# Applying New Scheduling Theory to

# Static Priority Pre-emptive Scheduling

*N. Audsley*
*A. Burns*
*M. Richardson*
*K. Tindell*
*A. J. Wellings*

Department of Computer Science
University of York
England

ABSTRACT

The paper presents exact schedulability analyses for real-time systems scheduled at run-time with a static priority pre-emptive dispatcher. The tasks to be scheduled are allowed to experience internal blocking (from other tasks with which they share resources) and (with certain restrictions) release jitter — such as waiting for a message to arrive. The analysis presented is more general than that previously published, and subsumes, for example, techniques based on the Rate Monotonic approach. In addition to presenting the theory, an existing avionics case study is described and analysed. The predictions that follow from this analysis are seen to be in close agreement with the behaviour exhibited during simulation studies.

## 1. INTRODUCTION

One proposed method of building a hard real time system is from a number of periodic and sporadic tasks, and a common way of scheduling such tasks is by using a static priority pre-emptive scheduler — at run-time the highest priority runnable task is run, pre-empting other lower priority tasks. This scheme was employed in the Rate Monotonic approach defined by Liu and Layland[12] where the (unique) static priority of a task is obtained from the period of that task — for any two tasks L and M, period(L) > period(M) => priority(L) < priority(M). Liu and Layland derived schedulability analysis which determines if a given task set will always meet all deadlines under all possible release conditions. The schedulability test given is *sufficient* (*i.e.* all task sets passing the test are guaranteed to be schedulable), but *not necessary* (*i.e.* a task set failing to pass the test is not necessarily unschedulable). Sha, Ding, and Lehoczky provided an exact Rate Monotonic test that is both sufficient and necessary[8].

The original Rate Monotonic approach had a number of restrictions:

- all tasks are independent of each other (e.g. they do not interact)
- all tasks are periodic
- no task can block waiting for an external event
- all tasks share a common release time (called the *critical instant)*
- all tasks have a deadline equal to their period.

The restriction that tasks cannot interact has been removed by the Priority Ceiling protocol[17] (and other similar protocols such as the Stack Resource Protocol[5] ). A method which allows sporadic tasks to be accommodated using periodic *servers* has been proposed by Lehoczky *et al*[9] (analysis is provided which can guarantee the worst-case response time of a single sporadic task). Rajkumar[16] used external blocking (*i.e.* when a task is blocked awaiting an external event, such as a delay expiry) with the Rate Monotonic approach to model the operation of a multi-processor Priority Ceiling protocol, and provided schedulability analysis to bound its effects. The restriction that tasks are assumed to share a critical instant has been relaxed by Audsley[4].

The final restriction that the deadline of a task must be equal to the period has not been relaxed for the Rate Monotonic approach. This is perhaps the most important restriction to lift: requiring the deadline of a task to be less than the period of that task is essential if *jitter* requirements are to be met (*i.e.* the result of a piece of computation must be produced within precise intervals); furthermore, when building distributed systems the deadline of a task often needs to be shortened to allow time for communication between tasks on different processors. In general hard sporadic tasks have deadlines that are not related to their minimum inter-arrival time, and hence they cannot be modelled as simple periodic tasks with period equal to deadline.

For tasks with deadlines less than (or equal) to periods, Leung and Whitehead [11] showed that Deadline Monotonic priority assignment is optimal†. Task priorities are now assigned in inverse order to task deadlines — a task with a short deadline (measured relative to the release time of the task) should have a high priority. A task with a long period but short deadline would have a low priority according to the Rate Monotonic priority assignment, but a high priority according to the Deadline Monotonic priority assignment. Consequently, the Rate Monotonic assignment will be sub-optimal for such task sets. If two or more tasks have the same deadline then they are assigned an arbitrary priority order (amongst themselves).

To apply the Deadline Monotonic approach, scheduling tests must be available which will allow deadlines to be guaranteed. Such analysis is provided by Joseph and Pandya[6], Lehoczky[10], and Audsley *et al*[1]. All provide sufficient and necessary schedulability tests, differing only in the complexity of their computations. The basic approach is expanded by Audsley *et al* to permit sporadic task deadlines to be guaranteed without the use of the servers required by the Rate Monotonic approach[9]. It should be pointed out that both Audsley *et al*, and Joseph and Pandya, provide schedulability tests for a task set with any arbitrary priority ordering (i.e. they do not just apply to task sets with priorities ordered by the Deadline Monotonic scheme). They also have the useful property that they furnish estimations of the actual worst case response times for each task. The actual schedulability test is then a trivial comparison of each task’s response time and deadline. The calculation of response time is particularly important when deadline requirements are assigned to the behaviour of a collection of tasks (in, for example, a distributed system). No one task has a hard "deadline", but each task’s response time contributes to some system-wide timing requirement that must be satisfied.

This paper is concerned with providing schedulability analysis to predict the worst-case response times for a set of periodic and sporadic tasks under any given priority assignment, and scheduled by a static priority pre-emptive scheduler. Section 2 describes the computation model assumed by the analysis. Section 3 provides analysis for guaranteeing the worst-case response times of periodic and sporadic tasks. This section also includes a glossary of notation. Section 4 modifies the analysis to take account of the so-called ‘release jitter’ problem (a special case of the general external blocking problem). Section 5 shows how the general approach is easily extended to more complex scenarios. Section 6 discusses the analysis and applies it to a small avionics case-study. Conclusions are offered in Section 7.

---

† optimal is the sense that if a task set can be scheduled by any static priority algorithm it can also be scheduled by the Deadline Monotonic algorithm.

## 2. COMPUTATIONAL MODEL

In this paper, unless explicitly mentioned, we consider the scheduling of tasks on a single processor. The techniques can also be used in a distributed environment with static task allocation[18].

