AN EXTENDIBLE APPROACH FOR ANALYSING FIXED PRIORITY HARD REAL-TIME TASKS

K. W. Tindell[1]
Department of Computer Science, University of York, England YO1 5DD

ABSTRACT

As the real-time computing industry moves away from static cyclic executive-based scheduling towards more flexible process-based scheduling, so it is important for current scheduling analysis techniques to advance and to address more realistic application areas. This paper extends the current analysis associated with static priority pre-emptive based scheduling; in particular it derives analysis for tasks with arbitrary deadlines that may suffer release jitter due to being dispatched by a tick driven scheduler. We also consider bursty sporadic activities, where tasks arrive sporadically but then execute periodically for some bounded time. The paper illustrates how a window-based analysis technique can be used to find the worst-case response time of a task set, and shows that the technique can be easily extended to cope with realistic and complex task characteristics.

1. INTRODUCTION

One commonly proposed way of constructing a hard real-time system is to build the system from a number of sporadic and periodic tasks, each assigned static priorities, and dispatched at run-time according to the static priority pre-emptive scheduling algorithm. The main thrust of research with this approach has been to derive *a priori* analysis that can bound the behaviour of the tasks at run-time. Original work by Liu and Layland [11] provides *a priori* analysis to determine if a set of periodic tasks would be guaranteed to meet their deadlines. Task deadlines are taken to be at the end of the periods of the tasks, and tasks are not permitted to block at run-time. Furthermore, each task is assigned a unique priority monotonically with task period, and hence the name *rate monotonic scheduling*. The analysis provides a *schedulability test* by giving a utilisation bound. The Liu and Layland analysis is *sufficient* (an essential property of any schedulability test — if the test passes a set of tasks then those tasks must always meet their deadlines) but not *necessary* (a measure of the 'efficiency' of the test — if the test is necessary then a rejected task set will be unschedulable in the worst-case); Sha *et al* [9] extended the analysis to provide an *exact* test (*i.e.* both sufficient and necessary). Sha *et al* [15] derived a run-time algorithm to permit tasks to lock and unlock semaphores according to a protocol, termed the *priority ceiling protocol*. With this protocol a system is guaranteed to be free of deadlock (on single processor systems), and a given task can be blocked at most once by a lower priority task. Sha *et al* extended the rate monotonic analysis to account for the behaviour of this protocol, adding a *blocking factor* to the schedulability equations (the blocking factor accounts for the worst-case time a given task can be blocked).

The priority pre-emptive dispatching algorithm has also been analysed by Joseph and Pandya [5] to find the *worst-case response time* of a given task. Analysis is derived that finds the worst-case time between a task being released (*i.e.* placed in a notional priority-ordered run-queue) and the task completing the execution of a worst-case required computation time. This permits task deadlines to be less than task periods: the worst-case response time can be compared to a static deadline. The optimal priority ordering for tasks with deadlines less than periods has been shown to be the *deadline monotonic* ordering [10]. Allowing task deadlines to be less than task periods is useful for many real-time applications: in distributed systems deadlines can be shortened to allow time for messages to pass between processors [17]; for control systems input and output *jitter* can be controlled [12]. However, some real-time applications are not so stringent, and can accept task deadlines greater than task periods: a task is permitted to re-arrive before the previous invocation has finished (and is then delayed until the previous invocation terminates). Lehoczky [8] describes qualitative analysis which can determine the worst-case response time of a given task with such arbitrary deadlines. Lehoczky points out that neither the rate monotonic

[1]The author can be contacted by e-mail at `ken@minster.york.ac.uk`

nor deadline monotonic priority ordering policies are optimal for tasks with arbitrary deadlines. We will reproduce here an algorithm which finds the optimal priority ordering for any task set.

Many static priority pre-emptive dispatchers are implemented using *tick scheduling* – a periodic clock interrupt runs the scheduler; a *budget timer* ensures that control is returned to the scheduler if a task exceeds its permitted worst-case execution time. One of the problems with pure tick scheduling is that sporadic task arrivals are polled by the scheduler, which means that a sporadic task can *arrive* (*i.e.* want to run) but be delayed before being *released*. It is then possible that this leads to so-called *release jitter*: variability in the release of a task (by at most the tick period). Periodic tasks which have a period that is not an integer multiple of the tick period can also in general suffer a release jitter. This release jitter leads to the possibility of a task appearing to arrive sooner than the worst-case inter-arrival time, and violates the assumptions of current analysis. Rajkumar [14] extends the exact rate monotonic analysis of Lehoczky *et al* [9] to permit tasks to be blocked on an external event (release jitter is a special case of external blocking) in order to permit the priority ceiling protocol to be extended to distributed systems. However, these extensions result in a non-exact test; we will derive exact analysis which accounts for release jitter.

Some real-time systems have tasks that behave as so-called *sporadically periodic* tasks [2]: a task arrives at some time, executes periodically for a bounded number of periods (called *inner periods*), and then does not re-arrive for a larger time (called the *outer period*). Examples of such tasks are interrupt handlers for bursty interrupts (for example, packet arrivals from a communications device), or certain monitoring tasks. Existing analysis is not exact (and hence pessimistic) for these tasks: tasks would be assumed to execute continuously at their inner period rate. We will derive exact analysis for tasks with this behaviour.

This paper will derive analysis for static priority pre-emptive systems that permits tasks to have arbitrary deadlines, release jitter, and behave as sporadically periodic tasks. The derivation of this analysis will illustrate how using a *window approach* to finding worst-case response times for these tasks is an appropriate way of obtaining an analysis tailored to the behaviour of real real-time tasks. This approach is easily extended to deal with other application characteristics.

