Robust Priority Assignment for Fixed Priority Real-Time Systems

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

Abstract

This paper focuses on priority assignment for real-time systems using fixed priority scheduling. It introduces and defines the concept of a "robust" priority ordering: the most appropriate priority ordering to use in a system subject to variable amounts of additional interference from sources such as interrupts, operating system overheads, exception handling, cycle stealing, and task execution time overruns. The paper describes a Robust Priority Assignment algorithm that can find the robust priority ordering for a wide range of fixed priority system models and additional interference functions. Proofs are given for a number of interesting theorems about robust priority assignment, and the circumstances under which a "Deadline minus Jitter" monotonic partial ordering forms part of the robust ordering. The paper shows that "Deadline minus Jitter" monotonic priority ordering is the robust priority ordering for a specific class of system, and that this property holds essentially independent of the additional interference function.

1. Introduction

1.1. Background and motivation

Fixed priority scheduling is used in a wide range of embedded real-time applications, from systems on spacecraft, to engine controllers and communications networks in automobiles, from industrial process control to digital set-top boxes, from medical systems to mobile phones, the list of applications using fixed priority scheduling is extensive and growing each year.

One of the most common problems faced by engineers involved in the development of fixed priority real-time systems is, how best to assign priorities so that the system will meet its time constraints.

Previous research into priority assignment has succeeded in providing answers to this question for a number of well defined, if somewhat restrictive, system models. Unfortunately, commercial real-time systems are seldom if ever fully compliant with the system models used in research. For example, tasks in real systems may be subject to *additional interference* of various types, for example:

- Effects of interrupts; interrupts occurring in bursts / at ill-defined rates, using more execution time than expected.
- Ill-defined Real-Time Operating System (RTOS) overheads.
- Tasks exceeding their expected execution times.
- Processor cycle stealing by peripheral control units such as Direct Memory Access (DMA) devices.
- Ill-defined critical sections where interrupts and hence task switches are disabled, possibly due to the behaviour of the RTOS.
- Errors occurring at an unpredictable rate, causing check-pointing mechanisms to re-run part or all of a task.

This paper considers systems subject to variable amounts of additional interference and seeks to find the most robust priority ordering to use.

1.2. Related work

Research into priority assignment policies for fixed priority scheduling on single-processor systems has mainly focussed on finding the *optimal* priority assignment policy or algorithm for restricted system models.

For a given system model, a priority assignment policy or algorithm is referred to as *optimal* if it provides a feasible priority ordering (resulting in a schedulable system) whenever such an ordering exists.

Work on priority assignment for fixed priority pre-emptive systems effectively began in 1967, when Fineberg and Serlin [1] considered priority assignment for two tasks. They noted that if the task with the shorter period is assigned the higher priority, then the least upper bound on the schedulable utilisation is $2(\sqrt{2}-1)$ or 82.8%. This result was generalised by both Serlin [2] in 1972 and Liu and Layland [3] in

1973, both of whom showed that for *synchronous* tasks (that share a common release time), that comply with a restrictive system model, and that have deadlines equal to their periods $(D_i = T_i)$, *Rate Monotonic*[1] priority ordering (RMPO) is optimal.

In 1982, Leung and Whitehead [4] showed that for synchronous tasks with deadlines less than or equal to their periods $(D_i \le T_i)$, but otherwise compliant with Liu and Layland's system model, *Deadline Monotonic*[2] priority ordering (DMPO) is optimal. They noted that for *asynchronous* tasks (that do not share a common release time), DMPO is not optimal.

More recently, Zuhily [9] confirmed that "*Deadline minus Jitter"* monotonic priority ordering (D-JMPO) is optimal for synchronous task sets with $D_i \le T_i$ and non-zero release jitter. We note that both DMPO and RMPO are special cases of D-JMPO.

In 1990, Lehoczky [5] showed that DMPO is not optimal for synchronous tasks with so called *arbitrary* deadlines, which may be greater than their periods $(D_i > T_i)$.

In 1991, Audsley [6] solved the problem of priority assignment for asynchronous task sets. Audsley's priority assignment algorithm is optimal in the sense that it finds a schedulable priority ordering if one exists. This algorithm is also applicable to systems where tasks have arbitrary deadlines.

In 1995, Davis and Burns [15] addressed the problem of assigning priorities to aperiodic tasks with firm deadlines. They provided an optimal priority assignment rule for inserting aperiodic tasks into the Deadline Monotonic priority ordering used for periodic tasks.

In 1996, George et al. [7] provided schedulability analysis for non-pre-emptive fixed priority scheduling. They showed that in the non-pre-emptive case, DMPO is no longer optimal for synchronous tasks with deadlines less than or equal to their periods $(D_i \le T_i)$. George et al. [7] showed that Audsley's optimal priority assignment algorithm is however applicable in this case.

In 2001, Audsley [8] showed how his original priority assignment algorithm could be adapted to also minimise the number of priority levels required.

In 2006, Bletsas and Audsley [10] showed that both Audsley's algorithm and DMPO remain optimal in the presence of blocking when resources are accessed according to the Stack Resource Policy (SRP) [12] developed by Baker from the Priority Ceiling Protocol (PCP) of Sha et al. [13].

[1] RMPO assigns priorities in order of task periods, such that the task with shortest period is given the highest priority.

[2] DMPO assigns priorities in order of task deadlines, such that the task with the shortest deadline is given the highest priority.

The Pseudo code for Audsley's algorithm is given below. For n tasks, the algorithm performs at most $n(n+1)/2$ schedulability tests and is guaranteed to find a schedulable priority assignment if one exists. This is a significant improvement compared to inspecting all $n!$ possible orderings. However, Audsley's algorithm does not specify the order in which tasks should be tried at each priority level. This order heavily influences the priority assignment chosen, if there is more than one ordering that is schedulable. Thus a poor choice of initial ordering can result in a priority assignment that leaves the system only just schedulable.

```
Optimal Priority Assignment Algorithm

for each priority level i, lowest first
{
   for each unassigned task τ
   {
       if τ is schedulable at priority i
       {
           assign τ to priority i
           break (continue outer loop)
       }
   }
    return unschedulable
}
return schedulable
```

Related research by Lehoczky et al. [24], Katcher et al. [25], Punnekkat et al. [26], and Regehr [27] used the *critical scaling factor*[3] as a metric for examining schedulability. In [27], Regehr explored the idea of a robust-optimal class of scheduling algorithms that maximise the critical scaling factor. Regehr showed that for tasksets where DMPO is optimal, it is also robust-optimal with respect to the critical scaling factor.

The research described in the rest of this paper was inspired by the need to provide appropriate advice on priority assignment to engineers developing complex commercial real-time systems for use in automotive systems and consumer electronics.

