Implementing Analysable Hard Real-time Sporadic Tasks in Ada 9X

A. Burns and A.J. Wellings

Real-time Systems Research Group, Department of Computer Science, University of York, UK

ABSTRACT

In this paper we illustrate how systems containing hard real-time sporadic tasks can be analysed for their worst case behaviour. In order to undertake this schedulability analysis, it is necessary to define the minimum inter-arrival time and/or maximum arrival frequency of sporadic tasks. Furthermore, at run-time it is essential to ensure that sporadic tasks are not invoked more often than has been guaranteed by the analysis. We assume that sporadics are invoked by interrupts and that interrupts can be masked under software control.

Sporadic tasks are often analysed using the notion of bandwidth preserving sporadic servers within the Rate Monotonic Scheduling Analysis scheme. At run-time this requires the underlying kernel to support complex execution time monitoring mechanisms. Unfortunately such mechanisms are not generally supported by Ada 9X. This paper shows that by using Deadline Monotonic Scheduling Analysis there is no need to resort to bandwidth preserving sporadic servers, and the facilities available in Ada 9X can be used.

1. Introduction

In a hard real-time system, tasks are often classified as being either *periodic* or *aperiodic.* Periodic tasks typically sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic, or sporadic, tasks usually arise from asynchronous events outside the embedded computer. These tasks have specified response times associated with them. Both periodic and aperiodic tasks must, if possible, be analysed to give their *worst case response times.*

In general, aperiodic tasks are viewed as being activated randomly, following for example a Poisson distribution. Such a distribution allows for "bursty" arrivals of external events but does not preclude any possible concentration of aperiodic activity. It is therefore not possible to do worst case schedulability analysis (there is a non-zero probability of any number of aperiodic events occurring). To allow worst case calculations to be made, there is often defined a minimum interval between any two aperiodic events (from the same source) or a maximum arrival frequency. If this is the case, the task involved is said to be *sporadic*.

Once analysis has been undertaken and the system is deemed to be schedulable at run-time, it may be necessary to monitor the run-time performance of the system to ensure that the system does not execute outside its design parameters. Consequently at run-time it is essential to ensure that sporadic tasks are not invoked more often than has been allowed for by the analysis. We assume that sporadics are invoked by interrupts and that interrupts can be masked under software control by two mechanisms: the first is by raising the priority of the processor to the priority of the interrupt; the second is by setting a status bit in the associated device control register.

Sporadic tasks are often analysed using the notion of bandwidth preserving sporadic servers[5, 9, 10] within the Rate Monotonic Scheduling Analysis scheme. [6] At run-time this requires the underlying kernel to support complex execution time monitoring mechanisms. Unfortunately such mechanisms are not generally supported by Ada 9X. This paper shows that by using Deadline Monotonic Scheduling Analysis there is no need to resort to bandwidth preserving sporadic servers, and a more natural run-time representation of sporadic tasks can be achieved. Furthermore, the facilities available in Ada 9X can be used to monitor and enforce sporadic arrival rates such that the

system does not execute outside its assumed worst case behaviour.

In section 2 we first present the deadline monotonic scheduling analysis technique and its application to Ada. In section 3 we provide the analysis for sporadic tasks and illustrate their implementation and enforcement in Ada 9X. In section 4 we consider bursty sporadics with a maximum arrival frequency, and then in section 5, we extend the approach for sporadics with a maximum arrival frequency and a minimum arrival interval. Our conclusions are given in section 6.

2. The Underlying Scheduling Analysis Techniques and their Application to Ada

The basic scheduling analysis techniques assume the following about the application code:

1. It consists of a defined static number of periodic and sporadic tasks which may communicate data asynchronously, using a protected object that provides the required mutual exclusion.
2. All tasks have known worst case execution time (referred to as either WCET or C).
3. All periodic tasks have known periods, T.
4. All sporadic tasks have known minimum arrival intervals M. They may also have a worst case rate over a longer period; This is expressed as a number of releases, N in an interval W.
5. All tasks have a deadline, D, by which they must complete (i.e., this is a requirement). For each task $D \leq T$.

