DEADLINE MONOTONIC SCHEDULING THEORY†

N.C. Audsley, A. Burns, M. F. Richardson and A.J. Wellings

Department of Computer Science
University of York, UK

Abstract Scheduling theories are now sufficiently mature that a genuine engineering approach to the construction of hard real-time systems is possible. In this paper we discuss the application of Deadline Monotonic Scheduling Theory (DMST). This theory is an extension of the more familiar approach based on rate monotonic priority assignment. The model presented can accommodate periodic and sporadic processes, different levels of criticality, process interaction and blocking, precedence constrained processes and multi-deadline processes. It is particularly well integrated with the use of Immediate Priority Ceiling Inheritance for control over process blocking. A basic pseudo-polynomial schedulability test is outlined and then supplemented by the introduction of offsets to control jitter, and period transformation to enable critical (hard) processes to be "protected" during potential transient overloads. These mathematical techniques derived within DMST can help designers experiment with alternative formulations and prove essential properties of systems before they are deployed.
Keywords hard real-time scheduling, rate monotonic, schedulability constraints

1. INTRODUCTION

The significance of the recent developments in Rate Monotonic Scheduling Theory (RMST) is that it has enabled an engineering basis to be provided for the production of hard real-time systems. Hard real-time systems being those that can give rise to catastrophic failure if the timing requirements are not met. RMST prescribes a run-time environment of preemptive priority based dispatching and, essentially, static priorities. Priorities are assigned to processes according to their periods (the shorter the period the higher the priority); importance is not used directly to effect priority assignment. Although the original rate monotonic scheme[7] required all process to be periodic and independent of each other, the theory can now accommodate sporadic activities[11], blocking[10] (due to process interactions) and transient overloads[8] (so that critical processes do not miss their deadlines if a transient overload occurs).

Deadline Monotonic Scheduling Theory (DMST) is an extension of RMST. It relaxes one of its axioms; namely it allows the deadline of a process to be less than the period. With DMST priorities are assigned in accordance with deadline (the shorter the deadline the higher the priority). Note that priorities are again static and that preemptive priority based dispatching is required; *it is not an earliest deadline formulation.*

To enable DMST to be used in an engineering context, adequate schedulability tests must be available. In a recent paper[2] we presented a necessary and sufficient test (among others) for a process set where deadlines were less than or equal to period length. The tests assume a critical instant[7] for which, as Leung *et al* showed[6], the deadline monotonic priority assignment is optimal (for static priorities with deadline not necessarily equal to period). The test presented is pseudo-polynomial in computational complexity.

An important aspect of the test is that it checks the feasibility of a process set with arbitrary priority assignments — it is not just applicable to deadline monotonic assignment. This enables *importance* to have an influence over the priority mapping (this is discussed in Section 7 on transient overloads). The test also provides information about the actual deadline that could be guaranteed with that process set. This is useful to control engineers who, in general, would like all deadlines to be as short as possible (as well as being guaranteed).

† This work is supported, in part, by the Information Engineering Advanced Technology Programme (UK), Grant GR/F 35920/4/1/1214; and the European Space Agency (ESTEC Contract 9198/90/NL/SF).

The advantages of the deadline monotonic approach over the rate monotonic approach can be summarised as follows:

- It allows a greater range of requirements to be accommodated (including all those appropriate for rate monotonic analysis).
- It allows the completion of a process' execution to be defined more precisely (i.e. it minimises jitter).
- It allows sporadic processes to be easily incorporated.
- It enables precedence constrained processes to be scheduled.
- It can accommodate multi-deadline processes.
- It is a more useful abstraction in distributed systems.

Many of these advantages are discussed in detail in the remainder of the paper. In particular, we describe, in the following sections, the application and refinement of DMST to: sporadic activities, process blocking, precedence constrainted processes, multi-deadline processes, jitter control and transient overloads. Space considerations do not allow a comprehensive analysis of the role of deadline monotonic priority assignment in distributed systems. In short: the deadline of a process can be interpreted to mean the time by which a generated message should arrive at some destination node. If P is the (worst case) time interval needed for the message to be communicated across the network, and T is the period of the process then its computation must be complete by the deadline $T-P$.