This paper builds upon previous research into priority assignment; it defines and explores a new concept of *robust* priority ordering; the most appropriate priority ordering to use in complex real-time systems which have a basic analysable system model, but are subject to all manner of additional interference which may impinge upon system schedulability.

1.3. Organisation

Section 2 describes the terminology, notation and system models used in the rest of the paper. Section 3 defines the concept of robust priority ordering. Section

[3] The critical scaling factor is the largest factor by which the execution time of every task can be increased and the system remain schedulable.

4 derives an algorithm which finds the most robust priority ordering. Section 5 illustrates the operation of the robust priority assignment (RPA) algorithm via examples of pre-emptive and non-pre-emptive task scheduling. In Sections 6 and 7 we consider "Deadline minus Jitter" monotonic priority ordering and examine the conditions under which it is the most robust partial or complete priority ordering. Finally, Section 8 summarises the key contributions of the paper and suggests directions for future research.

2. System model, terminology and notation

We are interested in the problem of priority assignment and scheduling for a real-time application executing on a single processor. The application is assumed to comprise a static set of *n* tasks, each assigned a unique priority *i*, from 1 to *n* (where *n* is the lowest priority), according to some priority assignment policy or algorithm.

We consider various fixed priority scheduling schemes. Scheduling may be *pre-emptive*, *non-pre-emptive* or *co-operative*. With pre-emptive scheduling, at any given time the ready task with the highest priority is executed. Thus the release of a high priority task may cause a low priority task to be pre-empted at any point during its execution. With non-pre-emptive scheduling, once a task has started executing, it continues to execute until completion[4]. At completion of a task, the highest priority ready task is allocated the processor. With co-operative scheduling, there is a limited form of pre-emption, with tasks offering pre-emption points within their execution via some form of reschedule call. At these reschedule points a switch may occur to a higher priority task.

Application tasks may arrive either *periodically* at fixed intervals of time, or *sporadically* after some minimum inter-arrival time has elapsed. Each task τ_i, is characterised by: its relative *deadline* D_i, *worst-case execution time* C_i, minimum inter-arrival time or *period* T_i, and *release jitter* J_i, defined as the maximum time between the task arriving and it being released (ready to execute). It is assumed that once a task starts to execute it will never voluntarily suspend itself.

Tasks may make mutually exclusive access to shared resources according to the Stack Resource Policy (SRP) [12]. A task at priority *i* may be blocked by a lower priority task, as a result of the operation of the SRP, for at most B_i, referred to as the *blocking time*.

A task's *worst-case response time* R_i, is the longest time from the task arriving to it completing execution. A task is referred to as *schedulable* if its worst-case response time is less than or equal to its deadline. A system is referred to as schedulable if all its tasks are schedulable. A priority assignment is said to be *feasible* if it leads to a schedulable system.

We consider various constraints on task deadlines: $D_i = T_i$, $D_i \le T_i$, and so called *arbitrary* deadlines where some tasks may have $D_i > T_i$. We also consider systems where task deadlines may be at some intermediate point during task execution, so called *deadlines prior to completion* [16].

A set of tasks is referred to as *synchronous* if the arrival times of the tasks are assumed to be independent and thus the tasks may share a common release time. A set of tasks is referred to as a*synchronous* if the arrival times of some of the tasks are related to each other via non-zero offsets and therefore the tasks may or may not share a common release time.

The term *transaction* [21] is used to describe a group of tasks with arrival times that are related by fixed offsets O_i. The start of a transaction is defined by the arrival time of the first task in the transaction. Thus the offset of the first task in a transaction is by definition zero, whilst the offsets of other tasks in the transaction are measured relative to the arrival of this task.

In this paper, we discuss a number of different *system models*. A system model is a combination of scheduling policies (for example fixed priority pre-emptive scheduling using the Stack Resource Policy for resource access) and a tasking model, describing the constraints on task attributes (for example $D_i \le T_i$, $J_i = 0$, no transactions / offset release times). For ease of reference, we will refer to the system model for which "Deadline minus Jitter" monotonic priority assignment is known to be optimal [9] as the *D-JM system model*.

3. Robust priority ordering

In this section, we define the concepts of *additional interference* and *robust priority ordering*. We first formalise the idea of additional interference before using it in the definition of robust priority ordering.

3.1. Additional interference

Our aim is to model additional interference in as general a way as possible, ensuring that our analysis is applicable to a wide range of sources of such interference, whilst also being able to derive interesting and useful results about systems that are subject to this interference. With that aim in mind, we assume that additional interference takes the form of a function $E(\alpha, w, i)$, where α is a scaling factor, used to model variability in the amount of interference, w is the

[4] Although tasks executing non-pre-emptively may not be pre-empted by higher priority tasks, typically, their execution may be interrupted and hence delayed by the execution of interrupt handlers.

length of the time interval over which the interference occurs and i is a priority level affected by the interference.

We are interested in systems whose schedulability is *sustainable* [19] with respect to the additional interference function. In other words, if the system is schedulable for a value of $\alpha = \alpha'$, it should also be schedulable for a value of $\alpha = \alpha'' \le \alpha'$. We require that $E(\alpha, w, i)$ is a monotonic non-decreasing function of its parameters. Hence for any fixed values of α and w, $E(\alpha, w, j) \ge E(\alpha, w, k)$ if and only if priority level j has a higher numeric value (i.e. a lower priority) than k. Similarly, if time interval $w' > w''$, then $E(\alpha, w', i) \ge E(\alpha, w'', i)$ for any fixed values of α and i and finally, if the scaling factor $\alpha' > \alpha''$, then $E(\alpha', w, i) \ge E(\alpha'', w, i)$ for any fixed values of w and i.

We note that these *monotonicity requirements* on $E(\alpha, w, i)$ represent little if any restriction in practice: α is a scaling factor and so by definition, $E(\alpha, w, i)$ can be formulated to be monotonically non-decreasing in α. Interference from just about any conceivable source is never less in a longer time interval than it is in a shorter one, and finally, interference affecting a high priority level typically also affects lower priority levels and so additional interference $E(\alpha, w, i)$ is naturally monotonically non-decreasing with respect to priority level.

As an example, consider a system subject to additional interference (i) from an interrupt handler of indeterminate duration that is activated at most every 100µS, and (ii) from an error recovery block that executes at priority j for a maximum duration of C_j^{REC}, at most every 10,000 µS. The additional interference function is as follows:

$$E(\alpha, w, i) = \alpha \left\lceil \frac{w}{100} \right\rceil + \begin{cases} if\,(i \ge j) & \left\lceil \frac{w}{10000} \right\rceil C_j^{REC} \\ else & 0 \end{cases}$$

Provided that they meet the monotonicity criteria, then significantly more complex additional interference functions can be accommodated by the analysis given in subsequent sections.