A *task* $i$ is assumed to consist of an infinite number of *invocation requests*, each separated by a minimum time $T_i$. Each invocation is a request to perform a bounded amount of computation $C_i$, and to lock and unlock semaphores from a bounded set $s(i)$ according to the Priority Ceiling Protocol[17]. Some tasks will have a deadline requirement such that all computation for an invocation must take place before a certain time measured relative to the invocation request. Deadlines, where required, are assumed to be constant and known *a priori*. The deadline requirement of a task $i$ is denoted $D_i$.

The notional *arrival* of a task at time $t$ will subsequently be recognised by the run-time dispatcher and the task will be placed in a notional queue of runnable tasks. The task is then said to be *released*. The time between a task's arrival and its release is known as *release jitter*. In Section 3 this will be assumed to be zero. This is the assumption normally applied in scheduling theory (e.g. in the Rate Monotonic approach).

A task has a static *base* priority assigned to it *a priori* (using, for example, the Deadline Monotonic priority assignment algorithm). It may also inherit a higher *dynamic* priority due to the operation of the Priority Ceiling Protocol. The dispatcher chooses to run the highest dynamic priority runnable task, pre-empting lower priority tasks when necessary.

## 3. FINDING WORST-CASE RESPONSE TIMES

Before proceeding further we introduce some simple notation in a glossary below. All internal blocking is assumed to be the result of semaphore use (other synchronisation primitives also could be used and analysed[2] ).

| | |
|---|---|
| $C_i$ | The worst-case computation time required by task $i$ on each release. At run-time we assume that any computation time from 0 to $C_i$ could be required for a single invocation of $i$ |
| $T_i$ | The lower bound on the time between successive arrivals of $i$. If $i$ is a periodic task then this lower bound will also be the upper bound (*i.e.* the period is fixed and equal to $T_i$) |
| $D_i$ | The deadline *requirement* of task $i$, measured relative to a given release of $i$. Note that we require $D_i \le T_i$. |
| $B_i$ | The worst-case blocking time task $i$ can experience due to the operation of the priority ceiling protocol (or equivalent concurrency control protocol). $B_i$ is normally equal to the longest critical section of lower priority tasks accessing semaphores with ceilings higher than (or equal to) the priority of $i$. |
| $J_i$ | The worst-case time task $i$ can spend waiting to be released after arrival (the *release jitter time*). |
| $I_i$ | The worst-case interference a task $i$ can experience. Interference on $i$ is defined as the time higher priority tasks can pre-empt and execute, and hence prevent $i$ from executing. |
| $r_i$ | The worst-case response time for a task $i$ measured from the time the task is released. For a schedulable task $r_i \le D_i$ (if there was no deadline requirement for $i$ we would require that $r_i \le T_i$). |

$hp(i)$ The set of tasks of higher base priority than the base priority of $i$ (these tasks could pre-empt $i$).

We now turn to the problem of computing the worst-case response time for a task $i$, denoted $r_i$. Initially tasks are assumed to be released when they arrive. This time can be viewed as a computational 'window': the release of $i$ marks the start of the window, and the completion of $i$ marks the end of the window. The maximum width of the 'window' is $r_i$. In this 'window' of duration $r_i$, task $i$ must (at worst) complete an amount of computation equal to $C_i$, and be delayed when locking semaphores by at most $B_i$. Additionally, task $i$ could be pre-empted by at most $I_i$. Therefore we can say that:

$$r_i = C_i + B_i + I_i \tag{1}$$

If a task $i$ has a deadline then we must have $r_i \le D_i$.

The worst-case computation time $C_i$ is constant and known *a priori* by some means[14, 15]. The worst-case blocking time $B_i$ is found according to the analysis given by Sha *et al*[17], and is equal to the longest critical section of any lower priority task accessing a semaphore with ceiling of equal or higher priority than task $i$.

The rest of this section presents analysis to find $I_i$. Note that similar analysis, cast in a different form, was first produced by Joseph and Pandya[6], and later by Audsley *et al*[1]. The method described below has the advantage that it is easily extended to cover situations such as non-zero release jitter time. Note also that the analysis is not based on the notion of processor utilisation. Although process sets with deadline equal to period can be assessed according to their utilisation, such techniques are not general purpose. For example, two tasks with deadlines equal to their computation time will never be schedulable, regardless of processor utilisation.

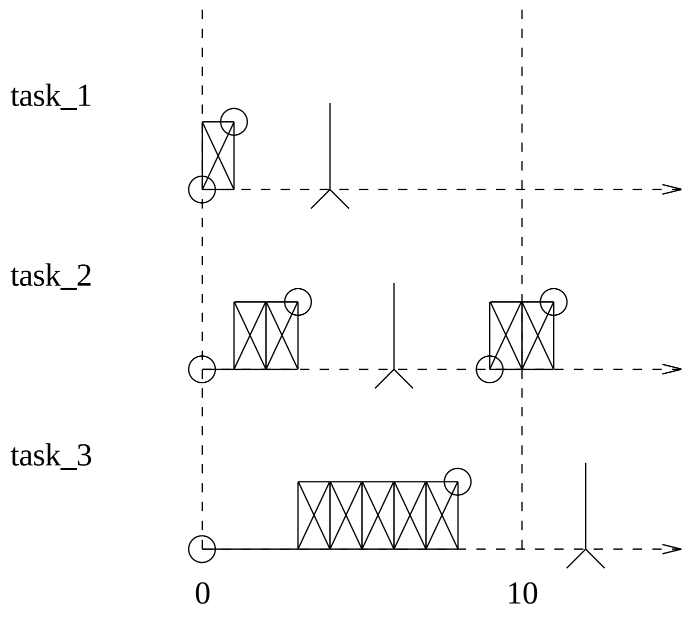


Figure 1

To find a formulae for the interference consider the sequence of computations illustrated in Figure 1. The diagram was produced by a tool called *STRESS*[3] written by the Real Time Systems Research Group at York; Appendix 1 describes the notation used in these diagrams. The diagram shows part of a schedule of a system consisting of three tasks, displayed in priority order. Task 1 is a task with worst-case computational requirement of $C_1 = 1$, a deadline of $D_1 = 4$, and a worst-case inter-arrival time of $T_1 = 50$. Tasks 2 and 3 have their characteristics defined in Table 1. Figure 1 shows the worst-case scheduling point described by Liu and Layland, where all tasks are released simultaneously (at time 0).

| Task | C | T | D |
|---|---|---|---|
| Task 1 | 1 | 50 | 4 |
| Task 2 | 2 | 9 | 6 |
| Task 3 | 5 | 20 | 12 |

Table 1

As can be seen from the diagram, task 3 is prevented from executing by task 1 for 1 tick and task 2 for 2 ticks, completing by time 8, giving $r_3 = 8$. Task 2 can never pre-empt task 3 more than once since task 3 finishes before task 2 can re-arrive (*i.e.* $r_3 \le T_2$).

