REAL-TIME SYSTEM SCHEDULING

N. Audsley

A. Burns

Department of Computer Science,
University of York, UK.

ABSTRACT

Recent results in the application of scheduling theory to dependable real-time systems are reviewed. 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 bound and reduce blocking are discussed. A review of specific real-time kernels is also included.

This paper is also available as part of the first year report from the ESPRIT BRA Project (3092), Predicatably Dependable Computer Systems, Volume 2, Chapter 2, Part II.

1. INTRODUCTION

Many papers have been published in the field of real-time scheduling; these include surveys of published literature[12, 14]. Unfortunately much of the published work consists of scheduling algorithms that are defined for systems that are extremely constrained and consequently of limited use for real applications. However it is possible to extract from this published material some general purpose techniques. The work presented here is an extension and amalgamation of two recently undertaken reviews[11, 1].

A common characteristic of many real-time systems is that their requirements specification includes timing information in the form of deadlines. An acute deadline is represented in Figure 1. The time taken to complete an event is mapped against the "value" this event has to the system. Here "value" is loosely defined to mean the contribution this event has to the system's objectives. With the computational event represented in Figure 1 this value is zero before the *start-time* and returns to zero once the *deadline* is passed. The mapping of time to value between start-time and deadline is application dependent (and is shown as a constant in the figure).

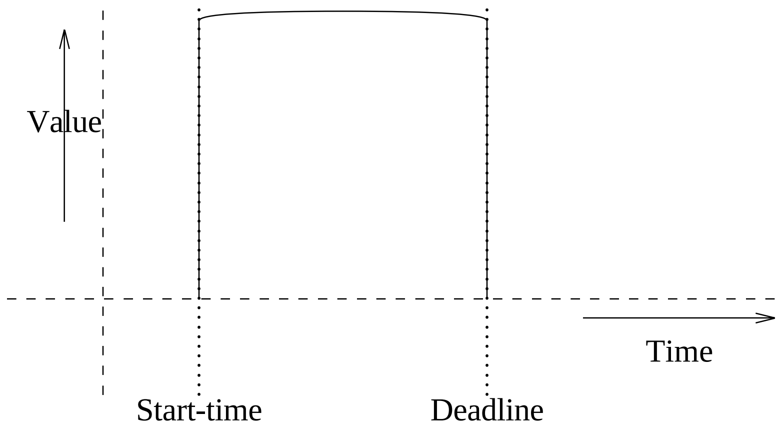

Figure 1: A Hard Deadline

In a safety critical system the situation may indeed be worse with actual damage (negative value) resulting from an early or missed deadline, see Figure 2. With the situation depicted in Figure 1 the failure that arises if the deadline is missed is benign. In Figure 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 value 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[41, 82] (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. Figure 3 shows a typical soft deadline. Note that although the value of the deadline depicted in Figure 3 does diminish after the deadline it is always positive (as time goes to infinity). The distinction between hard and soft deadlines is a 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 Figure 4 has, in effect, three deadlines (D1, D2 and D3). The first represents "maximum value", the second defines the time period for at least a positive contribution, whilst the third signifies the point at which actual damage will be done. The *time-value functions* represented in Figures 1-4 are a useful descriptive aid; they are actually used directly by Jenson in the Alpha Kernel[33, 34] and in the Arts kernel[79]. Hard real-time systems are needed in a number of application domains

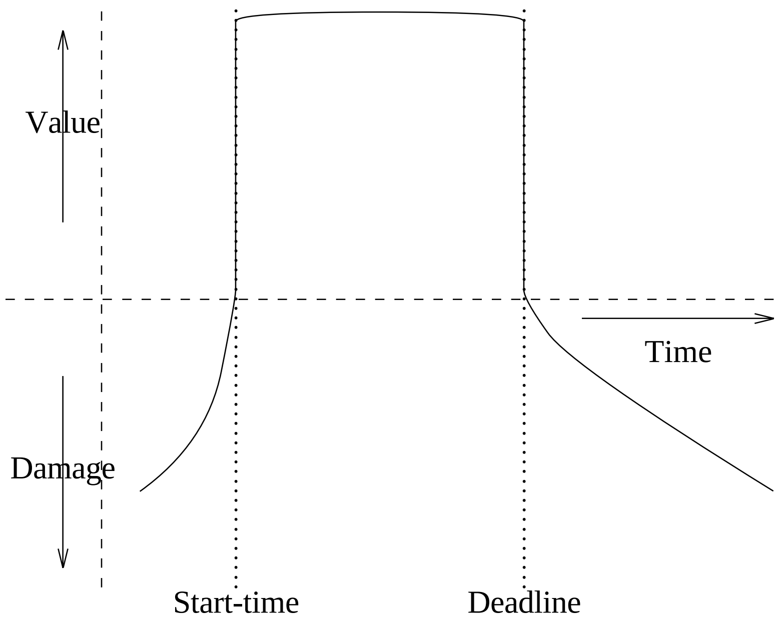

Figure 2: A Safety Critical System

including air-traffic control, process control, and "on-board" systems such as those proposed for the forthcoming space station[7]. The traditional approach to designing these systems has focussed on meeting the functional requirements; simulations and testing being employed to check the temporal needs. A system that fails to meet its hard deadlines 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.

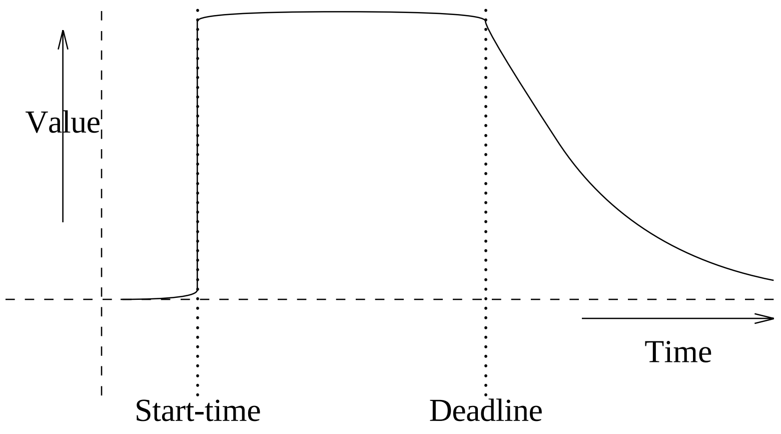

Figure 3: A Soft Deadline

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; the other focuses on the issue of scheduling[36]. The purpose of this paper is to review the current state of scheduling theory as it can be applied to dependable 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[74]. For a review of the use of semantic models to describe the properties of real-time systems see Joseph and Goswami[36].

The problem space for scheduling can be defined to stretches between the extremes:

- Reliable uniprocessor system with independent periodic processes.

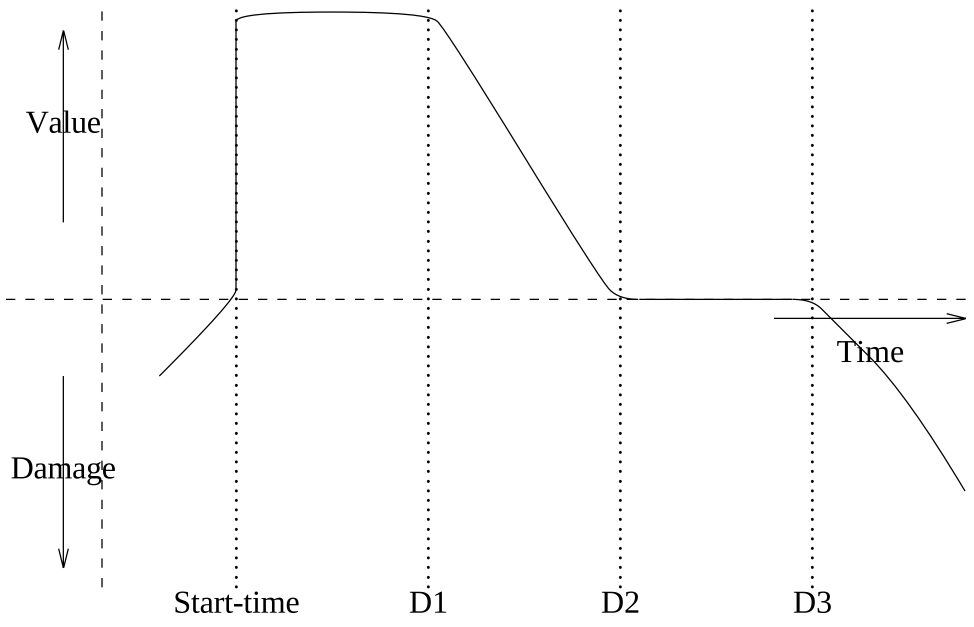

Figure 4: A Hybrid System

- Distributed system with dynamic interdependent periodic and aperiodic process experiencing transient overloads, processor failure and network partitioning.