With the above assumption Deadline Monotonic priority assignment represents the optimal static priority scheduling scheme†.[6] Each task is given a unique priority (P); the shorter the task's deadline, the higher the task's priority (i.e. $D_i < D_j \Rightarrow P_i > P_j$). In the following discussions, the task set is assumed to be ordered by priority such that P_1 is the highest priority.

2.1. Analysis for Periodic Tasks

The worst case response time, R, for each periodic task can be calculated using the following relationships (see Audsley et al[2] for a derivation of these equations). If the method converges to a value of R which is greater than D, the deadline requirement, then the task cannot be guaranteed. The relationships assume the following:

- There exists the possibility that all tasks will be released at the same time (the *critical instant*).
- Kernel costs such as context switch time and clock overheads are assumed to be zero or included in the computation time for each task.

If it can be proved that the tasks do not share a critical instant then the predictions made by these relationships are pessimistic. The basic scheduling equation is simple (for all tasks τ_i):

$$R_i = C_i + B_i + I_i$$

Where B_i is the blocking time that the task experiences, and I_i is the interference τ_i experiences from higher priority tasks.

A task is blocked when it is prevented from running by a lower priority task. For example, if a high priority task shares an Ada 9X protected object with a low priority task then it is possible for the high priority task to be delayed while the other task is actually executing within the protected object (as the protected object guarantees mutual exclusive access to its data).

In order to bound the blocking time, some form of priority inheritance is needed.[8] The Ada 9X Real-time Annex prescribes the use of *immediate ceiling priority inheritance* (also called *ceiling priority emulation*). Protected objects are assigned ceiling priorities. This represents the highest

† Optimal in the sense that if a task set is schedulable by a given static scheduling scheme it is schedulable using Deadline Monotonic priority assignment.

priority of any task that uses that object. Whenever a task accesses a protected object, its priority is immediately raised to that of the ceiling. As a consequence, mutual exclusion (on a single processor) is assured (the current task is running with a priority at least as high as any task that could also wish to enter the protected object). It is also the case that a task is blocked at most once during its execution.[8, 7] The worst case blocking time for τ_i is equal to the longest subprogram or entry of any protected object which is used by a lower priority task but has a ceiling higher than the τ_i.

The blocking time is calculated on the assumption that the task, once released, does not voluntarily suspend itself. If this is not the case then the blocking time must be increased.

An estimation of I_i is obtained by noting that in any discrete time interval between 0 and R_i the maximum load to be asserted by higher priority periodic tasks is

$$\sum_{j \in hpp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Where $hpp(i)$ is the set of higher priority periodic tasks (than task i). The notation $\lceil X \rceil$ indicates the ceiling function which gives the smallest integer greater than or equal to X. Hence we get the relationship:

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

To solve this an iterative approach is used:[2]

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

Where R_i^0 represents the initial iterative value of R_i (which is taken to be C_i) and R_i^n is the nth iterative value. This iteration terminates when either $R_i^{n-1} = R_i^n$ or (unsuccessfully) when $R_i^n > D_i$.

To illustrate the use of equation (1), consider the simple three task set given in Table 1. Let the blocking time be 2 units of computation for Task_1 and Task_2.

	Period	Computation Time	Priority	Deadline
Task_1	8	2	1	6
Task_2	12	3	2	10
Task_3	20	7	3	20

Table 1

Task_1 has the highest priority and has a worst case response time of 4 (i.e., its own computation time plus the blocking time). Task_2 has an earliest possible response time of 3; putting this value into equation (1) gives a right hand side value of 7, 7 then balances the equation:

$$7 = 3 + 2 + \left\lceil \frac{7}{8} \right\rceil 2$$

For Task_3 the initial estimate is 7; the left hand side of equation (1) is thus (note the blocking factor is zero as this is the lowest priority task):

$$7 + \left\lceil \frac{7}{8} \right\rceil 2 + \left\lceil \frac{7}{12} \right\rceil 3$$

This yields 12. Hence:

$$7 + \left\lceil \frac{12}{8} \right\rceil 2 + \left\lceil \frac{12}{12} \right\rceil 3$$

This now yields a value of 14. A further two iterations then gives a value of 19; this value is stable (i.e. causes equation (1) to actually balance) and therefore the actual worst response time of Task_3 is 19. Hence all tasks will complete before their deadlines.