3.2. Optimal and robust priority assignment

Following the definitions given in the literature, an optimal priority assignment policy may be defined as follows:

Definition 1: *optimal priority assignment policy*: For a given system model, a priority assignment policy P is referred to as *optimal* if there are no systems, compliant with the system model, that are schedulable using another priority assignment policy that are not also schedulable using policy P.

Similarly, given an additional interference function $E(\alpha, w, i)$, we can define a robust priority assignment policy as follows:

Definition 2: *robust priority assignment policy*: For a given system model and additional interference function, a priority assignment policy P is referred to as *robust* if there are no systems, compliant with the system model, that are both schedulable and can tolerate additional interference characterized by a scaling factor α using another priority assignment policy Q that are not also both schedulable and can tolerate additional interference characterized by the same or larger scaling factor using priority assignment policy P.

Stated otherwise, using the robust priority ordering, a system can tolerate additional interference that is at least as great as the additional interference tolerated by the system when any other priority ordering is used.

4. Robust priority assignment algorithm

In this section, our focus is on fixed priority real-time systems that can be described by a system model that is analysable[5], but are subject to additional interference.

Examples of analysable systems include those using pre-emptive, non-pre-emptive or co-operative scheduling of a static set of tasks with bounded execution times. The tasks may be periodic, arriving at a well-defined rate, or sporadic with a defined minimum inter-arrival time. Tasks may make mutually exclusive access to shared resources according to the Stack Resource Policy; they may be grouped together into transactions, with non-zero offsets, they may have non-zero release jitter, arbitrary deadlines and deadlines prior to completion. Examples of additional interference were given in Section 1.1.

We now derive an algorithm that provides a robust priority assignment whenever such an ordering exists. This algorithm is based on Audsley's optimal priority assignment algorithm [6,8] and is applicable to any analysable fixed priority system model where the following holds:

Condition 1: The worst-case response time of a task is dependent on the set of higher priority tasks, but not on the relative priority ordering of those tasks.

Condition 2: The worst-case response time of a task may be dependent on the set of lower priority tasks, but not on the relative priority ordering of those tasks.

[5] By analysable, we mean that the worst-case response times of tasks can be computed either by response time analysis or by some other method such as construction of a schedule.

Condition 3: When the priorities of any two tasks are swapped, the worst-case response time of the task being assigned a higher priority cannot increase with respect to its previous value.

Condition 4: When the priorities of any two tasks are swapped, the worst-case response time of the task being assigned a lower priority cannot decrease with respect to its previous value.

We observe that as the additional interference function $E(\alpha, w, i)$ is monotonically non-decreasing in both priority i and time interval w, then for any system model where the four conditions stated above hold, then they also hold when the worst-case response times are increased due to additional interference (assuming a fixed value of α).

The Robust Priority Assignment algorithm determines a schedulable priority ordering P, for any system where such an ordering exists. Further, the algorithm computes the maximum additional interference represented by α_i^P that can be tolerated by each task under priority ordering P. The maximum additional interference that can be tolerated by the system as a whole is given by:

$$\alpha^P = \min_{\forall i}(\alpha_i^P) \tag{1}$$

The algorithm performs $n(n+1)/2$ binary searches to determine this priority ordering. The starting values for the binary search can be set as follows: lower limit: zero, upper limit: some reasonable value based on inspection of the interference function. This upper limit is then doubled on each iteration of the binary search, if found to be schedulable.

```
Robust Priority Assignment Algorithm

for each priority level i, lowest first
{
   for each unassigned task τ
   {
       binary search for the largest value
       of α for which task τ is schedulable
       at priority i
   }
   if no tasks are schedulable at priority i
       return unschedulable
   else
       assign the schedulable task that
       tolerates the max α at priority i to
       priority i
}
return schedulable
```

We note that an alternative structuring of the Robust Priority Assignment algorithm is possible with a binary search at the outermost level, effectively enclosing Audsley's algorithm. This alternative structure could be used to derive the maximum additional interference that can be tolerated by the system as a whole, however; it would not provide information on the amount of additional interference tolerated at each priority level. For that reason, we prefer the formulation presented above.

Theorem 1: The Robust Priority Assignment (RPA) algorithm is an *optimal priority assignment policy* (see Definition 1).

Proof: Follows directly from equivalence with Audsley's algorithm. (Equivalence with the RPA algorithm can be seen by noting that Audsley's algorithm is an optimal priority assignment policy, irrespective of the initial ordering of the tasks and therefore irrespective of which unassigned but schedulable task is assigned to each priority level) □

Theorem 2: The Robust Priority Assignment (RPA) algorithm is a *robust priority assignment policy* (see Definition 2).

Proof: We assume (for contradiction) that there is an alternative priority ordering Q, which tolerates greater additional interference than the priority ordering P found by the RPA algorithm; so $\alpha^Q > \alpha^P$. For the purposes of the proof, we will refer to this alternative priority ordering as Q_n. We will iteratively transform Q_n into $Q_{n-1} .. Q_1$, where Q_1 is the same ordering as P. The transformation will be such that $\alpha^{Q_{k-1}} \geq \alpha^{Q_k}$, thus proving the theorem via the contradiction: $\alpha^P \geq \alpha^Q$.

We use k as an iteration count and also the priority level that we will transform. Thus k counts down from an initial value of n to 1. We note that as a result of the transformations, the tasks at priority levels lower than k become the same in both Q_k and P, hence Q_1 and P represent the same priority ordering.

On iteration k, we transform priority ordering Q_k as follows: First we find the priority level i in Q_k of the task assigned to priority level k in P. We refer to this task as τ_k, as we intend to assign it to priority level k. Note that as the tasks of lower priority than k are the same in both Q_k and P, priority level i must be either higher than or equal to k.

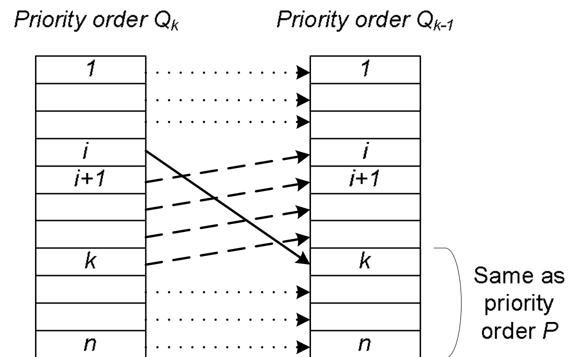

Figure 1: Transformation of priority order

There are two cases to consider:

1. Task τ_k is at priority k in both P and Q_k, in which case no transformation is required on this iteration, and so Q_{k-1} is identical to Q_k.
2. Task τ_k is at a higher priority i in Q_k. In this case, we form priority ordering Q_{k-1} by modifying Q_k as follows: Task τ_k is moved down in priority from priority level i to priority level k, and the tasks at priority levels i+1 to k are all moved up one priority level (see Figure 1).

