss only the modifications required to support the new algorithms.

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 job 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 job release and completion. FPZL also requires the maintenance of a *laxity queue* of ready, but non-running jobs, ordered by increasing laxity. The laxity of the job 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 critical-laxity tasks, hence the laxity queue need only track the laxity of at most m jobs. On expiry of a zero-laxity event, the scheduler runs and promotes the priority of the job at the head of the laxity queue to the highest level.

In contrast to FPZL, FPSL requires no additional timer events / scheduling points, other than those provided by a standard global FP scheduler, i.e. at job release and completion. 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 laxity threshold 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 $O(m)$. FPSL reduces the number of scheduling points compared to FPZL. With FPSL, 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). The implementation of FPSL is highly efficient; however, it requires that the set of critical-laxity tasks and their laxity thresholds are known off-line. This is only possible for tasksets that are deemed schedulable by one of the schedulability tests given in Sect. 4.

By comparison with FPSL the implementation of FPCL is less efficient; however, it does not require prior knowledge of which tasks are critical-laxity tasks. Like FPZL, the performance of FPCL can therefore be explored via simulations and experimental implementations, without the constraint that all of the tasksets examined must to be deemed schedulable by a schedulability test.

The 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 of FPCL 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 Sect. 8.

6 Priority assignment

In this section, we briefly discuss priority assignment for FPSL, FPCL and FPZL. Davis and Burns (2009, 2010a) showed that priority assignment is a key factor in global FP scheduling. As FPSL, 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 FPSL 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 (Davis and Burns 2009, 2010a). When there are no critical-laxity tasks, FPSL reduces to global FP scheduling. In this case, the Optimal Priority Assignment (OPA) algorithm (Audsley 1991, 2001) 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 priority promotion, then the following question arises: Which task should be assigned to that priority level? For the purposes of the empirical evaluation in Sect. 7, we answered this question via a simple heuristic. We computed the critical-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 priority promotion.

7 Empirical investigation

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

[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.

7.1 Taskset parameter generation

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

- Task utilisations were generated using the UUnifast-Discard algorithm (Davis and Burns 2009), giving an unbiased distribution of task utilisations. 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 1 ms 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. 1–10 ms, 10–100 ms etc.), thus providing reasonable correspondence with real systems.
- Task execution times were set based on the utilisation and period selected: $C_i = U_i T_i$.
- To generate constrained-deadline tasksets, task deadlines were assigned according to a uniform random distribution, in the range $[C_i, T_i]$. For implicit-deadline tasksets, deadlines were set equal to periods.

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 FPSL uses precomputed laxity thresholds and relies on the identification of critical-laxity tasks at the schedulability analysis stage, because of this, it was not possible to simulate the behaviour of FPSL 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.

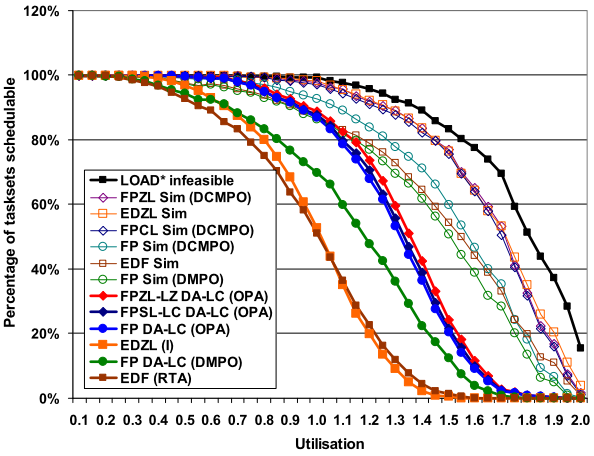


Fig. 3 2 processors, 10 tasks, $D \leq T$

7.3 Schedulability test effectiveness

We investigated the performance of the FPSL and FPZL DA-LC, schedulability tests using the OPA algorithm (Audsley 1991, 2001) to assign priorities, and compared their performance to that of the equivalent test for global FP scheduling, and to schedulability tests for global EDF by Bertogna et al. (2009) (the "EDF-RTA" test) and EDZL scheduling (Baker et al. 2008) (the "EDZL-I test"). Also shown on the graphs are results for the necessary infeasibility test of Baker and Cirinei (2006) (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) (Davis and Burns 2009, 2010a), which was found in the simulation studies to be significantly more effective than Deadline Monotonic Priority Ordering (DMPO). It was not possible to simulate optimal priority assignment as simulation of all possible priority orderings is intractable.