The review presented here is structured as a progression from the possible to the desirable. It attempts to illustrate the levels of predicatability that can be obtained for different application characteristics. In the following section necessary definitions are given; and ways of characterising scheduling policies are introduced. Section 3 then considers a simple uniprocessor system with independent processes. This model forms the basis of other more realistic scenarios. The first of which removes the assumption that the processes are independent (Section 4). All scheduling models require calculation (or estimation) of the time needed to execute sections of sequential code. Section 5 deals with execution times. Multiprocessor and distributed systems are considered in Section 6. Section 7 briefly outlines transformation techniques. Finally in Section 8 a short review of existing real-time kernels is given.

2. DEFINITIONS, STRATEGIES AND CHARACTERISTICS

The concept of *deadline*, and *hard* and *soft* real-time systems, were introduced above. Throughout this section definitions are given of the terms commonly used in the scheduling literature.

A *system* consists of one or more *nodes*, each node being an autonomous computing system. A node contains processor(s) and processor support hardware. This is termed the *internal environment*. Also, each node has an interface to its *external environment*. The external environment of a node can contain other nodes (connected via an arbitrary network) and interfaces to external sensors, actuators or other controllable devices. The nodes in a system interact to achieve a common objective. For a real-time application, this objective is typically the control and monitoring of the system's external environment[8].

Processes or *tasks* (the terms are used interchangeably in this review) form the logical units of computation in a processor. A single *application program* will typically consist of many processes. Each process has a single thread of control. For multiple processes on a single processor, the execution of the processes is interleaved. With a system containing multiple processors (i.e. multi-processor or distributed), the processes may be interleaved over all the processors in the system.

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

(a) Periodic Processes

(b) Aperiodic Processes (also known as non-periodic)

Periodic processes, as their name implies, execute on a regular basis; they are characterised by:

(i) their period;

(ii) 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, 100ms of CPU time; this may rise to 300ms 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 have 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 for example a Poisson distribution. Such a distribution allows for "bursty" arrivals of external events but does not preclude any possible concentration of aperiodic activity. It is therefore not possible to do worst case 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 there is often defined a minimum period between any two aperiodic events (from the same source)[55]. 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.

The maximum response time of a process is termed the *deadline*. Between the invocation of a process and its deadline, the process requires a certain amount of *computation time* to complete its execution. Computation times may be static, or vary within a given maximum and minimum (which may be infinity and arbitrarily close to zero). The computation time (which excludes contention between processes) must not be greater than the time between the invocation and deadline of a process.

Hence, the following relationship should hold for all processes:

$$C \leq D$$

where

C - computation time

D - deadline

For each periodic processes, its period must be at least equal to its deadline. That is, one invocation of a process is complete before successive invocations. This is termed a *runnability constraint*[51]. Hence, for periodic processes, the following relationship holds:

† 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.

$$C \leq D \leq T$$

where

T - period of the process

Non-periodic processes can be invoked at any time.

A process can *block* itself, by requesting a resource that is unavailable. This results in the process being suspended. This occurs, for example, when a process performs a "wait" on a semaphore guarding a critical region currently occupied. A process can also *suspend* itself voluntarily. This is of use when a process performs a "delay" or other such operation.

Tasks whose progress is not dependent upon the progress of other processes are termed *independent*. This definition discounts the competition for processor time between processes as a dependency. *Interdependent* processes can interact in many ways including communication and precedence relationships. The latter is a partial (perhaps total) ordering on the execution sequences of processes[15].

2.2. Scheduler Characteristics

A *scheduler* provides an algorithm or *policy*[50] for ordering the execution of the outstanding processes on the processor according to some pre-defined criteria. In a conventional multitasking operating system, processes are interleaved with higher importance (or priority) processes receiving preference. Little or no account is taken of deadlines. This is clearly inadequate for real-time systems. These systems require scheduling policies that reflect the timeliness constraints of real-time processes.

Schedulers produce a *schedule* for a given set of processes. If a process set can be scheduled to meet given pre-conditions the process set is termed *feasible.* A typical pre-condition for hard real-time periodic processes is that they should always meet their deadlines. An *optimal scheduler* is able to produce a feasible schedule for all feasible process sets conforming to a given pre-condition. For a particular process set an *optimal schedule* is the best possible schedule according to some pre-defined criteria. Typically a scheduler is optimal if it can schedule all process sets that other schedules can.

2.2.1. Static and Dynamic Algorithms

Scheduling algorithms themselves can be characterised as being either static or dynamic[14]. A static approach calculates (or pre-determines) schedules for the system. It requires prior knowledge of a process's characteristics but requires little run-time overhead. By comparison a dynamic method determines schedules at run-time thereby furnishing a more flexible system that can deal with non-predicted events. It has higher run-time cost but can give greater processor utilization. Whether dynamic algorithms are appropriate for hard real-time systems is, however, a matter of some debate[9, 55]. 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. Dynamic approaches do, nevertheless, have an important role;

- 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 application's 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 prior to the running of the system. A table is generated that contains all the scheduling decisions for use during run-time. This relies completely upon *a priori* knowledge of process behaviour. Hence this scheme is workable only if all the processes are effectively periodic.

An *online* scheduler makes scheduling decisions during the run-time of the system. It can be either static or dynamic. The decisions are based on both process characteristics and the current state of the system. A scheduler is termed *clairvoyant*

> *"if it has an oracle which can predict with absolute certainty the future request times of all processes."*[55]

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

Schedulers may be *preemptive* or *non-preemptive*. The former can arbitrarily suspend a process's 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[78].

Hybrid systems are also possible[78]. 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. The code might, for example, 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 and can be accommodated in a further 10 cycles[8].

2.2.2. Computability and Decidability

The computational complexity of scheduling is of concern for hard real-time systems. Scheduling algorithms with exponential complexities are clearly undesirable for online scheduling schemes - their impact on the processor time available for application software is extreme. Aspects of scheduling can be computationally intractable - unsuitable for even offline scheduling. To deal with computational complexity two separate considerations are necessary: computability and decidability. For scheduling an arbitrary process set, the two terms can be described as follows:

- decidability - deciding whether a feasible schedule exists.
- computability - given a schedule, computing whether that schedule is feasible.

These concepts are well illustrated using the travelling salesman problem[84]. Whilst deciding if a solution to the problem exists (i.e. a route with cost $\leq k$) is not possible in polynomial time (and is in fact NP-complete); computing whether a particular route has a cost below a given value is trivial. NP-complete problems[28] are considered computationally intractable for arbitrary n, and so NP-complete scheduling algorithms are not desirable.

2.3. A System Model

As a basis for discussing scheduling a system model is required. A common model in the literature is given as follows[51, 16].

A process set **T** has elements $\{\tau_1 ... \tau_i ... \tau_n\}$ where *n* is the cardinality of **T**. Each τ_i has the following characteristics.

D_i - deadline

P_i - period

C_i - computation time

Simplistic constraints upon this process set are.

(i) $C_i \leq D_i = P_i$ (i.e. the processes have computation time less than their deadline, and the deadline is equal to their period).

(ii) Computation times for a given process are constant.

(iii) All processes are periodic.

(iv) No precedence relations exist between processes.

(v) No inter-process communication or synchronisation is permitted.

(vi) No process resource requirements are considered.

(vii) Context switchs have zero cost.

(viii) All processes are allocated to a single processor.