2.2. Kernel Overheads

If we move away from the simple assumptions about overheads then each task must be "charged" with the cost of context switches. Furthermore if the Ada run-time support system has mutually exclusive sections then the blocking time from these must also be considered (for example, a context switch is normally non-preemptible). In addition, the manipulation of the delay queue and the handling of interrupts from the clock must be accounted for. In a recent paper, we have shown how equation (1) can be enhanced to take into account these kernel activities.[4] In the following sections we consider sporadic tasks and the interrupts that release them.

3. Sporadic Tasks with a Minimum Arrival Interval

Simple sporadic tasks may be characterised by their minimum arrival interval and their deadline. In general there is no direct relationship between these two factors and to force arrival rate to be reduced to the deadline value leads to poor utilisation, or the need to use complex bandwidth preserving algorithms.[10] Deadline Monotonic Scheduling Analysis directly support sporadic tasks; any number of which can potentially be guaranteed.[3]

In addition to the sporadic tasks themselves, the scheduling equations must take account of the cost of the interrupts. We assume that each sporadic task is invoked by a unique interrupt and that the priority of all interrupts is higher than any application tasks.

3.1. Analysis of Sporadic Activities

Assume that the expected minimum inter-arrival time for some sporadic task is M; and that the deadline associated with each release of the sporadic task is D. In the worst-case, the sporadic behaves exactly like a periodic task with period M and deadline D where $D \leq M$. The characteristic of this behaviour is that a maximum of one release of the task can occur in any interval $[t, t + M)$, where release time t is at least M time units after the previous release of the task. This implies that to guarantee the deadline of the sporadic task, the computation time must be available within the interval $[t, t + D)$ — noting that the deadline will be at least M after the previous deadline of the sporadic. This is exactly the guarantee given by the deadline-monotonic schedulability tests explained in section 2, and hence equation (1) can be extended as follows:

$$R_i^n = C_i + B_i + \sum_{j \in hpp(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j + \sum_{s \in hps(i)} \left\lceil \frac{R_i^{n-1}}{M_s} \right\rceil C_s \qquad (2)$$

Where $hps(i)$ is the set of higher priority sporadic tasks (than task i). Note task i can be either periodic or sporadic.

Equation (2) still ignores the cost of the interrupt handling. However to quantify this requires the implementation model to be defined. This is given next.

3.2. Implementation in Ada 9X

To implement a sporadic activity in Ada 9X requires a task and a protected object. The protected object is used as the interface between the task and the interrupts which invoke it. The basic approach is as follows (we assume that the reader is familiar with the Ada 9X language as defined by[1]):

```
protected type Interface(Id : System.Interrupts.Interrupt_Id) is
  procedure Start;
  pragma Attach_Handler(Start, Id);
  pragma Interrupt_Priority(Device_Priority(Id));
  entry Wait_Start(<Parameter Part>);
private
  -- store for parameters
  Start_Open : Boolean := False;
end Interface;

task Thread;

protected body Interface is

  procedure Start is
  begin
    -- read any device registers and save
    Start_Open := True;
  end Start;

  entry Wait_Start (<Parameter Part>)  when Start_Open is
  begin
    -- write params
    Start_Open := False;
  end Wait_Start;
end Interface;

task body Thread is
   Interrupt_Handler : Interface(Interrupt_To_Be_Handled);
begin
   loop
     Interrupt_Handler.Wait_Start(<Parameter>);
     -- code of the sporadic
   end loop;
end Thread;
```

The interrupt calls the `Start` procedure which executes at the hardware priority of the device. It simply lowers the barrier thereby allowing the sporadic thread to be invoked. The `Wait_Start` entry also executes at the hardware priority of the device and thus inhibits further interrupts whilst it is executing.