Figures 3, 4, 5 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 tasksets 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 tasksets deemed schedulable at a utilisation of approx. 4.9 ($= 0.61m$). As expected, this is slightly better than the DA-LC test for FPSL, again using Audsley's OPA algorithm and a binary search to bound critical-laxity execution time (marked FPSL-LC on the graph),

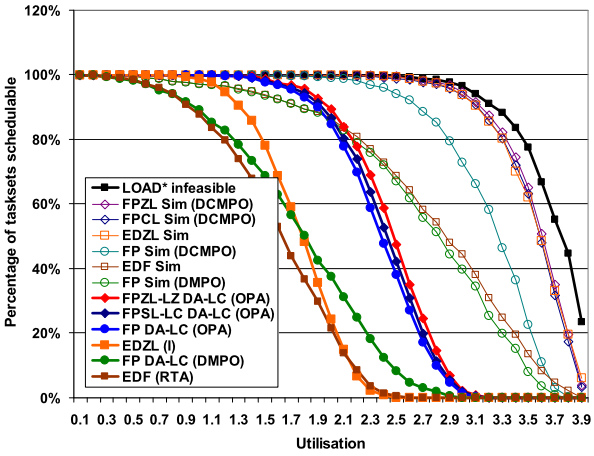


Fig. 4 4 processors, 20 tasks, $D \leq T$

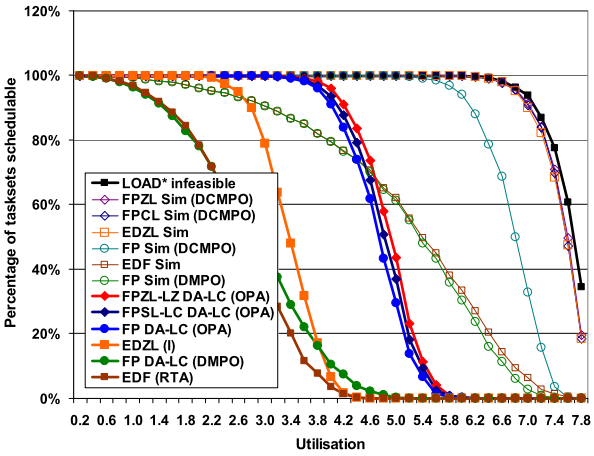


Fig. 5 8 processors, 40 tasks, $D \leq T$

with 50 % of tasksets deemed schedulable at a utilisation of approx. 4.8 ($= 0.60m$). Both FPSL 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.

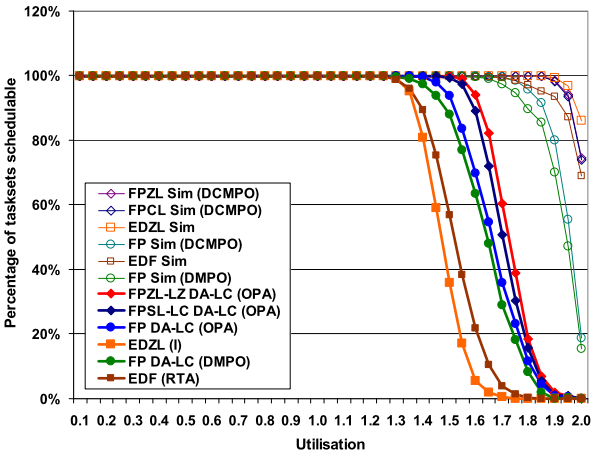


Fig. 6 2 processors, 10 tasks, $D = T$

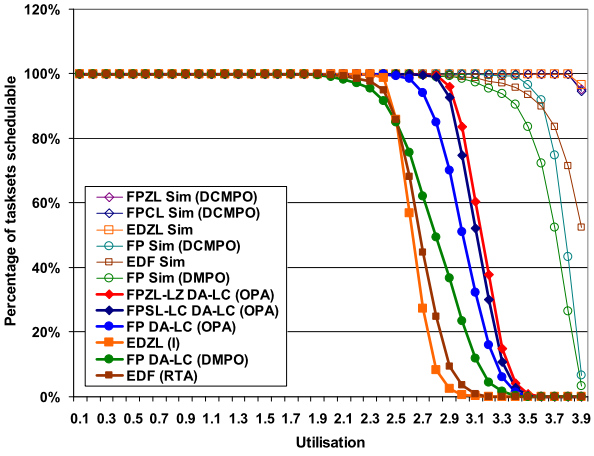


Fig. 7 4 processors, 20 tasks, $D = T$

Figures 6, 7, 8 show the results of the same experiments, repeated for implicit-deadline tasksets. These graphs show that the performance of the schedulability tests for FPSL and FPZL significantly exceed that of the best known tests for global FP, global EDF and EDZL, with an increased gap between both FPSL 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 FPSL (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 FPSL (and FPZL) is mainly due to the calculation of a less pessimistic bound on the amount of critical-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.

