Fixed Priority Scheduling with Deadlines Prior to Completion

A. Burns

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

ABSTRACT

Standard analysis for fixed priority scheduling assumes that all the computations for each task must be completed by the task's deadline. In practice this is rarely the case. The deadline is more correctly associated with the last observable event of the task, interval events and kernel overheads (such as the context switch away from the task) can occur after the deadline. New analysis is presented that caters for this situation. The result is enhanced schedulability without modification to the preemptive behaviour of the run-time system.

1. Introduction

The fundamental rate monotonic model is based upon a number of stringent assumptions. Since the initial analysis of Liu and Layland[10] in 1973 many of these assumptions have been removed (or at least weakened). For example:

- *tasks must be independent* — the priority inheritance protocols[12] allow tasks to share data that is protected by mutual exclusion;
- *deadline must equal period* — deadline monotonic theory[9, 4, 1] can by applied if deadline (D) is less than period (T), and arbitrary deadline analysis[14, 8] can by used for the case of $D > T$;
- *tasks are periodic* — hard sporadic tasks can by supported directly once the $D = T$ requirement is relaxed[3], and aperiodic tasks can be accommodated by various bandwidth preserving algorithms[13, 6].

One further restriction is relaxed in this paper; namely that the deadline of each task occurs at the very end of its computation - i.e. a task's worst case computation time, denoted by C, must always be completed by the deadline D.

In a recent report, Gerber and Hong[7] argue that it is only meaningful to attach a deadline to the last observable event of a task. Moreover this last observable event may not be at the end of the task's execution, i.e. there may be a number of internal actions after the last output event. Even if the task, as written, has the last observable event at the end of its execution sequence it may be possible to transform the code so that statements not dependent on this last event are moved beyond it.

When the model for analysis is enhanced to include kernel overheads[5], it is necessary to "charge" to each task the cost of the context switch that allows it to preempt a lower priority task plus the cost of the context switch back to the preempted task once the higher priority task has completed. For realistic context switch times (i.e. not zero), it is meaningless to attach the "deadline" to the end of the context switch. Figure 1 gives a block representation of a task's execution (excluding preemptions for higher priority tasks). Phase *a* is the initial context switch to the task, phase *b* is the task's actual execution time up to the last observable event, phase *c* represents the internal actions of the task following the last observable event, and finally phase *d* is the cost of the context switch away from the task. The real deadline of the task is at the end of phase *b*.

In the following discussions we shall denote by C^D the computation time required by the real deadline (i.e. phases $a + b$ only), and by C^T the total computation time of the task in each period (i.e. all four phases). Note there is no requirement to complete C^T by T as long as C^D is completed by D. We assume throughout that preemptive priority based dispatching is being undertaken, tasks have unique priorities, tasks do not suspend themselves during their execution, and tasks are released as soon as they arrive (and hence they suffer no release jitter).

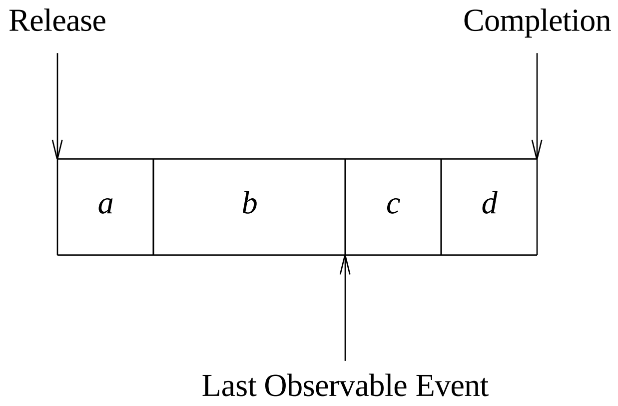

Figure 1: Four Phases of a Task's Execution

The rest of the report is organised as follows. In section 2 simple analysis is introduced that is adequate for task sets that can accommodate C^T within T. Section 3 presents the main result of this paper, it applies (modified) arbitrary deadline analysis[14] to the case in which C^D must be completed before D, but C^T can, occasionally, run over T. This is the case examined by Gerber and Hong[7]. Finally in section 4 we present our conclusions.

