blank

The Impact of an Ada Run-time System's Performance Characteristics on Scheduling Models†

A.Burns and A.J.Wellings

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

A.D.Hutcheon

York Software Engineering Ltd, UK

Abstract: When analysing hard real-time systems for their performance properties, most scheduling models either ignore the timing characteristics of the underlying run-time support kernel entirely or have a very simple model of its behaviour. However, when engineering an actual implementation, it is necessary to tailor the scheduling theory so that the actual performance characteristics of the run-time system are adequately addressed. Failure to do this may result in a "guaranteed" task missing its deadline. In this paper we investigate an actual Ada run-time kernel and give performance measures for clock handling and delay queue manipulation. The impact these measures have on the scheduling models are discussed.

1. Introduction

A common approach to building hard real-time systems is to undertake pre-runtime analysis of the application to ensure that, even in the worst possible situation (within the failure hypothesis), all deadlines are met. The use of the available processors is the most critical factor in deciding if deadlines are met; scheduling analysis is therefore used to derive the necessary guarantees.

Unfortunately, most scheduling models either ignore kernel timing characteristics entirely or have a very simple model of the behaviour of the kernel. When engineering an actual implementation, it is necessary to tailor the scheduling theory so that the actual performance characteristics of the run-time system are adequately addressed. Failure to do this may result in a "guaranteed" task missing its deadline. In this paper we consider the timing properties of a particular Ada run-time

†The work presented in this paper has been supported, in part, by the European Space Agency (ESTEC Contract 9198/90/NL/SF).

kernel. This enables realistic scheduling models to be developed that can be instantiated for actual kernel properties. We focus on the handling of the clock interrupt and the delay queue.

In this paper we first describe the typical scheduling equations that are used for this type of application. In section 3 we then present the kernel under investigation and provide performance measures for clock handling and delay queue manipulation. The impact these measures have on the scheduling models is considered in section 4.

2. Scheduling Models

Scheduling analysis usually assumes that the application consists of a set of periodic and sporadic processes. These process are characterised by the values C (worst case computation time per iteration), D (deadline by which the computation must be completed), T (the time between successive iterations), and P (priority). For periodic processes the value of T is assumed constant; for sporadic processes T is taken to be the minimum time interval between releases of the process. The priority may be assigned via an optimal scheme such as rate monotonic (P inversely related to T) or deadline monotonic (P inversely related to D), or via the application programmer to reflect issues of criticality.

When the rate monotonic scheme[6] (D equal to T) is extended to include blocking times (due to process interaction) the result is a set of inequalities that must all be satisfied if the process set is to be guaranteed. Assume that the processes are ordered according to priority with process 1 having the highest priority and process n having the lowest. For each process (i):

$$\sum_{j=1}^{i} \frac{C_j}{T_j} + B_i \leq i\,(2^{1/i} - 1) \qquad (1)$$

The blocking time represent the maximum time a process can be prevented from executing by a lower priority process. Equation (1) is sufficient but not necessary; i.e. there exist process sets that fail this test but which execute without missing a deadline. Exact analysis has been undertaken by Ding *et al*[5].

The above formulae (and those below) assume that there are no overheads associated with the kernel; for example, the time of a context switch is taken to be zero. To compensate for this unrealistic assumption the computation time of each process is often increased to cater for switching time:

$$C_i := C_i + 2 * Switch\_Time$$

The *Switch_Time* is assumed to be a known (single) constant.

When the deadline monotonic scheme is used, the tests are more complicated. Joseph and Pandya[4] showed how an iterative relation can be developed. This scheme has the advantage that it also caters for arbitrary priority assignment. For each process (i) the following relationship is true[2]:

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

where R_i is the actual worst case response time of the process. It is obtained by summing the process' execution time, its blocking time, and the interference it gets from higher priority process executing in the interval $(0, R_i]$.

Equation (2) can be solved[3] by giving an initial estimate to R_i:

$$R_i^0 := R_{i-1} - B_{i-1} + C_i + B_i$$

and then iterating around on the calculated values of the left hand side of the equation:

$$C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j = R_i^n \qquad (3)$$

In general there may be more than one solution to equation (3). The smallest (positive) value of R_i is obtained if the initial estimate for the iteration is less than this required value. The above initial estimates ensure this. The iteration is terminated once R_i^n equals R_i^{n-1} or when $R_i^n > D_i$ at which point the process cannot be guaranteed to meet its deadline.