If task 3 were to take a little longer to complete (because, say, task 1 executes for an extra two ticks) then a first guess at $r_3$ would be $8 + 2 = 10$. However, now that task 3 is a little longer, task 2 can re-arrive and pre-empt task 3 a second time, giving a worst-case interference of 4 from task 2. Figure 2 shows this situation.

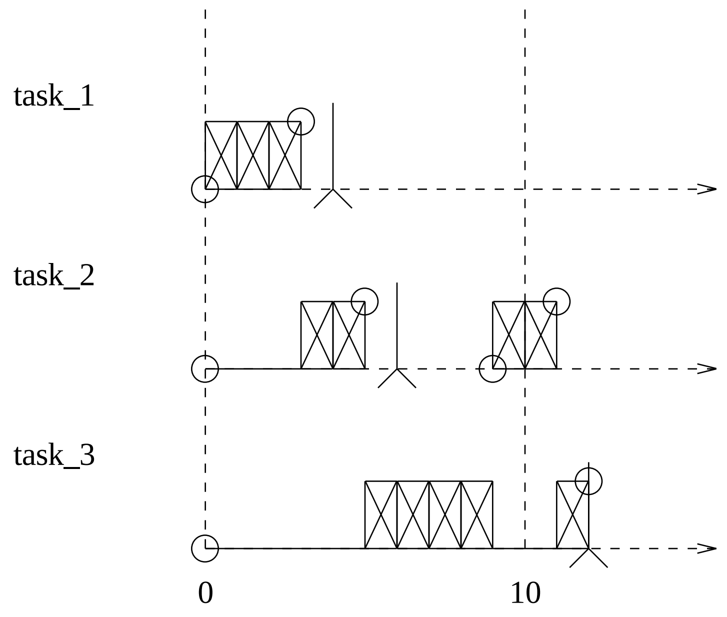


Figure 2

As can be seen from Figure 2 the worst-case response time of task 3 is now 12, just meeting its deadline.

In general, given prior knowledge of the worst-case response time $r_i$, the interference on task $i$ from a task $j$ is $nC_j$ where $n$ has a value such that $(n-1)T_j < r_i \le nT_j$.

Since $\lceil x \rceil = n$ when $n-1 < \lceil x \rceil \le n$ we can say that the worst-case interference from a task $j$ on task $i$ is given by:

$$\left\lceil \frac{r_i}{T_j} \right\rceil C_j$$

Note that this value for the maximal interference holds regardless of whether $j$ is periodic or sporadic. This is an important result since it means that run-time techniques such as aperiodic servers [9] are not needed. In fact, a periodic task can be regarded as a sporadic task, released by a regular timing event.

The total interference $I_i$ is given by:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil C_j \tag{2}$$

where $hp(i)$ is the set of tasks with higher base priorities.

Unfortunately, when equations (1) and (2) are combined, the unknown term $r_i$ appears on both the left and the right hand sides of the equation:

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

It is possible to solve this equation using an iterative technique. Let $r_i^n$ be the $n^{th}$ approximation to the true value of $r_i$. These approximations are generated from the above equation:

$$r_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \tag{3}$$

The iteration starts with $r_i^0 = 0$, and terminates when $r_i^{n+1} = r_i^n$. It can easily be shown that $r_i^{n+1} \geq r_i^n$ and so the iteration can be halted early if either $r_i^{n+1} > D_i$ or if $r_i^{n+1} > T_i$. It can also be shown that the iteration is guaranteed to converge if the processor utilisation is $\leq 100\%$[6]. Equation (3) can be embodied into a software tool that analyses a task set. Note that if the priority of task $a$ is greater than the priority of task $b$ then $r_b > r_a$; thus the task set should be analysed in priority order, with the starting value $r_b^0$ set to $r_a$ — this will enable the test to be evaluated more quickly.

Note, in this analysis we have not made use of any information about priority assignment. Both the Rate Monotonic policy and Deadline Monotonic policy could be used. In more complex situations, for example in a distributed system with complex tradeoffs, finding an optimal priority ordering may be NP-Hard, and other sub-optimal techniques such as Simulated Annealing[7] are appropriate[18].

## 4. THE RELEASE JITTER PROBLEM

In this section we show how *release jitter* causes problems with the analysis presented so far. We then show how the analysis presented can be extended to allow for such *external* blocking (and indicate how this type of blocking is often encountered in real systems).

The release jitter problem arises when we change the assumption that a task is always released as soon as it arrives. With release jitter, a task may be released at any time up to a bounded time after it arrives, denoted $J_i$. This can occur if, for example, the scheduler mechanism takes a bounded time to recognise the arrival of a task.

The analysis presented in the previous section is not sufficient when tasks can experience release jitter. Consider the task set defined in Table 2.

| Task | C | T | D | J |
|---|---|---|---|---|
| T1 | 3 | 12 | 8 | 4 |
| T2 | 6 | 20 | 10 | 0 |

Table 2

Task T1 is of higher priority than task T2. In this example we are concerned with the schedulability of T2. T1 experiences the external block because, say, it needs a message before it can commence. The message is sent at the same time as T1 arrives (T1 could, for example, be a sporadic task with the arrival triggered by some external event which also triggers the sending of a message from another processor). The message is guaranteed to arrive no later than 4 ticks after the arrival of T1, and hence we have a release jitter of $J_1 = 4$.