2. SPORADIC ACTIVITIES

Sporadic processes are characterised by their maximum arrival rate (or their minimum arrival interval) and their deadline. In general there is no direct relationship between these two factors. To force arrival rate to be reduced to the deadline value leads to poor utilisation or the need to use complex bandwidth preserving algorithms[11]. DMST will directly support sporadic processes; any number of which can be guaranteed[2].

Assume that the minimum inter-arrival time for some sporadic process is m; and that the deadline associated with each release of the sporadic process is D. In the worst-case the sporadic behaves exactly like a periodic process with period m and deadline D where $D \leq m$. The characteristic of this behaviour is that a maximum of one release of the process can occur in any interval $[t, t+m]$ where release time t is at least m time units after the previous release of the process. This implies that to guarantee the deadline of the sporadic process, 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.

3. PROCESS BLOCKING

In realistic real-time systems processes interact in order to satisfy system-wide requirements and to share resources. The forms that these interactions take are varied, ranging from simple synchronisation to mutual exclusion protection of a non-sharable resource, and precedence relations (see Section 4).

RMST incorporates a number of techniques for dealing with blocking. All strategies involve bounding the time a process is blocked by a lower priority process. A simple way of representing the timing requirement for a set of n processes is:

$$\forall i : 1 \leq i \leq n : C_i + I_i + B_i \leq D_i$$

Where C_i is the computation requirement of process i, B_i is the maximum blocking time process i suffers (from lower priority processes), and I_i is the interference process i gets from higher priority processes (note B_n and I_1 are 0; n is the lowest priority).

An identical approach is possible with the DMST. Priority inheritance, ceiling protocols[10] and Immediate Priority Ceiling Inheritance[3] are all applicable.

3.1. Immediate Priority Ceiling Inheritance

Although DMST can be used with any bounded blocking model it seems particularly well integrated with what we shall term *Immediate Priority Ceiling Inheritance*, IPCI. This protocol (though not explicitly named) is described by Baker[3] and alluded to by Klein and Ralya[5]. It is currently been proposed as the preferred implementation for *protected records* in Ada9X.

Processes interact via passive enties which may be called *protected objects*. Such protected object are critical regions that require mutual exclusion over their operations. Each protected object has a ceiling priority defined which is at least the maximum of the priorities of the processes that use it. It is a static attribute set before run-time.

When a process accesses a protected object its priority is immediately raised to the ceiling level. As a consequence a process can only experience a single block during any period. For example if a low priority process L shares a protected object with a high priority process H then it will run with at least the priority of H while in the protected object; as a result, no other process with priority less than or equal to H can execute (and enter another protected object used by H). When H becomes runnable it will experience at most a single block. Moreover this blocking will be experienced at the very beginning of its execution cycle (it will not be able to preempt L if L is running with the priority of H).

Once this initial block is over, H will run through its cycle without interference (other than possible preemption from an even higher priority process).

The use of immediate priority ceiling inheritance (IPCI) is more straightforward to implement than the ceiling protocol[10]. It also has the advantage that during execution it involves less context switching than the former method. Its blocking characteristics are equivalent in all cases apart from when a process only occasionally uses a protected object (i.e. not every cycle). With the ceiling protocol, the block is only experienced when a call is actually made on the protected object. The IPCI strategy can cause a block at the beginning of every period. Of course in both approaches blocking will only occur if a lower priority process happens to be using a critical region at the time a higher priority process becomes runnable.

4. PRECEDENCE CONSTRAINTS

Precedence relations between processes are often found in the software architecture of real-time systems. System wide requirements can be represented as a *transaction* flowing through a number of different processes. Precedence constraints can also be used to serialise access to system resources. In the following we consider a group of precedence constrained processes executing on the same processor.

A transaction is structured as, for example, three processes, P, Q & S. P runs first then Q and then S. Table 1 illustrates some typical values for these processes (the analysis assumes that all three processes run on the same processor).

	C	D	T	P
P	2	5	20	2
Q	2	4	20	1
S	4	7	20	3

Table 1

Note first that the periods of the three processes are the same, and that the overall deadline for the transaction is 16.