2. Restricted Analysis

Utilisation based analysis is only really applicable to task sets that conform exactly to the rate monotonic model. When D is not equal to T it is more appropriate to attempt to calculate the worst case response time of each task, R, and to then compare, trivially, each response time to the required deadline (i.e. test to confirm that $R \le D$). For deadline less than (or equal) to period we have for each task:

$$R_i = C_i + B_i + I_i \qquad (1)$$

i.e. response time equals computation time, C, plus blocking time, B (this is the maximum time a lower priority task can delay task i due to the operation of the priority ceiling protocol[12]), plus the interference, I, the task gets from higher priority tasks. As the amount of interference depends on the response time, R, we can solve equation (1) by extending a window, w†, into which we attempt to fit C, B and the maximum possible I. For each task j, of higher priority than i, the amount of interference occurring within w is given by I_i^j:

$$I_i^j = \left\lceil \frac{w_i}{T_j} \right\rceil C_j$$

On each iteration of the following equation w is extended and thus more interference may be experienced:

† this extending window is also called a *busy period*[8].

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

where $hp(i)$ is the set of higher priority tasks than i, and w_i^0 can be set to 0. Equation (2) is monotonically increasing in w, in the sense that $w_i^{n+1} \geq w_i^n$. If the equation balances (i.e. $w_i^{n+1} = w_i^n$) then R_i has been found:

$$R_i = w_i^n$$

Alternatively if $w_i^{n+1} > D_i$ then the task cannot be guaranteed to complete before its deadline and the iteration can be abandoned.

Equation (2) can be used with our new formulation that recognises two computation parameters for each task (i.e. C_i^D and C_i^T). In effect there are now two response times, R_i^D and R_i^T, corresponding to the two computation times. First the maximum response time for that part of the task that has a deadline can be found using the following formulation:

$$w_i^{n+1} = C_i^D + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j^T \qquad (3)$$

Here each task interferes with its full computation time (C_i^T) but is only trying to fit C_i^D into the window. If equation (3) converges on a fixed value, then R_i^D has been found and this can be compared with D_i.

Equation (2) can also be used to find the maximum response time for the entire computation time for each task:

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

This provides the value of R_i^T. But the analysis is only valid if each task's computations are completed within its period (i.e. $R_i^T \leq T_i$). If this were not the case then a task would, in effect, interfere with itself and equation (2) would be invalid. The next section of this paper removes this restriction. We conclude this section with a simple example. Table 1 contains a task set and response times derived from equations (3) and (4).

Task	T	C^D	C^T	D	R^D	R^T
task 1	10	2	3	10	2	3
task 2	12	3	4	12	6	7
task 3	20	2	4	10	9	18

Table 1: A Simple Task Set

Note that R_3^D is much shorter than R_3^T although the required computation time is only 2 less. Indeed the deadline of 10 is met by R_3^D (but not by R_3^T). Thus the tasks could not have been guaranteed by analysis requiring all computations to be completed by the deadline.

3. The Full Model

To cater for situations where R_i^T can be greater than T_i we must adapt the analysis produced for arbitrary deadlines[14, 15]. When deadline is less than (or equal to) period, it is only necessary to consider a single release of each task. The critical instant, when all higher priority tasks are released at the same time, represents the maximum interference and hence the response time following a release at the critical instant must be the worst case. However when deadline is greater than period a number of releases must be considered. We assume that the release of a task will be delayed until any previous releases of the same task have completed. For each potentially overlapping release we define a separate window $w_{i,q}$, where q is just an integer identifying a particular window (i.e. $q=0,1,2,\cdots$). Equation (2) can be extended to have the following form (for a single computational parameter C_i):

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

For example with q equal to two, three releases of task i will occur in the window. For each value of q a stable value of $w_{i,q}$ can be found ($w_{i,q}^n = w_{i,q}^{n+1}$) by iteration — as in equation (2). The response time is then given as

$$R_{i,q} = w_{i,q} - qT_i \tag{6}$$

e.g. with $q=2$ the task started $2T_i$ into the window and hence the response time is the size of the window minus $2T_i$.

The number of releases that need to be considered is bounded by the lowest value of q for which the following relation is true:

$$R_{i,q} \le T_i \tag{7}$$

At this point the task completes before the next release and hence subsequent windows do not overlap. The worst case response time is then the maximum value found for each q:

$$R_i = \max_{q=0,1,2,\ldots} R_{i,q} \tag{8}$$

Note that for $D \le T$ relation (7) is true for $q=0$ (if the task can be guaranteed), in which case equations (5) and (8) simplify back to the original equations.

