Mode Changes in Priority Pre-emptively Scheduled Systems

K. W. Tindell, A. Burns, A. J. Wellings

Department of Computer Science, University of York, England

Abstract

In many hard real time systems the set of functions that a system is required to provide may change over time. One way of providing this change is to allow currently running hard real time tasks to be deleted or changed, or new tasks to be added. We define this change as a mode change, and seek to guarantee a priori the timing constraints of all tasks across the change from one mode to another. This paper derives scheduling theory for static priority pre-emptive scheduling that can be used to make such guarantees.

1. Introduction

In many hard real-time systems the functions provided by the system may need to change over time — a number of *modes* are progressed through, with each mode providing a set of functions. Modes are changed in a controlled manner such that the behaviour of the system during the transitional period is known *a priori*. Mode changes can be used for a number of purposes: changes are often required to reduce the maximum load on a system (we use the term 'load' here to include both computational and communications load). Consider a hypothetical commercial airliner that has several software components, including:

- functions to control the stability of the aircraft (automatic trim, *etc.*)
- functions to control the undercarriage (raising and lowering, automatic braking, *etc.*)
- functions to automatically land the aircraft (using radio beacons, for example)
- functions to provide auto pilot and navigation (which controls the flight control systems to steer the aircraft according to planned way points)

The computers in the hypothetical aircraft go through three distinct modes: take-off, level flight, and landing. During take-off the auto pilot and auto land functions are not needed; in level-flight the auto land and undercarriage functions are not needed; during landing the auto pilot functions are not needed. Thus, the peak software load in any mode is lower than the total software load. However, the auto-stability software is needed in all three modes, and needs to run continuously without missing a deadline — a mode change must not cause a glitch where the software fails to meet its timing requirements (*e.g.* a task missing a deadline or release being delayed). In addition to planned mode changes there are changes that may occur sporadically: abnormal events, such as a processor failure, might require the system functions to degrade in a graceful manner (existing functions might be deleted, or be degraded to use fewer resources, to accommodate functions from the failed processor). Similarly, the failure of a piece of controlled equipment may cause a mode change: in our hypothetical commercial airliner there may be many sensor and actuator failure patterns, with many different controller algorithms that could be used, depending on the sensors and actuators remaining. After an actuator failure (say), the system could be required to remove old control functions, change some functions, and add new ones. The requirements on a mode change protocol therefore include the timely changing and addition of functions.

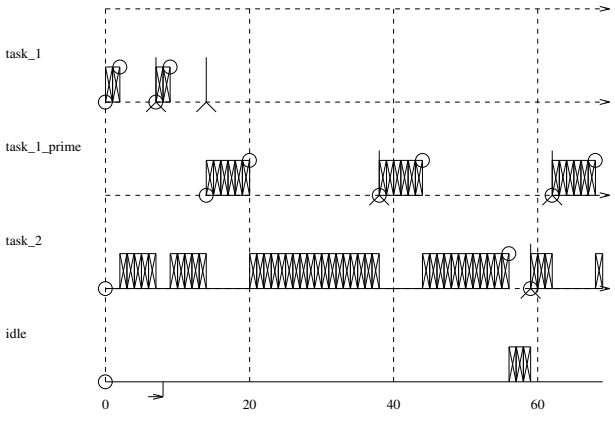
Current industrial practice in scheduling tasks is to compute *a priori* a cyclic schedule with a so-called major period equal to the least common multiple of the task periods. Mode changes are implemented by changing to a new schedule at the end of the major period of the old schedule. If the major period is long then a mode change could be deferred for a long time. New scheduling approaches, such as static priority pre-emptive scheduling (with deadline monotonic [6] and rate monotonic analysis [5]) and earliest deadline first scheduling [3], can potentially provide faster and more flexible mode changes — new tasks can potentially be released immediately without waiting for the end of a scheduling cycle.

Sha *et al* [8] attempt to address the problems of analysing *a priori* a single processor system, scheduled according to the rate monotonic scheduling policy, with tasks able to lock and unlock semaphores according to the priority ceiling protocol. The mode change protocol the propose is a simple one, where tasks are either deleted or added across a mode change. An "on the fly"

record of the processor utilisation is kept, and a task is added if the resulting task set is schedulable according to the exact-case rate monotonic scheduling equations [4]. The utilisation of an old task can be 'reclaimed' at the end of the period of the old task. Furthermore, the behaviour of the tasks under the priority ceiling protocol is addressed by a set of rules that specify when semaphore ceilings can be raised and lowered, and when new tasks locking semaphores can be added to the system. However, the protocol and analysis have some disadvantages — the analysis is not sufficient (*i.e.* the test may pass a task set that is unschedulable). The following simple example shows this.