The paper is structured as follows: Section 2 will describe the computational model assumed throughout this paper, and define the notation used. Section 3 will derive basic analysis from that of Joseph and Pandya and extend it to permit arbitrary deadlines, using the approach of Lehoczky. Section 4 will extend the analysis to permit release jitter to be accounted for. Section 5 will further extend the analysis to be exact for sporadically periodic tasks. Section 6 will also discuss the implementation of static priority pre-emptive dispatching using timer interrupts (*i.e.* tick scheduling); the analysis will be extended to exactly account for the overheads due to this means of scheduling. Section 7 will give an algorithm to find the optimal priority ordering for a task set. Section 8 will summarise the scheduling theory developed. Section 9 will present an example task set and determine the worst-case response times of each task in the set. Appendix A gives a table of notation used in this paper. Appendix B gives details on how a program implementing the analysis described in this paper can be obtained.

2. COMPUTATIONAL MODEL AND ASSUMPTIONS

A number of tasks are statically assigned to a single processor. Tasks have unique priorities; the run-time system provides pre-emptive priority-based dispatching. Each task may lock and unlock semaphores according to the priority ceiling protocol [15] (or equivalent [3]); although tasks are assigned unique static priorities, they may have their priorities temporarily increased due to priority inheritance (as part of the operation of the priority ceiling protocol). Tasks can *arrive* at any time (*i.e.* want to run), but can be delayed for a variable but bounded amount of time (termed the *release jitter*) before being placed in a notional priority-ordered run-queue (*i.e. released*). Tasks are given a worst-case inter-arrival time, termed the *period* (T): a task cannot re-arrive sooner than this time. For each arrival a task may execute a bounded amount of computation, termed the *worst-case execution time* (C). This value is deemed to contain the overheads due to context switching. The cost of pre-emption, within the model, is thus assumed to be zero.

The *worst-case response time* of a task (r) is the longest time ever taken by that task from the time it *arrives* until the time it completes its required computation. If a task has a worst-case response time greater than its period then the possibility exists for a task to re-arrive before the previous invocation has completed. There are two ways of dealing with this situation: the previous invocation is deemed to have a lower priority than the new arrival, and the new arrival pre-empts the old, *or* the new arrival is deemed to have a lower priority, and is therefore delayed from executing until after the previous invocation terminates. We adopt the latter approach for several reasons: the implementation of this approach is easier, since the run-time system does not have to support concurrent threads of the same task. Also, it makes little sense to have a task processing an event earlier in time delayed by the processing of a later one; a general rule in real-time systems is to preserve the order of events. Finally, the worst-case response time bounds derived are in general lower than those found if an earlier invocation is pre-empted.

Sporadically periodic tasks are assigned two periods: the *inner period* (t) and the *outer period* (T). The outer period is the worst-case inter-arrival time between 'bursts'; the inner period is the worst-case inter-arrival time between tasks within a burst. There are a bounded number of arrivals to each burst; furthermore, the total time for the burst (*i.e.* the number of inner arrivals multiplied by the inner period) must be less than or equal to the outer period. A task that is not a bursty task is simply modelled as one that has an inner period equal to the outer period, and at most one 'inner arrival'.

3. BASIC ANALYSIS AND ARBITRARY DEADLINES

This section derives simple analysis for the computational model described above. We assume that there is no release jitter and that tasks are not sporadically periodic.

Joseph and Pandya [5] derived simple analysis to find the worst-case response time of a given task i, assuming sporadic tasks with minimum inter-arrival times, and worst-case computation times. The analysis assumes a *critical instant*, where all tasks are assumed to be released together; this is the worst-case scheduling scenario for simple tasks. The following analysis gives the worst-case response time of a task i (r_i), assuming that all tasks are released as soon as they arrive, and that tasks do not suspend themselves (see Appendix A for a summary of the notation used):

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

As can be see, the response time r_i appears on both sides of the equation. Joseph and Pandya give a method for evaluating the equation, but a simple approach can be used by iterating to a solution:

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

Where $r_i^0 = C_i$

Theorem 3.1

If the utilisation of the i highest priority levels is less than 1, then the sequence r_i^n converges to r_i (the task completion time) in a finite number of steps (*i.e.* there exists an n such that $r_i^{n+1} = r_i^n$)

Proof:

See Tindell [16] or Joseph and Pandya [5]

The task meets its deadline if and only if r_i is less than or equal to D_i, the deadline of task i. If the sequence converges to a value of r_i greater than T_i then the value is invalid, since the computation of previous invocations of i has not been accounted for. For systems with task deadlines less than task periods (*i.e.* $D_i \leq T_i$) this is not a problem, since such a value of r_i would result in an unschedulable task anyway. However, for the arbitrary deadline situation the equation is no longer sufficient.

Lehoczky [8] describes qualitative analysis to determine the schedulability of a task set with arbitrary deadlines. The approach uses the notion of ‘busy periods’: a “level *i* busy period” is defined as the maximum time for which a processor executes tasks of priority greater than or equal to the priority of task *i*. Lehoczky shows how the worst-case response time of a task *i* can be found by examining a number of windows, each defined to be the length of the busy period starting at the window, and each window starting at an arrival of task *i* (hence at some multiple of T_i before the current invocation of task *i*). A number of windows back in time need to be examined to find the worst-case response time (in general, the worst-case response time can be the response time corresponding to any one of the windows). Figure 1 illustrates this.