3.1. Extended Analysis

The above discussion is taken from previous analysis[14, 15]. We now extend this model by introducing the two phases of computation (C^D and C^T). Note that the priority of the task does not change during execution; the two phases are merely for purposes of analysis. In order to analyse the correct number of releases the full response time $R_{i,q}^T$ (using C_i^T) for each q must be calculated — using equation (5). Equation (7) is then used to bound q. Having obtained a bound for q the actual response time for C_i^D is obtained via a modification to equation (5):

$$w_{i,q}^{n+1} = qC_i^T + C_i^D + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_{i,q}^n}{T_j} \right\rceil C_j^T \tag{9}$$

This assumes that the interference due to previous releases of task i is C_i^T, but the current release only requires C_i^D. For each value of q (when equation (9) balances) a response time $R_{i,q}^D$ is found. The worst case response time is obtained from each q value:

$$R_i^D = \max_{q=0,1,2,\ldots} R_{i,q}^D \tag{10}$$

It can be shown, trivially, that for utilisation less than 100% there exist bounded response times for all tasks†. It also follows that q is bounded (by the number of times the task's period

† Consider a set of periodic tasks with 100% utilisation, all of which have deadlines equal to the LCM of the task set; clearly within the LCM no idle tick is used and no task executes for more than it needs and hence all deadlines must be met.

divides into the LRM of the task set's periods), and hence each R_i^T and R_i^D value can be found. The schedulability test is simply a check that R_i^D is less than D_i for all i.

The above analysis can be applied to the simple task set introduced and discussed by Gerber and Hong[7]. Table 2 shows the characterisation of three tasks, note that $D = T$ for all entries, and that no task experiences blocking. With rate monotonic analysis task 3 cannot be guaranteed. Gerber and Hong note that task 3 can be transformed so that its last observable event can be moved to be some 60 time units prior to completion of the full 653. They then show how a two-thread implementation (with priority exchange) can be used to guarantee task 3. However the analysis embodied in equations (9), (10), (6) and (7) furnishes a value for R_3^D of 2493 which is just before the deadline (and period). Hence standard preemptive priority based dispatching will satisfy the timing requirements of this task set. No extra implementation scheme is needed. Note, for completeness, that the worst case response time of R_3^T is 2653.

Task	T	C^D	C^T	D	R^D	R^T
task 1	1000	400	400	1000	400	400
task 2	1600	400	400	1600	800	800
task 3	2500	493	653	2500	2493	2653

Table 2: Gerber and Hong's Task Set

Although $D = T$ in this example (and $D \le T$ in the extended example given in section 3.3) the analysis presented above is based upon arbitrary deadlines and hence D can be greater then T.

3.2. Optimal Priority Assignment

Where deadlines can be greater than period there is no simple algorithm for calculating the optimal priority ordering; neither rate monotonic nor deadline monotonic assignment is optimal. This is also the case with the formulation discussed in this paper. To illustrate this, consider the simple two task system defined in Table 3.

Task	T	C^D	C^T	D
task 1	6	1	4	2
task 2	3	1	1	3

Table 3: Priority Order Example

If task 1 is given the highest priority (because it has the shortest deadline) it will run for the first 4 ticks and task 2 will miss its first deadline. However the assignment of task 2 to the top priority level will allow all deadlines to be met.

Audsley has recently constructed an algorithm for assigning priorities when the simple models do not apply[2]. He formulates, and proves, the following theorem:

> **Theorem**: *If task k is feasible at the lowest priority level, then if the task set is feasible, there must exist a feasible priority ordering with task k at this lowest priority level.*

By feasible we mean satisfy all deadlines in the worst case. If a task k is found then the problem becomes one of assigning priorities to the remaining task set. The theorem can then be reapplied, leading to an algorithm in which priorities are assigned in inverse order.

This algorithm is applicable to the model derived above. It is used in the following example.

3.3. Extended Example

We conclude this section with a more extensive example which has been run through a tool that implements the above formulations. The key elements of the software of this tool are included in an appendix. The example is derived from one discussed by Tindell[14] which was itself based on one described by Locke *et al*[11]. Table 4 gives the details of the task set. Times are in units of 10 microseconds. Task 1 represents the overhead of the clock (ie period of 10ms computation time of 2.5ms). The context switch away from another task costs 1ms (i.e. 100 time units). Tasks 11 to 18 have various amounts of computation after the last observable event. For example tasks 14 and 16 have considerable activity after their last output. These tasks are typical of those that take an input, produce an output and then undertake a detailed analysis of the input in order to add a correction factor to the next output (in the next period).