A realistic models requires these constraints to be weakened, for example the inclusion of:

- Both periodic and aperiodic processes.
- Arbitrary precedence relationships defined between processes in the process set **T**; i.e. if $\tau_i < \tau_j$ then the execution of process τ_i precedes that of τ_j.
- Protected resource sharing between processes.
- Computation times variable between a minimum (best-case) and a maximum (worst-case).
- Multi-node systems with static (or dynamic) process allocation.

3. UNIPROCESSOR SYSTEMS WITHOUT BLOCKING

In order to give a basis for analysing more realistic systems we consider first 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.

3.1. 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[51]. By "optimal" we mean that if a process can be scheduled by any fixed priority algorithm then it can also be scheduled by the rate monotonic scheduling algorithm[64].

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. The scheme is essentially offline. Performance is also acceptable.

The rate-monotonic algorithm has two parts: the first to decide whether a given process set is schedulable under the rate-monotonic scheme; the second to assign a static priority ordering to the processes.

The first part of the algorithm provides a schedulability constraint upon the process set. It is given as[51]

$$n \left(2^{\frac{1}{n}} - 1 \right) \geq U \qquad (1)$$

where

n - number of processes

U - total processor utilisation by the processes, such that

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

The utilisation converges on 69% for large n. Thus if a process set has utilisation of less than 69% it is guaranteed to be scheduled by the rate-monotonic algorithm. However, this schedulability constraint is a necessary, but not a sufficient condition. That is there are process sets that can be scheduled using a rate monotonic priority ordering, but which break the utilisation bound.

A necessary and sufficient schedulability constraint has been found[67, 42]. The process set is described in the model above; it consists of n processes:

$$\left\{ \tau_1, \tau_2, .., \tau_j, .., \tau_n \right\}$$

where the processes are ordered in decreasing priority. The period of process τ_j is T_j. The workload over [0, t] (for arbitrary t > 0) due to all processes of equal or higher priority than τ_j is given by

$$W_I(t) = \sum_{i=1}^{j} C_j \left\lceil \frac{t}{T_j} \right\rceil$$

Task τ_j is schedulable under all process phasings if

$$\min_{(0 < t \le T_j)} \frac{W_j(t)}{t} \le 1$$

The entire process set is schedulable if the above holds for all processes.

The equations take into account all possible process phasings. The effect is to increase the allowable bound in equation (1).

3.2. Dynamic Online Scheduling

The alternative to making all scheduling decisions offline is to make them as the system is running. Several dynamic online scheduling algorithms are now discussed.

3.2.1. Earliest Deadline

Earliest deadline scheduling[51] uses the simple system model given earlier. Like the rate-monotonic algorithm, a preemptive priority-based scheduling scheme is assumed. The algorithm used for scheduling is: *the process with the (current) closest deadline is assigned the highest priority in the system and therefore executes.* The schedulability constraint is given thus:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \le 1 \tag{2}$$

Hence, 100% processor utilisation is possible. This is a sufficient and necessary condition for schedulability. It has been shown[47] that for an arbitrary process set in which process timing constraint is relaxed to allow deadlines not equal to periods, condition (2) is necessary but not sufficient. Liu and Layland state that given the constraints in the simple model,

> *"the deadline driven scheduling algorithm is optimum in the sense that if a set of processes can be scheduled by any algorithm, it can be scheduled by the deadline driven algorithm."*[51]

Treatment of a non-preemptable process scheme is given by Jeffay[32] in which non-preemptable processes are shown to be scheduled by the earliest deadline heuristic if they can be scheduled by any other non-preemptive scheduler.

3.2.2. Least Laxity

The laxity of a process is defined as the deadline minus remaining computation time. With the least laxity approach[54], the schedulability constraint is again given by equation (2) above. The process which has the least laxity is assigned the highest priority in the system and is therefore executed. Whilst a process is executing it can be preempted by another whose laxity has decreased to below that of the running process. An executing process has constant laxity.

A problem arises with this scheme when two processes have similar laxities. One process will run for a short while and then get preempted by the other and visa versa. Thus, many context switches occur in the lifetime of the processes. This can result in "thrashing", a term used in operating systems[50] to indicate that the processor is spending more time performing context switches than useful work.

The least laxity heuristic is optimal in the same way as the earliest deadline heuristic when the cost of context switching is ignored[21].

3.3. 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 were to be 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 non-hard 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.

3.3.1. 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[65].

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' that 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 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:

(i) adding two delay requests into the body of the code, or

(ii) instructing the run-time 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 run-time support.

3.4. Independent Aperiodic Processes

Most real-time systems have a mixture of periodic and aperiodic processes. Mok[55] has shown that earliest deadline scheduling remains optimal with respect to such a mixture. He however assumes a minimum separation time between two consecutive arrivals of aperiodic processes (i.e. they are sporadic).

Where aperiodic events are not sporadic then one must use a dynamic approach. Again the earliest deadline formulation is optimal[20]. Recently Ramamritham and Stankovic[61] have described a guarantee scheme which also takes into account the run-time overhead.

The earliest deadline approach, although optimal, suffers from unpredictability (or instability) when experiencing transient overloads; i.e. deadlines are not missed in an order that corresponds (inversely) to the importance (value) 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[43].

The following describes three such algorithms.

Priority Exchange

A periodic process is declared as a server for non-periodic processes. When the server's period commences, the server runs if there is any outstanding non-periodic requests. If no requests exist, the priority exchange algorithm[43, 69] allows the high priority server to exchange its priority with a lower priority periodic process. In this way, the server's priority decreases but time reserved for non-periodic processes is maintained. Consequently the priority of non-periodic processes served by this mechanism decreases. This leads to deadlines being missed in an unpredictable manner under overload. The computation time allowance for the server is replenished at the start of its period.

Deferrable Server

A periodic process is declared to serve non-periodic processes. This is termed the deferrable server[43, 66, 70]. When the server is run with no outstanding non-periodic process requests, the server does not execute but defers its assigned computation time. The server's time is preserved at its initial priority (unlike the priority exchange algorithm). When a non-periodic request does occur, the server has maintained its priority and can thus run and serve the non-periodic processes. Hence, under overload, deadlines are missed predictably. The computation time allowance for the server is replenished at the start of its period.

Sporadic Server

The deferrable server algorithm maintains its computation time at a fixed level. The priority exchange algorithm allows for more non-periodic processes to be serviced. Both algorithms provide better response time than a conventional polling approach[43]. The sporadic server algorithm[70, 66] combines the advantages of the above two algorithms. This is achieved by varying the points at which the computation time of the server is replenished, rather than merely at the start of each server period. In this way, the sporadic server increases the amount of non-periodic processes that can be serviced. Also, the sporadic server services non-periodic processes without lowering its priority. The general structure behind this algorithm 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 whilst still guaranteeing periodic activities. A sporadic server can be declared for each critical non-periodic process in order to guarantee the process offline.

The three algorithms provide varying degrees of service for non-periodic processes. The sporadic server allows for critical non-periodic processes to be guaranteed. However, the associated cost for this guarantee is reduced processor utilisation when the non-periodic process is dormant. Deadlines can be missed predictably during overload due to the sporadic server not exchanging priorities. Hence the sporadic server solves some of the problems associated with scheduling non-periodic processes but suffers from the constraints of performing all schedulability checks offline.

4. UNIPROCESSOR SYSTEMS WITH BLOCKING

In more realistic real-time systems processes interact in order to satisfy system-wide requirements. The forms that these interactions take are varied, ranging from simple synchronisation to mutual exclusion protection of a non-sharable resource, and precedence relations. To help program these events, concurrent programming languages provide synchronisation primitives; for example semaphores, monitors[30] (with signals or condition variables), occam (CSP) type rendezvous[6] and Ada extended rendezvous[5].

To calculate the execution time for a process requires knowledge of how long it will be blocked on any synchronisation primitive it uses. Mok[55] has shown that the problem of deciding whether it is possible to schedule a set of periodic processes, that 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-complete[27, 46]. Similarly the analysis of a concurrent program that uses arbitrary inter-process 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-complete (i.e a program could fail a feasibility test but still be schedulable; however if it passes a feasibility test it *will* be schedulable).

