Scheduling hard real-time systems: a review

by A. Burns

Recent results in the application of scheduling theory to hard real-time systems are reviewed in this paper. The review takes the form of an analysis of the problems presented by different application requirements and characteristics. Issues covered include uniprocessor and multiprocessor systems, periodic and aperiodic processes, static and dynamic algorithms, transient overloads and resource usage. Protocols that limit and reduce blocking are discussed. Consideration is also given to scheduling Ada tasks.

1 Introduction

An inherent characteristic of real-time systems is that their requirements specification includes timing information in the form of deadlines. Hard real-time systems are those that have crucial deadlines. Failure to meet these deadlines being as much an error as a failure in the value domain.

An acute deadline is represented in Fig. 1. The time taken to complete an event is mapped against the 'utility' this event has to the system.* Here 'utility' is loosely defined to mean the contribution this event has to the system's objectives. With the computational event represented in Fig. 1, this utility is zero before the *start time* and returns to zero once the *deadline* is passed. The mapping of time to utility between start time and deadline is application-dependent (and is shown as a constant in the Figure).

In a safety-critical system the situation may indeed be worse, with the actual damage (negative utility) resulting from an early or missed deadline (Fig. 2). With the situation depicted in Fig. 1, the failure that arises if the deadline is missed is benign. In Fig. 2, the failure becomes more severe as time passes beyond the deadline. Informally, a safety-critical real-time system can be defined as one in which the damage incurred by a missed deadline is greater than any possible utility that can be obtained by correct and timely computation. A system can be defined to be a *hard* real-time system if the damage has the potential to be catastrophic, i.e. where the consequences are incommensurably greater than any benefits provided by the service being delivered in the absence of failure).

In most large real-time systems not all computational events will be hard or critical. Some will have no deadlines attached, and others will merely have *soft* deadlines. A soft deadline is one that can be missed without compromising the integrity of the system. Fig. 3 shows a typical soft deadline. The distinction between hard and soft deadlines is a

* The time-utility functions represented in Figs. 1–4 are often called *time-value* functions; this term is not used here because of potential confusion between the 'value' domain and the *value* of the time domain. An alternative term to utility is *benefit.*

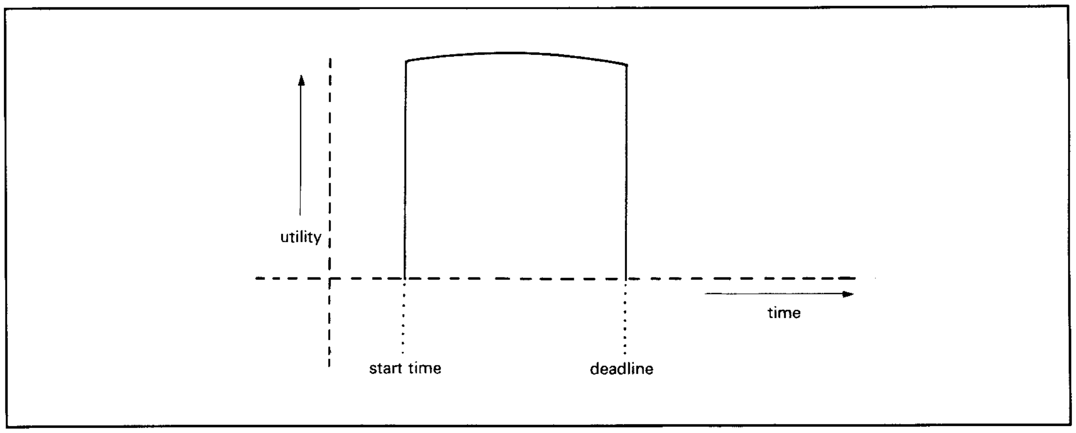

Fig. 1 A hard deadline

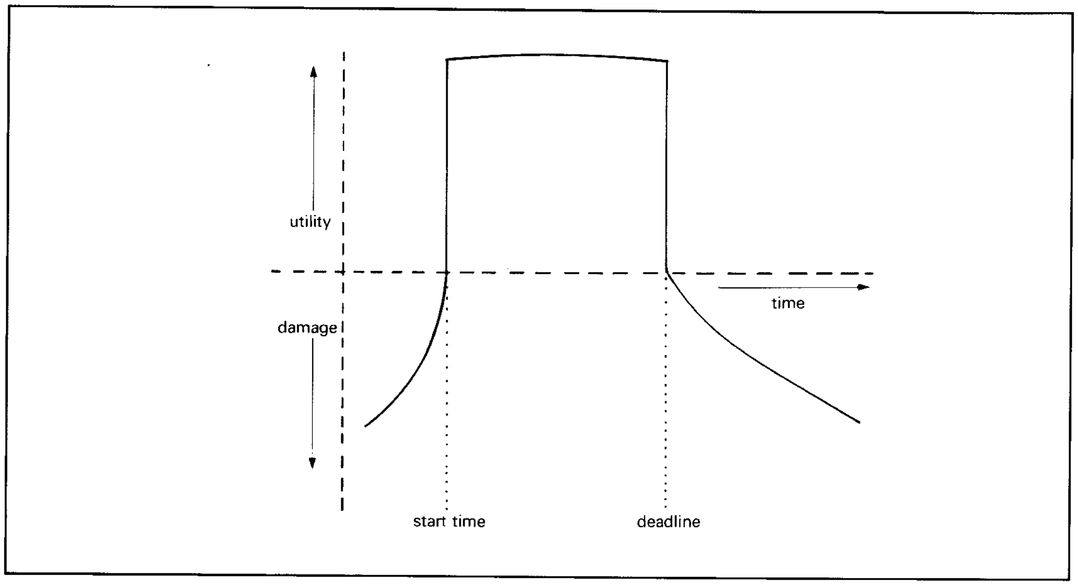

Fig. 2 A safety-critical system

useful one to make in a general discussion on real-time systems. An actual application may, however, produce hybrid behaviours. For example, the process represented in Fig. 4 has, in effect, three deadlines, D1, D2 and D3. The first represents 'maximum utility', the second defines the time period for at least a positive contribution and the third signifies the point at which actual damage will be done. The *time-utility functions* represented in Figs. 1–4 are a useful descriptive aid; they are actually used directly to control scheduling in some implementation strategies, for example, in the Alpha Kernel [1, 2].

Hard real-time systems are needed in a number of application domains, including air-traffic control, process control and 'on-board' systems such as those proposed for the forthcoming space station [3]. The traditional approach to designing these systems has focused on meeting the functional requirements; simulations and testing being employed to check the temporal needs. A system that fails to meet its hard deadlines during testing will be subject to hardware upgrades, software modifications and posthumous slackening of the original requirements. It is reasonable to assume that improvements can be made to this *ad hoc* approach.

In general, there are two views as to how a system can be guaranteed to meet its deadlines. One is to develop and use an extended model of correctness (and refinement); the other focuses on the issue of scheduling [4]. The purpose of this paper is to review the current state of scheduling theory as it can be applied to real-time systems. The development of appropriate scheduling algorithms has been isolated as one of the crucial challenges for the next generation of real-time systems [5]. For a review of the use of semantic models to describe the properties of real-time systems, see Joseph and Goswami [4].

1.1 *Deadline characteristics*

In the development of application programs it is usual to map system-timing requirements onto process deadlines. The issue of meeting deadlines therefore becomes one of

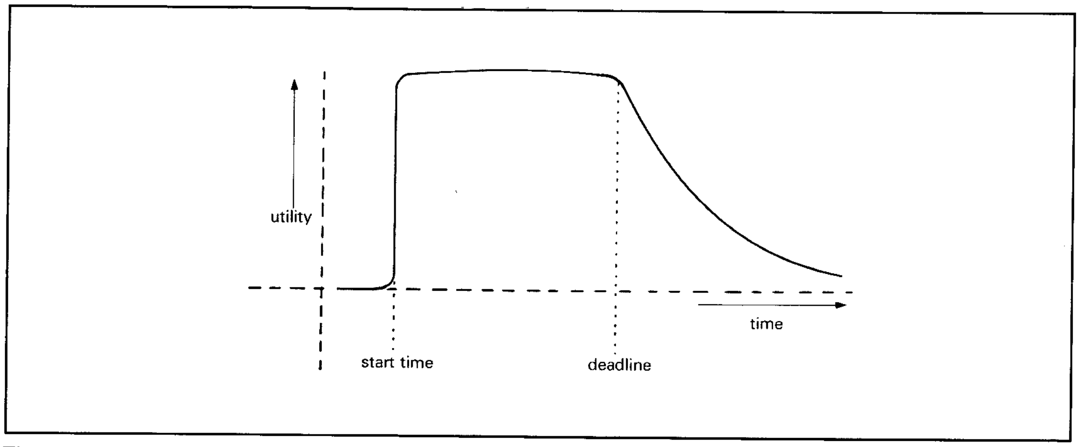

Fig. 3 A soft deadline

process scheduling, with two distinct forms of process structure being immediately isolated:

- periodic processes
- aperiodic processes.