Task	T	C^D	C^T	D	B
task 1	1000	250	250	1000	0
task 2	200000	3000	3100	5000	0
task 3	25000	700	800	5500	0
task 4	25000	1400	1500	12000	300
task 5	40000	1000	1100	40000	300
task 6	50000	3000	3100	50000	300
task 7	50000	5000	5100	50000	400
task 8	59000	8000	8100	59000	400
task 9	80000	9000	9100	80000	400
task 10	80000	2000	2100	80000	1350
task 11	100000	3000	7000	100000	1350
task 12	200000	4000	5000	150000	1350
task 13	200000	2000	3500	200000	1350
task 14	200000	300	1850	200000	1350
task 15	300000	1000	1500	300000	1350
task 16	300000	900	3600	300000	0
task 17	1000000	10000	15000	1000000	0
task 18	1000000	10000	15000	1000000	0

Table 4: Full Task Set

The overall utilisation of this task set is 99.5%. When standard deadline monotonic analysis is used, only the first 10 tasks can be guaranteed. The following represents the output from the tool given in the appendix. It shows priority (`P`) and response time (`Rd`) for C^D (`Cd`). All tasks are guaranteed to meet their real deadlines.

T	Cd	Ct	D	B	P	Rd	Met?
1000	250	250	1000	0	0	250	TRUE
200000	3000	3100	5000	0	1	4000	TRUE
25000	700	800	5500	0	2	5300	TRUE
25000	1400	1500	12000	300	3	7600	TRUE
40000	1000	1100	40000	300	4	8950	TRUE
50000	3000	3100	50000	300	5	13300	TRUE
50000	5000	5100	50000	400	6	20000	TRUE
59000	8000	8100	59000	400	7	33900	TRUE
80000	9000	9100	80000	400	8	47600	TRUE
80000	2000	2100	80000	1350	9	79550	TRUE
100000	3000	7000	100000	1350	10	99950	TRUE
200000	4000	5000	150000	1350	11	149450	TRUE
200000	2000	3500	200000	1350	12	197650	TRUE
200000	300	1850	200000	1350	13	199950	TRUE
300000	1000	1500	300000	1350	14	294900	TRUE
300000	900	3600	300000	0	15	294950	TRUE
1000000	10000	15000	1000000	0	16	783300	TRUE
1000000	10000	15000	1000000	0	17	999450	TRUE

Table 5: Output from the Analysis Tool

4. Conclusions

Preemptive priority based scheduling is an effective implementation strategy for real-time systems. The role of schedulability analysis is to test whether a given task set will meet all its deadlines when executing under such a strategy. Various forms of this analysis all make the assumption that the deadline is concerned with the final computational event of the task. In practice this is rarely the case, the last observable event may precede some internal computations, and the context switch away from the task does not itself have a deadline.

The scheduling analysis may therefore fail a task set that, in practice, always meets all its timing requirements. In this paper we have presented new analysis that more accurately reflects true engineering needs. We have shown how all computation up to the final observable event can be guaranteed to be completed before a deadline even when the full amount of computation may extend beyond the deadline and even beyond the end of the period of the task. The result is a test that effectively enhances the schedulability of a processor without changing, in any way, the behaviour of the processor or its run-time environment.

References

1. N. Audsley, A. Burns, M. Richardson, K. Tindell and A. Wellings, ‘‘Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling’’, *Software Engineering Journal* **8**(5), pp. 284-292 (September 1993).
2. 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).
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. N.C. Audsley, A. Burns and A.J. Wellings, ‘‘Deadline Monotonic Scheduling Theory and Application’’, *Control Engineering Practice* **1**(1) (1993).

5. 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).

6. R.I. Davis, K.W. Tindell and A. Burns, ‘‘Scheduling Slack Time in Fixed Priority Pre-emptive Systems’’, *Proceedings Real-Time Systems Symposium* (December 1993).