We now introduce a concise notation to aid in the discussion of groups of tasks within a priority ordering:

hep(k,P) is the set of tasks with priority higher than or equal to *k* in priority ordering *P*.

hp(k,P) is the set of tasks with priority strictly higher than *k* in priority ordering *P*.

lp(k,P) is the set of tasks with priority strictly lower than *k* in priority ordering *P*.

Comparing the tasks in priority order Q_{k-1} with their counterparts in Q_k. There are effectively four groups of tasks to consider:

1. $hp(i,Q_{k-1})$: These tasks are assigned the same priorities in both Q_k and Q_{k-1} and so can tolerate the same additional interference.
2. $hp(k,Q_{k-1}) \cap lep(i,Q_{k-1})$: These tasks retain the same partial order but are shifted up one priority level in Q_{k-1} and so can tolerate at least as much additional interference as they can in Q_k.
3. Task τ_k, which is at priority level i in Q_k and at the lower priority level k in Q_{k-1}: We know, from the RPA algorithm, that τ_k can tolerate at least as much additional interference when at priority k as any of the tasks in $hep(k,P)$, when they are assigned priority k. Now $lp(k,Q_k) = lp(k,P)$ implies that $hep(k,Q_k) = hep(k,P)$, and so τ_k can tolerate at least as much additional interference at priority k as the task at priority k in Q_k.
4. $lp(k,Q_{k-1})$: These tasks are assigned the same priorities in both Q_k and Q_{k-1}, and as $hep(k,Q_{k-1}) = hep(k,Q_k)$, they are subject to interference from the same set of higher priority tasks, and so can tolerate the same additional interference in each case.

For every task in Q_{k-1}, the above analysis identifies a task in Q_k which does not have a greater tolerance to additional interference. Thus Q_{k-1} can tolerate at least as much additional interference as Q_k and so $\alpha^{Q_{k-1}} \geq \alpha^{Q_k}$.

A total of n iterations of the above procedure (for values of k from n down to 1) are sufficient to transform any arbitrary priority ordering Q into the priority ordering P, generated by the RPA algorithm. Further, this transformation is achieved without any reduction in the maximum amount of additional interference that the system can tolerate □

5. Examples of robust priority ordering

In this section, we provide examples of robust priority assignment in the presence of additional interference. Our first example considers tasks that are scheduled non-pre-emptively, whilst the second example, considers tasks with arbitrary deadlines, scheduled pre-emptively. In each case, the systems are subject to additional interference due to an interrupt handler executing for an indeterminate duration.

5.1. Example 1: Non-pre-emptive tasks

This example considers robust priority assignment for tasks scheduled non-pre-emptively according to fixed priorities. The response times of non-pre-emptive tasks can be found via response time analysis; potentially this requires examining multiple invocations of the tasks within the worst-case busy period [7, 23].

Table 1: Task parameters

Task	C	T	D
τ_A	125	450	450
τ_B	125	550	550
τ_C	65	600	600
τ_D	125	1000	1000
τ_E	125	2000	2000

The example system comprises 5 tasks, the parameters of which are given in Table 1. Note here we use *A*, *B*, *C* etc. to distinguish the tasks irrespective of the priority levels to which they are assigned. The tasks are arranged in the table in DMPO, and are schedulable in this priority order with response times of 250, 375, 440, 565, and 565 respectively, assuming no additional interference. We note however, that as illustrated in [7], DMPO is not optimal for fixed priority non-pre-emptive scheduling.

In this example, we assume that the system is subject to additional interference from an interrupt that occurs infrequently (at most once during any task busy period), causing an interrupt handler to execute for an indeterminate amount of time. The additional interference function is therefore simply:

$$E(\alpha, w, i) = \alpha$$

where α represents the time for which the interrupt handler executes.

We now use the RPA algorithm to find a robust priority ordering for the tasks in Table 1. Recall that for each priority level, lowest first, the RPA algorithm selects the unassigned task that tolerates the most additional interference at that priority level. For each

priority level, the values of α computed[6] by the RPA algorithm are given in Table 2. The maximum value of α at each priority is highlighted in bold, indicating that the task is subsequently assigned to that priority level. Entries in the table marked as 'NS' mean that the task was not schedulable at that priority even with no additional interference. Entries in the table marked '-' indicate that no value was computed by the algorithm, as the task had already been assigned a lower priority.

Table 2: Computed values of α

Priority	Task				
	τ_A	τ_B	τ_C	τ_D	τ_E
5	NS	NS	NS	120	**354**
4	NS	NS	NS	**120**	-
3	10	**110**	74	-	-
2	135	-	**199**	-	-
1	**200**	-	-	-	-

The robust priority ordering for this example is ($\tau_A, \tau_C, \tau_B, \tau_D, \tau_E$). With this priority ordering, the system can tolerate infrequent interrupts that delay task execution by at most 110 time units. By comparison, DMPO ($\tau_A, \tau_B, \tau_C, \tau_D, \tau_E$) yields values of α of (200, 175, 74, 120, 354) respectively; hence using DMPO, the system can tolerate infrequent interrupts that delay task execution by at most 74 time units.

This example shows how the RPA algorithm determines a robust priority ordering for tasks scheduled non-pre-emptively according to fixed priorities. Further it illustrates that DMPO is not necessarily the most robust priority ordering to use for non-pre-emptive tasks.

5.2. Example 2: Pre-emptive tasks

Our second example considers pre-emptive tasks with arbitrary deadlines. The response times of arbitrary deadline tasks can be found via response time analysis [5, 18], again potentially examining multiple invocations of a task within the worst-case busy period.

The example system comprises two tasks, τ_A and τ_B, with the parameters[7] given in Table 3.

Table 3: Task parameters (D>T)

Task	*C*	*D*	*T*
τ_A	42	118	100
τ_B	52	154	140

We note that with no additional interference, the system is schedulable with either τ_A or τ_B at the higher priority.

Case 1: Additional interference of the form:

$$E(\alpha, w, i) = \alpha\left\lceil \frac{w}{100} \right\rceil$$

For example, from a interrupt handler of indeterminate duration, executing every 100 time units.