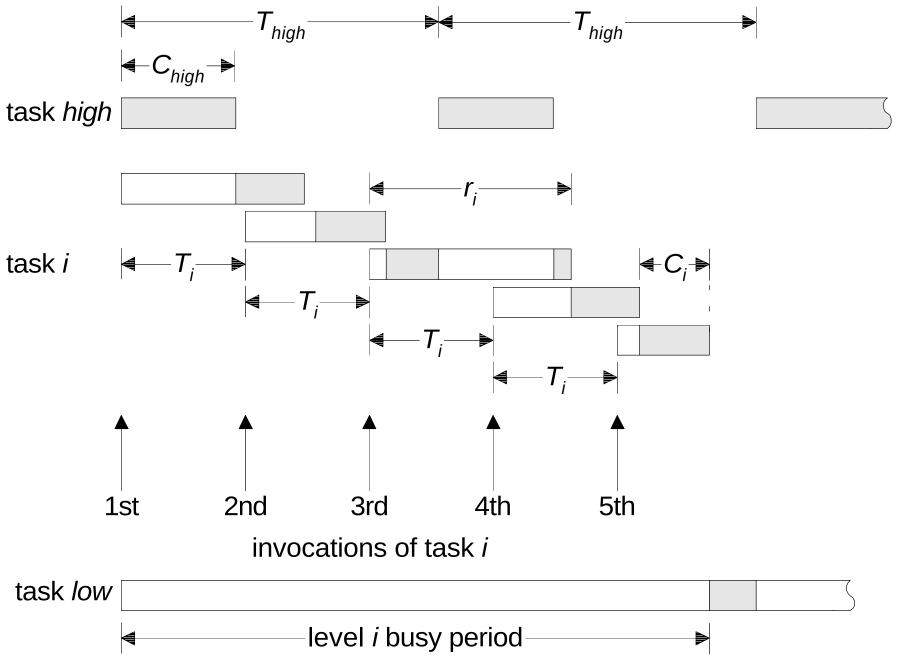


Figure 1: a level *i* busy period over five invocations of task *i*

Figure 1 shows part of the execution of a three task system: task *high* is a high priority task (included merely for illustrating response times of task *i*), task *i* is of medium priority (and can arrive before its previous invocation has completed), and task *low* is the lowest priority task (included to illustrate the length of the level *i* busy period). Time flows from left to right across the diagram. A task executing for some time is depicted as a shaded rectangle. The pre-emption of a task is depicted as a white rectangle. The response time of a task is therefore the length of the whole rectangle. The level *i* busy period is illustrated (the busy period ends when task *low* is able to begin execution). To find the worst-case response time all five invocations of task *i* in this busy period must be examined (in the diagram, the third invocation of task *i* is delayed the longest, and this response time is the worst-case response time). In the following discussion the ‘current invocation’ of task *i* is assumed to be the last (*i.e.* fifth) invocation.

For a level *i* busy period starting at time qT_i before the release of the current invocation of task *i* we can extend the original analysis by finding the computation that starts in this period:

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i(q)}{T_j} \right\rceil C_j \qquad (2)$$

The value of $w_i(q)$ can again be found by iteration, with $w_i(q)^0 = (q+1)C_i$; a faster convergence can be obtained by observing that $w_i(q+1) \geq w_i(q)$, and hence $w_i(q)^0 = w_i(q-1)$ will converge faster, with $w_i(0)^0 = C_i$.

Notice how all the computation of priority *i* or higher starting in the window is accounted for; the term qC_i accounts for the computation of previous invocations of task *i* starting in the window. Now, the first qT_i of the level *i* busy period falls before the current invocation of task *i* is released, and hence cannot contribute to the response time. The response time corresponding to the window starting qT_i before the current invocation of task *i* is therefore given by:

$(w_i(q) - qT_i)$

Now, as mentioned above, the worst-case response time can occur at any one of these response times, and thus the worst-case response time is given by:

$$r_i = \max_{q=0,1,2,\ldots} \left(w_i(q) - qT_i \right) \qquad (3)$$

The windows for increasing values of *q* need to be determined. The sequence is finite, however, because the search can stop if a level *i* busy period is found which finishes before task *i* starts (*i.e.* the processor is released to process lower priority tasks) — since the processor is executing lower priority tasks there can be no impact on task *i* from previous invocations of task *i* in busy periods starting earlier. Thus the above iteration over increasing values of *q* can stop if:

$w_i(q) \le (q+1)T_i$

The above analysis is illustrated by considering again figure 1: the third invocation of task *i* (*i.e.* $q = 2$) finishes at time *w*, where *w* is equal to $2C_{high} + 3C_i$ (this is the value of *w* to which equation 2 converges when $q = 2$). The third invocation of task *i* is released at time $2T_i$, and therefore the response time of this invocation is $2C_{high} + 3C_i - 2T_i$. After evaluating the response times for all five invocations starting in the busy period it can be seen that this is the largest response time (*i.e.* the worst-case response time).

We can now see that the original analysis is a special case of the above analysis: the original equation (Equation 1) is equivalent to equation 2 with $q = 0$. No windows starting earlier need be considered, since we constrain D_i, and hence $w_i(0)$, to be less than T_i.

At this point it is appropriate to discuss the extensions needed to analyse the priority ceiling protocol. The protocol has the property that any task can only be blocked at most once by a lower priority task. In fact, a stronger assertion can be proved: that during a level *i* busy period there can be at most one blocking due to a task of lower priority than *i*. This can be seen by considering the operation of the priority ceiling protocol: at any time only one task of lower priority than *i* can hold a semaphore with ceiling greater (or equal to) the priority of *i*. Thus any task executing in the level *i* busy period can only be blocked at most once. When this task terminates the processor will continue to execute at a priority greater than or equal to *i* until the end of the level *i* busy period. No task lower than the priority of *i* can execute (and hence cannot lock a semaphore). Therefore the level *i* busy period can be blocked at most once by tasks of lower priority than *i*.