Periodic processes, as their name implies, execute on a regular basis. They are characterised by

□ their period;
□ their deadline, often taken to be equal to period;
□ their required execution time (per period).

The execution time may be given in terms of an average measurement and/or a worst-case execution time. For example, a periodic process may need to execute every second using, on average, 100 ms of CPU time; this may rise to 300 ms in extreme instances.

The activation of an aperiodic process is essentially a random event and is usually triggered by an action external to the system. Aperiodic processes also having timing constraints associated with them, i.e. having started execution they must complete within a predefined time period. Often these processes deal with critical events in the system's environment, and hence their deadlines are particularly important.

In general, aperiodic processes are viewed as being activated randomly, following a Poisson distribution, for example. 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 analysis (there is a finite possibility of any number of aperiodic events). As a result, aperiodic processes cannot have hard deadlines. To allow worst-case calculations to be made, a minimum period between any two aperiodic events is often defined (from the same source). If this is the case, the process involved is said to be *sporadic.** In this paper, the term 'aperiodic' will be used for the general case, and 'sporadic' will be reserved for situations where hard deadlines are indicated.

1.2 Static and dynamic algorithms

Scheduling algorithms themselves can be characterised as being either static or dynamic [6]. A static approach calculates (or pre-sets) schedules for each process in advance; it requires prior knowledge of a process' characteristics but requires little runtime overhead. By comparison, a dynamic method determines schedules at runtime, thereby furnishing a more flexible system that can react to levels of activity beyond what was anticipated. Whether dynamic algorithms are appropriate for hard real-time systems is, however, a matter of some debate [7]. Certainly, in safety-critical systems it is reasonable to argue that no event should be unpredicted and that schedulability should be guaranteed before execution. This implies the use of a static scheduling algorithm. Nevertheless, dynamic approaches do have an important role.

* A slightly weaker definition of sporadic is possible if the deadlines of these processes are related to the actual number of active sporadic events. If there is an upper bound on the total processing requirement (per unit time) of all aperiodic events, then it is still possible to have a hard system. Note that, as the number of such sporadic processes approaches infinity, so must their deadlines.

- They are particularly appropriate to soft systems.
- They could form part of an error recovery procedure for missed hard deadlines.
- They may have to be used if the applications requirements fail to provide a worst-case upper limit; for example, the number of planes in an air traffic control area.

A scheduler is static and *offline* if all scheduling decisions are made before the running of the system. A table is generated that contains all the scheduling decisions for use during runtime. This relies completely upon *a priori* knowledge of process behaviour. Hence, this scheme is workable only if all the processes are effectively periodic.

A scheduler is termed *clairvoyant*

'if it has an oracle which can predict with absolute certainty the future request times of all processes.' [8]

This is difficult for systems which have non-periodic processes.

Schedulers may be *preemptive* or *non-preemptive.* The former can arbitrarily suspend a process' execution and restart it later, without affecting the behaviour of that process (except by increasing its elapse time). Preemption typically occurs when a higher priority process becomes runnable. The effect of preemption is that a process may be suspended involuntarily.

Non-preemptive schedulers do not suspend processes in this way. This is sometimes used as a mechanism for concurrency control for processes executing inside a resource whose access is controlled by mutual exclusion [9] (see later discussion on blocking).

Hybrid systems are also possible [9]. A scheduler may, in essence, be preemptive but allow a process to continue executing for a short period after it should be suspended. This property can be exploited by a process in defining a non-preemptable section of code. For example, the code might read a system clock, calculate a delay value and then execute a delay of the desired length. Such code is impossible to write reliably if the process could be suspended between reading the clock and executing the delay. These *deferred* preemption primitives must be used with care. The resulting blocking must be bounded and small, typically of the same magnitude as the overhead of context switching. The transputer's scheduler uses this approach to enable a fast context switch to be undertaken; the switch is delayed by up to 50 processor cycles. As a result, the context to be switched is small (the evaluation stack is empty) and can be accommodated in a further 10 cycles [10].

1.3 Problem space

The difficulties in providing deadline scheduling are heavily dependent on other characteristics of the real-time system. A problem space can be defined that stretches between the following extremes:

□ uniprocessor system with independent periodic processes only;
□ distributed system with interdependent periodic and aperiodic process experiencing transient overloads.

This review is structured as a progression from the possible

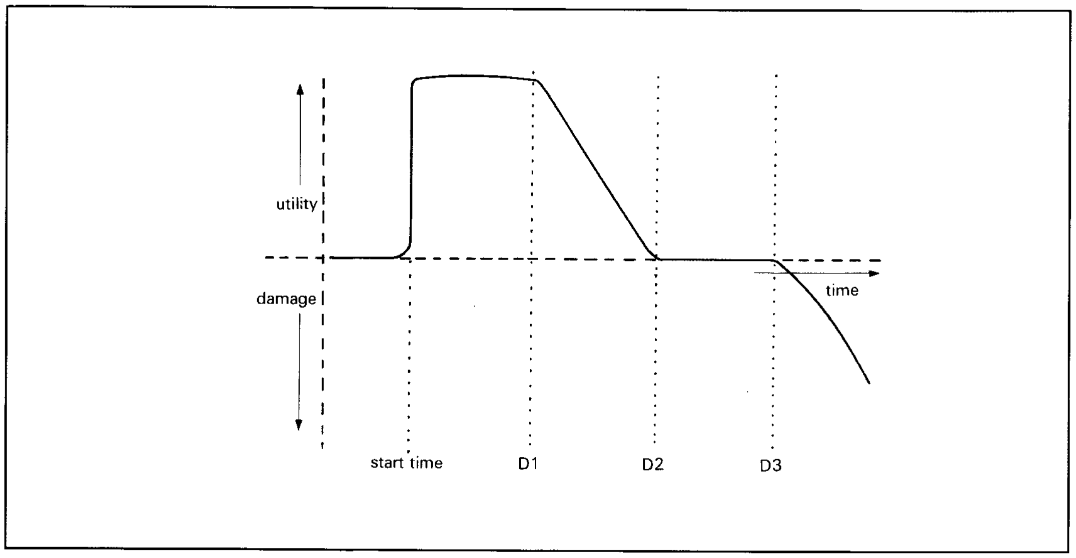

Fig. 4 A hybrid system

to the desirable. Further complications such as fault tolerance are not addressed.

2 Uniprocessor systems without blocking

First, we consider uniprocessor systems where there is a single scheduler and, by definition, only one process is executing at a time. Processes are independent of each other.

2.1 Independent periodic processes

In the simple case where the system processes are all periodic and independent of each other, it has been shown that the rate monotonic algorithm is an optimal static priority scheduling scheme [11]. By 'optimal' we mean that, if a process set can be scheduled by any fixed priority algorithm, then it can also be scheduled by the rate monotonic scheduling algorithm [12].

The rate monotonic algorithm requires a preemptive scheduler; all processes are allocated a priority according to their period. The shorter the period, the higher their priority. For this simple scheme, the priorities remain fixed and therefore implementation is straightforward. Overheads are also acceptably low. Moreover, schedulability can be checked before execution time, using either worst-case or average execution times, for example. For an arbitrary (random) collection of processes, the utilisation bound on the processor when schedulability is guaranteed is 88% [13]; worst-case utilisation is ln (2), i.e. 69%. When the periods for the processes are close to harmonic, utilisation approaches 100%; the worst case arises when the periods are relative primes. Necessary and sufficient schedulability constraint for the rate monotonic algorithm have been derived by Sha *et al.* [13, 14].

The rate monotonic formulation assumes that all tasks have period equal to deadline. If this is not the case, then the deadline monotonic algorithm is optimal [15]. New schedulability tests for deadline monotonic scheduling have recently been devised by Audsley [16].

As well as the rate and deadline monotonic approaches, there are at least two other ways of specifying optimal scheduling schemes for uniprocessors; earliest deadline [17, 18] and least slack time [19]. Slack time is the time a process has before its deadline, reduced by the execution time it still requires. These approaches allow high processor utilisation to be realised but at the cost of dynamic priorities and increased runtime overhead.