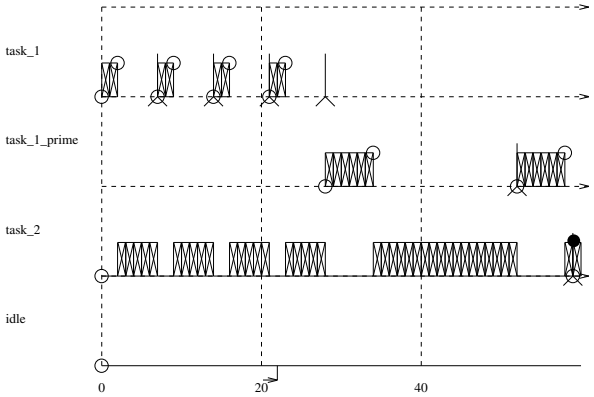
Consider a two task system that can execute in one of two modes, *A* or *B*. In mode *A*, task 1 has a worst-case computation time requirement C_1 of 2 ticks, and a period T_1 of 7 ticks. In mode *B* task 1 is deleted and replaced by task 1', with a worst-case computation time $C_{1'}$ of 6 ticks, and a period $T_{1'}$ of 24 ticks. Task 2 is required to execute unchanged through both modes. Task 2 has a worst-case computation requirement C_2 of 40 ticks, and a period T_2 of 59 ticks. For all three tasks the deadline of a task is equal to the period of that task.

The utilisation of the system in mode *A* is 96% and schedulable according to the exact rate monotonic scheduling equations [4]. Mode *B* has a utilisation of 92%, and is also deemed schedulable by the exact rate monotonic equations. Now, task 1' (in mode *B*) has a task utilisation of 25%, lower than the 28% of task 1 (in mode *A*). Since task 1' does not attempt to lock or unlock a semaphore, the only delay to the addition of task 1' into the system is the processor utilisation claimed by task 1. According to the Sha *et al* protocol, this utilisation can be reclaimed at the end of the period of task 1 after the mode change. The following STRESS [2] diagram illustrates this, with the mode change occurring at time 8:



As can be seen, task 1 has already started when the mode change request occurs at time 8, and runs to completion. The utilisation for task 1 cannot be reclaimed until the end of the period of task 1, so the introduction of task 1' is delayed until time 14. All tasks meet their deadlines across the change. Furthermore, when the mode change occurs at time 34, all tasks are still schedulable.

However, if the mode change occurs at time 22 then all is not well:



The large black blob for task 2 at time 59 indicates that task 2 has missed its deadline (the notation used in these diagrams is given at the end of this paper) — the Sha *et al* analysis predicts that this cannot happen. The analysis presented in the rest of this paper provides new sufficient analysis.

The rest of this paper is structured as follows: section 2 describes a hard real-time computational model and the proposed mode change protocol. Section 3 derives a schedulability test for simple mode changes. Section 4 extends the analysis to allow tasks to lock semaphores. Section 5 discusses some further issues for mode change analysis.

2. The Mode Change Protocol

In this section we describe the run-time behaviour of the system, indicating the restrictions on the behaviour of tasks; we also describe the operation of our mode change protocol.

A number of tasks are statically bound to a single processor. A task may arrive at any general time, and requires a bounded amount of computation time to complete. There is a lower bound on the time between successive arrivals of a task *i*, denoted T_i. Tasks may have a deadline requirement, denoted D_i, which

indicates the requirement placed on the worst-case time from arrival to completion of that task. Tasks are not permitted to voluntarily suspend themselves awaiting an external event, nor are they permitted to block awaiting a semaphore (although this latter restriction will be removed later in this paper). We require that, in the worst-case, a task i must complete before D_i after being released, and that this deadline is less than T_i (thus successive arrivals of a task do not interfere with each other)[1]. Tasks are assigned static priorities based upon deadline requirements, and as such are denoted deadline monotonic tasks [6]. The goal of the analysis presented in this paper is to calculate *a priori* the worst-case response time of a task; this worst-case time must hold across all modes in which i is active, and across a mode change.

The mode change protocol is a simple one. Firstly, there are two types of task defined: *changed* tasks, and *wholly new* tasks. A changed task consists of an *old mode version* and a *new mode version*. Old mode versions are always allowed to terminate gracefully — if they are running when the mode change occurs they run on unchanged until they terminate normally. The new mode version can be introduced any time later than the end of the period of the last release of the old mode version (note that a new mode version might therefore arrive immediately after the mode change, and hence cannot be allowed to suffer a delay before being released). A new mode version can be completely different from the old mode version, including different computation times, periods, deadlines, and priorities. A task that remains unchanged between modes is modelled by identical old and new mode versions. Similarly, a task that exists only in the old mode and is to be killed, is modelled as an old mode version task that is replaced by a null new mode version task.

A wholly new task i may be released no sooner than R_i after the mode change request occurs. The value of R_i is determined *a priori* from application specific timing requirements of the tasks. It may be appropriate for a task to be introduced to the system some large time after the mode change request (for example, a simple low criticality monitoring task); a more urgent task, such as a failure recovery task, may need to be introduced as soon as the mode change request occurs (indeed, the computational overheads due to the mode change can be modelled by a system task with a mode change offset of zero).

[1]This restriction can be removed, but the extensions to the scheduling theory to permit this are beyond the scope of this paper

We now turn to the *a priori* analysis required to determine the worst-case response times of tasks across a mode change, and hence determine the schedulability of a system.