In order to modify the scheduling tests so that they cater for the characteristics of a realistic kernel it is necessary to work with a formulation such as that described in equations (2) and (3). This is partially due to the need to move away from optimal priority assignments, but is also necessary as there are situation in which computation times vary and there is jitter on the release of periodic processes. The standard rate monotonic formulaes are not directly applicable as the prerequisites of that theory are no longer true.

All the above schemes apply to a "running" system and do not address the initialisation phase of the process set. Often the first iteration of a process takes more time as data structures need to be constructed and initialised. The kernel may also have considerable time penalties associated with creating processes. If these issues are important to an application then equations (2) and (3) can be modified to reflect the interference and blocking experienced during the initial phase of the system (or in a new mode of operation). These modifications are, however, not described here. Rather attention is focussed on the normal executing phase of the application. In this phase all the process are deemed to exist (i.e. none are created and none terminate).

3. The Kernel and Performance Measures

The kernel under investigation was a modified stand-alone Ada run-time system for York Ada[1].

The hardware platform consists of a Motorola 68020 (with an internal cache) and a floating point unit, a clock interrupt source and an interval timer. Other interrupts are allowed but no I/O devices use DMA. The code for the kernel is written in a mixture of C and assembler.

The structure of the kernel is as follows:

- Time management
  - A hardware clock source interrupts the kernel on a periodic basis.
  - Periodic processes execute a *delay_until* kernel routine at the end of their

period, and are placed on a delay queue (which is ordered according to the size of the outstanding delay).

  - The code associated with the clock interrupt handler is responsible for moving processes from the delay queue to the dispatch queue.

- Process management

  - A dispatch queue holds all runnable processes in priority order.
  - Ada tasking semantics are implemented and available for all non-real-time tasks.
  - An asynchronous IPC mechanism based on a passive object with a ceiling priority is provided for hard real-time task communication. A passive object is represented as an Ada passive task and has semantics similar to those provided by protected objects in Ada 9X.

- Interrupt management

  - Interrupts may be mapped to passive object entries. Sporadic tasks can be queued on different entries in the same passive object and be released by the interrupt.
  - Interrupts may be disabled.

- Memory management

  - Dynamic memory allocation is supported but assumed not to be used if the system contains hard real-time tasks.

- Exception management

  The kernel supports the Ada exception handling model; exception handlers are searched for dynamically at run-time.

3.1. Kernel Time Management Performance Characteristics

In this section we give the actual measurements that were obtained for the kernel for clock handling and manipulating the delay queues. These were obtained to an accuracy of 4 microseconds (µs).

The clock interrupt is set at 100 Hz; i.e. the time between clock interrupts is 10 milliseconds (ms). The interrupt handler for the clock takes 16µs if there are no processes on the delay queue. If there are then this rises to 24µs (cache disabled) or 20µs (cache enabled). When processes are moved from the delay queue to the dispatch queue then there is additional overhead. For one process moved, the total time in the interrupt handler is 88µs (cache disabled), 76µs (cache enabled); for each subsequent process an additional 40µs (cache disabled), 26µs (cache enabled) must be added.

The above results show that there is a great variation in the time spent in the clock handler. This would be the case even if queue operations were of constant length as it is the number of operations that is changing. For example if 20 processes are coming off the delay queue at the same time (which would occur at the LCM of their periods), the cost is 590µs compared with the minimum possible of 16µs. Because the clock interrupt handler has a priority higher than any application process then even high priority process must suffer long delays while lower priority processes are moved from one queue to the other. This is an example of priority inversion in the kernel and may well be responsible for the results reported in the case study by Locke *et al*[7].

3.2. Ramifications for the Scheduling Models

The values given above for the various kernel routines are constant; they do not depend on the size of the application (other than in the number of processes involved). If a typical application process has a worst case execution time in the order of seconds then a simple model of the kernel with pessimistic values will be perfectly adequate. There are however many applications with computation times and deadline needs much closer to the time values presented above. These applications need better models of kernel constraints and application behaviour.

The variability in the duration of the clock interrupt handler is a major concern. To assume that the maximum duration will occur on every iteration is excessively pessimistic. The behaviour of the clock handler must be modelled more accurately.

4. Adapting the Scheduling Models for Realistic Clock Handling

Four approaches are considered; they give a progressively more accurate model of the behaviour of the handler. The more exact models are appropriate when the pessimism introduced by the other models is significant for the application. Comparisons using the kernel characteristics detailed above are given in section 4.1.4.