With task τ_A at the higher priority, and τ_B at the lower priority, τ_A and τ_B can tolerate maximum additional interference given by $\alpha = 58$ and $\alpha = 9$ respectively. Alternatively, with τ_B at the higher priority, and τ_A at the lower priority, τ_B and τ_A can tolerate maximum additional interference given by $\alpha = 51$ and $\alpha = 10$ respectively. Hence (τ_B, τ_A) is the robust priority ordering in this case, tolerating additional interference characterised by $\alpha = 10$.

Case 2: Additional interference of the form:

$$E(\alpha, w, i) = \alpha\left\lceil \frac{w}{200} \right\rceil$$

For example, from a interrupt handler of indeterminate duration, executing every 200 time units.

With task τ_A at the higher priority, and τ_B at the lower priority, τ_A and τ_B can tolerate maximum additional interference given by $\alpha = 76$ and $\alpha = 18$ respectively. Alternatively, with τ_B at the higher priority, and τ_A at the lower priority, τ_B and τ_A can tolerate maximum additional interference given by $\alpha = 96$ and $\alpha = 15$ respectively. Hence (τ_A, τ_B) is the robust priority ordering in this case, tolerating additional interference characterised by $\alpha = 18$.

Case 3: Additional interference of the form:

$$E(\alpha, w, i) = \alpha\left(\left\lceil \frac{w}{100} \right\rceil K + \left\lceil \frac{w}{200} \right\rceil L\right)$$

Here the additional interference is from two sources, both of which cause interference of indeterminate duration. We use the (unknown) values *K* and *L* to describe the relative duration of the interference from these two sources. In this case, the form of the additional interference function is not well defined and it is in fact impossible to determine the robust priority ordering without further information about the values of *K* and *L*. If *K* = 1 and *L* = 0, then this is equivalent to case 1 and the robust priority ordering is (τ_B, τ_A), however, if *K* = 0 and *L* = 1, then this is equivalent to case 2 and so the robust priority ordering is (τ_A, τ_B).

Case 3 shows that the robust priority ordering is in general dependent on the form of the additional interference function. This is a significant but somewhat unfortunate result. It means that for general system models (such as the arbitrary deadline case examined here), it is only possible to determine the robust priority ordering if the form of the additional interference function is well defined, in other words, there are no unknowns save for the maximum value of α that the system can tolerate. We return to this point

[6] Using a granularity of 1 time unit.

[7] These parameters were chosen based on the example in [5].

in Section 7.

6. "D-J" monotonic partial ordering

In this section, we consider fixed priority systems where the tasks can be classified into two subsets:

1. Tasks that comply with the conditions under which "Deadline minus Jitter" monotonic priority ordering (D-JMPO) is known to be optimal, i.e. they are scheduled pre-emptively, have deadlines less than or equal to their periods, no offsets with respect to each other, and so on. We refer to these tasks as *D-JM system model tasks*.
2. Tasks that do not comply with the D-JM system model. These tasks may execute non-pre-emptively; they may be part of a transaction and thus have offset arrival times with respect to other tasks in the same transaction; they may have deadlines greater than their periods, deadlines prior to completion and so on. We refer to such tasks as *non D-JM system model tasks*.

We require that the maximum interference, on lower priority D-JM system model tasks, caused by the execution of the non D-JM system model tasks is monotonic in both time interval and priority. Thus non D-JM system model tasks are permitted to have offset arrival times with respect to each other, but these arrival times must be *independent* of the arrival times of the D-JM system model tasks.

We refer to systems containing the two classes of task as *mixed* systems. In general, a mixed system contains *n* tasks in total, *m* of which comply with the D-JM system model and *n-m* tasks which do not.

We assume that all of the tasks may make mutually exclusive access to shared resources according to the Stack Resource Policy. We note that blocking caused by non-pre-emptive execution may be viewed as a special case of the Stack Resource Policy where the ceiling priority is set to the highest priority in the system and the resource is effectively locked for the entire duration of the non-pre-emptive task. The blocking caused by tasks that are scheduled co-operatively, offering pre-emption points via some form of reschedule call may be similarly viewed as a special case of the Stack Resource Policy.

Recall that for each priority level, lowest first, the RPA algorithm selects the unassigned task that tolerates the most additional interference at that priority level. Intuitively, of all the unassigned D-JM system model tasks, the one with the largest value of "Deadline minus Jitter" is the one that can tolerate the most additional interference. Thus, we expect the priority ordering generated by the RPA algorithm to assign the D-JM system model tasks in "Deadline minus Jitter" partial order, interleaved in some way with the non D-JM system model tasks. We now prove this to be the case.

Theorem 3: Given a mixed system, and two D-JM system model tasks, τ_A and τ_B[8], where τ_A has a larger value of deadline minus jitter than τ_B, ($D_A - J_A \ge D_B - J_B$) then the additional interference characterised by α_i^A, tolerated by τ_A at an arbitrary priority *i* with τ_B at a higher priority, is at least as great as the additional interference α_i^B, tolerated by τ_B at priority *i* with τ_A at a higher priority.

Proof: As τ_A and τ_B are pre-emptable and have deadlines less than or equal to their periods, then the worst-case response time for τ_A assigned to priority level *i*, with τ_B at a higher priority, is given by $R_i = W_i + J_B$, where W_i is the smallest solution to the equality given in equation (2):

$$w_i = C_A + \left\lceil \frac{w_i + J_B}{T_B} \right\rceil C_B + I_i(w_i, A, B) + E(\alpha, w_i^n, i) \quad (2)$$

Where the function $I_i(w_i, A, B)$ represents the maximum time for which the other tasks, with the exception of τ_A and τ_B, prevent a ready task at priority *i* from executing, during an interval of length *w*. Note that the function $I_i(w_i, A, B)$ includes both interference from tasks of higher priority than *i* (with the exception of τ_A and τ_B) and also blocking effects due to tasks of lower priority than *i*. We return to this point about blocking later.

Similarly, the worst-case response time for τ_B assigned to priority level *i*, with τ_A at a higher priority, is given by $R_i = W_i + J_A$, where W_i is the smallest solution to the equality given in equation (3):

$$w_i = C_B + \left\lceil \frac{w_i + J_A}{T_A} \right\rceil C_A + I_i(w_i, A, B) + E(\alpha, w_i^n, i) \quad (3)$$

Now let W_i^B be the length of the busy period for task τ_B when it is at priority level *i* and is subject to the maximum amount of additional interference α_i^B, that it can tolerate at that priority level. The response time R_i^B, of τ_B is given by $R_i^B = W_i^B + J_B$. As $W_i^B \le D_B - J_B \le D_A - J_A$, and $D_A \le T_A$, it follows that $W_i^B + J_A \le T_A$, hence substituting W_i^B and α_i^B into equation (2) results in an identical equality to that obtained by substituting the same values into equation (3):

$$W_i^B = B_i + C_B + C_A + I(W_i^B, i, B) + E(\alpha_i^B, W_i^B, i) \qquad (4)$$

Thus $W_i^A = W_i^B$ and $\alpha_i^A = \alpha_i^B$ is also a solution to equation (2). As $R_i^B - J_B \le D_B - J_B \le D_A - J_A$, this solution gives a schedulable response time of

[8] Note again we use *A* and *B* to distinguish the tasks irrespective of the priority levels to which they are assigned.