4.1. 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 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. Firstly 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 is to either prohibit it altogether or to use priority inheritance.

4.2. Prevention of Priority Inversion

Priority inversion can be prohibited all together 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*[2]. 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 processes that uses S starts a new period.

The main problem with this approach is the enforced introduction 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 occurs in the following scenario:

- processes t_1 .. t_n in descending order of priority.
- each process consists solely of the use of critical section S.
- processes are released in the order t_n .. t_1 (i.e. in ascending priority order).
- t_n is released and requests S. The request is refused due to the next highest priority task t_{n-1} requiring S fractionally before t_n is projected to complete its use of S.
- this occurs successively to processes t_n to t_2. At this point, there has been an idle time approximately equal to the sum of computation times of t_n .. t_2.
- t_1 is released, requests and is granted S, and runs to completion.
- t_2 now runs to completion, followed by t_3 .. t_{n-1}, t_n.

Each process is suspended for a length of time equal to twice the (maximum) computation time of all higher priority processes. Thus the worst case utilisation is approximately 50%.

4.3. Priority Inheritance

The standard priority inheritance protocol[68] 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[8].

Sha *et al*[68] show that for this inheritance protocol there is a bound on 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*. That is, 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[10] for the languages CSP, occam and Ada.

4.4. Ceiling Protocol

Whilst the standard inheritance protocol gives an upper bound on the number of blocks a high priority process can encounter, this bound 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. (this is known as transitive blocking). Moreover there is nothing that precludes deadlocks in the protocol. All of these difficulties are addressed by the ceiling protocol[68]. When this protocol is used:

A. A high priority process can be blocked at most once during its execution (per activation).

B. Deadlocks are prevented.

C. Transitive 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 say, 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[68]:

- 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).

Whenever no system semaphores are locked then the locking of a first semaphore is always allowed. The effect of the protocol is that a second semaphore can not be locked if there could exist a higher priority process that may use both semaphores.

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 (A-C above) of the ceiling protocol has been given by Pilling, Burns and Raymond[57].

5. EXECUTION TIMES

To make effective scheduling decisions requires knowledge of the execution time of a process. Realistically, this time can vary with each execution of the process. Two alternatives clearly exist:

(i) to schedule using worst-case execution times.

(ii) to schedule using less than worse-case execution times.

To schedule according to worst-case times, when worst-case times are significantly greater than average-case times, results in a drop in processor utilisation[65]. For example, consider two processes submitted for a rate-monotonic scheduling scheme. Their combined worst-case utilisation times are 82.8%, exactly meeting the schedulability bound — equation (1). No other processes can be scheduled. If the processes have an average utilisation half of their worst-case, then the processor utilisation will be 41.4%, with no other processes able to be scheduled on that processor. This situation could occur if a process that samples an input has to raises an alarm if a high value is encountered, but in most cases does nothing.

Scheduling using execution times less than worst-case introduces the possibility of an overload occurring. If this condition arises, the requirement remains for a hard real-time system to meet the execution requirements of critical processes. An onus is placed upon the scheduler to achieve this. Two approaches have been proposed:

(i) ensure that processes that miss their deadlines are non-critical; i.e. the order in which processes miss their deadlines has to be predictable. Thus, non-critical processes can be forced to miss their deadlines[11].

(ii) in a multiprocessor or distributed system processes that are considered likely to miss their deadlines are migrated to other processors[75]. This approach is not possible in a statically scheduled system. Distributed scheduling is dealt with in a later sections.

5.1. Calculation of Execution Times

To calculate execution times requires an analysis of the actual program code. Leinbaugh[44] proposed a model of code to enable time analysis to be undertaken. Code is broken into segments, such that a segment consists of either a resource request, synchronisation primitive (or synchronous message) or a code block. Segments are now assigned worst case execution times. These worst case times are aligned end-to-end to calculate worst-case execution times for the process. Mok[55] proposes a similar model, but uses an arbitrary graph rather than a linear chain to represent the code.

When code performs an operation that may leave the process blocked, an upper bound on the blocking time is required for worst-case time calculation. Leinbaugh[44, 45] derives blocking time bounds from the necessity that all requests for devices etc. are serviced in a "*first-come-first-served*" manner. Thus a process can be delayed by one execution of every other process’s critical region for every resource it requires. Sha’s ceiling protocol (see above) provided an upper bound of a single block. In the general case, this time will be less than for Leinbaugh’s solution.

The work of Puschner and Koza[59] defines worst case times to programs by limiting the program constructs available to those that are analysable. The restrictions proposed include:

- no direct / indirect recursive calls.
- no function / procedure parameters.
- loops bounded either by a maximum number of iterations or a timeout.

Another assumption is that all resources are available when required: there is no blocking time.

All the approaches are valid, although it might be necessary to allow resource contention and therefore blocking times in a general hard real-time system. Hence, Puschner’s scheme could be augmented by use of the ceiling protocol; or Leinbaugh’s proposal could be utilised with a better bound on blocking time. The calculation of meaningful execution times remains one of the key issues in scheduling.

6. 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 Dertouzos[54] showed that the algorithms that are optimal for single processor systems are not optimal for increased numbers of processors. Consider, for example, 3 periodic processes P_1, P_2 and P_3 that must be executed on 2 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; let P_3 have requirements of 100 and 80. 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. This will leave P_3 with 80 units of execution to accomplish in the 75 units that are available. 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 65%. 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[28, 29, 44, 47]. It is therefore necessary to look for ways of simplifying the problem, and algorithms that give adequate feasible results.

6.1. Allocation of Periodic Processes

The above illustration showed that judicious allocation of processes can significantly effect schedulability. Consider another example; this time let 4 processes be executing on the 2 processors, and let their cycle times be 10, 10, 14 and 14. 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 schedulable even if execution times for the 4 processes are 5, 5, 10 and 4 (say). However if a 10 and a 14 were placed together on the same processor then utilisation drops to some 83% (i.e. execution times of 5, 5, 10 & 4 cannot be sustained).

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 run-time dispatcher it is better to keep processes on the same processor rather than try and utilise an idle processor (and risk unbalancing the allocation).

The pertinent considerations for (static) process allocation include:

(i) the cost of accessing the resources required by each process (remote or local invocations of code, servers etc.)[25],

(ii) the concurrency of the system,

(iii) fault-tolerant considerations[48], and

(iv) relative periods.

The cost of accessing resources or performing inter-process interactions is clearly more expensive for remote operations. The allocation of processes and resources to minimise cost of remote requests and interactions is clearly to place all resources and processes onto a single node. This reduces concurrency in the system. Thus, there exists a concurrency versus remote access cost trade-off.

Fault tolerant issues include the allocation of a process and its replicates. There is little point in allocating a process and some of its replicates onto the same node, if the replicates have been introduced into the system for hardware failures. Associated issues exist for other fault-tolerant methods such as dynamic reconfiguration.

Allocation can be viewed as the minimisation of the total cost of the system. Factors involved in cost calculation include [24]:

(i) cost of placing two processes on separate nodes (communication overheads);

(ii) cost of placing processes on the same node (reduction in concurrency).

Deriving the minimum cost has been shown to be an NP-complete problem[25, 24]. Indeed, only by restricting the search space can tractable solutions be found. Actual algorithms for suboptimal (but adequate) allocation of periodic processes can be found in the following references[19, 3, 22].

6.2. 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)[31]; this has been generalised more recently by Martel[52] to give one that is $O(m^2n^4 + n^5)$. However all these algorithms require the processes to be independent.

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 process 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*[61, 71] have proposed more flexible (dynamic) process 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 general protocol is used:

(a) Each aperiodic process arrives at some node in the network (this could be a processor in a multiprocessor system running its own scheduler).

(b) 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.

(c) 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 and by bidding for spare capacity in other nodes (see below).

(d) 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.

(e) In this way an aperiodic process is either scheduled (guaranteed) or it fails to meet its deadline†.

† It is possible for the protocol to cause a local periodic process to be displaced by a sporadic one if the latter is more important and cannot be guaranteed elsewhere.

The pertinent issues involved with process migration include:

(i) overheads of migrating code and/or environment data.