3. Schedulability Analysis

The goal of the analysis is to determine the worst case response time of a task across a mode change. The general approach to finding a worst-case response time is to determine the worst-case amount of computation that can occur in a window, and to expand the window until all the computation time required by the task of interest can be accommodated in the window [1]. The analyses for the old mode versions, the new mode versions, and the wholly new tasks are considered separately. Firstly, we reproduce here the simple iterative deadline monotonic analysis equation given by Audsley *et al* [1]. The equation below gives the worst-case response time of a given task i — to determine the schedulability of a system, the equation must be applied to all tasks in turn (through the rest of this paper i will denote the task for which we are finding the worst-case response time; j will denote a higher priority task).

$$r_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \qquad (1)$$

Where r_i^n is the nth iteration towards a value of r_i, the worst-case response time of a task i; $hp(i)$ is the set of all tasks of higher priority than i. C_i is the worst-case response time of task i, and B_i is the worst-case blocking time of task i due to the operation of the priority ceiling protocol [8] (although, for the moment, we assume no blocking, and hence $B_i=0$). The iteration begins with $r_i^0=0$, and proceeds until either $r_i^n = r_i^{n+1}$ (*i.e.* the equation has converged to a value), or until r_i^{n+1} exceeds a threshold such as D_i.

We wish to extend the above equation to the three types of task in a mode change. Firstly, we analyse old mode versions. Consider a window, of width W_i, which represents the response time of an old mode version i. A mode change request occurs at time x after the start of the window. The priorities of an old mode version j and the corresponding new mode version j' can straddle the priority of an old mode version i. There are four possible situations for a given changing task j: both tasks are lower, only the new version is higher, only the old version is higher, or both are higher.

Now, the first case is trivial, since neither task can interfere with task i. The second is also straightforward:

task j' can only act on the part of task i executing in the new mode. Since W_i is the total response time of task i, and x is the time after the start of W_i when the mode change request occurs, then the time over which the new mode j' can act is given by $W_i - x$. Thus, the worst-case time task i can be pre-empted (and hence delayed) by a new mode version j' is given by:

$$\left\lceil \frac{W_i - x}{T_{j'}} \right\rceil_0 C_{j'} \tag{2}$$

The notation $\lceil z \rceil_0$ denotes a modified ceiling function that returns zero if $z<0$.

Case 3 is similar. The worst-case time task i can be pre-empted by an old mode version j given by:

$$\left\lceil \frac{x}{T_j} \right\rceil C_j \tag{3}$$

Now, if the mode change occurs at time x, and this time is just after the release of an old mode version j (*i.e.* x is greater than some multiple of T_j), then j continues to execute. The new mode version j' is released at the end of the period of j (*i.e.* released T_j after the last release of j).

For case 4 it might be thought that the worst-case computational interference by a changed task j would be simply the sum of equations 2 and 3. However, this is pessimistic, since a new mode version j' cannot be released sooner than time T_j after the last release of old mode version j. Now, the worst-case interference occurs in one of two situations. In the first situation an old mode version j can share a critical instant with task i, with interference from the new mode version j' occurring at the end of the period of the last release of j. The second situation is when the new mode version j' is released immediately after the mode change, with the old mode version j interfering over the initial part of the window. The following diagrams illustrate the two situations.

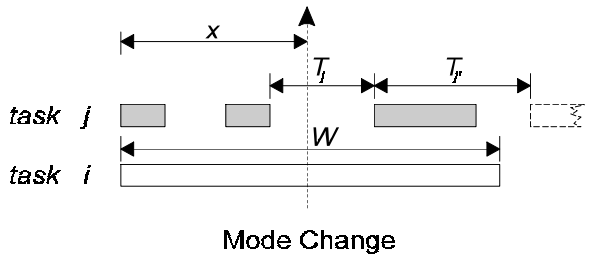


Figure 1: An old mode version shares a critical instant with task *i*

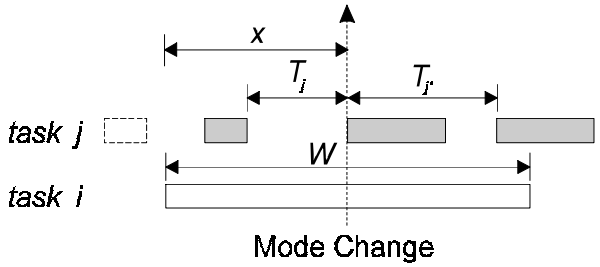


Figure 2: A new mode version is released at the mode change

In Figures 1 and 2 above, time flows from left to right. Rectangles represent task instances, with dashed rectangles representing tasks whose computation is *deemed* not to fall within the window marked *W*.

In Figure 1 the computational interference from a changed task j on an old mode version i is given by:

$$\left\lceil \frac{x}{T_j} \right\rceil C_j + \left\lceil \frac{W_i - \left\lceil \frac{x}{T_j} \right\rceil T_j}{T_{j'}} \right\rceil_0 C_{j'} \tag{4}$$

This can be re-written as:

$$k_j C_j + \left\lceil \frac{W_i - k_j T_j}{T_{j'}} \right\rceil_0 C_{j'} \tag{5}$$

Where k_j is given by:

$$k_j = \left\lceil \frac{x}{T_j} \right\rceil$$

In Figure 2, the computational interference from a higher priority old mode version j and the corresponding higher priority new mode version j' is given by:

$$\left\lfloor \frac{x}{T_j} \right\rfloor C_j + \left\lceil \frac{W_i - x}{T_{j'}} \right\rceil_0 C_{j'} \tag{6}$$