Using our current analysis we have (i.e. ignoring release jitter):

$$r_2^0 = 0$$

$$r_2^1 = C_2 + \left\lceil \frac{r_2^0}{T_1} \right\rceil C_1 = 6$$

$$r_2^2 = C_2 + \left\lceil \frac{r_2^1}{T_1} \right\rceil C_1 = 9$$

$$r_2^3 = C_2 + \left\lceil \frac{r_2^2}{T_1} \right\rceil C_1 = 9$$

The equation has converged, and hence $r_2 = 9$.

Since $r_2 \leq D_2$ T2 would be deemed schedulable. The following diagram shows a schedule for the two tasks when both are released together (the Liu and Layland worst-case).

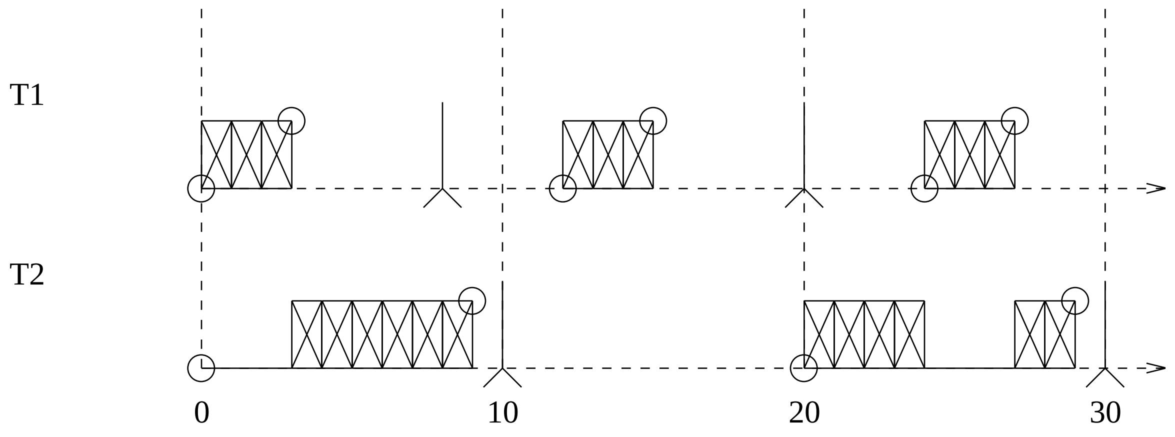


Figure 3

However, when release jitter is taken into account, there are situations when T2 will not always be schedulable. The following diagram shows such a situation:

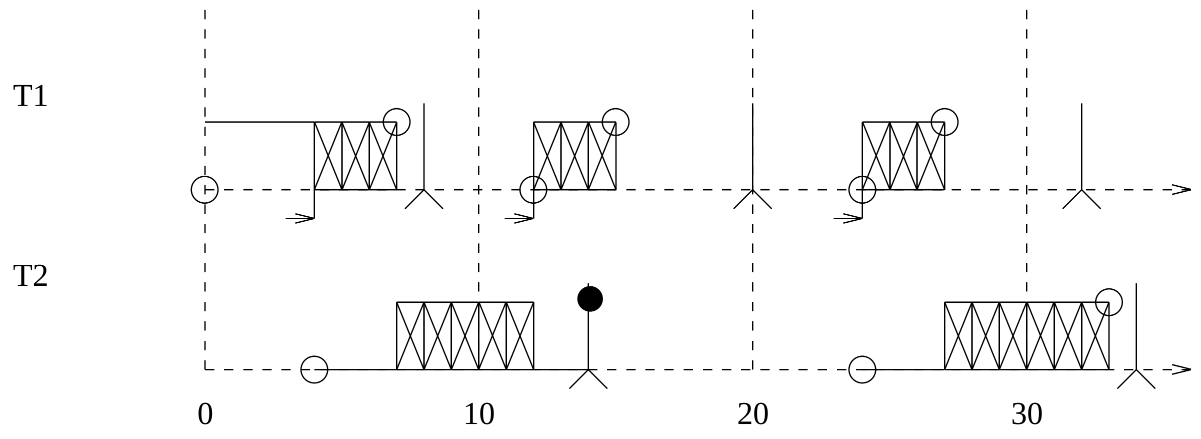


Figure 4

T1, although released at time zero, is suspended awaiting a message, which it receives at time 4 (this is also the time T2 arrives and is released). On the next release of T1, 12 ticks later, the next message is already available (it could have arrived in effectively zero time) and so the task can be released immediately. T2 misses a deadline (indicated by the large black 'blob' in the STRESS diagram) because of the release jitter of T1. The reason is that the worst-case scheduling point no longer occurs at the Liu and Layland critical instant (where all tasks are released together), but at the point when T2 is released at the same time as T1 finishes waiting — T1 can then effectively re-occur in a shorter time than the current analysis allows for, and so inflict a 'back to back hit'.

This phenomenon is described by Rajkumar[16] with reference to external blocking when locking remote semaphores in a distributed system — Rajkumar refers to this as an invasive effect due to deferred execution. This extra 'hit' can amount to at most an additional interference of $C_1$. The reason the current analysis fails is because the interference factor $I_i$ is not sufficient. An upper bound on the interference to allow for the extra 'hit' might therefore be obtained by simple adding in an extra computation time:

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

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

In effect we are saying here that an extra 'hit' occurs if $r_i + T_j > T_j$. This is pessimistic, since the extra 'hit' is not bound to occur in all systems. Consider again Figure 4. If $C_2$ were 5 ticks then $r_2$ would be 8, and all computation for T2 would be complete before T1 re-arrived and pre-empted T2. The extra 'hit' only occurs if $r_i + J_j > T_j$, and hence:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i + J_j}{T_j} \right\rceil C_j \qquad (4)$$

In Figure 4, $r_2$ would be 9, according to Equation (3). Since $r_2 + J_1 > T_1$ (or $9 + 4 > 12$), T2 gets an extra 'hit'. But if $C_2 = 5$ then $r_2$ would be 8, and since $8 + 4 \leq 12$, no extra 'hit' occurs.