$R_i^A = R_i^B - J_B + J_A$ for τ_A. Further, as $D_A - J_A \geq D_B - J_B$, there may be solutions to equation (3) for larger amounts of additional interference hence $\alpha_i^A \geq \alpha_i^B$ □

We now consider the effects of blocking. Recall that as far as blocking is concerned, co-operative and non-pre-emptive scheduling can be considered as special cases of the Stack Resource Policy. We therefore simply assume that tasks may share resources according to the Stack Resource Policy.

With the Stack Resource Policy, a task at priority *i* may be blocked for at most the duration of a single critical section executed by a task of lower priority than *i*, where that critical section involves access to a resource shared with a task of priority *i* or higher. As the task sets *lp(i)* and *hep(i)* remain the same irrespective of whether τ_A is at priority *i* and τ_B is at a higher priority or vice-versa, the set of critical sections that could cause blocking at priority *i* remains the same in both cases. This means that any blocking component of $I_i(w_i, A, B)$, as well as any interference components from tasks of higher priority than *i*, (not including τ_A and τ_B) are the same irrespective of whether τ_A is at priority *i* and τ_B is at a higher priority or vice-versa. Note that this remains the case when some of the non D-JM system model tasks form transactions with offset arrival times and thus the calculation of the exact worst-case blocking and interference effects becomes computational intractable.

Taking a sustainable approach[9] and computing blocking and interference independently, $I_i(w_i, A, B)$ comprises two components:

1. higher priority interference due to tasks in the set *hp(i)* with the exception of tasks τ_A and τ_B, and
2. a blocking factor B_i, representing the maximum time for which a task of lower priority than *i* can lock a resource shared with a task of priority *i* or higher.

As the task sets *lp(i)* and *hep(i)* remain the same independent of whether τ_A or τ_B is at priority *i*. Both of these factors are unchanged on interchanging the priorities of τ_A and τ_B.

Theorem 4: For a mixed system, where a schedulable priority ordering exists, there exists a robust priority ordering *P* with the D-JM system model tasks in "Deadline minus Jitter" monotonic partial order.

Proof: We assume (for contradiction) that there is an alternative priority ordering *Q*, which tolerates greater additional interference than priority ordering *P*, so $\alpha^Q > \alpha^P$. For the purposes of the proof, we will refer to this alternative priority ordering as Q_m.

We will iterative transform Q_m into $Q_{m-1} .. Q_1$, where Q_1 is a priority ordering with the D-JM system model tasks in "Deadline minus Jitter" monotonic partial order. Thus Q_1 is the equivalent of *P*. The transformation will be such that $\alpha^{Q_{k-1}} \geq \alpha^{Q_k}$, thus proving the theorem via the contradiction: $\alpha^P \geq \alpha^Q$.

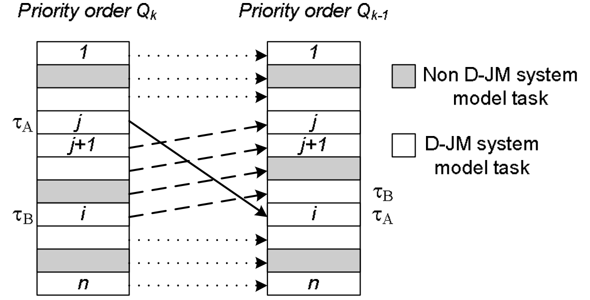

Figure 2: Transformation of priority order

We use *k* as an iteration count, where *k* counts down from *m* (the number of D-JM system model tasks) to 1. On iteration *k*, we select the D-JM system model task τ_A, with the *k*th largest value of deadline minus jitter. Thus on the first iteration (*k=m*), τ_A is the D-JM system model task with the smallest value of deadline minus jitter. Let *j* be the priority level of task τ_A in priority order Q_k. Next we identify the lowest priority level *i* in Q_k that is occupied by a D-JM system model task τ_B with a smaller value of deadline minus jitter than τ_A.

- If there is no such task τ_B or if *i* is a higher priority level than *j*, then no transformation is required on this iteration and so $Q_{k-1} = Q_k$.
- If *i* is a lower priority level than *j*, then we transform Q_k into Q_{k-1} by moving τ_A from priority *j* in Q_k to priority *i* in Q_{k-1} and shifting the tasks at priorities *j*+1 to *i* up one priority level (see Figure 2 above).

Comparing the tasks in priority order Q_{k-1} with their counterparts in Q_k. There are effectively four groups of tasks to consider:

1. $hp(i, Q_{k-1})$: These tasks are assigned the same priorities in both Q_k and Q_{k-1} and so can tolerate the same additional interference.
2. $hp(i, Q_{k-1}) \cap lep(j, Q_{k-1})$: These tasks retain the same partial order but are shifted up one priority level in Q_{k-1} and so can tolerate at least as much additional interference as they can in Q_k.
3. Task τ_A, which is at priority level *j* in Q_k and at the lower priority level *i* in Q_{k-1}: Theorem 3, tells us that τ_A can tolerate at least as much additional interference when it is at priority level *i* as task τ_B can tolerate when it is at that same priority level in priority ordering Q_k.

[9] Treating blocking and interference independently results in analysis which is sufficient but not necessary in the case of tasks with offset arrival times.

4. $lp(i, Q_{k-1})$: These tasks are assigned the same priorities in both Q_k and Q_{k-1}, and as $hep(i, Q_{k-1}) = hep(i, Q_k)$, they are subject to interference from the same set of higher priority tasks and so can tolerate the same additional interference in each case.

For every task in Q_{k-1}, the above analysis identifies a task in Q_k which does not have a greater tolerance to additional interference. Thus Q_{k-1} can tolerate at least as much additional interference as Q_k and so $\alpha^{Q_{k-1}} \geq \alpha^{Q_k}$.