The worst-case computational interference from either situation is thus given by the maximum of equations 5 and 6:

$$\max \left(\begin{array}{l} \left\lfloor \frac{x}{T_j} \right\rfloor C_j + \left\lceil \frac{W_i - x}{T_{j'}} \right\rceil_0 C_{j'}, \\ k_j C_j + \left\lceil \frac{W_i - k_j T_j}{T_{j'}} \right\rceil_0 C_{j'} \end{array} \right) \tag{7}$$

Now, the left hand clause in equation 6 represents the computation due to whole executions of old mode version j that fall in the window W_i. It might be thought that this does not properly reflect the actual computation from the old mode version j — an instance of an old mode version j released momentarily before task i would be ignored by equation 6, when in reality it would still execute substantially in the window W_i. We will defer for the moment the explanation of the validity of equation 6.

From equations 2 to 7 above, we can see that the total computational requirement, in the window W_i, from higher priority changed tasks j, is given by:

$$\sum_{j \in omv} \max \left(\begin{array}{l} \left\lfloor \frac{x}{T_j} \right\rfloor \overset{\uparrow i}{C_j} + \left\lceil \frac{W_i - x}{T_{j'}} \right\rceil_0 \overset{\uparrow i}{C_{j'}}, \\ k_j \overset{\uparrow i}{C_j} + \left\lceil \frac{W_i - k_j T_j}{T_{j'}} \right\rceil_0 \overset{\uparrow i}{C_{j'}} \end{array} \right) \tag{8}$$

Where omv is the set of all old mode versions, and j' is the new mode version corresponding to old mode version j. The notation $\overset{\uparrow i}{C_j}$ is a conditional, which returns C_j if j is of higher priority than i, and zero otherwise.

The computation required by higher priority wholly new tasks in a window W_i, where the mode change occurs at time x after the start of the window is given by:

$$\sum_{n \in wnt} \left\lceil \frac{W_i - R_n - x}{T_n} \right\rceil_0 \overset{\uparrow i}{C_n} \tag{9}$$

Where wnt is the set of all wholly new tasks, and R_n is the earliest time after the mode change request that task n can be released.

The window W_i is equal to the time taken for a task i to execute for time C_i, and for higher priority tasks j to pre-empt task i and execute. Hence W_i is equal to the total computation that occurs in the window. Thus, W_i, for a given value of x, is equal to:

$$W_i(x) = \sum_{j \in omv} \max \left(\begin{array}{l} \left\lfloor \frac{x}{T_j} \right\rfloor \overset{\uparrow i}{C_j} + \left\lceil \frac{W_i - x}{T_{j'}} \right\rceil_0 \overset{\uparrow i}{C_{j'}}, \\ k_j \overset{\uparrow i}{C_j} + \left\lceil \frac{W_i - k_j T_j}{T_{j'}} \right\rceil_0 \overset{\uparrow i}{C_{j'}} \end{array} \right) + \sum_{n \in wnt} \left\lceil \frac{W_i - R_n - x}{T_n} \right\rceil_0 \overset{\uparrow i}{C_n} + C_i \tag{10}$$

Note that W_i occurs on both the left and right hand sides of equation 10 — the equation can be re-cast as a recurrence relation:

$$W_i^{n+1}(x) = \sum_{j \in omv} \max \left(\begin{array}{l} \left\lfloor \frac{x}{T_j} \right\rfloor \overset{\uparrow i}{C_j} + \left\lceil \frac{W_i^n - x}{T_{j'}} \right\rceil_0 \overset{\uparrow i}{C_{j'}}, \\ k_j \overset{\uparrow i}{C_j} + \left\lceil \frac{W_i^n - k_j T_j}{T_{j'}} \right\rceil_0 \overset{\uparrow i}{C_{j'}} \end{array} \right) + \sum_{n \in wnt} \left\lceil \frac{W_i^n - R_n - x}{T_n} \right\rceil_0 \overset{\uparrow i}{C_n} + C_i \tag{11}$$

The initial value of W_i is set to zero. It can be shown that $W_i^{n+1} \geq W_i^n$, and hence the equation is guaranteed either to converge (*i.e.* $W_i^{n+1} = W_i^n$), or to exceed some threshold, such as D_i.

We can now see why equation 6 does not underestimate W_i for an old mode version when the old mode version j is released momentarily before task i — if task j is released a time Δ before task i but still has a computational impact on i, then releasing task j at the same time as i must increase the computational impact on i by at least Δ. Hence, the estimate of the window W_i will increase by at least Δ. Therefore, in the next iteration of W_i, the current worst-case may then be the right hand clause of equation 6 (which fully takes account of all of the computation from j falling in the first x time of W_i). Note, however, that this will result in a non-exact test, since W_i can sometimes be over-

estimated — on-going research is investigating this problem.