Equation (3) can thus be modified to allow for release jitter:

$$r_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n + J_j}{T_j} \right\rceil C_j \tag{5}$$

Recall that $r_i$ is the worst-case response time measured from the point at which task $i$ is released. A more reasonable and useful measure might be from the time task $i$ *arrives*, so that the worst-case time from arrival to completion of task $i$ is given by:

$$J_i + r_i \tag{6}$$

Note that Equation (5) still allows semaphores to be locked and unlocked according to the Priority Ceiling Protocol.

Having extended the scheduling analysis to handle release jitter, we now indicate using two examples how this can occur in a system.

**Precedence Constrained Distributed Tasks**

A common method of representing computations in a distributed system is as a collection of tasks with precedence relationships between their executions. Each task is statically allocated to a single processor. Such task sets can be analysed with theory which assumes release jitter. All tasks are defined to arrive at the same time, but a precedence constrained task on one processor can have its release delayed awaiting an indication of termination of all direct predecessors on other processors (perhaps by the arrival of a message, in a similar way to the earlier example). The worst-case release jitter of such a subtask can be computed by knowing the worst-case response times of predecessor subtasks located on other processors, and by knowing the worst-case communications delay. By assuming a best-case response time of zero for the predecessors, and that best-case message transit times are zero, the release jitter (*i.e.* the variability in release) can be said to be the largest sum of the worst-case response time of each predecessor, computed by Equation (6), plus the worst-case transit time of the message sent by that predecessor, *i.e.*:

$$J_i = \max_{\forall k \in dpred(i)} \left( J_k + r_k + M_{k,i} \right)$$

where $dpred(i)$ is the set of all tasks which are direct predecessors of task $i$, and $M_{k,i}$ is the worst-case transit time of a message sent from task $k$ to task $i$.

Note that the above equation only holds if all the predecessors of task $i$ are on a different processor from task $i$; to allow predecessors to be on the same processor other analysis must be developed. For example, one approach is to assign a lower priority to task $i$ than the local predecessors to ensure that task $i$ never runs before a predecessor, and assign a release jitter of task $i$ such that it is greater than or equal to the release jitter of local direct predecessors (so that whenever a higher priority predecessor is deferred awaiting a message arrival, task $i$ is also deferred and hence prevented from running).

A more detailed analysis of distributed precedence constrained tasks is beyond the scope of this paper, and is the subject of current research.

**Tick Driven Scheduling**

The implementation of a priority scheduler can also introduce release jitter. Consider a single processor where periodic and sporadic tasks are scheduled by a scheduler which is invoked by a periodic clock interrupt — the so called 'tick driven scheduling'.

Assume the period of the scheduler is $T_{tick}$, and that the scheduler, once invoked, will take no more than $C_{tick}$ computation time. Consider the following sequence of events: the scheduler is released at time $t = 0$ and looks to see if the sporadic task $s$ is to be released (in a real tick-driven system the scheduler might poll an I/O register for the condition for the release of $s$). Assume the condition for the arrival is not true and the scheduler continues executing (ultimately terminating after taking time $C_{tick}$). Just after the time the scheduler has polled, the sporadic $s$ arrives (*i.e.* the condition becomes true). However, $s$ cannot be released until the scheduler is next invoked at time $t = T_{tick}$. Hence the sporadic is deferred for a maximum time $T_{tick}$, awaiting the timer which invokes the scheduler. The following STRESS diagram illustrates how a sporadic task is deferred by a tick-driven scheduler.

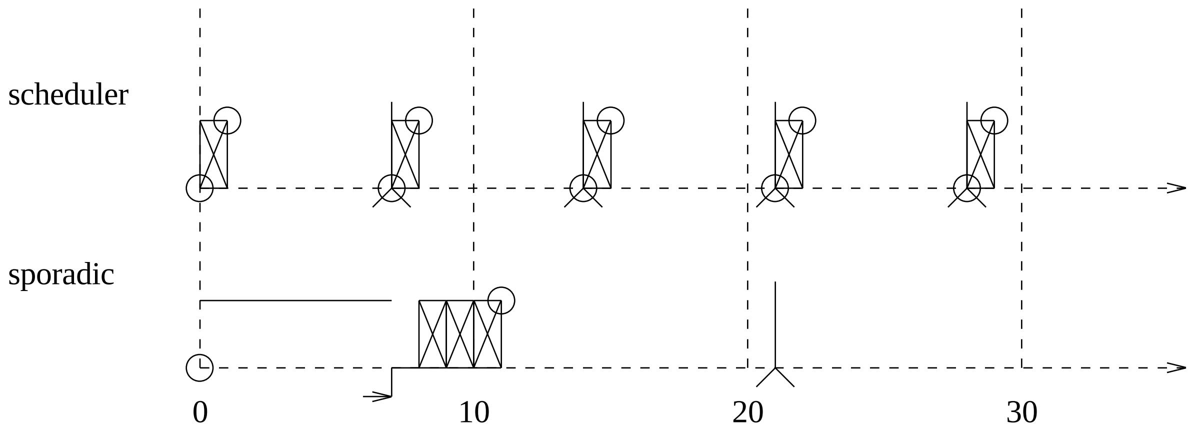


Figure 5

The tick-driven scheduler executes for $C_{tick} = 1$, with $T_{tick} = 7$. The worst-case execution time of the sporadic task is 3 time units. As can be seen, the sporadic task is deferred for 7 time units.