7. R. Gerber and S. Hong, ‘‘Semantic-Based Compiler Transformations for Enhanced Schedulability’’, CS-TR-3071, Department of Computer Science, University of Maryland (May 1993).

8. 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 December 1990).

9. 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).

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

11. 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).

12. 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).

13. B. Sprunt, L. Sha and J.P. Lehoczky, ‘‘Aperiodic Task Scheduling for Hard Real-Time Systems’’, *The Journal of Real-Time Systems* **1**, pp. 27-69 (1989).

14. K. Tindell, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, YCS189, Department of Computer Science, University of York (December 1992).

15. K. Tindell, A. Burns and A.J. Wellings, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, *Real-Time Systems (to appear)* (1994).

Appendix: Source code for the Tool

The code is structured as a number of Ada packages. The IO routines and main procedure are omitted.

```
package basic_types is
  type ticks is new natural;
  type attributes is
  record
    T : ticks; -- period
    C1,C2 : ticks; -- Cd and Ct
    D : ticks; -- deadline
    R : ticks := 0; -- response time for Cd
    B : ticks; -- blocking
    P : integer; -- priority
    met : boolean := false;
  end record;
  max_size : constant := 128;
  type process_set is array(1..max_size) of attributes;
  -- process set is assumed to be ordered by priority
  -- ie (1) has top priority
end basic_types;

with basic_types; use basic_types;
package routines is -- package body omitted
  function ceiling(top,bottom: ticks) return ticks;
  function max(first,second: ticks) return ticks;
end;

with basic_types; use basic_types;
with routines; use routines;
package routines2 is
  procedure utilisation(set : process_set; N : natural; U : out float);
  function interference(set : process_set; I : natural; W : ticks) return ticks;
  -- interference from tasks of higher priority than I in window W
end;

package body routines2 is
  procedure utilisation(set : process_set; N : natural; U : out float) is
  total : float := 0.0;
  begin
    for J in 1..N loop
      total := total + float(set(J).C2)/float(set(J).T);
    end loop;
    U := total*100.0;
  end;

  function interference(set : process_set; I : natural; W : ticks) return ticks is
  total : ticks := 0;
  begin
    for K in 1..I-1 loop
      total := total + ceiling(W,set(K).T)*set(K).C2;
    end loop;
    return total;
  end;

end routines2;
```

```
with basic_types; use basic_types;
with routines; use routines;
with routines2; use routines2;
package tests is

  procedure arb_test(set : in out process_set; N : natural);
  -- performs the full test and assigns response times

  procedure assign_pri(set :  in out process_set; N : natural; OK : in out boolean);
  -- performs optimal priority assignment of Audsley and assigned
  -- priorities if priority ordering found

end;

package body tests is
  procedure task_test(set : in out process_set; I : natural; OK : out boolean) is
    Q,W,W_old : ticks;
    factor_part, factor_all : ticks;
    worst : ticks;
  begin
    Q := 0; worst := 0;
    loop
      factor_part := Q*set(I).C2 + set(I).C1 + set(I).B;
      factor_all := (Q+1)*set(I).C2 + set(I).B;
      W_old := factor_part;
      loop -- calculate response time for C1
        W := factor_part + interference(set,I,W_old);
        exit when W = W_old;
        W_old := W;
      end loop;
      worst := max(worst, W-(Q*set(I).T));
      loop -- calculate response time for C2
        W := factor_all + interference(set,I,W_old);
        exit when W = W_old;
        W_old := W;
      end loop;
      exit when W <= (Q+1)*set(I).T;
      Q := Q + 1;
    end loop;
    set(I).R := worst;
    set(I).met := set(I).D >= set(I).R;
    OK := set(I).met;
  end;

  procedure arb_test(set : in out process_set; N : natural) is
    temp : boolean;
  begin
    for I in 1..N loop
      task_test(set,I,temp);
    end loop;
  end arb_test;
```

```
  procedure swap(set : in out process_set; I,J : natural) is
    temp : attributes := set(I);
  begin
    set(I) := set(J);
    set(J) := temp;
  end;

  procedure assign_pri(set :  in out process_set; N : natural;
                       OK : in out boolean) is
  begin
    for K in reverse 1..N loop
      for next in reverse 1..K loop
        swap(set,K,next);
        task_test(set,K,OK);
        set(K).P := K;
        exit when OK;
      end loop;
      exit when not OK;
    end loop;
  end;

end tests;
```