The window denoted $w_i(q)$ represents the execution of a level *i* busy period starting at time qT_i before the release of the invocation of *i* we are considering. The window $w_i(q)$ includes computation from multiple invocations of task *i* and higher priority tasks. To allow for blocking time we need only add the term B_i (equal to the longest critical section of a semaphore of ceiling priority greater than or equal to that of task *i*, where the critical section is executed by a task of priority lower than task *i*). The above analysis leads to the following theorem:

Theorem 3.2

The worst-case response time of a task *i* is given by:

$$r_i = \max_{q=0,1,2,\ldots} \left(w_i(q) - qT_i \right)$$

where $w_i(q)$ is given by:

$$w_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i(q)}{T_j} \right\rceil C_j \qquad (4)$$

Proof:

Follows directly from the derivation of the relations described above

4. THE RELEASE JITTER PROBLEM

So far we have assumed that as soon as a task arrives it is released. In general this is not the case (a task may be delayed by the polling of a tick scheduler, or perhaps awaiting the arrival of a message), and the above equations must be extended to include this release jitter time.

The release jitter problem occurs when the worst-case time between successive *releases* of a task is shorter than the worst-case time between *arrivals* of a task. Consider the following scenario: a task *j*, of higher priority than task *i*, *arrives* at time 0. At a later time J_j later task *j* is *released* (perhaps *j* is a sporadic which must be polled for by a tick scheduler). At the same time task *i* is released. Task *j* immediately pre-empts task *i*, as expected. At time T_j task *j* re-arrives. This time *j* is immediately released (perhaps *j* arrived just before the tick scheduler polling period). From the view of task *i*, task *j* has arrived with time $T_j - J_j$ between arrivals (Figure 2). The current scheduling equations do not cater for this situation, and hence must be modified.

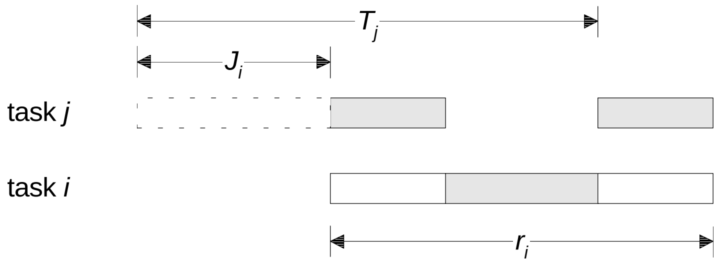


Figure 2: the problem of release jitter

Over a large number of periods task *j* will execute at the period T_j, but over a short period of time (between just two successive invocations of *j*) this rate is optimistic. Now, the worst-case scheduling scenario for this short-term inter-arrival 'compression' is as described above: a task *j* is *released* at the same time as the level *i* busy period. Without release jitter the total pre-emption time would be aC_i (where *a* is a positive integer); if release jitter is accounted for an extra pre-emption could occur, giving a total pre-emption of time $(a + 1)C_j$. This occurs if:

$$aT_j < w_i(q) + J_j \le (a+1)T_j \text{ and } (a-1)T_j < w_i(q) \le aT_j$$

i.e.:

$$a < \left\lceil \frac{w_i(q) + J_j}{T_j} \right\rceil \le a+1 \text{ and } a-1 < \left\lceil \frac{w_i(q)}{T_j} \right\rceil \le a$$

Lemma

The pre-emption due to higher priority tasks *j* is given by:

$$\sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j$$

Proof:

Follows directly from the definition of release jitter.