2.2 Transient overloads

It was noted earlier that a periodic process can be characterised by its average or its worst-case execution time. In general, execution times are stochastic. For hard real-time systems, it is necessary for all critical processes to be scheduled using worst-case estimates. However, it will usually be the case that some process deadlines are soft, in the sense that the occasional deadline can be missed. If total system schedulability was checked using only worst-case execution times, for all processes, then unacceptably low processor utilisations would be observed during 'normal' execution. Therefore, estimates that are nearer to the average may be used for soft deadline processes.

If the above approach is taken (or if worst-case calculations were too optimistic, or the hardware failed to perform as anticipated), then there may well be occasions when it is not possible to meet all deadlines. The system is said to be experiencing a transient overload. Unfortunately, a direct application of the rate monotonic algorithm (or the other optimal schemes) does not adequately address these overload situations. For example, with the rate monotonic approach, a transient overload will lead to the processes with the longest periods missing their deadlines. These processes may, however, be the most critical ones.

The difficulty is that the single concept of priority is used by the rate monotonic algorithm as an indication of period; it cannot therefore be used as an indication of the importance of the process to the system as a whole.

2.2.1 Period transformation: The simplest way of making the rate monotonic algorithm applicable to transient overloads is to ensure that the priority a process is assigned (due to its period) is also an accurate statement of its (relative) importance. This can be done by transforming the periods of important processes [20].

Consider, for example, two processes P_1 and P_2, with periods 12 and 30 and average execution times of 8 and 3 units, respectively. Using the rate monotonic algorithm, P_1 will be given the highest priority and all deadlines will be met if execution times stay at, or below, the average value. However, if there is an overload, P_2 will miss its deadline; this may or may not be what is required. To illustrate the use of period transformation, let P_2 be the critical hard real-time process that must meet its deadline. Its period is transformed so that it becomes a process P'_2, which has a cycle of 10 units with 1 unit of execution time (on average) in each period. Following the transformation, P'_2 has a shorter period than P_1 and hence will be given a higher priority; it therefore meets its deadlines in preference to P_1 (during a transient overload).

The process P'_2 is different from an ordinary process, with a cycle time of 10, in that it performs different actions in subsequent periods (repeating itself only every 3 periods). P'_2 is obtained from P_2 by

- either adding two delay requests into the body of the code
- or instructing the runtime system to schedule it as three shorter processes.

In general, it will not be possible to split a process into exactly equal parts. But, as long as the largest part is used for calculations of schedulability, the period transformation technique can deal adequately with transient overloads. Moreover, the transformation technique requires only trivial changes to the code or runtime support.

2.3 *Independent aperiodic processes*

Most real-time systems have a mixture of periodic and aperiodic processes. Mok [8] has shown that earliest deadline scheduling remains optimal with respect to such a mixture. However, he assumes a minimum separation time between two consecutive arrivals of aperiodic processes, i.e. they are sporadic.

Where aperiodic events are not sporadic, one must use a dynamic approach. Again, the earliest deadline formulation is optimal [21] and is the most common one used in these situations. Although optimal, it suffers from unpredictability (or instability) when experiencing transient overloads, i.e. deadlines are not missed in an order that corresponds (inversely) to the importance of that deadline to the system.

As an alternative, the rate monotonic algorithm can be adapted to deal with aperiodic processes. This can be done in a number of ways. The simplest approach is to provide a periodic process whose function is to service one or more aperiodic processes. This periodic server process can be allocated the maximum execution time commensurate with continuing to meet the deadlines of the periodic processes.

As aperiodic events can only be handled when the periodic server is scheduled, the approach is essentially polling. The difficulty with polling is that it is incompatible with the 'bursty' nature of aperiodic processes. When the server is ready, there may be no process to handle. Alternatively, the server's capacity may be unable to deal with a concentrated set of arrivals. To overcome this difficulty, a number of *bandwidth-preserving* algorithms have been proposed [22].

The general structure behind these algorithms is that any spare capacity (i.e. not being used by the periodic processes) is converted into a 'ticket' of available execution time. An aperiodic process, when activated, may run immediately if there are tickets available. This therefore allows these events to be highly responsive while still guaranteeing periodic activities. The spare capacity is derived from two sources:

☐ that which was designed into the system because of known aperiodic activity;
☐ that which has been released by periodic processes requiring less than their worst-case entitlement.

For an example of an algorithm that incorporates this model, see the sporadic server [23, 24].

3 Uniprocessor systems with blocking

In most realistic real-time systems, processes interact in order to satisfy system-wide requirements. The forms that these interactions take are varied, including simple condition synchronisations, precedence constraints and mutual exclusion protection of non-sharable resources. To help program these events, concurrent programming languages provide synchronisation primitives; for example, semaphores [25], monitors [26] (with signals or condition variables), occam (CSP) type rendezvous [27] and Ada extended rendezvous [28].

To calculate the execution time for a process requires knowledge of how long it will be blocked on any synchronisation primitive it uses. Mok [8] has shown that the problem of deciding whether it is possible to schedule a set of periodic processes, which use semaphores to enforce mutual exclusion, is NP-hard. Indeed, most scheduling problems for processes that have time requirements and mutual exclusive resources are NP-hard [29, 30]. Similarly, the analysis of a concurrent program, which uses arbitrary interprocess message passing (synchronous or asynchronous) appears to be computationally intractable. This does not imply that it is impossible to construct polynomial time feasibility tests but that necessary and sufficient checks are NP-hard, i.e. a program could fail a feasibility test but still be schedulable. However, if it passes a feasibility test it *will* be schedulable. Considerations are therefore restricted to those interprocesses interactions that enable feasibility tests to be constructed. The common form of mutual exclusion synchronisation is such a structure. This form of interaction gives rise to the client-server programming paradigm.

3.1 *Priority inversion*

The primary difficulty with semaphores, monitors or message-based systems is that a high-priority process can be blocked by lower priority processes an unbounded number of times. Consider, for example, a high-priority process H wishing to gain access to a critical section that is controlled by some synchronisation primitive. Assume that at the time of H's request, a low-priority process L has locked the critical section. The process H is said to be blocked by L. This blocking is inevitable and is a direct consequence of providing resource integrity, i.e. mutual exclusion.

Unfortunately, in the above situation H is not only blocked by L, but it must also wait for any medium-priority process M that wishes to execute. If M is executable, it will execute in preference to L, and hence further delay H. This phenomenon is known as *priority inversion*.

There are three main approaches that can minimise this effect. First, one can prohibit preemption of a process while it is executing in a critical section. This will prevent M from executing, but it will also delay H even when it does not wish to enter that particular critical section. If the critical sections are short (and bounded), then this may be acceptable. We shall return to this result in the multiprocessor Section. The other approaches to controlling priority inversion are considered below.

3.2 *Prevention of priority inversion*

Priority inversion can be prohibited altogether if no process is allowed to access a critical section, if there is any possibility of a higher priority process being blocked. This approach is described by Babaoglu *et al.* [31]. In the above example, process L would not be allowed to enter the shared section if H could become executable while L was still executing within it. For this scheme to be feasible, all processes must be periodic and all execution times (maximums) must be known. A process P can only enter a critical section S if the total elapse time of P in S is less than the 'free' time available before any higher priority process that uses S starts a new period.

The main problem with this approach is the enforced introducton of idle time into the system. For example, consider the following:

- a low-priority process L wishes to enter a critical region S at time t_1;
- a high-priority process H requires S at t_2. H has not yet been activated;
- L is not granted S because the time between t_1 and t_2 is less than that required to execute S. Hence, between t_1 and t_2 no process is utilising the processor.

The possibility of idle time reduces the processor utilisation that can be achieved. The worst-case utilisation is approximately 50%.

3.3 *Priority inheritance*

The standard priority inheritance protocol [32] removes inversion by dynamically changing the priority of the processes that are causing blocking. In the example above, the priority of L will be raised to that of H (once it blocks H) and, as a result, L will execute in preference to the intermediate priority process M. Further examples of this phenomena are given by Burns and Wellings [10].

Sha *et al.* [32] show that for this inheritance protocol there is a limit to the number of times a process can be blocked by lower priority processes. If a process has m critical sections that can lead to it being blocked, then the maximum number of times it can be blocked is m, i.e. in the worst case, each critical section will be locked by a lower priority process.

Priority inheritance protocols have received considerable attention recently. A formal specification of the protocol, in the Z notation, has been given for the languages CSP [33], Occam [34] and Ada [35].