We denote the worst-case response time a task i can experience as r_i. Now, the above equation for W_i gives the worst-case response time of an old mode version i for a given value of x — thus for an old mode version i we have

$$r_i = \max_{\forall x}\left[W_i(x)\right] \tag{12}$$

Many of the values of x result in the same value for W_i, and only a limited set of values of x are significant. Now, it might be thought that the largest value of W_i occurs for either $x=0$ or $x=T_i$ (this is the implicit assumption made by Sha *et al* [8]). However, upon examining the equations it soon becomes apparent that the maximum value need not necessarily occur at the end values of x (clearly from the example in section 1, the worst-case response time does not occur at $x=0$ or $x=T_i$). Consider again equation 5. One clause is a monotonic-increasing step function in k_j. Another is a monotonic-decreasing step function in k_j. There is no general way of determining the maximum value of the sum of increasing and decreasing step functions without examining the resulting function at points corresponding to all the steps of one of the functions. Consequently, to find the maximum value of W_i, equation 7 must be applied over all significant values of x. The significant values are ones that lead to different values of k_j, yet maximise the computational load due to wholly new tasks. The set of values of x are:

$$x \in \left\{0, \varepsilon, T_j + \varepsilon, 2T_j + \varepsilon, 3T_j + \varepsilon, \ldots\right\} \tag{13}$$

Where ε is the time quantum (in this paper we assume that ε is 1). Note that x must lie in the range 0 to W_i.†

We can now apply Equation 11 to the two task example of section 1. We wish to determine the schedulability of task 2 (since tasks 1 and 1' are always schedulable). The set of values of x over which to apply equation 11 are:

{0, 1, 8, 15, 22, 29, 36, 43, 50, 57, 64}

† We cannot know the value of W_i before we choose a value of x (since a value of x determines the value of W_i). To ensure that x is less than W_i we proceed to find W_i for successively larger values of x until we find an x for which the resulting value of W_i is less than x. We can then stop and discard that value of x and the value of W_i.

Taking x=22, we obtain the following:

$$W_2^1 = C_2 + \max\left(\begin{array}{l}\left\lfloor \frac{22}{T_1} \right\rfloor C_1 + \left\lceil \frac{W_2^0 - 22}{T_{1'}} \right\rceil_0 C_{1'}, \\ k_1 C_1 + \left\lceil \frac{W_2^0 - k_1 T_1}{T_{1'}} \right\rceil_0 C_{1'}\end{array}\right)$$

$$= 40 + \max(6+0, 8+0) = 48$$

$$W_2^2 = 40 + \max\left(\left\lfloor \frac{22}{7} \right\rfloor 2 + \left\lceil \frac{48-22}{24} \right\rceil_0 6, 8 + \left\lceil \frac{48-28}{24} \right\rceil_0 6\right)$$

$$= 40 + \max(6+12, 8+6) = 58$$

$$W_2^3 = 40 + \max\left(\left\lfloor \frac{22}{7} \right\rfloor 2 + \left\lceil \frac{58-22}{24} \right\rceil_0 6, 8 + \left\lceil \frac{58-28}{24} \right\rceil_0 6\right)$$

$$= 40 + \max(6+12, 8+12) = 60$$

We see that task 2 is unschedulable (this agrees with the results of the simulations discussed in Section 1).

We now turn to the problem of finding the worst-case response time of a new mode version *i.* A suitably long time after the mode change, the worst-case response time of a new mode version i will be given by equation 1 — the standard deadline monotonic scheduling equations derived by Audsley *et al* [1]. We term a response time found using these equations as the *steady-state* response time. Because old mode versions are not immediately killed after a mode change, but are allowed to run to completion, we need to find the response time that results when a new mode version is pre-empted by such higher priority old mode versions. In equation 1, the worst-case response time is determined by finding the smallest window that contains the computation of a task and all higher priority computation that could take place in that window. To find the worst-case response time for a new mode version i we adopt a similar approach. The worst-case scheduling point for a new mode version can be captured by one of three windows:

1. the *steady state* window, corresponding to the worst-case response time experienced by i a suitable long time after the mode change has occurred
2. a window of width W_i^{MC} starting at the mode change with task i released at the mode change
3. a window of width $W_i^{OMV}(x)$ starting at the release of the old mode version corresponding to task i, with the mode change occurring at time x after the window start, and i released at the end of the period of the old mode version

The steady-state response time can be found trivially from equation 1.

The second window (starting at the mode change) can be found by observing that the worst-case interference on a new mode version released at the mode change can occur in one of two situations:

1. The old mode version j is released momentarily before the mode change, thus sharing a critical instant with the window W_i^{MC}. The corresponding new mode version j' is released at time T_j later.

2. The new mode version j' is released momentarily after the mode change, thus sharing a critical instant with the window W_i^{MC}.

The worst-case of the above two situations is chosen. The computation from a wholly new task n occurs at least R_n after the mode change; the maximal interference from n will occur when it is released at R_n. The total computation in the window, and hence W_i^{MC} (the width of the window starting at the mode change), is found by:

$$W_i^{MC} = C_i + \sum_{n \in wnt} \left\lceil \frac{W_i^{MC} - R_n}{T_n} \right\rceil C_n + \sum_{j \in omv} \max \left(\begin{matrix} \uparrow i \\ C_j + \left\lceil \frac{W_i^{MC} - T_j}{T_{j'}} \right\rceil \end{matrix} \begin{matrix} \uparrow i \\ C_{j'}, \end{matrix} \left\lceil \frac{W_i^{MC}}{T_{j'}} \right\rceil \begin{matrix} \uparrow i \\ C_{j'} \end{matrix} \right) \tag{14}$$

Where *wnt* is the set of wholly new tasks, *omv* is the set of old mode versions. As with previous equations, W_i^{MC} must be found by iteration.