(ii) re-routing of messages involving a migrated process.

(iii) stability.

The overheads of migrating code can be avoided if copies of the code are distributed amongst a set of nodes. This set defines and limits the possible destinations of the process. Alternatively, code and data can be migrated. This is the approach adopted in DEMOS[58] and Sprite[23] whereby a process can be paused, migrated and continued on another node.

Re-routing of messages can be complex if a process is migrated several times. DEMOS places a forwarding address on each node that a process is migrated from. This can result in a long chain of forwarding addresses with the danger that a node holding a forwarding address may fail. In Sprite, each process has a home node through which all messages are routed thus avoiding the potentially long chain of forwarding addresses.

The concept of system stability is important for process migration[72]. Instability can be introduced in two ways:

- A process that is migrated successively from node to node without any significant progress being achieved.
- There is so much process movement in the system that the system performance degenerates.

The first affects only the process that is being successively migrated. The second relates to the whole system which spends a large proportion of its time migrating processes without really progressing. This is analogous to *thrashing* in operating systems[50]. Instability can be solved to a large degree by correct decisions as to where processes are migrated to.

Locating a node to migrate a process to is problematic. Two general approaches have been identified [13, 39]:

(i) receiver-initiated.

(ii) sender-initiated.

The first of these approaches involves a node asking for processes to be migrated to it. The second involves a node keeping a record of workloads of other nodes so that a process may be migrated to the most appropriate node. The sender-initiated method has been shown to be better under most system loads[13]. Stankovic *et al* have investigated both approaches[75, 86] and adopted a hybrid approach. Each node varies between the sender and receiver initiated according to the local workload.

The usefulness of Stankovic *et al's* 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-complete 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 that this movement is efficient. Some aperiodic processes may be tightly coupled to hardware unique to one node and will have at least some component that must execute locally.

6.3. Remote Blocking

If we now move to consider process interaction in a multiprocessor system then the complexity of scheduling is further increased[4, 27, 46, 81] (i.e. NP-hard). 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[87, 88, 89]. 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. Multiprocessor systems give rise to a new form of blocking— *remote blocking*. To minimise remote blocking the following two properties are desirable:

(a) Wherever possible a process should not use remote resources (and thereby be subject to remote blocking).

(b) Remote blocking should only be a function of remote critical sections, not process execution time due to remote preemption.

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

In a uniprocessor system it is correct for a high priority process to preempt a lower priority one. But in a multiprocessor 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 another 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. H is more important than either M or L, but is it more important than M *and* L?

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 processes in the system. Non-preemption is the protocol used by Mok[55]. As critical sections are often short when compared to non-critical sections of code this non-preemption of high priority critical section (which is another way of blocking a local higher priority process) is an acceptable rule.

Rajkumar *et al*[60] 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.

6.4. 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 following properties (during transient overload):

(a) Aperiodic deadlines are missed, rather than periodic ones.

(b) Aperiodic deadlines are not missed in an order that reflects importance.

Point (a) may, or may not, correspond to an applications requirement; point (b) is however always significant if the application has any hard deadlines attached to aperiodic activity.

A compromise situation is possible[11] (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 is 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 these processes that are destined to miss their deadlines. Note that this set could contain aperiodic and/or periodic processes. The movement of a periodic process 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. It must however be emphasised that migration is not a normal action; rather it is a form of error recovery after transient overload.

The advantage of this approach to transient overloads is that there is an increased chance that deadlines will be missed in the less important (soft) 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.

6.5. 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 has lead to a model of parallelism that is, essentially, loosely coupled. For example static allocation is a method usually considered more appropriate for distributed systems. Also the need to minimise remote action (remote blocking) is commensurate with good distributed design.

We have seen 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 (c.f. Spring Kernel[73, 76]). Note that copies of the hard deadlined processes may also be kept on each node to give increased availability or 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.

6.5.1. 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. One means of providing predicatable communication delays is to pre-allocate the use of the medium — this approach is taken in the MARS architecture (see below).

The ability of existing ISO/OSI standards to solve real-time communication problems must be seriously questioned[40]. Schedulability aspects of message communication has not been investigated extensively; recent publications by Kurose *et al*[17] and Strosnider *et al*[77] are, however, the exception.

6.5.2. Remote Procedure Calls

In all the analysis reported so far, it has been assumed that a program is partitioned between processors (or nodes) using the process as the unit of distribution. Access to remote resources being 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[56].

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 than 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.

6.6. Graph-Based Offline Scheduling Approachs

One obvious advantage of using offline methods of finding a schedule is that exhaustive search can be used[84]. This is computationally intractable for a large number of processes. Heuristic means can however be used to cut the search space to an acceptable size. Stankovic *et al*[88] for example begin with an empty schedule. Periodic processes are placed into the schedule until all processes meet their deadlines or not. In the latter case, backtracking is performed to consider other options. Heuristic functions are used in two places in the search:

(i) to limit the scope of backtracking - achieved by having a feasibility function which computes whether any feasible schedules can result from the current unfinished schedule.

(ii) to provide suggestions as to which process to insert into the schedule next. Options at this stage include the process with the least laxity or the earliest deadline.

Stankovic *et al* then take this static offline scheme and developed it to cope with resource usage and precedence constraints. Aperiodic processes are catered for by an online guarantee function (see above).

The graph based scheduling scheme proposed by Koza and Fohler[26] for the MARS system[37, 38, 18] is similar. It caters particularly for precedence constraints. However, all processes are periodic and so no facility is provided to cope with scheduling non-periodic processes online (although it is possible to switch to another static schedule if a specified external event takes place). All message passing operations are also scheduled offline. The motivation behind this is to ensure the system is predictable by completely scheduling every action in the system pre-runtime. MARS provides a verification tool for checking specified timing behaviours. This scheduling approach is further examined in Section 6.

Another scheduling scheme based upon graph search has been proposed by Xu *et al* [85]. This scheme finds schedules for task sets that allow arbitrary precedence between tasks. A branch and bound technique is used that defines the root of the search as a schedule generated by the earliest deadline algorithm, and then uses successive approximation to find an optimal schedule given the pre-defined precedence constraints. The authors acknowledge that in the general case this is an NP-complete problem, but assert that in the normal case a solution can be found within a reasonable time.

6.7. Process Abstraction

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

(a) using shared memory (shared primitives); or

(b) using a remote procedure that contains the necessary calls to the local primitives.

Typically (a) would be used in a multiprocessor system, and (b) 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 (c.f. Ada or occam). 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 inter-process 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 then remote preemption is minimised only if these processes are given higher priorities than other processes.

7. TRANSFORMATION TECHNIQUES

Scheduling is concerned with the allocation of scarce resources, the most important of which being the processors themselves. Whether any particular program with hard deadlines can indeed be scheduled depends upon

(a) the actual timing constraints,

(b) the available resources (and their characteristics), and

(c) the software architecture of the program.

We have noted already that an arbitrary architecture (with "unstructured" process interaction) presents a scheduling problem that is at least NP-complete. The software architecture must therefore be constrained. The review presented here (and elsewhere[9]) has shown that dependable hard real-time systems can be constructed if:

- aperiodic events are sporadic,
- the communication media performs in bounded time,
- worst case execution times are obtainable,
- resource usage is predictable,
- the architecture of the application leads to a decomposition into client and server processes,
- all timing constraints (requirements) can be represented as deadlines on either periodic or sporadic client processes,
- periodic client processes are statically pre-allocated,
- sporadic client processes are statically pre-allocated,
- server processes (and the resources they control) are also statically pre-allocated, and
- a preemptive scheduler with priority inheritance is used.

This list contains however a severe restriction on the software architecture and there is a clear need to define more flexible topologies that are nevertheless still tractable.

A distinctively different approach to this problem has been defined recently by Joseph *et al*[53]. They suggest that a real-time system should be designed in two stages. First the functional specifications are used to control decomposition. The resulting units can however assume *sufficient* resources. This uses the theoretically familiar maximum resources formulation (ie enough processors with adequate speeds, memory etc). The first phase of design need not concern itself with blocking etc but defines the maximum resources needed by the program. It is maximum in the sense that any more resources would not be utilised.