3.4 *Ceiling protocol*

Whereas the standard inheritance protocol gives an upper limit to the number of blocks that a high-priority process can encounter, this limit is high and can lead to unacceptably pessimistic worst-case calculation. This is compounded by the possibility of chains of blocks developing, i.e. P_1 being blocked by P_2, which is blocked by P_3 etc. Moreover, there is nothing that precludes deadlocks in the protocol.

All of these difficulties are addressed by the ceiling protocol [32]. When this protocol is used

□ a high-priority process can be blocked, at most, once during its execution (per activation),
□ deadlocks are prevented,
□ transient blocking is prevented.

The ceiling protocol can best be described in terms of binary semaphores protecting access to critical sections. In essence, the protocol ensures that, if a semaphore is locked, by process P_1, for example, and could lead to the blocking of a higher priority process P_2, then no other semaphore that could block P_2 is allowed to be locked. A process can therefore be delayed by not only attempting to lock a previously locked semaphore, but also when the lock could lead to multiple blocking on higher priority processes.

The protocol takes the following form [32]:

- all processes have a static priority assigned (perhaps by the rate monotonic algorithm);
- all semaphores have a ceiling value defined; this is the maximum priority of the processes that use it;
- a process has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher priority processes;
- a process can only lock a semaphore if its dynamic priority is higher than the ceiling of any currently locked semaphore (excluding any that it has already locked itself).

If there are currently no system semaphores locked, then on all occasions the locking of the first semaphore is allowed. The effect of the protocol is that a second semaphore can only be locked if a high priority process that uses both semaphores does not exist.

The benefit of the ceiling protocol is that a high-priority process can only be blocked once (per activation) by any lower priority process. The cost of this result is that more processes will experience this block.

A formal proof of the important properties above of the ceiling protocol has been given by Pilling, Burns, and Raymond [36].

3.5 *Ceiling priority*

An equivalent behaviour to that exhibited by the ceiling protocol can be obtained by raising the priority of each process to the ceiling level immediately the semaphore is locked [37]. Priorities are assigned to semaphores (critical regions) as before, i.e. maximum priority of processes that use it. If a low-priority process L shares a critical region with a high-priority process H, then it will run with at least the priority of H while in the critical region. As a result, no other process with priority less than H can execute (and lock a second semaphore). 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 priority 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 ceiling priority is more straightforward to implement than ceiling protocol. It also has the advantage that during execution it involves less context switching than the other method. Its blocking characteristics are equivalent in all cases, apart from when a process only occasionally uses a critical region (i.e. not every cycle). With the ceiling protocol, the block is only experienced when a call is actually made on the critical region. The ceiling priority 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'.

The recent analysis on the use of ceiling priorities is having an important influence on the design of programming languages [38] and real-time kernels [39]. This is considered again in Section 5.

4 Multiprocessor systems

The development of appropriate scheduling schemes for multiprocessor systems is problematic. Not only are uniprocessor algorithms not directly applicable but some of the apparently correct methods are counter-intuitive.

Mok and Dertouzes [19] showed that the algorithms that are optimal for single-processor systems are not optimal for increased numbers of processors. Consider, for example, three periodic processes P_1, P_2 and P_3 that must be executed on two processors. Let P_1 and P_2 have identical deadline requirements, namely a period of 50 units and an execution requirement (per cycle) of 25 units, and let P_3 have requirements of 100 and 52, respectively. If the rate monotonic (or earliest deadline) algorithm is used, P_1 and P_2 will have highest priority and will run on the two processors (in parallel) for their required 25 units. They will then be delayed for 25 units before repeating the 50 unit cycle. This will leave P_3 with 52 units of execution to accomplish in the 50 units that are available (on each processor). The fact that P_3 has two processors available is irrelevant (one will remain idle). As a result of applying the rate monotonic algorithm, P_3 will miss its deadline even though average processor utilisation is only 76%. However, an allocation that maps P_1 and P_2 to one processor and P_3 to the other easily meets all deadlines.

This difficulty with the optimal uniprocessor algorithms is not surprising, as it is known that optimal scheduling for multiprocessor systems is NP-hard [40–43]. It is therefore necessary (as Garey *et al.* pointed out in 1978 [44]) to look for ways of simplifying the problem and algorithms that give adequate suboptimal results.

4.1 Allocation of periodic processes

The above illustration showed that judicious allocation of processes can significantly affect schedulability. Consider another example; this time let four processes be executing on the two processors and let their cycle times be 10, 10, 14 and 14, respectively. If the two 10s are allocated to the same processor (and by implication the two 14s to the other), then 100% processor utilisation can be achieved. The system is able to be scheduled even if execution times for the four processes are 5, 5, 10 and 4, for example. However, if a 10 and a 14 were placed together on the same processor, then utilisation drops to 83%.

What this example appears to show is that it is better to statically allocate periodic processes, rather than let them migrate, and, as a consequence, potentially downgrade the system's performance. Even on a tightly coupled system running a single runtime dispatcher, it is better to keep processes on the same processor, rather than try and utilise an idle processor (and risk unbalancing the allocation).

If static deployment is used, then the rate monotonic algorithm (or other optimal uniprocessor schemes) can test for schedulability on each processor. In performing the allocation, processes that are harmonically related should be deployed together, i.e. to the same processor.

4.2 Allocation of aperiodic processes

As it appears expedient to statically allocate periodic processes, then a similar approach to aperiodic processes would seem to be a useful model to investigate. If all processes are statically mapped, then the bandwidth-preserving algorithms discussed earlier can be used on each processor (i.e. each processor, in effect, runs its own scheduler/dispatcher).

The problem of scheduling n independent aperiodic processes on m processors can be solved (optimally) in polynomial time. Horn presents an optimal algorithm that is $O(n^3)$ for identical processors (in terms of their speeds) [17]; this has been generalised more recently by Martel [45] to give $O(m^2n^4 + n^5)$.

One of the drawbacks of a purely static allocation policy is that no benefits can be gained from spare capacity in one processor when another is experiencing a transient overload. For hard real-time systems, each processor would need to be able to deal with worst-case execution times for its periodic processes, and maximum arrival times and execution times for its sporadic load. To improve on this situation, Stankovic *et al.* [46, 47] have proposed more flexible (dynamic) task scheduling algorithms.

In their approach, which is described in terms of a distributed system, all periodic processes are statically allocated, but aperiodic processes can migrate. The following protocol is used.

- ☐ Each aperiodic process arrives at some node in the network; this could be a processor in a multiprocessor system running its own scheduler.
- ☐ The node at which the aperiodic process arrives checks to see if this new process can be scheduled together with the existing load. If it can, the process is said to be guaranteed by this node.
- ☐ If the node cannot guarantee the new process, it looks for alternative nodes that may be able to guarantee it. It does this using knowledge of the state of the whole network (called focused addressing) and by bidding for spare capacity in other nodes (see below).

☐ The process is thus moved to a new node where there is a high probability that it will be scheduled. However, because of race conditions, the new node may not be able to schedule it once it has arrived. Hence, the guarantee test is undertaken locally; if the process fails the test, then it must move again.
☐ In this way, an aperiodic process is either scheduled (guaranteed) or it fails to meet its deadline.

Locating a node to which a process can migrate is problematic. Two general approaches have been identified [48, 49]:

- receiver-initiated
- sender-initiated.

The first of these approaches involves a node asking for processes to migrate to it. The second involves a node keeping a record of workloads of other nodes, so that a process may migrate to the most appropriate node. The sender-initiated method has been shown to be better under most system loads [48]. Stankovic *et al.* have investigated both approaches [50, 51]] and adopted a hybrid approach. Each node varies between the sender- and receiver-initiated, according to the local workload.

The usefulness of their approach is enhanced by the use of a linear heuristic algorithm for determining where a non-guaranteed process should move. This heuristic is not computationally expensive (unlike the optimum NP-hard algorithm) but does give a high degree of success, i.e. there is a high probability that the use of the heuristic will lead to an aperiodic process being scheduled (if it is schedulable at all).

The cost of executing the heuristic algorithm and moving the aperiodic processes is taken into account by the guarantee routine. Nevertheless, the scheme is only workable if aperiodic processes can be moved and if this movement is efficient. Some aperiodic processes may be tightly coupled to hardware unique to one node and will have at least one component that must execute locally.

The use of this dynamic scheduling approach has been incorporated into the design of the Spring Kernel [52, 53]. To minimise the overhead of migrating aperiodic processes, copies of the code of the process are held at nodes that are likely to be asked to guarantee the process.