All the schedulability tests for the deadline, and rate, monotonic approaches are based upon the concept of a critical instant. This assumes that all process are released (at some time) together. If we apply the critical instant concept to the above process set then the system is clearly unschedulable — the optimal priority assignment is as shown in the table (i.e. deadline monotonic), hence Q will run first (which it shouldn't) for 2 units followed by P (a further 2 units) and then S will be unable to complete its 4 units of execution before its deadline.

The critical instant concept is clearly too pessimistic for precedence constrained processes. We know that they never wish to execute together. Both Q and S require an *offset*. That is they cannot execute at the start of the period.

A simple transformation can be applied to processes with offsets that share the same period. We relate the deadlines of all processes not to their start times but to the start time of the transaction. This will not effect P but it will mean that Q and S have their deadlines stretched (we refer to the new process as Q' and S'). Table 2 now has the new deadlines and priorities for the process set:

	C	D	T	P
P	2	5	20	1
Q'	2	9	20	2
S'	4	16	20	3

Table 2

The priority model will now ensure that P executed first then Q' and then S'. Moreover the new process set is schedulable and would actually allow other processes to be given priorities interleaved with this transaction. As the processes share the same period only one of them will experience a block at the start of its execution — if IPCI is used.

It must be emphasised that this represents just one way of scheduling precedence related processes. It is possible to construct process sets that are only schedulable if priorities increase down the precedence relationship (and some form of signal is used to stop a process running before its predecessor)[4].

5. MULTI-DEADLINE PROCESSES

The above approach for dealing with precedence constrained processes has a further property that will enable multi-deadline processes to be accommodated. Processes can exist that have more than one deadline, they are required to complete part of their computations by one time and the remainder by a later time. This can occur when a process must read an input value very early in the period and must produce some output signal at a later time.

To implement multi-deadline processes it is necessary for the run-time system interface to facilitate dynamic priority changes. The process is modelled as a precedence related transaction. Each part of the transaction is thus assigned a priority (as described above). The process actually executes in a number of distinct phases, each with its own priority: for example a high priority to start with until its first deadline is met, then a lower priority for its next deadline.

6. JITTER CONTROL

Where real-time software is incorporated into control systems there is a clear need for output jitter to be reduced. Control algorithms are particularly sensitive to variations in the times at which outputs

are generated or in the intervals between an input value been obtained and an output produced. Proponents of cyclic executives often argue that their techniques give much better predicatability than kernels based on preemptive priority based dispatching. The combination of the use of the deadline monotonic approach and process offsets can however give close control over jitter.

With the rate monotonic scheme the completion of a process can occur at any time in the interval (C_{min}, T]; where C_{min} is the minimum execution time of the process (T is the period). If more control is needed but the cycle time should not be reduced then the deadline (D) for the end of the computation must be moved away from the period end (i.e. reduced). The process can then be guaranteed to have finished in the interval (C_{min}, D]. D can be made arbitrarily close to C_{min} as long as the system is still schedulable. Further control comes from the use of *related* processes and offsets. Related process are those that have precedence constraints but cannot immediately execute one after the other. They are more typical of distributed transactions in which communication delays must be taken into account. If the process described above is given an offset of O then it can be guaranteed to complete in the interval ($O+C_{min}$, D].

By assuming a critical instance, the schedulability tests are tractable but pessimistic if processes have offsets that imply that they are never released together. Schedulability tests for processes set with offsets have recently been derived[1].

7. CRITICALITY ISSUES

In safety critical applications it is important that the executing system is predictable at all times. This is particularly true when there is the possibility of deadlines being missed. The cause of a missed deadline is assumed to be a transient overload which can occur for a number of reasons:

- Calculation of worst case execution times being too optimistic.
- System overheads being greater than anticipated.
- Sporadic activities occurring more frequently than anticipated.
- Hardware failures of different kinds.

With rate monotonic static priority assignment, a transient overload will lead to the processes with the longest periods (i.e. lowest priorities) missing their deadlines. As these may nonetheless be the most critical, the period transformation technique can be used to "shorten" (e.g. halve) the period to the critical processes so that their priorities increase[9]. It should be noted however that this technique is not as straightforward as the literature would imply if processes can block; the transformed process must be constructed so that it is not delayed while holding a system resource.

On a single processor system (or node of a distributed system) it is not really the deadlines that are used at run-time but the relative priorities of the processes. A critical process can therefore be given a higher priority than a non critical one and the tests reported in Section 1 can be applied immediately. Recall that they were constructed to work with arbitrary priority assignments.