The distinctive second phase of their method involves transforming the initial design so that the resource requirement is reduced whilst still satisfying the specified time constraints (ie the program continues to be feasible). In the paper cited above they consider a number of commutative and associative transformations.

8. CURRENT KERNELS

A number of current real-time kernels are now reviewed with respect to their approaches to scheduling. Emphasis is placed on Arts, MARS and Spring.

The MARS (MAintainable Real-time System)[37] kernel is designed to support hard real-time systems for peak load conditions. All processes and inter-process communications are scheduled offline. This enables complete system predictability. The MARS view of real-time systems is that

> *"real-time control systems [are] very regular, i.e. the same sequence of actions is performed periodically."*[38]

A similar viewpoint is taken by the Arts kernel whose objective is to provide

> *"a predictable, analyzable, and reliable distributed computing system."*[79]

In a similar manner to the MARS kernel, Arts ensures the runnability of the system under all conditions pre-runtime. This is achieved by use of the rate-monotonic algorithm and the ceiling protocol.

The Spring kernel[73] is also designed for hard real-time systems. Hard real-time processes are allocated and guaranteed offline. Other processes are able to migrate around the system until they find a node with enough spare capacity to guarantee execution.

The majority of other kernels support only soft real-time in that they have little or no concept of process deadlines. However one such system of interest is Alpha[34]. The rationale behind this kernel is that real-time systems are inherently non-deterministic. This leads to the view that guaranteeing processes offline is not always possible in a system that provides degraded service under overload or failure. Hence, an effort is made by the kernel to run the process that is deemed the most important to the mission at a given time (i.e. has the greatest value — see Figures 1-4).

The Chaos[63] kernel is aimed at adaptable applications. For example, if an external event occurs with increasing frequency, the kernel adapts the period of the process dealing with that event to cope with the increased frequency.

In contrast, the Dragon/Slayer kernel[83] is aimed at vulnerable applications in which parts of the system are expected to fail frequently. The objective of this kernel is to ensure the proper replication of process and data to ensure a degraded service can continue after substantial failure.

8.1. Process Architectures

Most kernels offer an object based model for the applications. Typically, this involves active and passive objects corresponding to processes and the external code that the processes may execute. Processes may execute code in an object asynchronously or synchronously.

This model is seen in the Chaos system with one modification. Threads (processes) are resident inside an object and are not permitted to move outside of that object. The same restriction is seen in the Maruti kernel[49] where the processes and resources required for a specific part of the application are grouped together into a single object. The StarOS kernel[35] restricts processes to exist within a single object also. However, new instances of an object can be created dynamically together with the processes (threads) that they contain. These objects then broadcast their capabilities to all other objects.

Logically, the most flexible object system is the Alpha kernel. Active objects are permitted to move between passive objects, even if these objects are distributed. This entails an overhead for moving process state from one node to the next.

The strictly hard real-time kernels are less specific regarding the computational model they provide the application. Generally, their emphasis is placed upon the guarantees given to hard real-time processes rather than their object model. The MARS system imposes the strictest (realistic) process model upon the application. All processes must be periodic with any communication between processes taking place at exactly the same time in each execution of the processes involved. Processes with hard deadlines are guaranteed offline.

The Spring kernel allows processes to be periodic or aperiodic and have to have hard or soft deadlines. Periodic processes and those with a hard deadline are allocated pre-runtime to a node. All other processes are initially assigned a node, but are free to migrate during runtime. Precedence relationships may exist between processes, although it is not clear whether such relationships affect a process’s ability to migrate during runtime.

The Arts kernel divides processes into periodic and aperiodic. Critical timing characteristics can be associated with hard real-time processes in terms of a "time-fence" past which the process must not execute. Soft real-time processes have their importance represented by a "time-value" function. In this way, the system can decide which soft real-time processes should miss their deadlines during overload. Hard real-time aperiodic processes are executed in time reserved by a deferrable server (see section 2.4).

8.2. Scheduling

Kernels not aimed specifically at hard real-time applications tend to use a static priority preemptive scheduling scheme where the priorities are assigned by the application programmer. This is seen in the VRTX[62] and Fados kernels[80]. The disadvantage of this scheme is the emphasis on the designer to assign the appropriate priorities to the processes:

> *"By carefully setting the priorities of the tasks in the system, the designer can allocate the CPU time appropriately to meet any real-time deadlines."*[62]

The MARS kernel adopts a fixed schedule approach. In this the schedule is completely calculated offline and is given to the nodes in the system as part of system initialisation. All inter-process communications, resource requests, etc. are included in the schedule. The effect of this is that resources are always available when requested, and the communication medium is always available to send a message. Problems arise because all nodes must have schedules that are in lock-step with each other. This is achieved in the MARS system by having special purpose hardware on each node supporting a global synchronised time grid. Nodes may change schedules (again at a pre-defined time) simultaneously to another pre-calculated schedule.

The Arts kernel uses the rate-monotonic algorithm to analyse and guarantee hard real-time processes offline. When the system is run, other scheduling algorithms can be used in preference to the rate-monotonic algorithm (earliest deadline, least laxity, etc.). Non-periodic hard real-time processes are scheduled using execution time reserved by a deferrable server. All other processes are scheduled dynamically using a value-function scheme.

The Spring kernel has a similar approach to Arts. The schedulability of a set of critical processes is calculated offline. A graph-based exhaustive search method is used. Non-periodic processes are scheduled dynamically by the scheduler at a node attempting to guarantee the non-periodic process locally. If this is unsuccessful the process is passed to another node.

In contrast to the kernels that provide guarantees to hard real-time processes, the Alpha kernel provides a completely value-function oriented approach. The process with the highest value to the system is executed. This applies to both periodic and aperiodic processes.

8.3. Global Scheduling

Two of the kernels supply global scheduling facilities. The MARS system permits all the schedules in the system to change simultaneously. In the Spring kernel, not only can the schedules change, but also the scheduling policy. This is controlled by the meta-level scheduler which monitors the environment of the system and alters local scheduling policies to maximise system performance.

A second function of the global scheduling policy in the Spring kernel is in migrating processes that cannot be guaranteed on a local node. Once a process cannot be guaranteed locally, another node is found that might be able to meet the process’s deadline. The global scheduling policy is responsible for deciding which node, if any, the process is migrated to. This is achieved by a combination of receiver and sender initiated schemes.

None of the kernels reviewed provide a global scheduling policy that permits general migrating of processes under a load-balancing scheme. This would involve too high an overhead for practical hard real-time systems.

9. CONCLUSIONS

In this paper, the current state of scheduling for hard real-time systems has been reviewed. The major findings and associated conclusions are now given.

The complexity of general scheduling has been seen to be NP-complete. Hence, the emphasis in the literature has been to address the limited problems set by 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.

Due to these complexity considerations, sub-optimal scheduling schemes must be used in realistic hard real-time systems. Such schemes include the developments of the rate-monotonic algorithm and the scheduling approaches of the Spring and Arts kernels. 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 for response to changing environment conditions or local overload.

Guaranteeing of process deadlines introduces a predictability/flexibility tradeoff. In the MARS kernel, all processes are periodic and have guaranteed deadlines. However, the constraints imposed upon the process characteristics of the system reduce flexibility. The Alpha kernel takes a more relaxed viewpoint based upon 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 offline or statically, but with the kernel able to make dynamic value 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 schedulability constraints to be used that are based upon worst-case execution times and arrival rates. The calculation of these times relies upon 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, schedulability analysis and scheduling algorithms must be able to cope with processes that have generalised timing characteristics. For example, period 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 schedulability tests and scheduling algorithms for inter-dependent 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.

One can be confident that basic research in these areas will lead to more dependable real-time systems.

Acknowledgements

The authors would like to express thanks to the many constructive comments made on an earlier version of this text by Gerhard Fohler, Mike Richardson and Werner Schütz.