4.3 *Remote blocking*

If we now move to consider process interaction in a multiprocessor system, then the complexity of the scheduling is further increased (i.e. NP-hard) [29, 30, 54, 55]. As with the uniprocessor discussion, we restrict ourselves to the interactions that take the form of mutual exclusive synchronisation for controlling resource usage.

In the dynamic (flexible) system described above, in which aperiodic processes are moved in order to find a node that will guarantee them, heuristics have been developed that take into account resource usage [56–58]. Again, these heuristics give good results (in simulation) but cannot guarantee all processes in all situations.

If we return to a static allocation of periodic and aperiodic processes, then schedulability is a function of execution time which is a function of blocking time. It was noted earlier that multiprocessor systems give rise to a new form of blocking, ***remote blocking***. To minimise remote blocking, the following two properties are desirable.

☐ Wherever possible, a process should not use remote resources (and thereby be subject to remote blocking).
☐ Remote blocking should only be a function of remote critical sections, not process execution time caused by remote preemption.

The first property, which also reduces interprocessor communication, can be achieved, to a certain degree, by judicious allocation of the processes, so that all processes that use a particular critical section (together with the critical section itself) reside on the same processor. This is a further argument for static allocation, and it suggests that migration of aperiodic processes could be counter-productive.

Reducing remote access represents a second criterion for allocation; the two now being

- group processes according to their periods (i.e. harmonic periods together);
- group processes according to their resource usage.

These criteria are independent and therefore not simultaneously achievable on all occasions. A compromise allocation would need to be made based on the particular characteristics of the application.

Although remote blocking can be minimised by appropriate processes deployment, it cannot, in general, be eliminated. We therefore must consider the second approach.

In a uniprocessor system, it is correct for a high-priority process to preempt a lower priority one. However, in a multi-processor system, this is not necessarily desirable. If the two processes are on different processors, then we would expect them to execute in parallel. However, consider the following example of three processes H, M and L with descending priority levels. Processes H and L run on one processor; M runs on the other but 'shares' a critical section that resides with, and is used by, L. If L is executing in the critical section when M wishes to use some data protected by it, then M must be delayed. But, if H now starts to execute, it will preempt L and thus further delay M. Even if L was given M's priority (i.e. remote inheritance), H would still execute.

To minimise this remote preemption, the critical section can be made non-preemptable (or at least only preemptable by other critical sections). An alternative formulation is to define the priority of the critical section to be higher than all local processes (using a ceiling priority). Non-preemption is the protocol used by Mok [8]. As critical sections are often short when compared to non-critical sections of code, this non-preemption of a high-priority critical section (which is another way of blocking a local higher priority process) is an acceptable rule.

Rajkumar *et al.* [59] have proved that, with non-preemption, remote blocking can only take place when a required resource is already being used, i.e. when an external critical section is locked. They go on to define a form of ceiling protocol appropriate for multiprocessor systems.

4.4 *Transient overloads*

It has already been noted that, with static allocation schemes, spare capacity in one processor cannot be used to alleviate a transient overload in another processor. Each processor must deal with the overload as best it can, i.e. by making sure that missed deadlines correspond to less important processes. If a dynamic scheme is used to allocate aperiodic processes, then some migration can be catered for. Unfortunately, the schemes discussed earlier have the follow properties (during transient overload).

- □ Aperiodic deadlines are missed, rather than periodic ones.
- □ Aperiodic deadlines are not missed in an order that reflects importance.

The first point may, or may not, correspond to an applications requirement; the second is, however, always significant if the application has any hard deadlines attached to aperiodic activity.

A compromise situation is possible (between the static and dynamic approaches) if a static allocation is used for normal (non-overload) operation, with controlled migration being employed for tolerance of transient overloads. With this approach, each processor attempts to schedule all aperiodic and periodic processes assigned to it. If a transient overload is experienced (or better still predicted), then a set of processes that will miss their deadlines are isolated. This set will correspond to the least important collection of active processes; this could be achieved using rate monotonic scheduling and period transformation.

An attempt is then made to move those processes that are destined to miss their deadlines. Note that this set could contain aperiodic and/or periodic processes. The movement of a periodic processes will be for one complete cycle of its execution. After the execution of this cycle, it will 'return' to its original processor. At the receiver processor, all incoming processes will arrive as aperiodic, and unexpected, events.

In the new processor, the new processes will be scheduled according to their deadlines. This may even cause local processes to miss their deadlines (potentially) if their importance is less than the new arrivals. A chain of migrations could then ensue (although this is unlikely). It must, however, be emphasised that migration is not a normal action, rather it is a form of error recovery after transient overload. A well specified system may never experience such events.

The advantage of this approach to transient overloads is that there is a reduced risk of deadlines being missed in the more important processes. It also deals adequately with systems in which aperiodic deadlines are more crucial than the periodic ones. Of course, an optimal scheme would miss the least important deadlines in the entire system (rather than just locally), but the computations necessary to obtain this optimum are too intensive for real-time applications.

4.5 Distributed systems

When moving from consideration of shared memory systems to distributed architectures, it is usual to encounter increased complexity. However, in the case of scheduling, the differences are not significant. This is because the analysis of shared memory systems had led to a model of parallelism that is, essentially, loosely coupled. For example, static allocation is a method usually considered more appropriate for distributed systems. In addition, the need to minimise remote action (remote blocking) is commensurate with good distributed design.

Actual algorithms for the suboptimal (but adequate) allocation of periodic processes in distributed systems can be found in References 60–64. These all use heuristics to reduce the search space. Recent work in applying *simulated annealing* [65, 66] to process allocation has produced promising results [67]. The energy function (which is used to drive the search algorithm) takes into account

- schedulability at each processor (using either rate monotonic, deadline monotonic or shortest deadline);
- memory utilisation at each processor (must be less than 100%);
- replicated processes (for fault tolerance) that must be allocated different processors;
- constrained processes; these must be allocated to a single processor or a subset of those generally available;
- network traffic (the algorithm attempts to minimise communication subject to the above four constraints).

Tindell shows [67] that the algorithm finds the optimal allocation (with test examples for which it is feasible to find the optimal by exhaustive search) and can cope adequately with large systems, for example, allocating 42 tasks to eight processors. This last example took 40 CPU seconds on a RC3260 MIPS machine; it is estimated that exhaustive search would take 10^{28} years!

We have seen in the earlier discussion on multiprocessor systems that process migration can be used to give flexibility or to counter transient overloads. One method of reducing the time penalty associated with moving a complete process from one node to another is to anticipate the migration and to have a copy of the code at the receiver node (see Spring Kernel [52, 53]). Note that copies of the hard deadlined processes may also be kept on each node to enable reconfiguration to take place after processor failure.

For instance, a two-processor system could have statically allocated jobs and be structured to meet all hard deadlines locally, even during worst-case computation times. At each node, there are also soft processes that may miss their deadlines during extreme conditions. Copies of these soft processes could be held on each node, so that a potential overload could be tolerated by moving some form of process control block between nodes.

This is a general approach; whether it is appropriate for any particular application would depend on the characteristics of the application and the hardware on which it is implemented. It would be necessary to make sure that a deadline would not be missed by an even greater margin as a result of migration. This behaviour, which is part of the phenomena known as *bottleneck migration*, can dictate the use of a strategy that precludes migration. After all, transient overloads are indeed 'transient', and so some form of local recovery may be more desirable.

4.5.1 Communication delays: Almost all distributed systems incorporate some form of communication media; typically, this is a local area network. With loosely coupled systems, communication delays are significant and must be incorporated into the analysis of worst-case behaviours. To do this, network delays must be bounded. Unfortunately, many commercially available networks are non-deterministic or use FIFO queuing, or at best only have a small range of priorities available. This gives rise to priority inversion and prohibits the calculation of useful worst-case delay times.

This is a fundamental distinction between systems (such as a communications network) that are designed to have good average performance and those that have good worst-case performance. The needs of the real-time community for time-limited behaviour are rarely taken into account in architectural designs. The ability of existing ISO/OSI Standards to solve real-time communication problems must be seriously questioned [68], although recent Standards work is aimed at improving this situation [69]. Schedulability aspects of message communication have not been investigated extensively; recent publications by Kurose *et al.* [70], Strosnider *et al.* [71] and Rodd *et al.* [72] are, however, noteworthy exceptions.

4.5.2 Remote procedure calls: In all the analysis reported so far, it is has been assumed that a program is partitioned between processors (or nodes) using the process as the unit of distribution. Access to remote resources is via shared memory under the control of a remote critical section. We have seen that execution of such critical sections must be undertaken at a high priority if remote blocking is to be minimised. An application may, however, choose to distribute a process between more than one processor. If this is done, then the process can be said to reside on one processor but its 'thread of control' may pass to another processor. In general, this will be accomplished by remote procedure calls.