Real-time processes can often be partitioned into a number of distinct levels of criticality. In the following discussion assume there are two levels: hard and soft. The priority assignment activity proceeds as follows. All priorities for hard processes are greater than the priorities of soft. In each partition, priorities are assigned according to the processes' deadlines (e.g. via the deadline monotonic scheme). When all real-time processes have been assigned priorities then the schedulability of the process set is tested.

The application of the schedulability test could have one of a number of possible outcomes:

(a) All processes are schedulable.

(b) Some hard processes are not schedulable.

(c) All hard processes are schedulable, but some soft are not.

Eventuality (b) is clearly serious as it implies that not all hard safety-critical deadlines can be guaranteed. To alleviate this requires the deadlines to be re-examined (lengthened if possible) and/or the worst case execution times reduced (by reducing functionality).

If it is only soft processes that fail the test then one of three possible actions must be taken:

- make no change to the priorities
- promote a soft process
- period transform a hard process

The motivation for making no change is that at run-time more capacity will be available and there is a good chance that the soft process will meet its deadline, or not miss it by much. This may be quite satisfactory for soft process that do not have acute deadlines. Capacity being generated from computations taking less than worst case and sporadics not arriving at their maximum frequency.

If minimum or average execution times are known for all processes then it would be possible to see under what circumstances the soft process will meet its deadline. Clearly if it cannot be met under minimum load then it will never meet its deadline during actual execution and hence some slackening of requirements must be made. It would (with sufficient data) be possible to produce a probabilistic assessment of how often the soft process will meet its deadline; and when it does not, by how much it

overruns.

If a soft process is promoted then it will be allocated a priority according to its deadline within that set. For all intents and purposes it is now a hard process. The main candidates for this promotion are soft processes that have short deadlines and small computational requirements. They will benefit from having a higher priority but should not interfere too much with the original hard set. Obviously once a process is promoted the tests must be re-applied.

The use of period transformation is considered below. It is an appropriate technique when there exists a hard process with a long deadline and not insignificant computation requirement. Such a process may make a number of soft processes with shorter deadlines unschedulable.

7.1. Period Transformation

It was noted above that the assignment of priorities in the deadline monotonic approach takes no account of criticality. A transient overload may lead to a missed deadline in the most critical process — if it has the longest deadline. This property of deadline monotonic scheduling is identical to that found in rate monotonic theory. To counter this in the rate monotonic theory, period transformation is advocated[9]. The purpose of this section is to show how that approach can be applied to deadline monotonic priority assignment.

Consider, for illustration, two process P_1 and P_2. They have the following characteristics:

	C	D	T	P
P_1	4	18	20	HIGH
P_2	7	10	10	LOW

Table 7

If P_1 is given the highest priority because of its criticality then P_2 will miss its first deadline. Alternatively if P_2 is given the highest priority because it has the shortest deadline then an overrun by this process could cause P_1 to miss its deadline.

The transformation technique allows P_1 to be transformed into a process with a shorter deadline then P_2. The means of undertaking the transformation is however not as straightforward as in the rate monotonic theory.

Consider transforming P_1 into a process with half the period. The new process executes the first half of P_1 in its first cycle and the second half in the next. The period of the new process would be 10 and the computational requirement 2, but the new deadline for this process is 8 (not 9); this is because the deadline of the second half (second iteration) must be such that the overall deadline is met after the full period of the first iteration (i.e. 10).

The new process P_1^* has a deadline (8) that is less then that of P_2 and will therefore have the highest priority as well as the highest criticality. Note the new process set is schedulable.

General Formula

If a process (C, D, T) wishes to be transformed into m iterations of a new process then the new process will have a computation time of $\frac{C}{m}$; a period of $\frac{T}{m}$ and a deadline of $\frac{T - m\,(T - D)}{m}$. It is important to note that such a transformation does not preserve schedulability (although it does preserve utilisation) — the new process set must always be re-tested

For example if P_1 is transformed into 4 then the new process will have a period of 5, a computation of 1 and a deadline of 3.