Now, task i is not guaranteed to be released at the mode change time. However, if task i is released a time Δ after the mode change, then the computation from old mode versions must decrease by Δ (or become zero — if it becomes zero then the steady-state response time must be the worst). The response time of i must therefore reduce by Δ, and hence there will be no extra computation time from new tasks. Clearly, this is a monotonic function and thus the worst-case value of the equation occurs when the window does indeed start at the mode change.

Finally, we define the window $W_i^{OMV}(x)$: a window starting when the old mode version is released, and where the mode change occurs at time x after this start. In this window falls the computation from i, the computation from the old mode version of i (denoted *old(i)*), plus the interference on the window due to higher priority changed and new tasks. Hence the window $W_i^{OMV}(x)$ is given by:

$$W_i^{OMV}(x) = C_i + \sum_{n \in nmt} \left\lceil \frac{W_i^{OMV}(x) - R_n - x}{T_n} \right\rceil \begin{matrix} \uparrow i \\ C_n \end{matrix} + \sum_{j \in omv} \max \left(\begin{matrix} \left\lfloor \frac{x}{T_j} \right\rfloor \begin{matrix} \uparrow i \\ C_j \end{matrix} + \left\lceil \frac{W_i^{OMV}(x) - x}{T_{j'}} \right\rceil_0 \begin{matrix} \uparrow i \\ C_{j'}, \end{matrix} \\ k_j \begin{matrix} \uparrow i \\ C_j \end{matrix} + \left\lceil \frac{W_i^{OMV}(x) - k_j T_j}{T_{j'}} \right\rceil_0 \begin{matrix} \uparrow i \\ C_{j'} \end{matrix} \end{matrix} \right) \tag{15}$$

Note that task i is a member of the set *omv* but that this does not result in C_i being included twice. Note also that we must evaluate the equation for all significant values of x in the same manner as for equation 11. It can now be seen that equation 14 is a special case of equation 15 where $x \to 0$.

Thus the worst-case occurs for one of the three windows defined. Now, for W_i^{OMV} the task i is released $T_{old(i)}$ after the start of the window, and consequently the computation occurring before this does not contribute to the worst-case response time of task i. Therefore the worst-case response time for a new mode version i is given by:

$$\max\left(r_i^{SS}, W_i^{MC}, W_i^{OMV} - T_{old(i)}\right) \tag{16}$$

Where r_i^{SS} is the steady-state worst-case response time, as calculated by equation 1, W_i^{OMV} is the maximum value of $W_i^{OMV}(x)$ over all significant values of x, and W_i^{MC} is the value of the window starting at the mode change.

We now turn to the final problem of finding the worst-case response time for wholly new tasks, and again define a window W_i to start at the time of the mode change. W_i in this case is the same as that given equation 14. Now, task i cannot be released sooner than R_i after the mode change, and hence the computation in W_i from the mode change to R_i occurs before i is released, and can be discounted. If the value of W_i is less than R_i then the old mode can have no effect on task i — task i is released later than the time taken to complete all the higher priority old mode computation. Hence, the worst-case response time of a wholly new task i is given by:

$$r_i = \max\left(r_i^{SS}, W_i - R_i\right) \tag{18}$$

As check, we can take the mode change equations developed in this section and assume 'null' mode change: where there are no wholly new tasks, and all old mode versions are the same as their corresponding new mode versions (*i.e.* $\forall j\ T_{j'}=T_j,\ C_{j'}=C_j$). It is easy to show that the analysis presented so far reduces to the original deadline monotonic equation (equation 1) under these conditions. Therefore we can conclude that the simple non-blocking deadline monotonic analysis derived by Audsley *et al*, and given in equation 1, is merely a special case of this more general mode change analysis.

4. Semaphore Locking

In this section we consider further analysis resulting from removing the restriction that tasks cannot lock semaphores. We assume now that semaphores are locked and unlocked according to the priority ceiling protocol [7], and seek to extend the analysis of the previous section to include blocking factors.

Sha *et al* [8] examine the problem of choosing the ceiling values of semaphores which can be locked and unlocked across mode changes. They formulate strict rules for raising and lowering the ceiling values, and for when a new mode task can be added to the system. These rules preserve the property that a task cannot be blocked more than once by a lower priority task. However, it is not possible to use this approach here: a new mode version may need to be released arbitrarily close to the mode change. The Sha *et al* rules governing the addition of new tasks lead to a potentially long delay if semaphores are locked by this new task. Sha *et al* go on to give an alternative approach where system ceilings remain static between modes and new tasks can be added immediately after the mode change. We adopt this approach, and hence define the ceiling of a semaphore to be the priority of the highest priority task in any mode which uses that semaphore - the so-called "ceiling of ceilings".