A total of *m* iterations of the above procedure (for values of *k* from *m* down to 1) are sufficient to transform priority ordering *Q* into a priority ordering *P*, with the D-JM system model tasks in "Deadline minus Jitter" monotonic partial order. Further, this transformation is achieved without any reduction in the maximum amount of additional interference that the system can tolerate □

Theorem 5: For a mixed system, if a feasible priority ordering exists, then the RPA algorithm always generates a robust priority ordering with the D-JM system model tasks in "Deadline minus Jitter" monotonic partial order, irrespective of the task execution times, and irrespective of the form of the additional interference function $E(\alpha, w, i)$, provided that $E(\alpha, w, i)$ meets the monotonicity criteria stated in Section 3.1

Proof: Follows directly from the proof of Theorems 3 and 4, which are independent of both task execution times and the form of the function $E(\alpha, w, i)$, requiring only that $E(\alpha, w, i)$ meets the monotonicity criteria.

We now use this result to improve the efficiency of both the RPA algorithm, and Audsley's optimal priority assignment algorithm.

6.1. Priority assignment algorithm efficiency

The proof of Theorem 3 shows that at each priority level, the unassigned D-JM system model task with the largest value of deadline minus jitter can tolerate at least as much additional interference at that priority level as any other unassigned D-JM system model task. Thus at each priority level, the RPA algorithm need only calculate the value of α for at most a single unassigned D-JM system model task (the one with the largest value of $D_i - J_i$). In the worst-case, the non D-JM system model tasks are assigned last and so the total number of computations of α required is given by:

$$m(n-m+1) + (n-m)(n-m+1)/2$$

which simplifies to:

$$(n(n+1) - m(m-1))/2 \qquad (6)$$

If *m* = *0*, then equation (6) reduces to the familiar $n(n+1)/2$ computations. If *m*=*n*, then only *m* computations are required. Note, an additional O(*m*log*m*) operations are required to determine the "Deadline minus Jitter" monotonic partial ordering.

Theorem 3 also implies that the unassigned D-JM system model task with the largest value of deadline minus jitter is guaranteed to be schedulable at a particular priority level if any of the other unassigned D-JM system model tasks are schedulable at that priority. Thus at each priority level, Audsley's optimal priority assignment algorithm need only check the schedulability of at most one D-JM system model task (the one with the largest value of $D_i - J_i$).

Often in real-world systems, the overwhelming majority of tasks comply with the D-JM system model, with just a few tasks having arbitrary deadlines, deadlines prior to completion, non-pre-emptive execution, or forming transactions with offset arrival times. In this case, the efficiency improvement in the priority assignment algorithms is significant. For a system of 50 tasks, a number of which do not comply with the D-JM system model, Table 4 shows the number of computations required, and the factor by which the efficiency of the priority assignment algorithms is improved compared with the previous worst-case bound of *n*(*n*+1)/2 = 1275.

As an example, suppose that a system has four (non D-JM system model) tasks, which form a transaction with a period of 100ms and offsets of 0ms, 25ms, 50ms and 75ms respectively. Also part of the system are a further 46 D-JM system model tasks that are unrelated to the offset of this transaction. For this system, utilising the improvements described above, the optimal priority ordering can be found with a factor of 5.3 times less computation than before.

Table 4: Priority assignment algorithm efficiency improvement

Number of non D-JM system model tasks	Number of computations	Improvement factor
1	99	12.9
2	147	8.7
3	194	6.6
4	240	5.3
5	285	4.5
10	495	2.6
25	975	1.3

We note that Theorem 5 proves a conjecture made by Bernat in [22]. Bernat studied mixed systems comprising two types of tasks; those with "weakly hard" timing constraints, specifying the pattern of deadlines that must be met / may be missed, and those with "strongly hard" time constraints, where all deadlines must be met. Bernat showed that DMPO is

not optimal for weakly hard tasks and conjectured that the optimal priority assignment for a system containing both types of task would have the strongly hard tasks in deadline monotonic partial order. Bernat used this conjecture to improve the efficiency of Audsley's algorithm in a similar manner to that described above. In our terminology, weakly hard tasks are non-D-JM system model tasks, whilst strongly hard tasks are D-JM system model tasks.

7. "D-J" monotonic priority ordering

In this section, we consider fixed priority systems where all of the tasks comply with the D-JM system model, i.e. they are scheduled pre-emptively, have deadlines less than or equal to their periods, no offsets with respect to each other, and so on.

Theorem 6: "Deadline minus Jitter" monotonic priority assignment is the *robust priority assignment policy* (see Definition 2) for systems where all tasks comply with the D-JM system model, irrespective of the task execution times, and irrespective of the form of the additional interference function $E(\alpha, w, i)$, provided that $E(\alpha, w, i)$ meets the monotonicity criteria stated in Section 3.1.

Proof: Follows directly from Theorem 5.

Theorem 6 is a highly significant result. It tells us that for real-world systems, which have a scheduling policy and tasking model that comply with the D-JM system model, but are subject to ill-defined or unknown additional interference in the form of interrupts, operating system overheads, DMA cycle stealing, budget overruns and so on, then "Deadline minus Jitter" monotonic priority ordering is always the most robust priority ordering to use. Theorem 6 shows that this is the case even if the exact form of the additional interference function is unknown, that is, if we have little or no information about the type of additional interference, its extent or its exact timing behaviour. It is also the case if we have only rough estimates or indeed no information at all, about task worst-case execution times.

Theorem 6 has important implications when upgrading a real-time system which has a scheduling policy and tasking model that comply with the D-JM system model. Using a different microprocessor and operating system affects interrupt latencies, execution times, OS overheads and other forms of additional interference as well as task execution times. However, assuming only that the additional interference meets the broad monotonicity criteria, "Deadline minus Jitter" monotonic priority ordering remains the most robust priority ordering to use.

We note that Theorem 6 subsumes the result of Regehr [27], which showed that DMPO maximises the critical scaling factor for task execution times.

8. Summary and conclusions

In this paper we introduced the concept of *robust* priority ordering and provided an algorithm that can determine the robust priority ordering for a wide range of real-time systems scheduled using fixed priorities.

8.1. Applicability

The motivation for finding a robust priority ordering is that it is the optimal fixed priority ordering to use in single-processor, real-time systems which are subject to additional interference from sources such as interrupts, RTOS overheads, DMA cycle stealing, execution of checkpoints and error recovery blocks or indeed any other additional interference that can be characterised by a monotonically non-decreasing function of time interval and priority level.

As well as processor scheduling, the Robust Priority Assignment algorithm is also applicable to communications networks scheduled according to fixed priorities. For example, the RPA algorithm could be used to obtain a robust priority ordering for messages on Controller Area Network (CAN) [20] in the presence of additional interference due to errors on the bus.