Tasks which always arrive as the scheduler is released do not experience external blocking — in the case study described later all periodic tasks have periods which are exact multiples of $T_{tick}$, with release times measured in scheduler ticks, and hence these tasks can be considered to always arrive as the scheduler is released. However a periodic task will experience release jitter if its period is not an integer multiple of the clock period.

## 5. SPORADICALLY PERIODIC TASKS

Another illustration of the strength of the analytical approach taken in this paper is to adapt the scheduling analysis to more accurately describe the behaviour of so-called ‘sporadically periodic tasks’. Very often a task will arrive at some time, execute, and then re-arrive periodically for a number of times, and then not re-arrive for a longer time. This is illustrated by the following STRESS diagram:

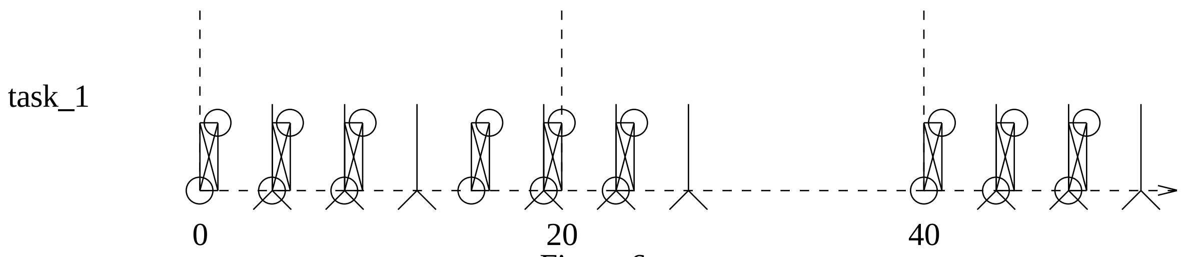


Figure 6

The task illustrated has an ‘inner’ period of 4 ticks, a minimum ‘outer’ period of 15 ticks, and a worst-case execution time of 1 tick. The task arrives periodically 3 times for each outer arrival. This behaviour is quite common in real systems — a task is initiated in response to some event, and then for a short period of time periodically monitors or controls some part of the system.

The model also caters for bursty sporadics. An interrupt, that releases a sporadic, may be defined to have a very short minimum arrival time but have the maximum number of arrivals over some larger interval. The maximum being much lower than the minimum interval would dictate. For example in a satellite control system (for which this scheduling model has been applied) bus interrupts can occur every 960μs up to a maximum of four. There must then be a gap of 10ms.

If the analysis developed so far were to be applied to these situations, the predictions would be pessimistic since the theory has to assume that the task executed continually. This might result in a higher assumed interference than could actually occur. However, the general analytical approach is well-suited to extending the current analysis to remove this pessimism.

Our general approach to ascertaining the schedulability of a task is to determine the interference over a given window (usually the worst-case response time of a task). This interference is summed, and the window widened if necessary. We require that a wider window always leads to a higher interference. Hence to ascertain the schedulability of a task $i$ in the presence of higher priority sporadically periodic tasks we need to find an upper bound on the interference over a window of size $r_i$. We adopt the following additional notation:

|  |  |
|---|---|
| $n_j$ | The number of times task $j$ executes for each 'outer' arrival (in Figure 6, $n = 3$). |
| $t_j$ | The 'inner' period of task $j$ (in Figure 6, $t = 4$). |
| $T_j$ | The 'outer' period of task $j$ (in Figure 6, $T = 15$). |
| $C_j$ | The worst case computation time required by the 'inner' task (in Figure 6, $C = 1$). |

For the moment we assume that tasks do not experience release jitter. The number of full outer periods completing within the window of size $r_i$ is bounded by:

$$\left\lfloor \frac{r_i}{T_j} \right\rfloor$$

The total interference due to full outer arrivals is therefore bounded by:

$$n_j \left\lfloor \frac{r_i}{T_j} \right\rfloor C_j \tag{7}$$

At most one partially complete outer arrival can interfere over the remaining part of the window not already accounted for by whole arrivals. This remaining time amounts to:

$$r_i - T_j \left\lfloor \frac{r_i}{T_j} \right\rfloor$$

and lies in the range $(0 .. T_j]$. We shall denote this value by $Q_{ij}$. The interference over this remaining time is bounded by:

$$\left\lceil \frac{Q_{ij}}{t_j} \right\rceil C_j \tag{8}$$

The above equation assumes that task $j$ executes as a continual periodic task (with period $t_j$) over the remaining interval. However, task $j$ cannot execute for more than $n_j$ periods in this interval (since the interval covers only a partially complete outer arrival), and another bound can be obtained:

$$n_j C_j \tag{9}$$

The least upper bound can therefore be used, *i.e.*:

$$\min\left(\left\lceil \frac{Q_{ij}}{t_j} \right\rceil, n_j\right) C_j \tag{10}$$

Combining Equations (10) and (7) and summing over all higher priority tasks we obtain:

$$I_i = \sum_{\forall j \in hp(i)} \left[ \min\left(\left\lceil \frac{Q_{ij}}{t_j} \right\rceil, n_j\right) + n_j \left\lfloor \frac{r_i}{T_j} \right\rfloor \right] C_j \tag{11}$$

If a task $j$ is not sporadically periodic then we choose $n_j = 1$ and $t_j = T_j$. As a check for Equation (11) we assume that all tasks are not sporadically periodic, and hence for all tasks $j$ $n_j = 1$ and $t_j = T_j$. From Equation (11) we have:

$$I_i = \sum_{\forall j \in hp(i)} \left[ \min\left(\left\lceil \frac{r_i - T_j \left\lfloor \frac{r_i}{T_j} \right\rfloor}{T_j} \right\rceil, 1\right) + \left\lfloor \frac{r_i}{T_j} \right\rfloor \right] C_j$$

$$= \sum_{\forall j \in hp(i)} \left[ \left\lceil \frac{r_i - T_j \left\lfloor \frac{r_i}{T_j} \right\rfloor}{T_j} \right\rceil + \left\lfloor \frac{r_i}{T_j} \right\rfloor \right] C_j$$

$$= \sum_{\forall j \in hp(i)} \left[ \left\lceil \frac{r_i}{T_j} - \left\lfloor \frac{r_i}{T_j} \right\rfloor \right\rceil + \left\lfloor \frac{r_i}{T_j} \right\rfloor \right] C_j$$

$$= \sum_{\forall j \in hp(i)} \left[ \left\lceil \frac{r_i}{T_j} \right\rceil - \left\lfloor \frac{r_i}{T_j} \right\rfloor + \left\lfloor \frac{r_i}{T_j} \right\rfloor \right] C_j$$

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

Which is equal to Equation (2). Hence Equation (11) is a generalisation of Equation (2).