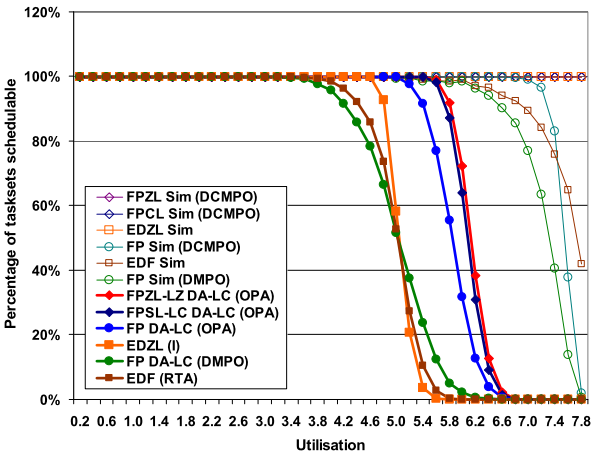


Fig. 8 8 processors, 40 tasks, $D = T$

Table 1 Approximate optimality degree

Algorithm / test	#Processors		
	2	4	8
FPZL LZ DA-LC (OPA)	84.7 %	79.4 %	77.4 %
FPSL CL DA-LC (OPA)	83.7 %	78.5 %	76.7 %
FP DA-LC (OPA)	81.4 %	75.7 %	73.6 %
FP DA-LC (DMPO)	80.1 %	70.0 %	62.8 %
EDZL(I)	71.7 %	66.2 %	63.5 %
EDF (RTA)	74.2 %	67.4 %	62.5 %

For implicit-deadline tasksets, we used our experimental results to obtain approximate values for the *Optimality Degree* (OD) (Bini and Buttazzo 2005) 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 (Funk and Nadadur 2009) 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, expressed as a percentage.

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 as-

signment 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 by Baker et al. (2008). By comparison, FPSL scheduling has an Optimality Degree that is approx. 1 % worse than FPZL and thus 2–3 % better than global FP scheduling.

We repeated our experiments for smaller numbers (2) and larger numbers (20) of tasks per processor and for a smaller range of task periods (with a factor of ten difference between the minimum and maximum possible period). In each case, although the data points changed, the relationships between the effectiveness of the different methods and the conclusions that can be drawn from them remained essentially the same. As the number of tasks per processor increased, we observed the following minor changes:

- The effectiveness of the schedulability tests for FPSL, FPZL and global FP scheduling increased, while the effectiveness of the schedulability tests for global EDF and EDZL declined.
- The potential performance of EDZL exceeded that of FPZL (DCMPO) by a small margin.

Further, as the range of task periods reduced, the performance potential of FPCL declined by a small margin compared to that of FPZL.

We note that it is possible to form more effective tests for EDZL, FPZL, global FP and global EDF by combining a number of existing sufficient tests, as was done by Bertogna (2009) for global EDF. In particular, we note that due to the dominance of EDZL over global EDF, any taskset deemed schedulable by a test for global EDF (such as EDF-RTA) is also guaranteed to be schedulable using EDZL. Similarly, a number of different sufficient tests for global FP scheduling could be used to show schedulability under FPZL or FPSL. In this paper, we have chosen to compare representative state-of-the-art tests for each specific scheduling algorithm rather than combinations of tests, which could potentially achieve yet higher performance.

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 `SCHED_FIFO` and `SCHED_RR`. The `SCHED_FIFO` policy does not pre-empt tasks executing at the same priority level, whereas the `SCHED_RR` policy defines a *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 `SCHED_FIFO` policy.