This partitioning of a process is in keeping with an object-oriented view of distribution and is supported, for example, in the Alpha Kernel [73].

The use of a remote procedure introduces yet another form of remote blocking. As with critical sections, remote preemption can be minimised only if the procedure is given top priority. The invocation of a remote procedure (or object) is undertaken by a surrogate process that is aperiodic in nature). The local scheduler must execute this aperiodic process immediately if remote preemption is not to take place. If more than one 'remote' procedure requires execution at the same time, then some preemption is inevitable.

Giving surrogate processes high priorities could lead to transient overload. The local scheduler would therefore need to know how important the external process is, in order to decide which deadlines to forfeit.

4.6 Process abstraction

In most of the above discussion, it has been assumed that critical sections are protected by some low-level primitive. Remote assess can therefore be accommodated in two ways:

☐ using shared memory (shared primitives);
☐ using a remote procedure that contains the necessary calls to the local primitives.

Typically, the first of these would be used in a multiprocessor system and the second in a distributed system, but this would not be universally true.

There is, however, a different structure that is more in keeping with the use of a higher level of abstraction. Here, the critical section is constructed as part of a process, and therefore mutual exclusion is ensured. Usage of the critical section is undertaken by this server process on behalf of client processes. Requests take the form of interprocess communications, e.g. rendezvous.

To access a remote server process requires some sort of remote call. This could take the form of a procedure activation but could also be a remote rendezvous. Where critical sections are embodied in processes, remote preemption is minimised only if these processes are given higher priorities than other processes.

5 Scheduling Ada tasks

Given the importance of Ada in the real-time domain, this review will conclude by assessing the impact of the above analysis is having on the use and future of Ada.

Ada's limitations for real-time work have been the subject of four international workshops and numerous studies. Many of Ada's difficulties accrue from its inability to accommodate deadline scheduling. These difficulties have been well documented [74–81].

One class of problems concerns the specification of period activity and deadline information. As Ada does not directly provide these features, it has been proposed that extensions in the form of runtime libraries can be used [39, 82]. The other major difficulty concerns the priority model defined in the Ada Language Reference Manual (Ada LRM) [83]. There are two important issues here; the requirement that priorities are static, and the FIFO queues that are defined for entries.

In order to apply the results discussed earlier, which incorporate critical sections and mutual exclusion synchronisation, Ada programs must be structured so that resources are controlled by server tasks and used by client tasks. Server tasks are characterised as having entries; client tasks do not (there are other restrictions on server [84]). Without this structure, the schedulability of the program would be exceedingly difficult to analyse (NP-hard).

The requirement that priorities are static precludes priority inheritance but can be circumvented in one of two ways.

- Assign all tasks the same priority and use a new measure ('importance' or 'preference') that is defined to be dynamic.
- Do not assign priorities to any task (server) that could inherit a priority. The Ada LRM says that a task without a priority can be assigned one of any value [83]. A creative interpretation of this 'feature' would allow the priority of such a task to change dynamically.

If either of these approaches is taken, and the non-determinacy of the select statement is used to prescribe a partial ordering (of alternatives) based on priority, then only the FIFO entry queue presents an (almost) insurmountable hurdle. To stop the 'fair' queue protocol from undermining the priority-based view of the system, it is necessary to ensure that, at all times, the maximum number of tasks on any entry is one. This can possibly be achieved by judicious preemptions on the part of the runtime systems. Alternatively, programs themselves can be designed so that each entry is used by only one calling task. With this approach, methods such as the ceiling protocol could be applied to Ada [75, 84, 85].

In a distributed Ada program, the non-preemption of critical sections can be interpreted as executing complete tasks or accept statements at high priority. Yet again, the Ada LRM (just) allows the latter if one of the tasks involved (the server) does not have a priority defined:

'If only one of the two priorities is defined, the rendezvous is executed with at least that priority' [83].

5.1 Protected records in Ada 9X

Clearly, Ada, as it is currently defined, is deficient in terms of its priority model. A specifically tailored runtime system, together with a creative reading of the Ada LRM, will allow solutions to be constructed for hard real-time systems, but more appropriate solutions must be sought in the revisions to the language which are due by 1993. The form these changes will take is not yet fully defined; however, it seems likely that mandatory FIFO queues and static priorities will be removed. A less prescribed language model, which leaves more freedom to the implementation, will have the advantage of allowing real-time schedulers to be constructed, where necessary, but will allow other applications to use simpler schemes.

One of the more far-reaching changes being considered is the incorporation into the language of a monitor-like passive synchronisation primitive [37]. This *protected record* will encapsulate a critical section and allow ceiling priorities to be assigned (see Section 3.5). It will also have guards to allow condition synchronisation to be programmed. This represents quite a fundamental addition to the concurrency model of Ada; but, it will significantly improve the effectiveness of Ada in building hard real-time systems.

6 Conclusion

The complexity of general scheduling has been seen to be NP-hard. Hence, the emphasis in the literature has been to address the limited problems generated from a constrained model of hard real-time systems. This was seen in the discussion on scheduling algorithms in simple uniprocessor systems, where resources, process precedence constraints and arbitrary process timing constraints were not initially considered.

Owing to these complexity considerations, suboptimal scheduling schemes must be used in architectures designed for realistic hard real-time systems. Such schemes include the prescheduling approach of MARS [86], the developments of the rate-monotonic algorithm in ARTS [87], and the heuristic scheduling approaches of the Spring Kernel [52] and Alpha Kernel [73]. Together, these indicate the feasibility of scheduling the following aspects of realistic hard real-time systems:

□ guaranteeing both periodic and non-periodic hard real-time processes on the same processor;
□ utilisation of spare time by non-critical processes;
□ initial static allocation of processes;
□ migration of processes in response to changing environment conditions or local overload.

Guaranteeing process deadlines introduces a predictability/flexibility trade-off. In the MARS kernel, all processes are periodic and have guaranteed deadlines. However, the constraints imposed on the process characteristics of the system reduces flexibility. The Alpha kernel takes a more dynamic viewpoint, based on the use of utility functions (called, in Alpha, time-value functions). This enables great flexibility, but no absolute guarantees are given to process deadlines.

The point at which a scheduling scheme resides in predictability/flexibility space is determined, to a large extent, by the degree of non-determinism, failure, degradation and dynamic change expected in the system's lifetime. If this is high, then an inflexible solution may not be appropriate. For most systems, a hybrid scheme is suitable. This entails guaranteeing hard real-time process deadlines off-line or statically, but with the kernal able to make dynamic utility-related scheduling decisions when the system changes.

In order to perform adequate hard real-time scheduling, the following problems must be addressed.

- Guaranteeing hard deadlines requires constraints to be used that are based on worst-case execution times and arrival rates. The calculation of these times relies on the accuracy of maximum blocking bounds.
- The utilisation of the CPU during runtime is low when worst-case execution times are used to calculate a schedulability bound. Thus, the scheduler is required to make decisions regarding the best usage of the spare time available. Decisions are likely to be of better quality the earlier the scheduler knows that a hard real-time process is running within its maximum execution time. Methods of communicating such information to the scheduler and algorithms, to utilise such information, are required.
- For generalised hard real-time systems, schedule analysis and scheduling algorithms must be able to cope with processes that have generalised timing characteristics; for example, periods not equal to deadline and processes with multiple deadlines in a single execution.
- Tasks in hard real-time systems are unlikely to be independent. Hence, consideration needs to be given to schedule tests and scheduling algorithms for interdependent processes.
- The implications of fault-tolerant programming techniques require consideration. For example, the recovery block technique allows for an additional block of code to be executed in an attempt to overcome a failure. This block of code must be accounted for during any worst-case execution time analysis. Similar problems exist for other fault-tolerant programming techniques.

The current research in this problem domain should lead to the development of execution environments appropriate for the next generation of real-time systems.

7 Acknowledgments

The author would like to thank the many constructive comments made on an earlier version of this text by Neil Audsley, Gerhard Fohler, Mike Richardson and Werner Schütz.