Within this approach, the main issue to be considered is the action to be taken if the start events come in too quickly. As the source of the sporadic invocation is an interrupt, it is necessary to inhibit the interrupts (via setting the appropriate status bit in the device control register) for the period that the sporadic is not prepared to accept them. This is because the analysis will have assumed that the interrupt is itself a sporadic event with a minimum arrival interval, and it will be necessary to ensure that the interrupt handler does not consume more execution time than has been allowed for.

The following illustrates one possible approach; it assumes that the deadline of the sporadic handler is less than the minimum arrival interval.

```
with System; use System;
with System.Real_Time; use System.Real_Time;
with System.Interrupts;
  ...
  protected type Interface(Id : System.Interrupts.Interrupt_Id) is
    procedure Start;
    pragma Attach_Handler(Start, Id);
    pragma Interrupt_Priority(Device_Priority(Id));
    entry Wait_Start(<Parameter Part>; Started : out Time);
  private
    -- store for parameters
    Start_Open : Boolean := False;
    Start_Time : Time;
  end Interface;

  task Thread is
    pragma Priority(Thread_Priority);
  end Thread;

  protected body Interface is

    procedure Start is
    begin
      -- save device params
      Start_Time := Clock; -- log time of event
      Start_Open := True;
      -- turn off interrupts, using the device control register
      -- this allows only one outstanding interrupt at once
    end Start;

    entry Wait_Start (<Parameter Part>; Started : out Time)
            when Start_Open is
    begin
      -- write params
      Started := Start_Time;
      Start_Open := False;
      -- turn interrupts on using the device control register
    end Wait_Start;
  end Interface;
```

```
task body Thread is
  T: Time;
  Started : Time;
  Mat : constant Fine_Duration := -- set to the minimal inter-arrival time;
  Interrupt_Handler : Interface(Interrupt_To_Be_Handled);
begin
  T:= Clock;
  loop
    Interrupt_Handler.Wait_Start(<Parameter>, Started);
    -- action

    -- calculate the next delay period, this equals either the time of
    -- the last interrupt plus the minimum arrival interval, or the time
    -- the thread was last invoked plus the minimum arrival interval
    if T > Started then
      T := T + Mat;
    else
      T := Started  + Mat;
    end if;
    delay until T;
  end loop;
end Thread;
```

Assume that the sporadic is waiting for the interrupt to arrive; it has therefore called the `Wait_Start` entry and has been queued. When the interrupt occurs, the event is noted inside the `Start` interrupt handler, and further interrupts are disabled. The barrier on the `Wait_Start` entry is lowered, and as there is a task waiting, the protected action is executed (at the hardware priority of the interrupt). Interrupts are turned on again and the interrupt handler exits. The sporadic thread is now free to run at its own priority to respond to the interrupt. It is guaranteed by the analysis to call the `Wait_Start` entry before the minimum arrival interval between the interrupts (because its response time is less than the minimum arrival interval). If the interrupt arrives before the minimum arrival interval, it will be noted and interrupts will be disabled. As the task is not waiting, interrupts will remain disabled until the interrupt is serviced. We assume that interrupts will be queued by the hardware for at least a period of M before being lost.

By adopting the above approach, any overrun of the device will not affect the performance of the software.

3.3. Analysis of Interrupt Handlers

We can assume that all interrupt handlers have the same computing cost C_{IH} as they are all derived from the same protected type. Each sporadic task has associated with it an interrupt handler. However it is not possible to simply add in the cost of the interrupt handling into the cost of the sporadic. This is because the interrupt handler is allowed to execute twice in an interval smaller than M. To understand this consider the sporadic executing immediately after an interrupt has occurred. This will turn interrupts back on again, and hence a second interrupt could occur soon after the previous one. However at most 3 interrupt handling activities can occur in any interval $2M$, as illustrated in Figure 1. In general, $N+1$ interrupts can occur in N intervals.

Also, because the sporadic task potentially suspends itself twice during its execution (once on the delay and once on the protected object when the interrupt is not outstanding), it can potentially suffer two blocks from lower priority tasks (if they share other protected objects with higher priority tasks). However, only one of these can ever be significant in terms of calculating the response time. Unfortunately, it is still necessary to add any extra computation time or context switch time to the WCET of the sporadic.