References

1. N. Audsley, *Survey: Scheduling Hard Real-Time Systems*, Department of Computer Science, University of York (1990).
2. O. Babaoglu, K. Marzullo and F.B. Schneider, ‘‘Priority Inversion and its Prevention in Real-Time Systems’’, PDCS Report No.17, Dipartimento di Matematica, Universita di Bologna (1990).
3. J A. Bannister and K.S. Trivedi, ‘‘Task Allocation in Fault-Tolerant Distributed Systems’’, *Acta Informatica* **20**, pp. 261-281 (1983).
4. S.H. Bokhari and H. Shahid, ‘‘A Shortest Tree Algorithm for Optimal Assignment Across Space and Time in a Distributed Processor System’’, *IEEE Transactions on Software Engineering* **SE-7**(6), pp. 583-589 (1981).
5. A. Burns, *Concurrent Programming in Ada*, Ada Companion Series, Cambridge University Press, Cambridge (1985).
6. A. Burns, *Programming in occam 2*, Addison Wesley, Wokingham (1988).
7. A. Burns and C.W. McKay, ‘‘A Portable Common Execution Environment for Ada’’, pp. 80-89 in *Ada: the Design Choice*, ed. A. Alvarez, Cambridge University Press, Madrid (1989).
8. A. Burns and A.J. Wellings, *Real-time Systems and their Programming Languages*, Addison Wesley, Wokingham (1990).
9. A. Burns, ‘‘Distributed Hard Real-Time Systems: What Restrictions are Necessary?’’, pp. 297-304 in *Proc. 1989 Real-Time Systems Symposium: Theory and Applications*, ed. H. Zedan, North Holland (1990).
10. A. Burns and A.J. Wellings, ‘‘The Notion of Priority in Real-time Programming Languages’’, *Computer Languages* **15**(3), pp. 153-162 (1990).
11. A. Burns, ‘‘Scheduling Hard Real-Time Systems: A Review’’, *Software Engineering Journal* **6**(3), pp. 116-128 (1991).
12. T. L. Casavant and J. G. Kuhl, ‘‘A Taxonomy of Scheduling in General Purpose Distributed Computing Systems’’, *IEEE Transactions on Software Engineering* **14**(2), pp. 141-154 (February 1988).
13. H. Y. Chang and M. Livny, ‘‘Distributed Scheduling Under Deadline Constraints: a Comparison of Sender-Initiated and Receiver-Initiated Approaches’’, pp. 175-180 in *Proceedings 7th IEEE Real-Time Systems Symposium* (December 1986).
14. S. Cheng, J.A. Stankovic and K. Ramamritham, ‘‘Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey’’, pp. 150-173 in *Hard Real-Time Systems: Tutorial*, ed. J.A. Stankovic and K. Ramamritham, IEEE (1988).
15. S.C. Cheng, J.A. Stankovic and K. Ramamritham, ‘‘Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems’’, pp. 166-174 in *Proceedings 7th IEEE Real-Time Systems Symposium* (December 1986).
16. H. Chetto and M. Chetto, ‘‘Some Results of the Earliest Deadline Scheduling Algorithm’’, *IEEE Transactions Software Engineering* **15**(10), pp. 1261-1269 (October 1989).
17. R. Chipalkatti, J.F. Kurose and D. Towsley, ‘‘Scheduling Policies for Real-Time and Non-Real-Time Traffic in a Statistical Multiplexer’’, pp. 774-783 in *Proceedings IEEE INFOCOM 89* (1989).
18. A. Damm, J. Reisinger, W. Schwabl and H. Kopetz, ‘‘The Real-Time Operating System of MARS’’, *ACM Operating Systems Review* **23**(3 (Special Issue)), pp. 141-157 (1989).

19. S. Davari and S.K. Dhall, ‘‘An On-line Algorithm for Real-Time Task Allocation’’, pp. 194-200 in *Proceedings 7th IEEE Real-Time Systems Symposium* (December 1986).

20. M. Dertouzos, ‘‘Control Robotics: The Procedural Control of Physical Processes’’, pp. 807-813 in *Artificial Intelligence and Control Applications, IFIP Congress*, Stockholm (1974).

21. M.L. Dertouzos and A.K.L. Mok, ‘‘Multiprocessor On-Line Scheduling of Hard Real-Time Tasks’’, *IEEE Transactions on Software Engineering* **15**(12), pp. 1497-1506 (December 1989).

22. S.K. Dhall and C.L. Liu, ‘‘On a Real-Time Scheduling Problem’’, *Operations Research* **26**(1), pp. 127-140 (1978).

23. F. Douglis and J. Ousterhout, ‘‘Process Migration in the Sprite Operating System’’, pp. 18-25 in *Proceedings Real-Time Systems Symposium* (December 1987).

24. R. D. Dutton and R. C. Brigham, ‘‘Note: The Complexity of a Multiprocessor Task Assignment Problem Without Deadlines’’, *Theoretical Computer Science* **17**, pp. 213-216 (1982).

25. D. Fernandez-Baca, ‘‘Allocating Modules to Processors in a Distributed System’’, *IEEE Transactions on Software Engineering* **15**(11), pp. 1427-1436 (Novenber 1989).

26. G. Fohler and C. Koza, ‘‘Heuristic Scheduling for Distributed Real-Time Systems’’, Research Report Nr. 6/89, Instiut fur Technische Informatik, Technische Universitat Wien, Austria (April 1989).

27. M.R. Garey and D.S. Johnson, ‘‘Complexity Results for Multiprocessor Scheduling under Resource Constraint’’, *SIAM J. Comput.* **4**, pp. 397-411 (1975).

28. M.R. Garey and D.S. Johnson, *Computers and Intractability*, Freeman, New York (1979).

29. R.L. Graham et al., ‘‘Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey’’, *Ann. Discrete Math* **5**, pp. 287-326 (1979).

30. C.A.R. Hoare, ‘‘Monitors: An Operating System Structuring Concept’’, *CACM* **17**(10), pp. 549-557 (1974).

31. W.A. Horn, ‘‘Some Simple Scheduling Algorithms’’, *Navel Research Logistics Quarterly* **21** (1974).

32. K. Jeffay, ‘‘On Optimal, Non-Preemptive Scheduling of Periodic Tasks’’, Technical Report 88-10-03, University of Washington, Department of Computer Science (October 1988).

33. E.D. Jenson, C.D. Locke and H. Tokuda, ‘‘A Time-Driven Scheduling Model for Real-Time Operating Systems’’, in *Proceedings 6th IEEE Real-Time Systems Symposium* (December 1985).

34. E.D. Jenson, J.D. Northcott, R.K. Clark, S.E. Shipman, F.D. Reynolds, D.P. Maynard and K.P. Loepere, ‘‘Alpha: An Operating System for Mission-Critical Integration and Operation of Large, Complex, Distributed Real-Time Systems’’, in *Proc. 1989 Workshop on Mission Critical Operating Systems* (September 1989).

35. A. K. Jones, R. J. Chansler, I. Durham, K. Schwans and S. R. Vegdahl, ‘‘StarOS, a Multiprocessor Operating System for the Support of Task Forces’’, pp. 117-127 in *Proceedings 7th ACM Symposium on Operating Systems Principles* (December 1979).

36. M. Joseph and A. Goswami, ‘‘Formal Description of Real-Time Systems: A Review’’, RR129, University of Warwick, Department of Computer Science (1988).

37. H. Kopetz and W. Merker, ‘‘The Architecture of MARS’’, *15th Fault-Tolerant Computing Symposium*, Ann Arbor, Michigan, pp. 274-279 (June 1985).

38. H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft and R. Zainlinger,

‘‘Distributed Fault-Tolerant Real-time Systems: The MARS Approach’’, *IEEE Micro*, pp. 25-39 (February 1989).

39. P. Krueger and M. Livny, ‘‘The Diverse Objectives of Distributed Scheduling Policies’’, pp. 242-249 in *Proceedings 8th IEEE Real-Time Systems Symposium* (December 1987).

40. G. Le Lann, ‘‘Critical Issues in Distributed Real-Time Computing’’, in *Proc. Workshop on Communication Networks and Distributed Operating Systems within the Space Environment*, ESA(ESTEC) (October 1989).