8 References

[1] JENSON, E.D., LOCKE, C.D., and TOKUDA, H.: 'A time-driven scheduling model for real-time operating systems'. Proc. 6th IEEE Real-Time Systems Symp., December 1985

[2] JENSON, E.D., NORTHCOTT, J.D., CLARK, R.K., SHIPMAN, S.E., REYNOLDS, F.D., MAYNARD, D.P., and LOEPERE, K.P.: 'Alpha: an operating system for mission-critical integration and operation of large, complex, distributed real-time systems'. Proc. 1989 Workshop on Mission Critical Operating Systems, September 1989

[3] BURNS, A., and McKAY, C.W.: 'A portable common execution environment for Ada' *in* ALVAREZ, A. (Ed.) 'Ada: the design choice (Cambridge University Press, Madrid 1989) pp. 80–89

[4] JOSEPH, M., and GOSWAMI, A.: 'Formal description of real-time systems: a review'. Report RR129, Department of Computer Science, University of Warwick, 1988

[5] STANKOVIC, J.A.: 'Real-time computing systems: the next generation' *in* STANKOVIC, J.A., and RAMAMRITHAM, K. (Eds.) 'Tutorial: hard real-time systems' (IEEE, 1988) pp. 14–38

[6] CHENG, S., STANKOVIC, J.A., and RAMAMRITHAM, K.: 'Scheduling algorithms for hard real-time systems: a brief survey' *in* STANKOVIC, J.A., and RAMAMRITHAM, K. (Eds.): 'Tutorial: hard real-time systems' (IEEE, 1988) pp. 150–173

[7] BURNS, A.: 'Distributed hard real-time systems: what restrictions are necessary?' Proc. 1989 Real-Time Systems Symposium: Theory and Applications, pp. 297–304

[8] MOK, A.K.: 'Fundamental design problems of distributed systems for hard real-time environments'. PhD Thesis, Laboratory for Computer Science, MIT/MIT/LCS/TR-297, 1983

[9] TOKUDA, H.: 'Real-time critical section: not preempt, preempt or restart?' CMU Technical Report, Computer Science Department, Carnegie-Mellon University, 1989

[10] BURNS, A., and WELLINGS, A.J.: 'Real-time systems and their programming languages' (Addison Wesley, Wokingham, 1990)

[11] LIU, C.L., and LAYLAND, J.W.: 'Scheduling algorithms for multiprogramming in a hard real-time environment', *J. ACM*, 1973, **20,** (1), pp. 46–61

[12] SHA, L., and LEHOCZKY, J.P.: 'Performance of real-time bus scheduling algorithms', *ACM Perform. Eval. Rev.*, 1986, **14,** (1)

[13] LEHOCZKY, J.P., SHA, L., and DING, V.: 'The rate monotonic scheduling algorithm: exact characterization and average case behavior'. Technical Report, Department of Statistics, Carnegie-Mellon University, 1987

[14] SHA, L., and GOODENOUGH, J.B.: 'A review of analytic real-time scheduling theory and its application to Ada' *in* ALVAREZ, A. (Ed.): 'Ada: the design choice' (Cambridge University Press, Madrid, 1989) pp. 137–148

[15] LEUNG, J.Y.T., and WHITEHEAD, J.: 'On the complexity of fixed-priority scheduling of periodic, real-time tasks', *Perform. Eval.*, 1982, **2,** (4), pp. 237–250

[16] AUDSLEY, N.: 'Deadline monotonic scheduling'. Report YCS.146, Department of Computer Science, University of York, 1990

[17] HORN, W.A.: 'Some simple scheduling algorithms', *Nav. Res. Logist. Q.*, 1974, **21**

[18] BLAZEWICZ, J.: 'Deadline scheduling of tasks — a survey', *Found. Control Eng.*, 1977, **1,** (4), pp. 203–216

[19] MOK, A.K., and DERTOUZOS, M.L.: 'Multiprocessor scheduling in a hard real-time environment'. Proc. 7th Texas Conf. Computer Systems, November 1978

[20] SHA, L., LEHOCZKY, J.P., and RAJKUMAR, R.: 'Task scheduling in distributed real-time systems'. Proc. IEEE Industrial Electronics Conf., 1987

[21] DERTOUZOS, M.: 'Control robotics: the procedural control of physical processes' *in* 'Artificial intelligence and control applications' (IFIP Congress, Stockholm, 1974) pp. 807–813

[22] LEHOCZKY, J.P., SHA, L., and STROSNIDER, J.K.: 'Enhancing aperiodic responsiveness in hard real-time environment'. Proc. 8th IEEE Real-Time Systems Symp., San Jose, California, December 1987

[23] SHA, L., GOODENOUGH, J.B., and RALYA, T.: 'An analytical approach to real-time software engineering'. Software Engineering Institute Draft Report, 1988

[24] SPRUNT, B., SHA, L., and LEHOCZKY, J.: 'Scheduling sporadic and aperiodic events in a hard real-time systems'. Department of Computer Science, Carnegie-Mellon University, 1988

[25] DIJKSTRA, E.W.: 'Cooperating sequential processes' *in* GENUYS, F. (Ed.): 'Programming languages' (Academic Press, London, 1968)

[26] HOARE, C.A.R.: 'Monitors: an operating system structure concept', *CACM*, 1974, **17,** (10), pp. 549–557

[27] BURNS, A.: 'Programming in occam 2' (Addison Wesley, Wokingham, 1988)

[28] BURNS, A.: 'Concurrent programming in Ada' (Ada Companion Series, Cambridge University Press, 1985)

[29] GAREY, M.R., and JOHNSON, D.S.: 'Complexity results for multiprocessor scheduling under resource constraint', *SIAM J. Comput.*, 1975, **4,** pp. 397–411

[30] LENSTRA, J.K., RINNOOY, A.H.G., and BRUCKER, P.: 'Complexity of machine scheduling problems', *Ann. Discrete Math.*, 1977, **1**

[31] BABAOGLU, O., MARZULLO, K., and SCHNEIDER, F.B.: 'Priority inversion and its prevention in real-time systems'. PDCS Report 17, Dipartimento di Matematica, Universita di Bologna, 1990

[32] SHA, L., RAJKUMAR, R., and LEHOCZKY, J.P.: 'Priority inheritance protocols: an approach to real-time synchronisation', *IEEE Trans.*, 1990, **C-39,** (9), pp. 1175–1185

[33] BURNS, A., and WELLINGS, A.J.: 'Priority inheritance and message passing: a formal treatment', *Real-time Syst.*, 1991

[34] BURNS, A., and WELLINGS, A.J.: 'Occam's priority model and deadline scheduling'. Proc. 7th Occam User Group Meeting, Grenoble, September 1987

[35] BURNS, A., and WELLINGS, A.J.: 'Priority inheritance and message passing — a formal treatment'. Report YCS.116, Department of Computer Science, University of York, February 1989

[36] PILLING, M., BURNS, A., and RAYMOND, K.: 'Formal specification and proofs of inheritance protocols for real-time scheduling', *Softw. Eng. J.*, 1990, **5,** (5), pp. 263–279

[37] KLEIN, M.H., and RALYA, T.: 'An analysis of input/output paradigms for real-time systems'. Software Engineering Institute Report CMU/SEI-90-TR-19, 1990

[38] BAKER, T.P.: 'Protected records, time management and distribution', *Ada Lett.*, 1990, **X,** (9), pp. 17–28

[39] Ada Run Time Environments Working Group: 'A catalog of interface features and options'. ACM, ARTEWG, 1991

[40] GAREY, M.R., and JOHNSON, D.S.: 'Computers and intractability' (Freeman, New York, 1979)

[41] GRAHAM, R.L., *et al.*: 'Optimization and approximation in deterministic sequencing and scheduling: a survey', *Ann. Discrete Math.*, 1979, **5,** pp. 287–326

[42] LEINBAUGH, D.W.: 'Guaranteed response times in a hard real-time environment, *IEEE Trans.*, 1980, **SE-6,** (1), pp. 85–91

[43] LEUNG, J.Y.T., and MERRILL, M.L.: 'A note on preemptive scheduling of periodic real-time tasks', *Inf. Process. Lett.*, 1980, **11,** (3), pp. 115–118

[44] GAREY, M.R., GRAHAM, R.L., and JOHNSON, D.S.: 'Performance guarantees for scheduling algorithms', *Oper. Res.*, 1978, **26,** (1), pp. 3–21

[45] MARTEL, C.: 'Preemptive scheduling with release times, deadlines, and due times', *ACM*, 1982, **29,** (3)

[46] RAMAMRITHAM, K., and STANKOVIC, J.A.: 'Dynamic task scheduling in hard real-time distributed systems', *IEEE*

Sofw., 1984, **1,** (3), pp. 65–75

[47] STANKOVIC, J.A., RAMAMRITHAM, K., and CHENG, S.: 'Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems', *IEEE Trans.*, 1985, **C-34,** (12), pp. 1130–1143