The scheduling equation for the response time of each task is thus:

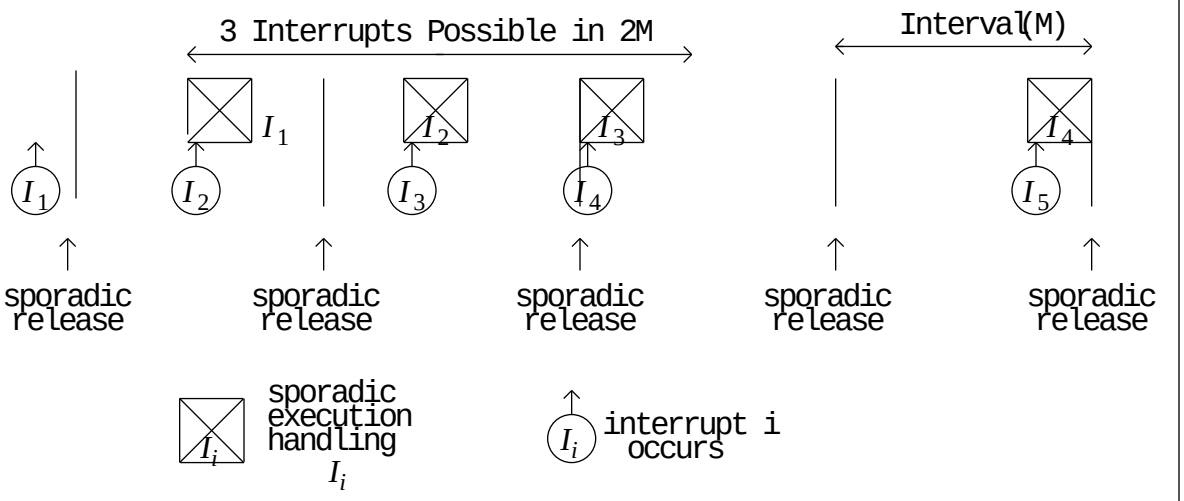

Figure 1: Sporadic Execution Jitter and Interrupt Handling Release Jitter

$$R_i^n = C_i + B_i + \sum_{j \in hpp(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j + \sum_{s \in hps(i)} \left\lceil \frac{R_i^{n-1}}{M_s} \right\rceil C_s + \sum_{h \in sih} \left\{ C_{IH} + \left\lceil \frac{R_i^{n-1}}{M_h} \right\rceil C_{IH} \right\} \quad (3)$$

Where *sih* is the set of all sporadic interrupt handlers. Note that the two sets *hps* and *sih* are of equal size, and that for each member of *hps* there is a member of *sih* with the same M value.

The value of C_{IH} is equal to the worst case execution time of the `Start` procedure plus the time taken to evaluate the barrier expression plus the worst case execution time of the `Wait_Start` entry.

4. Enforce a Maximum Arrival Frequency

In the previous discussions sporadic tasks were defined to have a minimum inter-release time. This was then used to bound the interference other (lower priority) tasks suffer. As sporadic tasks are released by the action of an external agent via an interrupt, it may be necessary to model the "bursty" nature of some interrupt events.

4.1. Analysis of Sporadic with a Maximum Arrival Frequency

In general the model presented in section 3.1 can be trivially extended. As the sporadic events can arrive all together, a sporadic which is characteristed by:

- a WCET of C,
- a deadline D and
- a maximum arrival frequency of N in a W interval

is equivalent to a sporadic with :

- a WCET of NC,
- a deadline of D and
- a minimum arrival interval of W.

The calculated worst case response time will be the response time of the last invocation of the sporadic in the burst.

4.2. Implementation in Ada 9X

As the interrupts can come in bursts, they must be buffered. The time spent in the interrupt handler must also be taken into account in the scheduling analysis. When the maximum interrupt burst has been received in any one period, interrupts must be disabled to ensure that the system does not operate outside its design parameters. The approach taken is to disable interrupts when the buffer becomes full. The sporadic then enables interrupts when its starts a fresh period. Note however, because of release jitter on the sporadic task, the size of the buffer must be twice the number of interrupts allowed during the period.