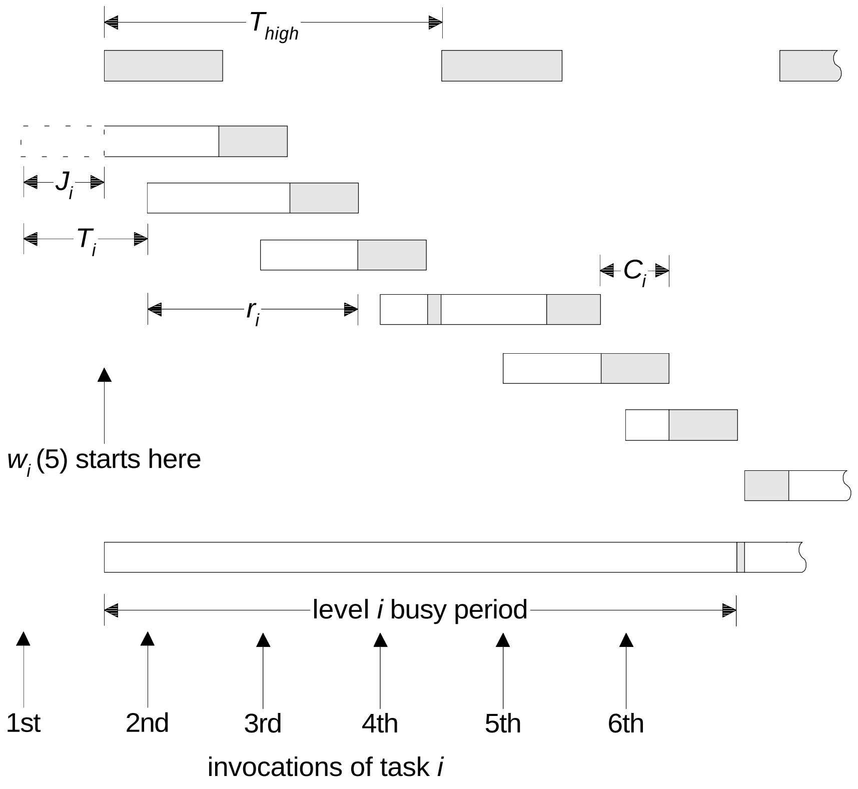


Figure 3: release jitter and busy periods

We now consider the join effects of $D > T$ and release jitter. Task i itself could potentially suffer release jitter (as described in the previous section), and thus the level i busy periods from previous invocations of task i can start J_i later (Figure 3). This does not affect the computation from i starting in $w_i(q)$ (it remains at $(q + 1)C_i$), but does affect the corresponding response time: the window $w_i(q)$ needs to start J_i later than the first invocation of i in the busy period, and hence equation 3 is updated to:

$$r_i = \max_{q=0,1,2,3,\ldots} \left(w_i(q) + J_i - qT_i \right) \tag{5}$$

For completeness we mention that the equations are still correct in the case when J_i is large ($>> T_j$): the equations predict that, in the worst-case, a number of invocations of j could be released simultaneously. This is exactly what could happen in the worst-case: suppose that task j was a packet interrupt handler (processing packets from a broadcast bus). If a tick scheduler polled for the release of task j and had a polling period $T_{tick} >> T_j$ (*i.e.* several packets could arrive between scheduler ticks) then all outstanding packets would have to be processed by task j, taking at most C_j computation time for each packet. Thus a lower priority task i released at the same time would be pre-empted for a long time, and thus the equations for release jitter handle the case when the release jitter is large.

5. SPORADICALLY PERIODIC TASKS

This section describes how the schedulability analysis of the previous section can be updated to determine exactly worst-case response times when tasks can behave as 'sporadically periodic' tasks: executing with an inner period (t_i) and outer period (T_i).

Firstly, we have to place some restrictions on the model: we must have the 'bursty' behaviour finishing before the next burst (*i.e.* $n_i t_i \le T_i$). We also assume that the release jitter J_i is the inner release jitter (*i.e.* each invocation of task *i* can suffer this jitter).

Theorem 5.1

The worst-case response time of task *i* is given by:

$$r_i = \max_{q=0,1,2,3,\ldots} \left(w_i(q) + J_i - m_i t_i - M_i T_i \right)$$

where m_i is given by:

$$m_i = q - M_i n_i$$

and

$$M_i = \left\lfloor \frac{q}{n_i} \right\rfloor$$

and where $w_i(q)$ is given by:

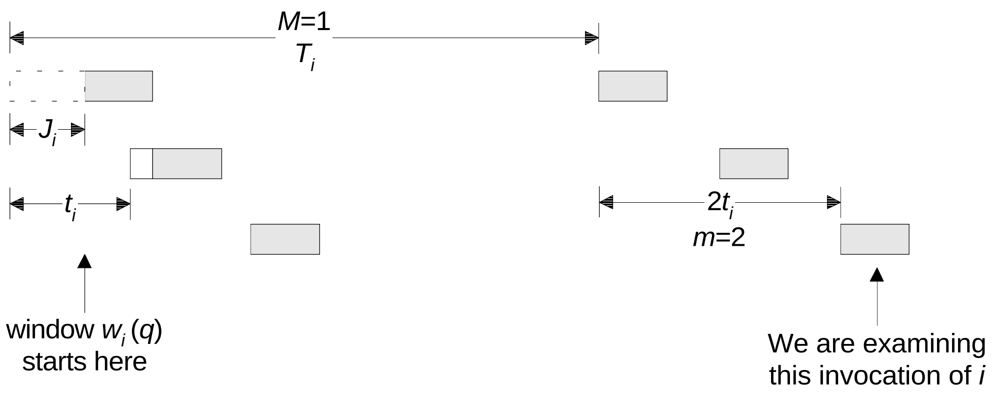
$$w_i(q) = \left(M_i n_i + m_i + 1 \right) C_i + B_i + \sum_{\forall j \in hp(i)} \left(\min\left(n_j, \left\lceil \frac{J_j + w_i(q) - F_j T_j}{t_j} \right\rceil \right) + F_j n_j \right) C_j$$

and F_j is given by:

$$F_j = \left\lfloor \frac{J_j + w_i(q)}{T_j} \right\rfloor$$

Proof:

We adopt the same approach to finding the worst-case response time as before: a number of windows are examined and the response time corresponding to each is determined. The worst-case response time is the maximum of these response times. However, instead of examining windows starting at time $J_i + qT_i$ before the current release of a task *i* we have to also consider windows starting at $J_i + qt_i$ before the release of task *i*.



M=1
T_i
J_i
t_i
$2t_i$
m=2
window $w_i(q)$ starts here
We are examining this invocation of i

Figure 4: sporadically period tasks; $n_i = 3$, $q = 5$

Firstly we consider the computation occurring in the window $w_i(q)$ from previous invocations of task i. We define two integers M_i and m_i, where M_i is the number of outer periods previously that the window w starts at, and m_i is the number of inner periods (Figure 4). M_i and m_i are given by:

$$M_i = \left\lfloor \frac{q}{n_i} \right\rfloor$$

$$m_i = q - M_i n_i$$

where q is an integer ≥ 0.