The problem now becomes one of determining the correct blocking factor for each task. Now, an old mode version can never be blocked by any new mode version, or wholly new task — if any new task were to have blocked an old mode version then the old mode version must have been released after the new task. Clearly this can never happen, since an old mode version must have been released in the old mode, and all new tasks must have been released in the new mode. Hence the blocking factor for an old mode version i is therefore equal to the longest critical section of any lower priority old mode version locking a semaphore with ceiling priority greater than or equal to the priority of task i. To add this blocking factor to our analysis of the worst-case response time for an old mode version i, we again consider the computational window W_i. Equation 10 gives the worst-case higher priority computation that would take place in the window for an old mode version i, including the computation for task i. We now simply add to that the time B_i, where B_i represents higher priority computation due to the operation of the priority inheritance mechanism of the priority ceiling protocol. Equation 10 is thus updated to:

$$W_i(x) = \sum_{j \in omv} \max \begin{pmatrix} \left\lfloor \frac{x}{T_j} \right\rfloor {}^{\uparrow i}C_j + \left\lceil \frac{W_i - x}{T_{j'}} \right\rceil_0 {}^{\uparrow i}C_{j'}, \\ k_j\, {}^{\uparrow i}C_j + \left\lceil \frac{W_i - k_j T_j}{T_{j'}} \right\rceil_0 {}^{\uparrow i}C_{j'} \end{pmatrix} + \sum_{n \in wnt} \left\lceil \frac{W_i - R_n - x}{T_n} \right\rceil_0 {}^{\uparrow i}C_n + C_i + B_i \tag{19}$$

Clearly, in the steady-state equation for new mode versions and wholly new tasks we must include a blocking factor that accounts for such a task being blocked by any lower priority new mode version or wholly new task. However, this blocking factor must also include the potential blocking from any lower priority *old* mode version — a lower priority old mode version can generally exist for a long time, and cause task i to block well into the new mode. The blocking factor B_i for the steady-state equation (Equation 1) is therefore equal to the longest critical section of any lower priority old mode version, new mode version, or wholly new task locking a semaphore of ceiling priority greater than or equal to the priority of task i.

To find the blocking factor B_i for a new mode version i released after the mode change consider how i can be blocked by lower priority tasks: if either W_i^{OMV} or W_i^{MC} results in a larger response time than equation 1, then the window W_i must consist of higher priority computation and, possibly, computation from a critical section of a lower priority old mode version. No lower priority *new* task can have been released and performed computation to lock a semaphore — all computation occurring in W_i would pre-empt lower priority new tasks. Hence, if a lower priority new task does manage to obtain a semaphore to block task i then the processor must have been released to this lower priority task

before task i was released, and therefore all higher priority old mode versions would have completed. Hence the steady-state equation would represent the worst-case response time (a similar argument holds for the blocking factors of wholly new tasks). Therefore we can say that the blocking factor B_i added to equation 14 is equal to the longest critical section of any lower priority old mode version locking a semaphore of priority greater than or equal to the priority of task i. The report by Tindell *et al* [10] discusses in detail this approach to reducing the blocking pessimism and also the approach of examining multiple windows to determine the worst-case scheduling points for tasks with offset information.

5. Outstanding Issues

So far in this paper we have restricted the discussion and analysis to include only single processor task sets. We have not addressed the issue of mode changes in distributed systems. The analysis in this paper can be easily extended to simple distributed systems by merely applying the analysis to each processor in turn, assuming a static binding of tasks to processors. However, one major problem hinders this approach: inter-processor communication. The problem of mode changes and communications has been mostly neglected: current industrial practice, as mentioned at the start of this paper, has been to compute static processor schedules to permit mode changes. In distributed systems these static schedules are broadened to include static schedules for the communications media (usually a shared broadcast bus), with the major cycle times of the processors and the bus being the same. For more flexible systems the schedulability of mode changes across distributed systems must be determined.

The report by Tindell *et al* [9] addresses some of the problems in guaranteeing end-to-end deadlines for hard real-time messages. The report discusses several architectural approaches, and integrates the scheduling of tasks with the scheduling of messages: the sending and receiving of messages is controlled by software tasks on each processor. The timing analysis for message delivery is linked with the timing analysis of all tasks scheduled on the processors. In one simple approach, using a shared Time Division Multiple Access (TDMA) broadcast bus, a 'transmit' task on each processor takes messages from a buffer (shared with other tasks on that processor which send messages), and transmits the messages on the bus. The transmit task is a periodic task, with period equal to the TDMA cycle time, and released by the start of the processor TDMA slot. Similarly, a periodic 'receive' task takes packets from a receive buffer, shared with a communications adapter, assembles messages, and places them in buffers shared with destination tasks. Tindell *et al* give schedulability analysis that not only bounds end-to-end message delivery times (from queuing at the source processor to delivery to the destination task) but gives the computational load imposed by the transmit and receive tasks on each processor. Now, after a mode change the load on the communications bus could change: some tasks may transmit smaller messages, perhaps less frequently, and new tasks that use the bus could be introduced. This would change the computational load imposed by the transmit and receive tasks. Thus the transmit and receive tasks in the old mode would be old mode versions, changing to new mode versions with different computation times. Furthermore, reconfiguration of the TDMA slots after a processor failure (say) would change the TDMA cycle time, and affect the period of the transmit task (and perhaps the receive task). Again, these tasks would change periods between modes.