4.1. Simple Clock Models

It was noted above that the cost of moving n processes from the delay queue to the dispatch queue can be represented by the equation:

$$Cost = ct_s + (n - 1)ct_m$$

where ct_s is the cost of moving one process and ct_m is the addition cost (per process) of moving more than one process at a time. If N is the maximum number of periodic process in the application then the maximum time in the clock handler is given by

$$C_{clk} = ct_{fxd} + ct_s + (N - 1)ct_m$$

where ct_{fxd} is the fixed overhead associated with the clock interrupt.

The clock handler can therefore by modelled as a single process with period T_{clk} and computational load of C_{clk}. This however is pessimistic as it assumes this overhead on each clock tick.

Let $T_{\min}$ be the shortest period for any application process. The clock handler, which is of higher priority than this process, will execute M times in any interval of duration $T_{\min}$; where M is given by:

$$M = \left\lceil \frac{T_{\min}}{T_{clk}} \right\rceil$$

If M is much greater than 1, the maximum load can be spread over a longer period. In this situation, the clock handler is modelled as a process with period $T_{\min}$ and computation time C'_{clk}:

$$C'_{clk} \ = \ M{*}ct_{fxd} + M{*}ct_s + (N-M){*}ct_m$$

This assumes $M \le N$.

In both of the above approaches the priority of the clock P_{clk} is set to the highest value (regardless of its period or deadline). A simple check will determine if the clock handler itself can finish its computations in time to handle the next interrupt. The above values of T_{clk} and C_{clk} are used to calculate the interference on application processes (using equation 3).

4.2. Modelling the Action of the Clock Handler Directly

If the overhead of the clock handler is significant, in terms of the application's computation times and periods, then it is necessary to model the clock handler as a number of different processes. The fixed overhead ct_{fxd} is easily modelled as a periodic process with period equal to the clock frequency and computation time set at the value appropriate for no processes being released (but processes on the queue). This process is given a high (the highest) priority, P_{clk}.

To model the rest of the clock overhead, each periodic process must be coupled to a fictitious process that has the same period (as the process) but has a computation time equal to the cost of removing one process from the delay queue, ct_s. The priority of these fictitious processes is also P_{clk}. This set of fictitious processes, and the fixed overhead, all induce interference on the application processes.

This model is reasonably accurate if ct_s is not significantly greater than ct_m (which takes into account the reduced cost of removing more than one process from the delay queue). To benefit from this reduction in cost, a more complicated model is needed. This is derived in the next section.

4.3. A More Exact Model

Equation (2), and its derivative equation (3) are based on the notion of a critical instance. They assume that all higher priority processes are released at the same time as the process being investigated. They thus assume that N fictitious process are interfering; but they do not take into account any reduction in interference that would occur if more than one fictitious process is released at the same time.

If equation (3) balances (i.e. $R_i^n = R_i^{n+1}$) then the computation time C_i will fit into the interval (0, R_i^n]. In any such interval it is possible to calculate the number of times, K the clock handler could have executed:

$$K \ = \ \left\lceil \frac{R_i^n}{T_{clk}} \right\rceil$$

It is also possible to calculate the number of movements, V, there has been from the delay queue to the dispatch queue:

$$V \ = \ \sum_{j}^{F} \left\lceil \frac{R_i^n}{T_j} \right\rceil$$

The summation is over all fictitious processes. If $K \ge V$ then we must assume, for the worst case, that each movement occurs on a different clock tick (hence all must be costed at ct_s). If this is not the case a reduced cost can be used. Thus equation (3) becomes:

$$C_i \;+\; B_i \;+\; I^F \;+\; \left\lceil \frac{R_i^{n-1}}{T_{clk}} \right\rceil ct_{fxd} \;+\; \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil C_j \;=\; R_i^n \qquad (4)$$

where I^F is the interference from the fictitious processes given by:

$$I^F \;=\; \begin{cases} \text{if } K \geq V: & V{*}ct_s \\ \text{if } K < V: & K{*}ct_s + (V-K){*}ct_m \end{cases} \qquad (5)$$

There is a final refinement that can be applied to task sets that are guaranteed to execute in phase (i.e. all tasks are released at time 0 and do not drift in time relative to one another). The above analysis assumes that tasks can move between queues at any tick of the clock. But this is not true for tasks in phase. If GCD is the greatest common divisor of the task periods then the number of clock ticks that could give rise to task movement is given by:

$$\left\lceil \frac{R_i^n}{GCD} \right\rceil$$

Thus K (as used above) can be redefined to be:

$$K \;=\; MAX\left\{ \left\lceil \frac{R_i^n}{GCD} \right\rceil \;,\; \left\lceil \frac{R_i^n}{T_{clk}} \right\rceil \right\}$$

Equations (4) and (5) remain valid with the new value of K.