[48] CHANG, H.Y., and LIVNY, M.: 'Distributed scheduling under deadline constraints' a comparison of sender-initiated and receiver-initiated approaches'. Proc. 7th IEEE Real-Time Systems Symposium, December 1986, pp. 175–180

[49] KRUEGER, P., and LIVNY, M.: 'The diverse objectives of distributed scheduling policies'. Proc. 8th IEEE Real-Time Systems Symp., December 1987, pp. 242–249

[50] STANKOVIC, J.A., RAMAMRITHAM, K., and ZHAO, W.: 'Distributed scheduling of tasks with deadlines and resource requirements', *IEEE Trans.*, 1989, **C-38,** (8), pp. 1110–1123

[51] ZHAO, W., and RAMAMRITHAM, K.: 'Distributed scheduling using bidding and focussed addressing'. Proc. 6th IEEE Real-Time Systems Symp., December 1985, pp. 103–111

[52] STANKOVIC, J.A., and RAMAMRITHAM, K.: 'The design of the Spring Kernel'. Proc. 8th IEEE Real-Time Systems Symp., San Jose, California, December 1987, pp. 146–157

[53] STANKOVIC, J.A., and RAMAMRITHAM, K.: 'The Spring Kernel' a new paradigm for real-time operating systems', *ACM Oper. Sys. Rev.*, 1989, **23,** (3), pp. 54–71

[54] BOKHARI, S.H.: 'A shortest tree algorithm for optimal assignment across space and time in a distributed processor system', *IEEE Trans.*, 1981, **SE-7,** (6), pp. 583–589

[55] ULLMAN, J.D.: 'Complexity of sequence problems', *in* COFFMAN, E.G. (Ed.): 'Computers and job/shop scheduling theory' (Wiley, 1976)

[56] ZHAO, W.: 'A heuristic approach to scheduling hard real-time tasks with resource requirements in distributed systems'. PhD Thesis, Laboratory for Computer Science, MIT, 1986

[57] ZHAO, W., RAMAMRITHAM, K., and STANKOVIC, J.A.: 'Preemptive scheduling under time and resource constraints', *IEEE Trans.*, 1987, **C-38,** (8), pp. 949–960

[58] ZHAO, W., RAMAMRITHAM, K., and STANKOVIC, J.A.: 'Scheduling tasks with resource requirements in hard real-time systems', *IEEE Trans.*, 1987, **SE-13,** (5), pp. 564–577

[59] RAJKUMAR, R., SHA, L., and LEHOCZKY, J.P.: 'Real-time synchronization protocols for multiprocessors'. Department of Computer Science, Carnegie-Mellon University, April 1988

[60] DHALL, S.K., and LIU, C.L.: 'On a real-time scheduling problem', *Oper. Res.*, 1978, **26,** (1), pp. 127–140

[61] BANNISTER, J.A., and TRIVEDI, K.S.: 'Task allocation in fault tolerant distributed systems', *Acta Inform.*, 1983, **20,** pp. 261–281

[62] DAVARI, S., and DHALL, S.K.: 'An on-line algorithm for real-time task allocation'. Proc. 7th IEEE Real-Time Systems Symp., December 1986, pp. 194–200

[63] CHEN, G., and YUR, J.: 'A branch-and-bound-with-underestimates algorithm for the task assignment problem with precedence constraint'. 10th Int. Conf. on Distributed Computing Systems, 1990, pp. 494–501.

[64] RAMAMRITHAM, K.: 'Allocation and scheduling of complex periodic tasks'. 10th Int. Conf. on Distributed Computing Systems, 1990, pp. 108–115.

[65] AARTS, E.H.L., and KORST, J.: 'Simulated annealing and Boltzmann machines' (Wiley Interscience, 1988)

[66] KIRKPATRICK, S., GELATT, C.D., and VECCHI, M.P.: 'Optimisation by simulated annealing', *Science*, 1983, (220), pp. 671–680

[67] TINDELL, K.: 'Allocating real-time tasks (an NP-hard problem made easy)'. Report YCS.147, Department of Computer Science, University of York, 1990

[68] LE LANN, G.: 'Critical issues in distributed real-time computing'. Proc. Workshop on Communication Networks and Distributed Operating Systems within the Space Environment, ESA(ESTEC), October 1989

[69] GRANT, K.: 'Interim report of the TCCA rapporteurs' group of ISO/TC184/SC5/WG2 on time-critical communications architecture and systems'. N220, BSI, June 1990

[70] CHIPALKATTI, R., KUROSE, J.F., and TOWSLEY, D.: 'Scheduling policies for real-time and non-real-time traffic in a statistical multiplexer'. Proc. IEEE INFOCOM 89, 1989, pp. 774–783

[71] STROSNIDER, J.K., and MARCHOK, T.E.: 'Responsive, deterministic IEEE 802.5 Token Ring Scheduling', *J. Real-Time Syst.*, 1989, **1,** (2), pp. 133–158

[72] RODD, M., IZIKOWITZ, I., and ZHAO, W.: 'RTMMS — an OSI-based real-time messaging system', *Real-Time Syst. J.*, 1990, **2,** (3)

[73] NORTHCOTT, J.D.: 'Mechanisms for reliable distributed real-time operating systems: the Alpha Kernel' (Academic Press, Orlando, 1987

[74] BURNS, A., LISTER, A.M., and WELLINGS, A.J.: 'A review of Ada tasking' (Lecture Notes in Computer Science, Springer-Verlag, 1987) Vol. 262

[75] BORGER, M., KLEIN, M., WEIDERMAN, N., and SHA, L.: 'A testbed for investigating real-time Ada issues'. Proc. 2nd Int. Workshop on Real-Time Ada Issues, *Ada Lett.*, 1988, **8,** (7), pp. 7–11

[76] BURNS, A., and WELLINGS, A.J.: 'Real-time Ada issues'. Proc. 1st Int. Workshop on Real-Time Ada Issues, *ACM Ada Lett.*, 1987, **7,** (6) pp. 43–46

[77] CORNHILL, D., 'Tasking — session summary'. Proc. 1st Int. Workshop on Real-Time Ada Issues, *ACM Ada Lett.*, 1987, **7,** (6), pp. 29–32

[78] CORNHILL, D., SHA, L., LEHOCZKY, J.P., RAJKUMAR, R., TOKUDA, H.: 'Limitations of Ada for real-time scheduling'. Proc. 1st Int. Workshop on Real-Time Ada Issues, *ACM Ada Lett.*, 1987, **7,** (6), pp. 33–39

[79] CORNHILL, D., and SHA, L.: 'Priority inversion in Ada', *Ada Lett.*, 1987, **7,** pp. 30–32

[80] LOCKE, C.D., and VOGEL, D.R.: 'Problems in Ada runtime task scheduling'. Proc. 1st Int. Workshop on Real-Time Ada Issues, *ACM Ada Lett.*, 1987, **7,** (6), pp. 51–56

[81] McCORMICK, F.: 'Scheduling difficulties of Ada in the hard real-time environment'. Proc. 1st Int. Workshop on Real-Time Ada Issues, *ACM Ada Lett.*, 1987, **7,** (6), pp. 49–50

[82] Ada Run Time Environments Working Group. 'A catalog of interface features and options'. SIG-ADA, ARTEWG, December 1987

[83] U.S. Department of Defense. 'Reference manual for the Ada programming language'. ANSI/MIL-STD 1815 A, January 1983

[84] GOODENOUGH, J.B., and SHA, L.: 'The priority ceiling protocol: a method for minimizing the blocking of high priority Ada tasks'. Proc. 2nd Int. Workshop on Real-Time Ada Issues, *ACM Ada Lett.*, 1988, **8,** (7), pp. 20–31

[85] LOCKE, C.D., and GOODENOUGH, J.B.: 'A practical application of the ceiling protocol in a real-time system'. Report of the Software Engineering Institute, March 1988

[86] DAMM, A., REISINGER, J., SCHWABL, W., and KOPETZ, H.: 'The real-time operating system of MARS', *ACM Oper. Syst. Rev.*, 1989, **23,** (3), pp. 141–157

[87] TOKUDA, H., and MERCER, C.W.: 'ARTS: a distributed real-time kernel', *ACM Oper. Syst. Rev.*, 1989, pp. 29–53

The author is with the Department of Computer Science, University of York, Heslington, York YO1 5DD.

The paper was first received on 15th November 1989 and in final revised form on 5th February 1991.