We must also analyse the timing behaviour of the hard real-time messages during the mode change — a message queued by a task in the new mode may be delayed by messages still in the queue that were queued in the old mode. Furthermore, the computational load of the receive task could be higher just after the mode change: old mode messages could arrive while the receive task is dealing with new mode messages; the receive task would have to deal with both. We therefore propose that the way to analyse distributed hard real-time systems that can undergo mode changes is to integrate the analysis developed in this paper with the communications schedulability analysis developed by Tindell *et al*. This is beyond the scope of this paper and the subject of current research.

Another problem with mode changes in distributed systems is that of synchronising mode change requests: slight differences in the times when a mode change request is made on two different processors could lead to an erroneous mode change. Consider a system consisting of two tasks, each on separate processors. The tasks are synchronised to perform a certain action atomically — perhaps operating a device accessed by both processors — and are thus always released at the same time on both processors. Imagine a mode change request is received by some part of the system and forwarded to both processors. It is possible for the request to be received at different times on each processor — at one processor the request occurs before the release of the task pair, and thus the new version of

the task is run; at the other processor the request occurs after the release of the task pair, and thus the old version is run. This is not desired, and thus some way of co-ordinating mode changes is needed. The simplest way of accomplishing this is to use global time: a request for a mode change arrives, with the request stamped by the initiator with a time (in the near future) when the request is to be carried out. This time is interpreted as a *local* time, and hence each processor makes a consistent decision — in the above example the tasks are released at the same local times (both are released to within some global time value Δ of each other) and hence the tasks are released atomically.

6. Summary and Conclusions

This paper has described how a simple single processor system consisting of periodic and sporadic tasks can be allowed to undergo mode changes. Previous analysis has been seen to break down, and thus analysis has been derived which provides for the first time a schedulability test for such systems. Such a test could easily be incorporated into engineering support tools. This paper has also discussed some of the approaches that could be taken to extend the analysis to cope with more complex and interesting scheduling problems, and to handle distributed hard real-time systems.

7. References

[1] Audsley, N. C., Burns, A., Richardson, M. F., Tindell, K. W., and Wellings, A. J., *"Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling"*, Report RTRG/92/120, Department of Computer Science, University of York, February 1992

[2] Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J., *STRESS: A Simulator For Hard Real Time Systems*, Real Time Research Group, Department of Computer Science, University of York, October 1991.

[3] Chetto, H. and Chetto, M., *"Some Results of the Earliest Deadline Scheduling Algorithm"*, IEEE Transactions Software Engineering **15**(10) (October 1989), pp. 1261-1269.

[4] Lehoczky, J. P., Sha, L. and Ding, Y., *"The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour"*, Proceedings of the Real Time Systems Symposium (1989).

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

[6] Leung, J. Y. T., and Whitehead, J., *"On The Complexity of Fixed-Priority Scheduling of Periodic Real Time Tasks"*, Performance Evaluation (Vol. 2, Part 4, Dec 1982), pp. 237-250.

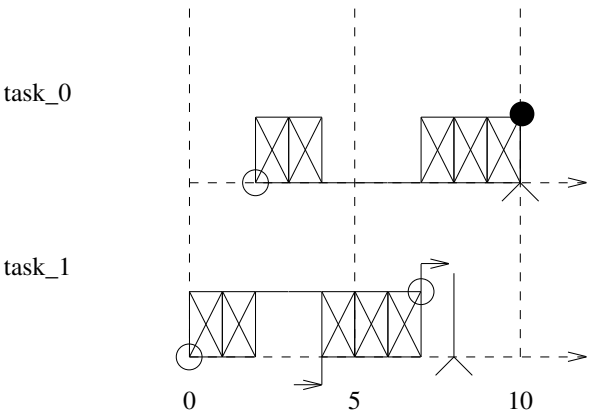
[7] Sha, L., Rajkumar, R., and Lehoczky, J. P., *"Priority Inheritance Protocols: An Approach to Real Time Synchronisation"*, IEEE Transactions on Computers **39**(9) (September 1990), pp. 1175-1185.

[8] Sha, L., Rajkumar, R., Lehoczky, J., and Ramamritham, K., *"Mode Change Protocols for Priority-Driven Pre-emptive Scheduling"*, Real Time Systems **1**(3) (1989), pp. 244-264.

[9] Tindell, K. W., Burns, A., and Wellings, A. J., *"Allocating Real Time Tasks (An NP-Hard Problem made Easy)"*, Real Time Systems **4**(2) (June 1992) pp. 145-165

[10] Tindell, K., "*Using Offset Information to Analyse Static Priority Pre-Emptively Scheduled Task Sets,*" Dept of Computer Science, University of York (August 1992).

Simulator Diagrams

In these diagrams, time increases across from left to right. Task execution is represented by boxes. A task which is preempted is shown by a line at the level of the bottom of the boxes; a task which is blocked by a line at the level of the top of the boxes. These states are annotated by a variety of symbols:

Task release is marked by a low-level circle, and successful task completion by a high-level circle. If a task fails to meet its deadline, or otherwise fails to complete, then a filled high-level circle is used. Task deadlines are marked by a vertical line with an inverted 'v' mark at the bottom. Message transmission is marked by high-level outgoing arrow, and message reception by a low-level incoming arrow. An example is shown below.