Consider the following program fragment:

```
with System; use System;
with System.Real_Time; use System.Real_Time;
  ...
  type Freq is record
    Interval : Fine Duration;
    Number : Integer; -- number of events in the above interval
  end record;
  Maf : constant Freq := -- set to appropriate values;

  subtype Buffer_Range is Integer range 0 .. 2*Maf.Number-1;
  subtype Event_Number is Integer range 0 .. 2*Maf.Number;
  type Event_Params is
    record
      <Device Parameters>;
      Occurred : Time;
    end record;
  type Param_Store is array (Buffer_Range) of Event_Params;

  protected type Interface(Id : System.Interrupts.Interrupt_Id) is
    procedure Start;
    pragma Attach_Handler(Start, Id);
    pragma Interrupt_Priority(Device_Priority(Id));
    entry Wait_Start(<Parameter Part>; Started : out Time);
  private
    Outstanding_Events :Param_Store;
    First : Buffer_Range  := 0;
    Last : Buffer_Range  := 0;
    Size : Event_Number := 0;
  end Interface;

  task Thread is
    pragma Priority(Thread_Priority);
  end Thread;

  protected body Interface is

    procedure Start is
    begin
      Outstanding_Events(Last).Occurred := Clock;
      -- save any device registers etc
      Last := (Last + 1) mod Events'Last;
      Size := Size + 1;
      if Size = Events'Last then
        -- disable interrupts;
      end if;
    end Start;
```

```
    entry Wait_Start (<Parameter Part>;  Started : out Time) when Size > 0 is
    begin
      -- assign params
      First := (First + 1) mod Events'Last;
      Count := Count -1 ;
      -- enable interrupts
    end Wait_Start;
  end Interface;

  task body Thread is
    subtype Invocation_Range is Integer range 0 .. Maf.Number-1;
    subtype Events is Integer range 0 .. Maf.Number;
    Last_Interrupt_Times : array ( 0 .. Invocation_Range) of Time;
    First, Last : Invocation_Range := 0;
    Last_Time : Time;
    Interrupt_Handler : Interface(Interrupt_To_Be_Handled);
  begin
    for I in 1 .. Maf.Number-1 -- take N - 1 interrupts
    loop
      Interrupt_Handler.Wait_Start(<Parameter>, Last_Time);
      Last_Interrupt_Times(Last) := Last_Time;
      Last := (Last +1) mod Events'Last;
      -- respond to event;
    end loop;

    loop
      Interrupt_Handler.Wait_Start(<Parameter>, Last_Time);
      Last_Interrupt_Times(Last) := Last_Time;
      Last := (Last +1) mod Events'Last;
      -- respond to event;
      delay until (Last_Interrupt_Times(First) + Maf.Interval);
    end loop;
  end Thread;
```

The thread ensures that it does not handle more than the required number of interrupts in a given period by calculating a delay value which is based on the time the N-1th previous interrupt arrived. This stops it calling the protected object and turning on interrupts.

4.3. Analysis of Interrupt Handler

Equation (3) is trivially updated as follows; where $hpbs(i)$ is the set of higher priority bursty sporadics (than task i), and $bsih$ is the set of all bursty sporadic interrupt handlers.

The scheduling equation for the response time of a task is thus:

$$R_i^n = C_i + B_i + \sum_{j \in hpp(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j + \sum_{s \in hps(i)} \left\lceil \frac{R_i^{n-1}}{M_s} \right\rceil C_s + \sum_{h \in sih} \left\{ C_{IH} + \left\lceil \frac{R_i^{n-1}}{M_h} \right\rceil C_{IH} \right\}$$
$$+ \sum_{bs \in hpbs(i)} \left\lceil \frac{R_i^{n-1}}{W_{bs}} \right\rceil NC_{bs} + \sum_{hb \in bsih} \left\{ NC_{IH} + \left\lceil \frac{R_i^{n-1}}{NM_{hb}} \right\rceil NC_{IH} \right\} \quad (4)$$

5. A maximum arrival frequency and a minimum arrival interval

In the previous model there was no minimum arrival interval between invocations of the bursty sporadic task. Consequently the analysis had to assume that the maximum number of events in a given period could all arrive at the start of the period. If the bursty sporadic task has a minimum arrival interval within its burst then less pessimistic analysis can be derived. We still assume that the

interrupts themselves can arrive arbitrarily close together.