4.4. Comparison of Models

The above approaches can only really be compared within the context of a specific application. However, for illustrative purposes an example will be given that uses the characteristics of the kernel discussed earlier. Table 1 holds the important kernel characteristics.

T_{clk}	ct_{fxd}	ct_s	ct_m
10ms	20µs	56µs	26µs

Table 1: Kernel Clock Characteristics

The application consists of 21 periodic processes; the shortest period is 50ms. There is one such process, three 100ms process, one 150 ms process, three 200ms processes, one 250ms process and one 300ms; ten other processes have periods of 500ms, the last process has a period of 1000ms. The clock device interrupts every 10ms and hence none of the application processes suffer release jitter. Note that the LCM of the task set is 1000ms and the GCD 50ms.

The following analysis provides results for the highest priority process (T_1 = 50ms), the 10th highest priority process (T_{10} = 300ms), and the lowest priority process (T_{21} = 1000ms). Five models are compared:

(1) Clock handler modelled as a single process executing every 10ms (i.e. the actual clock frequency).

(2) Clock handler modelled as a single process executing every 50ms (i.e. the period of the shortest application process).

(3) Clock handler's fixed overhead modelled as a single process executing every 10ms; each periodic process having a fictitious process to model its extra overhead (as described in section 4.1.2).

(4) Clock handler's fixed overhead modelled as a single process executing every 10ms; each periodic process having a fictitious process to model its extra overhead; equations (4) and (5) used to reduce the pessimism of model (3), K set to the number of clock ticks.

(5) Clock handler's fixed overhead modelled as a single process executing every 10ms; each periodic process having a fictitious process to model its extra overhead; equations (4) and (5) used to reduce the pessimism of model (3), K set to the number of clock ticks that can cause task movement.

The table below presents the results for the three processes. The R^n value represents the deadline that would be calculated from equation (3) if the overhead of the clock is ignored (this value is given in milliseconds, and its actual calculation is of no relevance here). All the other values in the table represent the total extra interference that must be added according to the model being used (the five models are represented in the last five columns); these values are all given in microseconds.

	T	R^n	(1)	(2)	(3)	(4)	(5)
T_1	50ms	32ms	2384	796	1256	746	656
T_{10}	300ms	192ms	11920	3184	1968	1728	1248
T_{21}	1000ms	595ms	35760	9552	5064	4794	3354

Table 2: Sample Comparisons

Note that for the shortest period process, model (2) is less pessimistic than model (3); this is because it has been able to take into account the lower overhead of removing groups of processes from the delay queue. The difference in interference predicted by the three models is very marked for the lowest priority process. Indeed models (1), (2) and (3) would all push the response time for this process over the 600ms value. At this point significant extra interference from the 50ms process, the three 100ms processes, the two 200ms processes and the 300ms process would need to be included. This has the result of pushing the response time close to 800ms. Models (4) and (5) keeps the response time below 600 and hence equation (5) will still balance.

This last point is significant. Although the differences between the models is small it can make a sizable difference to the response time calculated. If T_{21} had a deadline of 750ms then models (4) or (5) would, correctly, predict that it would make its deadline by over 150ms; the other models would predict that the deadline would be missed.

5. Conclusions

The tailoring of scheduling models to fit particular application requirements and kernel characteristics is key to the development of an engineering discipline for building real-time systems. This paper has reported on the properties of a specific kernel. The technology used in this kernel is typical of a broad class of kernels.

The movement of "low" priority processes from the delay queue to the dispatch queue was found to be a significant source of priority inversion. To reduce the pessimism of assuming a worst case behaviour, for the clock handler, on each iteration it is possible to model the extra overhead as the interference from a set of fictitious period process (one per application periodic process).

References

1. *York Ada Compiler Environment (York ACE) Reference Guide*, York Software Engineering Limited (1991). (Release 5.1)
2. 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).
3. N.C. Audsley, K. Tindell, A. Burns, M.F. Richardson and A.J. Wellings, ‘‘The DrTee Architecture for Distributed Hard Real-Time Systems’’, *Proceedings IEEE Workshop on Architecture Support for Real-time Systems*, San Antonio, Texas (December 1991).
4. 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).
5. 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).
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. 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).