From figure 4 it can be seen that the window starts at time $m_i t_i + M_i T_i - J_i$ before the current release of task i. The total computation from previous invocations of task i starting in the window $w_i(q)$ is given by:

$$M_i n_i Ci + m_i C_i \tag{6}$$

Consider now the computation occurring in the window $w_i(q)$ from higher priority sporadically periodic tasks j. If the window $w_i(q)$ is larger than a number of 'bursts' of j then the computation from each burst amounts to $n_j C_j$. For the partial 'burst' starting in $w_i(q)$ we can treat j as a simple periodic task executing with period t_j over the remaining part of $w_i(q)$. The whole number of 'bursts' starting and finishing in $w_i(q)$ is given by:

$$F_j = \left\lfloor \frac{J_j + w_i(q)}{T_j} \right\rfloor$$

the remaining part of the window $w_i(q)$ is of length $J_j + w_i(q) - F_j T_j$, and hence a bound on the number of invocations of j in this remaining time is:

$$\left\lceil \frac{J_j + w_i(q) - F_j T_j}{t_j} \right\rceil$$

Another bound on the number of invocations in this time is n_j, since a burst can consist of at most n_j invocations of task j. Therefore the least upper bound can be taken:

$$\min\left(n_j, \left\lceil \frac{J_j + w_i(q) - F_j T_j}{t_j} \right\rceil \right)$$

Thus the total computation from all higher priority tasks *j* occurring in the window $w_i(q)$ is:

$$\sum_{\forall j \in hp(i)} \left(\min\left(n_j, \left\lceil \frac{J_j + w_i(q) - F_j T_j}{t_j} \right\rceil \right) + F_j n_j \right) C_j \qquad (7)$$

and hence the theorem is derived.

Note that when evaluating the worst-case response time for a task *i* the iteration can stop as soon as $w_i(q) \le M_i T_i + m_i t_i - J_i$

6. TICK SCHEDULING

Tick scheduling is a common way of implementing a priority pre-emptive scheduler: a periodic clock interrupt runs a scheduler which polls for the arrivals of tasks; any arrived tasks are placed in a notional priority ordered run-queue. The scheduler then dispatches the highest priority task on the run-queue. Sporadic tasks can arrive at any time, and hence suffer a worst-case release jitter of T_{clk}, the period of the tick scheduler. Periodic tasks, in the general case, can also arrive at any time. However, a task with a period that is a multiple of T_{clk}, and defined to arrive at a tick interrupt, has a release jitter of zero. Our motivation for polling for sporadics is a safety one: if sporadics were released by a processor interrupt initiated by the arrival condition of the sporadic then some mechanism must exist to prevent the interrupt re-occurring before the worst-case re-arrival time. If sporadics are polled then the scheduler can decide not to release a sporadic if the minimum inter-arrival time has been violated.

The overheads due to operating tick scheduling can be accurately determined by applying the same scheduling analysis as described in previous sections. Before applying the analysis we will briefly describe some important characteristics of certain tick scheduling implementations. A common way of implementing tick scheduling is to use two queues: one queue holds a deadline ordered list of tasks which are awaiting their start conditions (such as a start time for periodics, or a start event — such as a value in an I/O register — for sporadics); we denote this queue the *pending queue*. The other queue is a priority-ordered list of runnable tasks, denoted the *run queue*. At each clock interrupt the scheduler scans the pending queue for tasks which are now runnable and transfers them to the run queue.

When a running task terminates it executes a system call (on a 68000-family processor this is called a 'trap') to transfer the task back to the pending queue to await the start condition for the next invocation (of course, for a task with deadline greater than period the start condition may already be true, and the task can be returned immediately to the run queue). The overheads due to this call can be allowed for by treating the call as a critical section of the calling task, guarded by a semaphore with a ceiling equal to the highest priority task in the system.

Most implementations of such a queuing system have the following characteristic: the computation cost to take a task from the pending queue to the run queue is lower if more than one task is taken at the same time. For example, in one implementation of a run-time system at York [4] the worst-case cost to handle a timer interrupt is 66µs; the cost to take the first task from the pending queue is another 74µs (we denote this time C_{QL}). For each subsequent task removed (as part of the processing within the same interrupt) the cost is 40µs (we denote this time C_{QS}). This is because there are one-off costs associated with setting up loops, *etc*. These costs are worst-case computation time costs with the processor cache disabled. We now develop analysis to accurately account for these overheads.

The costs of the periodic timer interrupt can be modelled as a simple task, with worst-case computation time $C_{clk}$and period T_{clk}; within a time window of width *w* the worst-case number of timer interrupts is given by:

$$L = \left\lceil \frac{w}{T_{clk}} \right\rceil$$

Now, within the same window the worst-case number of times tasks move from the pending queue to the run queue is (from equation 7) given by:

$$K = \sum_{\forall j \in tasks} \min\left(\left\lceil \frac{J_j + w - T_j F_j}{t_j} \right\rceil, n_j \right) + n_j F_j$$