We now return to the problem of release jitter. There are two potential places where release jitter could occur: on an outer arrival (where the first arrival of a succession of $n_j$ inner arrivals of a task $j$ is deferred), and on an inner arrival (where each of the $n_j$ arrivals could experience delay). For simplicity we assume that the outer arrival jitter and the inner arrival jitter are the same. For a task $j$ we assume that this jitter is denoted $J_j$. Following the same argument as for the derivation of jitter in Equation (5) we can modify Equation (11) to include release jitter:

$$I_i = \sum_{\forall j \in hp(i)} \left[ \min\left(\left\lceil \frac{J_j + r_i - T_j \left\lfloor \frac{J_j + r_i}{T_j} \right\rfloor}{t_j} \right\rceil, n_j\right) + n_j \left\lfloor \frac{J_j + r_i}{T_j} \right\rfloor \right] C_j \tag{12}$$

As with Equation (3) an iterative equation to find $r_i$ can be formulated. Again, the worst-case response time of a task, measured from arrival to termination, is given by $J_i + r_i$.

## 6. DISCUSSION AND CASE STUDY

In this section we analyse and discuss the task set of a small avionics case study undertaken by Locke *et al* and described in detail in their paper[13].

A number of mostly-periodic tasks implement an avionics weapons management subsystem. There is a single sporadic task, and a single task where the deadline of the task is less than the period (for reduced output 'jitter' requirements). Task priorities are assigned according to the Deadline Monotonic policy. Originally, the tasks were analysed using the Rate Monotonic schedulability analysis derived by Sha *et al*[8]. The study [13] reports that, using this analysis, out of a set of 18 tasks only the 8 highest priority tasks could be guaranteed to meet their deadlines. In simulations nearly all tasks were found to meet their timing requirements (two tasks are reported as missing their deadlines).

Equation (5) was applied to the task set described by Locke *et al*[13], using the given priority assignment. For the single sporadic task a release jitter of 1000 µs was assumed (to account for the worst-case delay due to the operation of the tick-driven scheduler — see earlier the discussion of induced jitter from tick-driven scheduling). The other tasks are all periodic with periods which are multiples of $T_{tick} = 1000$µs, and hence do not experience release jitter. Table 3 lists the tasks in priority order (task 1, the tick-driven scheduler, is the highest priority task), and gives the attributes and the derived response times, using Equation (5), of the tasks. All times given are in µs.

| $i$ | $C_i$ | $T_i$ | $D_i$ | $r_i$ | $B_i$ | $J_i$ |
|---|---|---|---|---|---|---|
| 1 | 51 | 1000 | 1000 | 51 | 0 | 0 |
| 2 | 3000 | 200000 | 5000 | 3504 | 300 | 0 |
| 3 | 2000 | 25000 | 25000 | 5906 | 600 | 0 |
| 4 | 5000 | 25000 | 25000 | 11512 | 900 | 0 |
| 5 | 1000 | 40000 | 40000 | 13064 | 1350 | 0 |
| 6 | 3000 | 50000 | 50000 | 16217 | 1350 | 0 |
| 7 | 5000 | 50000 | 50000 | 20821 | 750 | 0 |
| 8 | 8000 | 59000 | 59000 | 36637 | 750 | 0 |
| 9 | 9000 | 80000 | 80000 | 47798 | 1350 | 0 |
| 10 | 2000 | 80000 | 80000 | 48949 | 450 | 0 |
| 11 | 5000 | 100000 | 100000 | **115966** | 1050 | 0 |
| 12 | 1000 | 200000 | 200000 | 137488 | 450 | 1000 |
| 13 | 3000 | 200000 | 200000 | 140641 | 450 | 0 |
| 14 | 1000 | 200000 | 200000 | 141692 | 450 | 0 |
| 15 | 1000 | 200000 | 200000 | 143694 | 1350 | 0 |
| 16 | 3000 | 200000 | 200000 | 145446 | 0 | 0 |
| 17 | 1000 | 1000000 | 1000000 | 146497 | 0 | 0 |
| 18 | 1000 | 1000000 | 1000000 | 147548 | 0 | 0 |

Table 3

As can be seen from the table, our analysis expects that all deadlines can be met except for task 11. Locke *et al* found that task 11 did indeed miss its deadline occasionally. They also found that task 16 missed a deadline once. This discrepancy can be explained if the scheduler implementation does not exactly agree with the assumptions made in this paper.

The case study was implemented in Ada. Most Ada run-time systems make use of two queues; a *run-queue* which holds all runnable tasks and a *delay-queue* which holds all (periodic) task that are waiting for their next release. At any particular tick the number of tasks to be moved from the delay queue to the run-queue will vary between none and sixteen. A standard run-time system will not undertake this at a constant cost (in computation time); hence the value of $C_1$ of 51 μs is potentially an underestimation. Furthermore, the costs of context switches must be accounted for accurately, along with any blocking factors due to the operation of the system (for example, calls to the Ada run-time in most implementations are generally non-preemptable and hence can induce a blocking factor on all tasks). It is therefore unlikely that $B_{16}$ is actually zero. Equation (5) predicts a worst case response time for task 16 of 145446 μs which seems a long way from its deadline of 200000 μs. However if the above factors could increase the responses time by only 3.2% then this would push it over 150000 μs at which point it would suffer increased interference from tasks 3, 4, 6 and 7; and subsequently tasks 5, 9 and 10. This is sufficient for it to miss its deadline in the worst case. Without details of the exact implementation no fair comparison of the results of experiments and analysis can be made. In general, however, our analysis agrees with the observed behaviour. Moreover, it matches the observed behaviour more closely that the original Rate Monotonic analysis.