The results described in Section 6 are also relevant to hierarchical systems where a mix of Deferrable, Sporadic and Periodic servers are used to schedule applications in the same system [14]. Both Periodic and Sporadic servers comply with the D-JM system model, whereas Deferrable servers do not[10]. For such systems, Theorem 5 shows that the robust priority ordering always has the Periodic and Sporadic servers in "Deadline minus Jitter" monotonic partial order. This implies that the more efficient form of Audsley's optimal priority assignment algorithm can be used to assign server priorities. This is particularly useful when a new application is added to the system and revised server priorities need to be determined at run-time.

8.2. Contribution

The major contributions of this work are:

- The introduction of the concept of *robust* priority ordering.
- The definition of an algorithm that determines the most robust priority ordering if any feasible ordering exists. This algorithm is applicable to a wide range of system models; provided that they meet simple rules in terms of how worst-case response times depend upon priority.

[10] In [13], the authors show that rate/deadline monotonic priority assignment is not optimal for Deferrable servers.

- Showing that in general, the most robust priority ordering depends upon the exact form of the additional interference function.
- Proving that for systems where "Deadline minus Jitter" monotonic priority ordering is optimal, it is also the most robust priority ordering effectively *independent* of the form of the additional interference function.
- Proving that for mixed systems, where some tasks comply with the simple D-JM system model, but other tasks do not (because they have offsets, arbitrary deadlines, co-operative / non-pre-emptive execution etc.), then if a feasible priority ordering exists, then a robust priority ordering also exists that has the D-JM system model tasks in "Deadline minus Jitter" monotonic partial order.
- Using the above result to improve the efficiency of both the Robust Priority Assignment algorithm and Audsley's optimal priority assignment algorithm.

These contributions make significant improvements to the set of known priority assignment techniques that are appropriate for use in the design of complex real-world, real-time systems.

8.3. Acknowledgements and future work

This work was funded in part by the EU Frescor project. As part of this project, we aim to extend our research into priority assignment and to utilise the results described in this paper within the context of the Frescor contract framework.

9. References

[1] M.S. Fineberg and O. Serlin, "Multiprogramming for hybrid computation", In *proceedings AFIPS Fall Joint Computing Conference*, pp 1-13, 1967
[2] O. Serlin, "Scheduling of time critical processes". In *proceedings AFIPS Spring Computing Conference*, pp 925-932, 1972.
[3] C. L. Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of the ACM*, 20(1): 46-61, January 1973.
[4] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, 2(4): 237-250, December 1982.
[5] J. Lehoczky. "Fixed priority scheduling of periodic task sets with arbitrary deadlines". In *Proceedings 11th IEEE Real-Time Systems Symposium*, pp. 201–209, IEEE Computer Society Press, December 1990.
[6] N.C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times", *Technical Report YCS 164*, Dept. Computer Science, University of York, UK, December 1991.
[7] L. George, N. Rivierre, and M. Spuri. "Pre-emptive and non-pre-emptive real-time uni-processor scheduling. *Technical Report 2966*, Institut National de Recherche et Informatique et en Automatique (INRIA), France, September 1996
[8] N.C. Audsley, "Optimal priority assignment in fixed priority scheduling". Information Processing Letters Vol. 79, No. 1, pp39-44, 2001.
[9] A. Zuhily and A. Burns, "Optimality of (D-J)-monotonic priority assignment". *Information Processing Letters.* Volume 103, Number 6, pp. 247-250, April 2007.
[10] K. Bletsas, and N.C. Audsley, "Optimal priority assignment in the presence of blocking". *Information Processing Letters* Vol. 99, No. 3, pp83-86, August. 2006.
[11] T.P. Baker. "Stack-based Scheduling of Real-Time Processes." *Real-Time Systems Journal* (3)1, pp. 67-100, 1991.
[12] L. Sha, R. Rajkumar, and J.P. Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". *IEEE Transactions on Computers*, 39(9): 1175-1185, 1990.
[13] R.I. Davis, A. Burns. "Hierarchical Fixed Priority Pre-emptive Scheduling". *Technical Report YCS-2005-385*, University of York, Dept. of Computer Science, April 2005.
[14] R.I. Davis, A. Burns. "Hierarchical Fixed Priority Pre-emptive Scheduling" In *proceedings IEEE Real-Time Systems Symposium.* pp. 389-398. December 2005.
[15] R.I. Davis and A. Burns, "Optimal Priority Assignment for Aperiodic Tasks with Firm Deadlines in Fixed Priority Pre-emptive Systems". *Information Processing Letters* 53(5). 1995.
[16] A. Burns, K. Tindell, and A. J. Wellings. "Fixed priority scheduling with deadlines prior to completion". In *proceedings of the 6th Euromicro Workshop on Real-Time Systems*, pp 138-142, June 1994.
[17] N.C. Audsley, A. Burns, M. Richardson, A.J.Wellings. "Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling". *Software Engineering Journal*, 8(5) pp. 284-292, 1993.
[18] K.W. Tindell, , A. Burns, A.J.Wellings. "An extendible approach for analyzing fixed priority hard real-time tasks". *Real-Time Systems.* Volume 6, Number 2, pp133-151 March 1994.
[19] S. Baruah and A. Burns. "Sustainable Scheduling Analysis". In *proceedings 27th IEEE Real-Time Systems Symposium*, pp. 159–168, IEEE Computer Society Press, December 2006.
[20] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised". *Real-Time Systems*, Volume 35, Number 3, pp 239-272. April 2007.
[21] K. W. Tindell. "Using Offset Information to Analyse Static Priority Pre-Emptively Scheduled Task Sets". *Technical Report YCS-92-182*. Dept. of Computer Science, University of York, UK, 1992.
[22] G. Bernat "Specification and Analysis of Weakly Hard Real-Time Systems". *PhD Thesis*. Universitat de les Illes Balears. 1998.
[23] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed priority scheduling with deferred preemption revisited", *CS Report 06-34*, Technische Universiteit Eindhoven (TU/e), The Netherlands, December 2006.
[24] J.P. Lehoczky, L. Sha, Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behaviour". In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pp. 166–171, Santa Monica, CA, December 1989.
[25] D.I. Katcher, H. Arakawa, J.K. Strosnider, "Engineering and analysis of fixed priority schedulers". *IEEE Transactions on Software Engineering*, 19(9):920–934, September 1993.
[26] S. Punnekkat, R. Davis, A. Burns, "Sensitivity analysis of real-time task sets". In *Proceedings of the Asian Computing Science Conference*, pp72–82, Nepal, December 1997.
[27] J. Regehr, "Scheduling tasks with mixed pre-emption relations for robustness to timing faults". In *proceedings 23th IEEE Real-Time Systems Symposium*, pp. 315–326, IEEE Computer Society Press, December 2002.