There is a limit to how far a transformation can go such that the new deadline is more than the new computation time. In the limit

$$\frac{T - m\,(T - D)}{m} = \frac{C}{m}$$

which leads to a value of m of

$$m = \frac{T - C}{T - D}$$

And hence the new process characteristics (C^*, D^*, T^*) are:

$$C^* = D^* = \frac{C(T - D)}{T - C}$$

$$T^* = \frac{T(T - D)}{T - C}$$

As this process now has computation requirement equal to deadline it must be given the highest priority; i.e. D^* must now be the shortest deadline in the process set.

The maximum allowed transformation to P_1 is 8; this would give a period of 2.5, and a computation time equal to deadline with value 0.5.

Required Transformation

The actual problem that needs to be solved when performing a period transformation is: *by how much should a process be transformed so that its deadline becomes less than X*. Where X is the deadline of some soft process. This formula is simply:

$$m = \left\lceil \frac{T}{X + T - D} \right\rceil$$

A reduction by a factor of 2 for P_1 is thus adequate (as illustrated earlier).

Implementation

There are a number of ways in which the transformation can be implemented. For example the kernel can be used to achieve the effect. The

process will be allowed to run for the reduced number of time units and then be suspended until it is appropriate for it to run again. This suspension will be done by lowering its priority to below any hard or soft priority level. It can thus run if no other real-time process is runnable.

The only possible problem with this occurs when the process is holding a resource when it is due to have its priority lowered. If protected objects with IPCI are used then although its base priority has been reduced its current actual priority will remain at the ceiling level because it is inside the protected object. The required semantics are therefore delivered automatically, although the formulae would need to be altered to account for this.

Note that in the algorithms used above the computation time for the transformed process remains constant. In reality it will increase as it instigates more context switches; however this can easily be incorporated into the formulaes.

8. CONCLUSION

Scheduling theories are now sufficiently mature that a genuine engineering approach to the construction of hard real-time systems is possible. Mathematical analysis techniques are available that can help designers experiment with alternative formulations and to prove essential properties of systems before they are deployed. In this paper we have described Deadline Monotonic Scheduling Theory so that is can accommodate periodic and sporadic processes, different levels of criticality, process interaction and blocking, precedence constrained processes and multi-deadline processes. A basic schedulability test, which can also allow deadline alterations to be analysed, is supplemented by the introduction of offsets to control jitter, and period transformation to enable critical (hard) processes to be "protected" during potential transience overloads. The model is particularly well integrated with the use of Immediate Ceiling Priority Inheritance for control over process blocking.

The advantage of DMST (and RMST) over purely static approaches (such as the cyclic executive) is that it allows unused resources to be used effectively (by other processing activity). It can also deal with increased non-determinacy in the systems environment (i.e. sporadic activity). The advantage of DMST over RMST is that it can incorporate a wider set of requirements and facilitate the use of software structures such as precedence constrained processes and multi-deadline processes. It can easily guarantee hard sporadic processes where deadline is less than minimum arrival interval.

The schedulability tests are non-trivial but remain pseudo-polynomial. Space restrictions have not allowed them to be discussed in detail in this paper. The tests clearly need to be embodied in system design support tools. The existence of these tools and the mathematical foundation to their use is one of the characteristics of an engineering discipline.

References

1. 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).
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. T.P. Baker, "A Stack-Based Resource Allocation Policy for Realtime Processes", in *Proceedings of the 11th IEEE Real-Time Systems Symposium* (1990).
4. M.G. Harbour, M.H. Klein and J.P. Lehoczky, *Fixed Priority Scheduling of Periodic Tasks with varying Execution Priority*, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, USA (1991).
5. M.H. Klein and T. Ralya, *An Analysis of Input/Output Paradigms for Real-Time Systems*, Software Engineering Institute Report CMU/SEI-90-TR-19 (1990).
6. 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).
7. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *JACM* **20**(1), pp. 46-61 (1973).
8. L. Sha, J.P. Lehoczky and R. Rajkumar, "Task Scheduling in Distributed Real-time Systems", in *Proceedings of IEEE Industrial Electronics Conference* (1987).
9. L. Sha, J.B. Goodenough and T. Ralya, *An Analytical Approach to Real-Time Software Engineering*, Software Engineering Institute Draft Report (1988).
10. 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).
11. 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).