In our experience, even the tasks scheduled under the `SCHED_FIFO` policy may still be migrated on to different processors due to load balancing. To avoid such unexpected migrations, we force the `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 `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 `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. `syscall_run()` releases the first job of the task, and `syscall_wait_for_period()` generates a scheduling point for the Linux kernel. There is another interface, `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 (Beal et al. 2000; Calandrino et al. 2006; Faggioli et al. 2009; Oikawa and Rajkumar 1999; Srinivasan et al. 1998) provide a similar set of programming interfaces.

Our implementations of FPCL and global FP only dispatch new tasks in `syscall_run()` and `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 `cpus_allowed` mask for the dispatched task, and call the *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

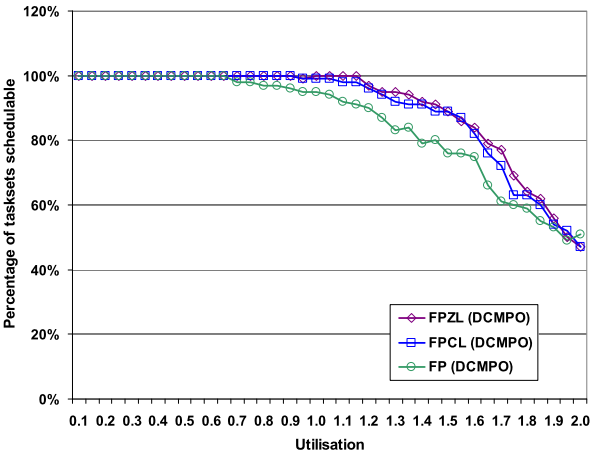
We now compare our implementations of FPCL, FPZL, and global FP scheduling, using a 2.0 GHz Intel Core 2 Quad processor (Q9650) with 2 GBytes of main memory. Since our goal is to evaluate implementation overheads in scheduling, rather than evaluating basic performance (such as kernel response times and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

Fig. 9 Sample code for user space tasks

```
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();
   }
}
```

Fig. 10 2 processors, 10 tasks, $D \leq T$



The implementation of the FPCL algorithm described in Sect. 5 was used. 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 Fig. 11 show the results of our experiments for constrained-deadline tasks, using the same basic taskset parameters as the simulations (Fig. 3 and Fig. 4 in Sect. 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.

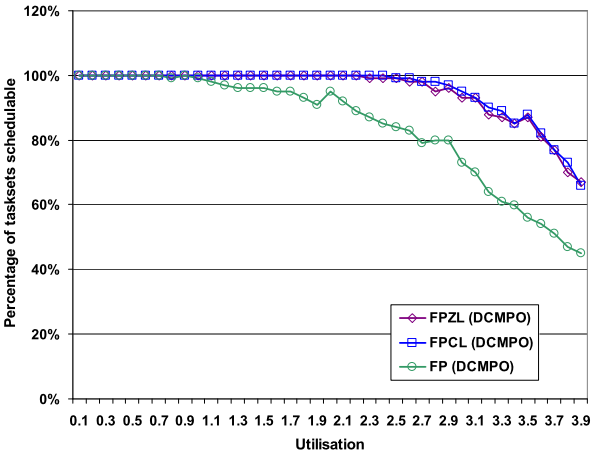


Fig. 11 4 processors, 20 tasks, $D \leq T$

Table 2 Average-case utilisation

Expt.	Utilisation	
	worst-case	average-case
$m = 4, n = 20, D \leq T$	1.4	1.32
	1.5	1.41
	1.6	1.51
	1.7	1.55
	1.8	1.61
	1.9	1.69

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. Using the minimum number of busy-wait loops that was found experimentally to produce the required WCET avoids execution time overruns; however, variability in the execution time of the loop, for example due to cache effects, then manifests itself in what are effectively execution time under-runs. This means 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.

On the Q9650 processor used for the experiments, there is variability in execution times, even for simple busy-wait loops, due to hardware effects (e.g. cache, bus conflicts etc.). An alternative approach to implementing the synthetic workload for each task would have been to monitor the amount of execution time actually used and continue to loop until close to the prescribed WCET. This approach would result in more consistent execution times; however, this would also be artificial, forcing the