5.1. Analysis Model

A sporadic task can be characterised by: M, the minimum inter-release time (as before), and the maximum rate at which releases can occur. This latter value is expressed as the maximum number of releases, N, in a given interval W (e.g. 6 in 10ms). The sporadic is now assumed to execute N times, with an inter-release time of M, and then be silent (i.e. not execute) until the end of W.

To take this behaviour into account, lower priority tasks are assumed to be released at the start of a burst. If their response time goes beyond the interval (W) a second burst will be experienced and so on. The scheduling equations can be easily changed to reflect this model.[2] Rather than assume an interference of C every M units of time, the interference can be reduced. Recall that equation (1) has the form for sporadics:

$$R_i^n = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{n-1}}{M_j} \right\rceil C_j$$

Consider one higher priority task (denoted by subscript j) that is bursty. In the time interval [0, R_i^{n-1}) a full cycle will be executed the following number of times:

$$\left\lfloor \frac{R_i^{n-1}}{W_j} \right\rfloor$$

This will give rise to the following interference:

$$\left\lfloor \frac{R_i^{n-1}}{W_j} \right\rfloor N_j \; C_j$$

The number of extra "hits" from part cycles will be given by

$$\left\lceil \frac{R_i^{n-1} - \left\lfloor \frac{R_i^{n-1}}{W_j} \right\rfloor W_j}{M_j} \right\rceil$$

But there are a maximum of N_j such hits so that the total interference becomes:

$$I_i^j = \left\lfloor \frac{R_i^{n-1}}{W_j} \right\rfloor N_j \; C_j + \min \left\{ \left\lceil \frac{R_i^{n-1} - \left\lfloor \frac{R_i^{n-1}}{W_j} \right\rfloor W_j}{M_j} \right\rceil , N_j \right\} C_j \qquad (5)$$

This can be applied in equations (3) and (4) for each bursty sporadic.

5.2. Ada 9X Implementation

This approach is very similar to the one in Section 4 except that we need to enforce a minimum invocation gap between invocations of the sporadic. We still, however, assume that the interrupt can come in a burst with effectively a zero inter-arrival interval.

The protected object remains the same. The sporadic thread becomes:

```
type Freq is record
  Interval : Duration; -- period
  Number : Integer; -- number of events in the above period
  Gap : Fine_Duration;
end record;

Maf : constant Freq := -- set to appropriate values;

function Max(A, B : Time) return Time is separate;
   -- returns the lastest time of A or B

task body Thread is
  subtype Invocation_Range is Integer range 0 .. Maf.Number-1;
  subtype Events is Integer range 0 .. Maf.Number;
  Last_Interrupt_Times : array ( 0 .. Invocation_Range) of Time;
  First, Last : Invocation_Range := 0;
  Last_Time : Time;
  Interrupt_Handler : Interface(Interrupt_To_Be_Handled);
  T : Time;
begin
  T := Clock;
  for I in Maf.Number-1
  loop
    Interrupt_Handler.Wait_Start(<Parameter>, Last_Time);
    Last_Interrupt_Times(Last) := Last_Time;
    Last := (Last +1) mod Events'Last;
    -- respond to event;
    if T > Last_Time then
      T := T + Maf.Gap;
    else
      T := Last_Time  + Maf.Gap;
    end if;
    delay until T;
  end loop;

  loop
    Interrupt_Handler.Wait_Start(<Parameter>, Last_Time);
    Last_Interrupt_Times(Last) := Last_Time;
    Last := (Last +1) mod Events'Last;
    -- respond to event;
    if T > Last_Time then
      T := (Max(Last_Interrupt_Times(First) + Maf.Interval),
                  (T + Maf.Gap));
    else
      T := (Max(Last_Interrupt_Times(First) + Maf.Interval),
                  (Last_Time + Maf.Gap));
    end if;
    delay until T;
  end loop;
end Thread;
```

Here the sporadic task takes $N-1$ interrupts with the required arrival interval. It then takes the next one and delays for the maximum of the minimum arrival interval and the time to ensure that the minimum inter-arrival frequency is not violated.