## 7. SUMMARY AND CONCLUSIONS

We have presented results in this paper which provide simple exact analysis for systems scheduled at run-time with a static priority pre-emptive dispatcher. The analysis has been extended to include release jitter, allowing tasks to arrive and then be deferred for a bounded amount of time. The analysis has been further extended to permit sporadically periodic tasks to be analysed exactly. A case study already presented elsewhere and analysed according to Rate Monotonic scheduling theory has been re-analysed using this theory. The basis of the analysis is the development of formulae which predict the worst-case interference a task can suffer from higher priority tasks; utilisation based analysis is not used as this cannot cater for systems which contain tasks with deadlines less than periods.

The two most important aspects of the scheduling theory presented here are: that older scheduling theory can be considered a special case of the analysis presented in this paper (systems previously analysed by the Rate Monotonic approach can now be re-analysed using more powerful techniques), and that the analysis presented here can be extended in a straightforward manner to allow more complex and powerful systems to be investigated.

## 8. ACKNOWLEDGEMENTS

The authors would like to thank the authors of the case study for help with the analysis of the case study and for comments on an earlier draft of this paper.

## 9. REFERENCES

1. N.C. Audsley, A. Burns, M.F. Richardson and A.J. Wellings, ‘‘Hard Real-Time Scheduling: The Deadline Monotonic Approach’’, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA (15-17 May 1991).
2. N. C. Audsley, ‘‘Resource Control For Hard Real-Time Systems: A Review’’, YCS 159, Department of Coputer Science, University of York (August 1991).
3. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘STRESS: A Simulator For

Hard Real-Time System'', RTRG/91/106, Real-Time Research Group, Department of Computer Science, University of York (October 1991).

4. N.C. Audsley, ``Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times'', YCS 164, Dept. Computer Science, University of York (December 1991).

5. T.P. Baker, ``Stack-Based Scheduling of Realtime Processes'', *Real Time Systems* **3**(1) (March 1991).

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

7. S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, ``Optimisation by Simulated Annealing'', *Science*(220), pp. 671-680 (1983).

8. J.P. Lehoczky, L. Sha and V. Ding, ``The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior'', Tech Report, Department of Statistics, Carnegie-Mellon (1987).

9. J.P. Lehoczky, L. Sha and J.K. Strosnider, ``Enhancing Aperiodic Responsiveness in Hard Real-Time Environment'', in *Proceedings 8th IEEE Real-Time Systems Symposium*, San Jose, California (December 1987).

10. J. P. Lehoczky, ``Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines'', *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, pp. 201-209 (5-7 Decmeber 1990).

11. J.Y.T. Leung and J. Whitehead, ``On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks'', *Performance Evaluation (Netherlands)* **2**(4), pp. 237-250 (December 1982).

12. C.L. Liu and J.W. Layland, ``Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment'', *JACM* **20**(1), pp. 46-61 (1973).

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

14. C.Y. Park and A.C. Shaw, ``Experiments with a program timing tool based on source-level timing schema'', *Computer* **24**(5), pp. 48-57, Computer (USA) (May 1991).

15. P. Puschner and C. Koza, ``Calculating The Maximum Execution Time Of Real-Time Programs'', *The Journal of Real-Time Systems* **1**(2), pp. 159-176 (September 1989).

16. R. Rajkumar, ``Real-Time Synchronisation Protocols for Shared Memory Multiprocessors'', *Proceedings 10th IEEE International Conference on Distributed Computing Systems*, Paris, IEEE Computer Society Press (28 May - 1 June 1990).

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

18. K. Tindell, A. Burns and A. Wellings, ``Allocating Real-Time Tasks (An NP-Hard Problem made Easy)'', *Real Time Systems* **4**(2), pp. 145-165 (June 1992).

## APPENDIX 1

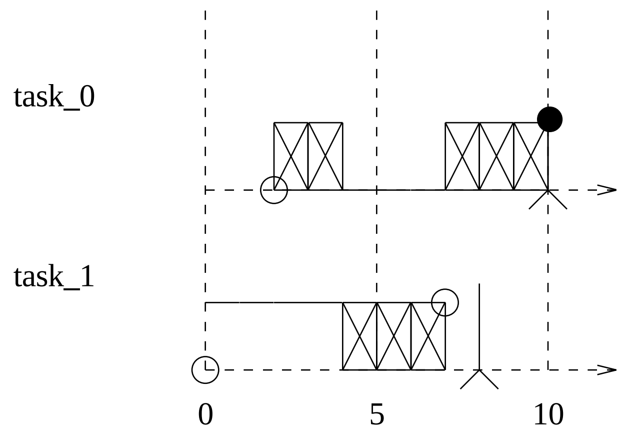
### A Brief Description of *STRESS* Diagrams

*STRESS* diagrams illustrate the execution of tasks under the *STRESS* simulator. In these diagrams, time increases from left to right.

Task execution is represented by boxes. A task which is pre-empted is shown by a line at the level of the bottom of the boxes; a task which is deferred 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 a $\wedge$ mark at the bottom.

An example is shown below. Tasks task_0 and task_1 are released at times 2 and 0 respectively, have deadlines at times 10 and 8 respectively, and require 6 and 3 computation ticks respectively; task_1 is deferred for four ticks, executes for three further ticks and then completes. task_0 executes for two ticks, before being pre-empted at tick 4 and resumed at tick 7; it fails to meet its deadline and is killed.



Example Stress Diagram