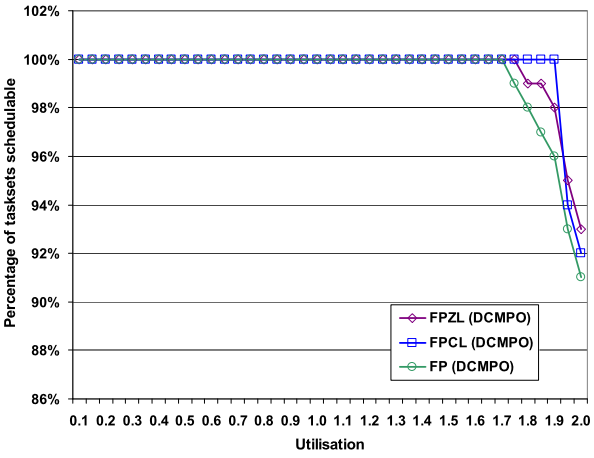


Fig. 12 2 processors, 10 tasks, $D = T$

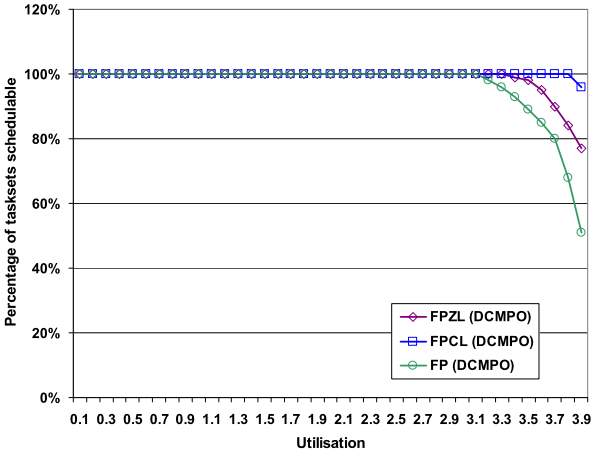


Fig. 13 4 processors, 20 tasks, $D = T$

implementation to behave more like a simulation. We preferred instead to use a more realistic approach where synthetic workloads are represented by simple busy-wait loops with their inherent execution time variability on this platform.

Despite these difficulties, the experimental results given in Fig. 10 and Fig. 11 provide a means of comparing the three algorithms.

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 critical-laxity job at its precise deadline; hence if execution times or scheduler invocation costs are underestimated, 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 somewhat before the deadline.

Figure 12 and Fig. 13 show the results for implicit-deadline tasks using the same basic taskset parameters as the simulations (Fig. 6 and Fig. 7 in Sect. 7.3). As mentioned previously, it is not straightforward to complete a critical-laxity job exactly at its deadline (as considered in theory) due to variations in execution times and scheduler invocation costs. This is, we believe, the main reason why FPCL outperformed FPZL in these experiments. While FPZL would be expected to perform better under a precise timing analysis, FPCL is an easier to implement and more robust scheduling algorithm.

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-emptions. This led us to consider minimally dynamic scheduling algorithms which permit each job to change priority at most once during its execution. We introduced three such algorithms, based on global FP scheduling, called FPSL, FPCL and FPZL. The number of context switches with FPZL is at most three per job for each critical-laxity task, and at most two per job for ordinary tasks. As there are at most m critical-laxity tasks, the increase in overheads compared to global FP scheduling is tightly bounded. With FPSL and FPCL task priorities only change at task release and completion events, thus the number of context switches is at most two per job.

The key contributions of this paper are as follows:

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

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

- The priority promotion 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 FPSL and FPZL result in a modest improvement over the equivalent test for global FP scheduling in the case of constrained-deadline tasksets, with an increased improvement for implicit-deadline tasksets.

- The schedulability tests for FPSL 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 critical-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 critical-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 critical-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 FPCL and FPZL can improve significantly upon the performance of global FP scheduling; however, FPCL is easier to implement and more robust that FPZL, when task execution times and scheduling overheads are subject to a small amount of uncertainty.

Finally, we note that semi-partitioned scheduling algorithms (Andersson et al. 2008; Burns et al. 2011; Guan et al. 2010; Kato and Yamasaki 2009a), 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. Comparisons of such methods with laxity-based global scheduling techniques, such as FPZL and EDZL, could potential improve our understanding of multiprocessor scheduling.