If, over a window of width w, the total number of task queue moves, K, is less than the number of clock interrupts L, then in the worst-case all of the queue manipulations are full cost (*i.e.* each taking a worst-case computation time of C_{QL}), and the full cost of tick scheduling overheads is:

$$LC_{clk} + KC_{QL}$$

If, over w, K is greater than L then only the first L task queue moves are at the full cost C_{QL}; the remaining $K - L$ require only C_{QS} each. Hence the tick scheduling overheads for a task i over a window of width w are:

$$LC_{clk} + \min(L, K)C_{QL} + \max(K - L, 0)C_{QS}$$

7. Optimal Priority Ordering

As has been mentioned before neither the deadline monotonic nor rate monotonic priority ordering policies are optimal[2] for tasks with arbitrary deadlines. We reproduce here the optimal priority ordering algorithm of Audsley [1]. The algorithm works as follows: a priority ordering is partition into two parts: a sorted part, consisting of the lower n priority tasks, and the remaining unsorted higher priority tasks. Initially the priority ordering is an arbitrary one, and all tasks are unsorted. All tasks in the unsorted partition are chosen in turn and placed at the top of the sorted partition and tested for schedulability. If the chosen task is schedulable then the priority of the task is left as it is, and the sorted partition extended by one position. If the task is not schedulable it is returned to its former priority. This continues until either all tasks in the unsorted partition have been checked and found to be unschedulable (in which case there is no priority ordering resulting in a schedulable system), or else the sorted partition is extended to the whole priority map (in which case the priority ordering is a feasible one).

An arbitrary priority ordering is chosen in an array, with 0 being the highest priority, and $N - 1$ the lowest (N denotes the number of tasks in the system; the algorithm assumes $N > 1$). The following pseudo-code details the algorithm:

```
ordered := N
repeat
   finished := false
   failed := true
   j := 1
   repeat
       insert j at priority ordered
       if j is schedulable then
           ordered := ordered - 1
           failed := false
           finished := true
       else
           insert j back at old priority
       end if
       j := j + 1
   until finished or j = ordered
until Ordered = 1 or failed
```

[2]optimal in the sense that if the algorithm is unable to find a priority ordering where all tasks are schedulable then no priority ordering exists where all tasks are schedulable

At all times the sorted partition is schedulable, since the priority ordering within the unsorted partition cannot affect the sorted tasks. The sorted partition increases in size until either all the tasks are schedulable, or none of the top *n* tasks are schedulable at priority *n*. The analysis has the property that decreasing the priority of a task cannot lead to a decrease in worst-case response time (*i.e.* a decrease in priority cannot increase schedulability). Therefore, in the case where none of the top *n* tasks is schedulable at priority *n* no priority ordering can exist where all tasks are schedulable. Therefore the algorithm must be considered optimal. Furthermore, this algorithm holds for any scheduling test where worst-case response time is monotonic with decreasing priority (*i.e.* where decreasing the priority of a task does not lead to a decrease in the worst-case response time of that task).

8. SUMMARY

We reproduce here the equations for the full schedulability test:

The worst-case response time of a task *i* is given by:

$$r_i = \max_{q=0,1,2,3,\ldots} \left(w_i(q) + J_i - m_i t_i - M_i T_i \right)$$

The iteration over values of q can stop as soon as $w_i(q) \le M_i T_i + m_i t_i - J_i$

M_i is given by:

$$M_i = \left\lfloor \frac{q}{n_i} \right\rfloor$$

and m_i is given by:

$$m_i = q - M_i n_i$$

$w_i(q)$ is given by:

$$w_i(q) = (M_i n_i + m_i + 1)C_i + B_i + \sum_{\forall j \in hp(i)} \left(\min\left(n_j, \left\lceil \frac{J_j + w_i(q) - F_j T_j}{t_j} \right\rceil \right) + F_j n_j \right) C_j$$
$$+ LC_{clk} + \min(L, K)C_{QL} + \max(K - L, 0)C_{QS}$$

where $w_i(q)$ can be found by iteration as for equation 1, with $w_i(q)^0 = w_i(q-1)$, and $w_i(0)^0 = 0$

F_j is given by:

$$F_j = \left\lfloor \frac{J_j + w_i(q)}{T_j} \right\rfloor$$

K is given by:

$$K = \sum_{\forall j \in tasks} \min\left(\left\lceil \frac{J_j + w_i(q) - T_j F_j}{t_j} \right\rceil, n_j \right) + n_j F_j$$

and L is given by:

$$L = \left\lceil \frac{w_i(q)}{T_{clk}} \right\rceil$$

Other symbols used in the above equations are defined in the glossary given in Appendix A. The schedulability of a task *i* can be assessed by comparing with the deadline:

$$r_i \leq D_i$$

An optimal priority ordering can be obtained using the algorithm described in the previous section.

9. EXAMPLE TASK SET

The following tables summarise an example task set, based upon the GAP task set described by Locke *et al* [13]. The tasks are listed in priority order, as found by the optimal priority ordering algorithm described earlier. All times are in microseconds. Task 11 is a sporadic task, whose arrival is polled for by the tick scheduler, and therefore has non-zero release jitter. All other tasks are strictly periodic, with periods that are multiples of T_{clk}, and hence do not suffer release jitter. Tasks lock and unlock semaphores according to the following pattern:

semaphore	*locked by*	*time held*
2	task9	300
4	task9	300
1	task9	900
2	task15	1350
3	task10	400
3	task6	400
4	task3	100
5	task11	750
5	task15	750

When the task set is assigned a deadline monotonic priority ordering, the worst-case response times are as follows:

ID	T_i	t_i	n_i	C_i	B_i	J_i	D_i	r_i
task1	200000	200000	1	3000	0	0	5000	4180
task2	25000	5000	3	700	0	0	5000	4880
task3	25000	5000	3	1400	300	0	12000	7660
task4	40000	40000	1	1000	300	0	40000	12740
task5	50000	50000	1	3000	300	0	50000	16140
task6	50000	50000	1	5000	400	0	50000	21706
task7	59000	59000	1	8000	400	0	59000	37506
task8	80000	80000	1	9000	400	0	80000	48306
task9	80000	80000	1	2000	1350	0	100000	78450
task10	100000	100000	1	5000	1350	0	115000	**117708**
task11	200000	200000	1	1000	1350	1000	200000	142184
task12	200000	200000	1	3000	1350	0	200000	144382
task13	200000	200000	1	1000	1350	0	200000	145448
task14	200000	200000	1	1000	1350	0	200000	146514
task15	200000	200000	1	3000	0	0	200000	148296
task16	1000000	1000000	1	1000	0	0	1000000	149362
task17	1000000	1000000	1	1000	0	0	1000000	195330