5.3. Analysis of Interrupt Handler

As the approach to interrupt handling is the same as in section 4.1, equation (4) becomes:

$$R_i^n = C_i + B_i + \sum_{j \in hpp(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j + \sum_{s \in hps(i)} \left\lceil \frac{R_i^{n-1}}{M_s} \right\rceil C_s + \sum_{h \in sih} \left\{ C_{IH} + \left\lceil \frac{R_i^{n-1}}{M_h} \right\rceil C_{IH} \right\}$$

$$+ \sum_{bs \in hpbs(i)} \left\{ \left\lfloor \frac{R_i^{n-1}}{W_{bs}} \right\rfloor N_{bs} C_{bs} + \min \left\{ \left\lceil \frac{R_i^{n-1} - \left\lfloor \frac{R_i^{n-1}}{W_{bs}} \right\rfloor W_{bs}}{W_{bs}} \right\rceil , N_{bs} \right\} C_{bs} \right\}$$

$$+ \sum_{hb \in bsih} \left\{ NC_{IH} + \left\lceil \frac{R_i^{n-1}}{NM_{hb}} \right\rceil NC_{IH} \right\} \qquad (6)$$

Note that this equation deals with each type of sporadic separately. It is possible to combine terms to give a simpler representation, however for clarity, we have distinguished between them.

6. Conclusion

In this paper we have presented schedulability analysis for sporadic tasks. Further, we have shown how sporadic tasks can be implemented in Ada 9X so that violations of the assumptions made by the analysis can be prohibited from occurring. Alternative implementations can be defined for where the sporadic tasks are invoked from software tasks. In this case, various actions on an overrun condition can be taken. These range from simply reporting the overrun to the sporadic or the invoker, to raising an exception or aborting the invoker.

The enforcement of gaps requires access to the real-time clock. The granularity and efficiency of the delay primitive will significantly influence the applicability of the model presented. If an on-board interval timer is available, this limitation may be removed by directly programming the timer and having a user implemented delay.

Acknowledgement

The authors would like to thank Offer Pazy, Neil Audsley, Rob Davis, and Ken Tindell for commenting on an earlier version of this paper.

The work has been supported, in part, by the European Space Agency (ESTEC Contract 9198/90/NL/SF) and the Defence Research Agency (Contract Number MOD 2191/023).

References

1. Ada 9X Mapping/Revision Team, Intermetrics, ‘‘Ada 9X Reference Manual, Draft Version 3.0’’, Ada 9X Project Report (June 1993).
2. N. Audsley, A. Burns, M. Richardson, K. Tindell and A. Wellings, ‘‘Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling’’, *Software Engineering Journal* (to appear).
3. 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).
4. A. Burns, A.J. Wellings and A.D. Hutcheon, ‘‘The Impact of an Ada Run-time System’s Performance Characteristics on Scheduling Models’’, pp. 240-248 in *Ada sans frontieres Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science 688,*

Springer-Verlag (1993).

5. J.P. Lehoczky, L. Sha and J.K. Strosnider, *Enhancing Aperiodic Responsiveness in Hard Real-Time Environment*, Proc. 8th IEEE Real-Time Systems Symposium, San Jose, California, December 1987.
6. C.L. Liu and J.W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment’’, *JACM* **20**(1), pp. 46-61 (1973).
7. M. Pilling, A. Burns and K. Raymond, ‘‘Formal Specification and Proofs of Inheritance Protocols for Real-Time Scheduling’’, *Software Engineering Journal* **5**(5), pp. 263-79 (1990).
8. 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).
9. B. Sprunt, J. Lehoczky and L. Sha, ‘‘Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm ’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 251-258 (December 1988).
10. B. Sprunt, L. Sha and J. P. Lehoczky, ‘‘Aperiodic Task Scheduling for Hard Real-Time Systems’’, *Real-Time Systems* **1**, pp. 27-69 (1989).