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

Andersson B, Jonsson J (2000) Some insights on fixed-priority pre-emptive non-partitioned multiprocessor scheduling. In: Proceedings real-time systems symposium (RTSS)—work-in-progress session

Andersson B, Bletsas K, Baruah SK (2008) Scheduling arbitrary-deadline sporadic tasks on multiprocessors. In: Proceedings real-time systems symposium (RTSS)

Audsley NC (1991) Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Dept Computer Science, University of York, UK, Dec 1991

Audsley NC (2001) On priority assignment in fixed priority scheduling. Inf Process Lett 79(1):39–44

Baker TP (2003) Multiprocessor EDF and deadline monotonic schedulability analysis. In: Proceedings real-time systems symposium (RTSS), pp 120–129

Baker TP (2006) An analysis of fixed-priority scheduling on a multiprocessor. Real-Time Syst 32(1–2):49–71

Baker TP, Baruah SK (2009) Sustainable multiprocessor scheduling of sporadic task systems. In: Proceedings Euromicro conference on real-time systems (ECRTS), pp 141–150

Baker TP, Cirinei M (2006) A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks. In: Proceedings real-time systems symposium (RTSS)—work-in-progress (WIP) session

Baker TP, Cirinei M, Bertogna M (2008) EDZL scheduling analysis. Real-Time Syst 40(3):264–289

Baruah SK (2007) Techniques for multiprocessor global schedulability analysis. In: Proceedings real-time systems symposium (RTSS), pp 119–128

Baruah SK, Baker TP (2009) An analysis of global EDF schedulability for arbitrary sporadic task systems. Real-Time Syst 43(1):3–24, ECRTS special issue

Baruah SK, Burns A (2006) Sustainable scheduling analysis. In: Proceedings real-time systems symposium (RTSS), pp 159–168

Baruah SK, Fisher N (2008) Global fixed-priority scheduling of arbitrary-deadline sporadic.... In: Proc of the 9th int'l conference on distributed computing and networking, pp 215–226

Baruah SK, Bonifaci V, Marchetti-Spaccamela A, Stiller S (2009) Implementation of a speedup-optimal global EDF schedulability test. In: Proceedings Euromicro conference on real-time systems (ECRTS), pp 259–268

Bastoni A, Brandenburg BB, Anderson JH (2010a) An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In: Proceedings real-time systems symposium (RTSS)

Bastoni A, Brandenburg B, Anderson J (2010b) Cache-related preemption and migration delays: empirical approximation and impact on schedulability. In: Proceedings of the sixth international workshop on operating systems platforms for embedded real-time applications (OSPERT 2010), July 2010, pp 33–44

Beal D, Bianchi E, Dozio L, Hughes S, Mantegazza P, Papacharalambous S (2000) RTAI: real time application interface. Linux J 29:10

Bertogna M (2007) Real-time scheduling analysis for multiprocessor platforms. PhD Thesis, Scuola Superiore Sant'Anna, Pisa

Bertogna M (2009) Evaluation of existing schedulability tests for global EDF. In: Proceedings of the first international workshop on real-time systems on multicore platforms: theory and practice

Bertogna M, Cirinei M (2007) Response time analysis for global scheduled symmetric multiprocessor platforms. In: Proceedings real-time systems symposium (RTSS), pp 149–158

Bertogna M, Cirinei M, Lipari G (2005) New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In: Proc 9th international conf on principles of distributed systems, pp 306–321

Bertogna M, Cirinei M, Lipari G (2009) Schedulability analysis of global scheduling algorithms on multiprocessor platforms. IEEE Trans Parallel Distrib Syst 20(4):553–566

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-Time Syst 30(1–2):129–154

Brandenburg BB, Calandrino JM, Anderson JH (2008) On the scalability of real-time scheduling algorithms on multicore platforms: a case study. In: Proceedings real-time systems symposium (RTSS), pp 157–169

Burns A, Davis RI, Wang P, Zhang F (2011) Partitioned EDF scheduling for multiprocessors using a $C = D$ scheme. Real-Time Syst 48(1):3–33

Calandrino J, Leontyev H, Block A, Devi U, Anderson J (2006) LITMUSRT: a testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings real-time systems symposium (RTSS), pp 111–123

Chao Y-H, Lin S-S, Lin K-J (2008) Schedulability issues for EDZL scheduling on real-time multiprocessor systems. Inf Process Lett 107(5):158–164

Cho S, Lee S-K, Han A, Lin K-J (2002) Efficient real-time scheduling algorithms for multiprocessor systems. IEICE Trans Commun E85-B(12):2859–2867