Tasks are listed in the table in priority order. As can be seen, task 10 misses its deadline (the worst-case response time of this task is marked in bold type). When Audsley's optimal priority ordering algorithm is applied, and the analysis given in this paper used, the following results are obtained:

ID	B_i	r_i
task2	0	1580
task1	0	4880
task3	300	7660
task8	300	21606
task7	300	34960
task4	300	38472
task6	400	45108
task5	400	48306
task10	300	96306
task9	1350	99554
task17	1350	141184
task16	1350	142250
task15	750	144782
task14	750	145848
task13	750	146914
task12	750	195080
task11	0	196330

Again, tasks are listed in the table in priority order. As can be seen, the worst-case response times of all tasks are less than the deadlines, and hence the task set is schedulable. Appendix B details how to obtain the program that implemented the analysis given in this paper.

10. Conclusions

This short paper has shown how the window approach to finding worst-case response times can give powerful scheduling theory tailored to the behaviour of real real-time systems. To use the window approach, all that need be done is to find the worst-case amount of higher priority computation starting in a given window, and examine a sufficient number of windows. This approach has led to schedulability analysis for complex task behaviour, such as mode changes [7] and task offsets [6]; the latter analysis has been shown to be a superset of static priority cyclic scheduling. The overheads due to operating a tick driven scheduler have also been accurately accounted for. The analysis derived in this paper permits tasks to be assigned arbitrary deadlines, suffer release jitter, and behave as sporadically periodic tasks.

11. References

[1] Audsley, N. C., "*Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times,*" Dept. Computer Science, University of York (December 1991).

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

[3] Baker, T. P., "*Stack-Based Scheduling of Realtime Processes,*" Real Time Systems 3(**1**) (March 1991).

[4] A.D. Hutcheon, "Timings of Run-time Operations in Modified York Ada," Task 8 Volume C, Deliverable on ESTEC Contract 9198/90/NL/SF, York Software Engineering Limited, University of York (July 1992).

[5] Joseph, M., P. Pandya, "*Finding Response Times in a Real-Time System,*" BCS Computer Journal (Vol. 29, No. 5, Oct 86) pp.390-395.

[6] K. Tindell, "Using Offset Information to Analyse Static Priority Pre-Emptively Scheduled Task Sets," YCS 182, Dept of Computer Science, University of York (August 1992d).

[7] K. W. Tindell, A. Burns and A. J. Wellings, “Mode Changes in Priority Pre-emptively Scheduled Systems,” Proceedings 13th IEEE Real-Time Systems Symposium (2-4 December 1992).

[8] Lehoczky, J. P., “*Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines*,” Proceedings 11th IEEE Real-Time Systems Symposium (5-7 Decmeber 1990) pp.201-209.

[9] Lehoczky, J., L. Sha and Y. Ding, “*The Rate Monotonic Scheduling Algorithm: Exact Characterisation and Average Case Behaviour*,” Proceedings of the Real-Time Systems Symposium (1989).

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

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

[12] Locke, C.D., “*Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives*,” Real-Time Systems 4(**1**) (March 1992) pp.37-53. Real-Time Syst. (Netherlands).

[13] Locke, C. D., D. R. Vogel and T. J. Mesler, “*Building a Predictable Avionics Platform in Ada: A Case Study*,” Proceedings of the 12th Real Time Systems Symposium (December 1991).

[14] Rajkumar, R., L. Sha and J. P. Lehoczky, “*Real-Time Synchronization Protocols for Multiprocessors*,” Proceedings of the IEEE Real-Time Systems Symposium (1988) pp.259-269.

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

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

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

APPENDIX A — GLOSSARY OF NOTATION

The following table gives the notation used in this paper:

i	Task i denotes the task for which we are trying to find the worst-case response time
j	Task j generally denotes a task of higher priority than i
tasks	The set of all tasks in the system
$hp(i)$	The set of all tasks of higher priority than task i
C_i	The worst-case computation time of task i
D_i	The static deadline of task i, measured relative to the arrival time of the task
B_i	The worst-case blocking time of task i, found according to the priority ceiling protocol
J_i	The worst-case release jitter of task i (*i.e.* the worst-case delay between a task arriving and being released)
T_i	The outer period of task i
t_i	The inner period of task i
n_i	The worst-case number of arrivals of task i per outer period
r_i	The worst-case response time of task i, measured from the arrival time to the completion time
C_{QL}	The worst-case computation time required to move a single task from the delay queue to the run queue
C_{QS}	The worst-case computation time to move each task from the delay queue to the run queue
T_{clk}	The period of the clock timer interrupt
C_{clk}	The worst-case computation time required to service the periodic clock timer interrupt
w	Generally a computation ‘window’ used to find worst-case response times

APPENDIX B — AN ANALYSIS PROGRAM

The source code to a program that implements the analysis described in this paper can be obtained via anonymous FTP from:

`minster.york.ac.uk`
(IP address: `144.32.128.41`)

in the directory:

`pub/realtime/programs/src/arbdead`