Cho H, Ravindran B, Jensen ED (2006) An optimal real-time scheduling algorithm for multiprocessors. In: Proceedings real-time systems symposium (RTSS), pp 101–110
Cirinei M, Baker TP (2007) EDZL scheduling analysis. In: Proceedings Euromicro conference on real-time systems (ECRTS), pp 9–18
Davis RI, Burns A (2009) Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In: Proceedings real-time systems symposium (RTSS), pp 398–409
Davis RI, Burns A (2010a) Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. Real-Time Syst 47(1):1–40. doi:10.1007/s11241-010-9106-5
Davis RI, Burns A (2010b) FPZL schedulability analysis. Technical Report YCS-2010-452, Dept of Computer Science, University of York, April 2010
Davis RI, Burns A (2011a) FPZL schedulability analysis. In: Proceedings real-time and embedded technology and applications symposium (RTAS), pp 245–256
Davis RI, Burns A (2011b) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv 43(4):35 doi:10.1145/1978802.1978814
Faggioli D, Trimarchi M, Checconi F (2009) An implementation of the earliest deadline first algorithm in Linux. In: Proceedings ACM symposium on applied computing, pp 1984–1989
Fisher N, Baruah SK (2006) Global static-priority scheduling of sporadic task systems on multiprocessor platforms. In: Proceedings IASTED international conference on parallel and distributed computing and systems
Funk S, Nadadur V (2009) LRE-TL: an optimal multiprocessor algorithm for sporadic task sets. In: Proceedings real-time and network systems (RTNS), pp 159–168
Guan N, Stigge M, Yi W, Yu G (2009) New response time bounds for fixed priority multiprocessor scheduling. In: Proceedings real-time systems symposium (RTSS), pp 387–397
Guan N, Stigge M, Yi W, Yu G (2010) Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound. In: Proceedings real-time and embedded technology and applications symposium (RTAS)
Kato S, Yamasaki N (2008) Global EDF-based scheduling with efficient priority promotion. In: Proceedings of real-time computing systems and applications (RTCSA), pp 197–206
Kato S, Yamasaki N (2009a) Semi-partitioned fixed-priority scheduling on multiprocessors. In: Proceedings real-time and embedded technology and applications symposium (RTAS), pp 23–32
Kato S, Yamasaki N (2009b) Real-time scheduling module for Linux kernel. IPSJ Trans Adv Comput Syst 2(1 (ACS25)):75–86 (in Japanese)
Kato S, Takeda A, Yamasaki N (2010) Global rate-monotonic scheduling with priority promotion. Technical Report CMU-ECE-TR10-05, May 2010
Lee SK (1994) On-line multiprocessor scheduling algorithms for real-time tasks. In: Proc IEEE region 10's ninth annual international conference, pp 607–611
Leung JY-T (1989) A new algorithm for scheduling periodic real-time tasks. Algorithmica 4:209–219
Oikawa S, Rajkumar R (1999) Portable RT: a portable resource kernel for guaranteed and enforced timing behaviour. In: Proceedings real-time and embedded technology and applications symposium (RTAS), pp 111–120
Park M, Han S, Kim H, Cho S, Cho Y (2005) Comparison of deadline-based scheduling algorithms for periodic real-time tasks on multiprocessor. IEICE Trans Inf Syst E88-D(3):658–661
Piao X, Han S, Kim H, Park M, Cho Y, Cho S (2006) Predictability of earliest deadline zero laxity algorithm for multiprocessor real time systems. In: Proc of the 9th IEEE international symposium on object and component-oriented real-time distributed computing, Gjeongju, Korea
Srinivasan B, Pather S, Hill R, Ansari F, Niehaus D (1998) A firm real-time system implementation using commercial off-the shelf hardware and free software. In: Proceedings real-time and embedded technology and applications symposium (RTAS), pp 112–119
Takeda A, Kato S, Yamasaki N (2009) Real-time scheduling based on rate monotonic for multiprocessors. IPSJ Trans Adv Comput Syst 2(1 (ACS25)):64–74 (in Japanese)

Robert I. Davis is a Senior Research Fellow in the Real-Time Systems Research Group at the University of York, and a Director of Rapita Systems Ltd. He received his DPhil in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial products. Robert's research interests include scheduling algorithms and schedulability analysis for real-time systems and networks.

Shinpei Kato is a Visiting Assistant Research Computer Scientist in the Department of Computer Science at University of California, Santa Cruz. He received his B.S., M.S., and Ph.D. degrees from Keio University in 2004, 2006, and 2008 respectively. He has also worked at The University of Tokyo and Carnegie Mellon University from 2009 to 2011 as a postdoctoral scientist. His research interests include real-time systems, operating systems, and parallel and distributed systems.