41. J.C. Laprie, ‘‘Dependability: A Unified Concept for Reliable Computing and Fault Tolerance’’, pp. 1-28 in *Resilient Computer Systems*, ed. T. Anderson, Collins and Wiley (1989).

42. J.P. Lehoczky, L. Sha and V. Ding, ‘‘The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior’’, Tech Report, Department of Statistics, Carnegie-Mellon (1987).

43. J.P. Lehoczky, L. Sha and J.K. Strosnider, ‘‘Enhancing Aperiodic Responsiveness in Hard Real-Time Environment’’, in *Proceedings 8th IEEE Real-Time Systems Symposium*, San Jose, California (December 1987).

44. D.W. Leinbaugh, ‘‘Guaranteed Response Times in a Hard Real-Time Environment’’, *IEEE Transactions on Software Engineering* **6**(1), pp. 85-91 (January 1980).

45. D. W. Leinbaugh and M. R. Yamini, ‘‘Guaranteed Response Times in a Distributed Hard Real-Time Environment’’, *IEEE Transactions on Software Engineering* **12**(12), pp. 1139-1144 (December 1986).

46. J.K. Lenstra, A.H.G. Rinnooy and P. Brucker, ‘‘Complexity of Machine Scheduling Problems’’, *Ann. Discrete Math.* **1** (1977).

47. J.Y.T. Leung and M.L. Merrill, ‘‘A Note on Preemptive Scheduling of Periodic Real-Time Tasks’’, *Information Processing Letters* **11**(3), pp. 115-118 (1980).

48. S.T. Levi, D. Mosse and A.K. Agrawala, ‘‘Allocation of Real-Time Computations Under Fault-Tolerance Constraints’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 161-170 (December 1988).

49. S.T. Levi, S.K. Tripathi, S.D. Carson and A.K. Agrawala, ‘‘The MARUTI Hard Real-Time Operating System’’, *ACM Operating Systems Review* **Special Issue**, pp. 90-105 (1989).

50. A. M. Lister, ‘‘Fundamentals of Operating Systems’’, 3rd. Edition, Macmillan Computer Science Series (1984).

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

52. C. Martel, ‘‘Preemptive Scheduling with Release Times, Deadlines, and Due Times’’, *ACM* **29**(3) (1982).

53. A. Moitra and M. Joseph, ‘‘Implementing Real-Time Systems by Transformation’’, pp. 143-158 in *Proc. 1989 Real-Time Systems Symposium: Theory and Applications*, ed. H. Zedan, North Holland (1990).

54. A.K. Mok and M.L. Dertouzos, ‘‘Multiprocessor Scheduling in a Hard Real-Time Environment’’, in *Proc. 7th Texas Conf. Comput. Syst.* (November 1978).

55. A.K. Mok, ‘‘Fundamental Design Problems of Distributed Systems for Hard Real Time Environments’’, PhD Thesis, Laboratory for Computer Science (MIT), MIT/LCS/TR-297. (1983).

56. J.D. Northcott, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*, Academic Press, Orlando (1987).

57. M. Pilling, A. Burns and K. Raymond, ‘‘Formal Specification and Proofs of Inheritance Protocols for Real-Time Scheduling’’, *Software Engineering Journal* **5**(5), pp. 263-279 (1990).

58. M. Powell and B. Miller, ‘‘Process Migration in DEMOS/MP’’, pp. 110-118 in *Proceedings of the Ninth ACM Symposium on Operating System Principles*, Bretton Wood, New Hampshire (October 1983).

59. P. Puschner and C. Koza, ‘‘Calculating The Maximum Execution Time Of Real-Time Programs’’, *The Journal of Real-Time Systems* **1**(2), pp. 159-176 (September 1989).

60. R. Rajkumar, L. Sha and J.P. Lehoczky, *Real-Time Synchronization Protocols for Multiprocessors*, Department of Computer Science, Carnegie-Mellon University (April 1988).

61. K. Ramamritham and J.A. Stankovic, ‘‘Dynamic Task Scheduling in Hard Real-Time Distributed Systems’’, *IEEE Software* **1**(3), pp. 65-75 (July 1984).

62. J. F. Ready, ‘‘VRTX: A Real-Time Operating System for Embedded Microprocessor Applications’’, *IEE Micro* **6**(4), pp. 8-17 (August 1986).

63. K. Schwan, P. Gopinath and W. Bo, ‘‘CHAOS - Kernel Support for Objects in the Real-Time Domain’’, *IEEE Transactions on Computers* **36**(8), pp. 904-916 (August 1987).

64. L. Sha and J.P. Lehoczky, ‘‘Performance of Real-Time Bus Scheduling Algorithms’’, *ACM Performance Evaluation Review,Special Issue* **14**(1) (May 1986).

65. L. Sha, J.P. Lehoczky and R. Rajkumar, ‘‘Task Scheduling in Distributed Real-time Systems’’, in *Proceedings of IEEE Industrial Electronics Conference* (1987).

66. L. Sha, J.B. Goodenough and T. Ralya, *An Analytical Approach to Real-Time Software Engineering*, Software Engineering Institute Draft Report (1988).

67. L. Sha and J.B. Goodenough, ‘‘A Review of Analytic Real-Time Scheduling Theory and its Application to Ada’’, pp. 137-148 in *Ada: the Design Choice*, ed. A. Alvarez, Cambridge University Press, Madrid (1989).

68. 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).

69. B. Sprunt, J. Lehoczky and L. Sha, ‘‘Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm’’, pp. 251-258 in *Proceedings 9th IEEE Real-Time Systems Symposium* (December 1988).

70. 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).

71. J.A. Stankovic, K. Ramamritham and S. Cheng, ‘‘Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems’’, *IEEE Transactions on Computers* **34**(12), pp. 1130-1143 (1985).

72. J.A. Stankovic, ‘‘Stability and Distributed Scheduling Algorithms’’, *IEEE Transactions on Software Engineering* **11**(10), pp. 1141-1152 (October 1985).

73. J.A. Stankovic and K. Ramamritham, ‘‘The Design of the Spring Kernel’’, pp. 146-157 in *Proceedings 8th IEEE Real-Time Systems Symposium*, San Jose, California (Dec 1987).

74. J.A. Stankovic, ‘‘Real-Time Computing Systems: The Next Generation’’, pp. 14-38 in *Tutorial Hard Real-Time Systems*, ed. J.A. Stankovic and K. Ramamritham, IEEE (1988).

75. J.A. Stankovic, K. Ramamritham and W. Zhao, ‘‘Distributed Scheduling of Tasks with Deadlines and Resource Requirements’’, *IEEE Transactions on Computers* **38**(8), pp. 1110-1123 (August 1989).

76. J.A. Stankovic and K. Ramamritham, ‘‘The Spring Kernel: A New Paradigm for Real-time Operating Systems’’, *ACM Operating System Review* **23**(3), pp. 54-71 (July 1989).

77. J.K. Strosnider and T.E. Marchok, ‘‘Responsive, Deterministic IEEE 802.5 Token Ring Scheduling’’, *The Journal of Real-Time Systems* **1**(2), pp. 133-158 (Sep 1989).

78. H. Tokuda, ‘‘Real-Time Critical Section: Not Preempt, Preempt or Restart?’’, CMU Technical Report, Computer Science Department, Carnegie-Mellon University (1989).

79. H. Tokuda and C.W. Mercer, ‘‘ARTS: A Distributed Real-Time Kernel’’, *ACM Operating Systems Review* **Special Issue**, pp. 29-53 (1989).

80. F. Tuynman and L. O. Hertzberger, ‘‘A Distributed Real-Time Operating System’’, *Software Practice and Experience* **16**(5), pp. 425-441 (May 1986).

81. J.D. Ullman, ‘‘Complexity of Sequence Problems’’, in *Computers and Job/Shop Scheduling Theory*, ed. E.G. Coffman, Wiley (1976).

82. A. Vrchoticky and W. Schütz (eds.), *Real-Time Systems (Specific Closed Workshop), ESPRIT PDCS Workshop Report W3*, Institut für Technische Informatik, Technische Universität Wien, Vienna (January 1990).

83. H. F. Wedde, G. S. Alijani, W. G. Brown, S. D. Chen, G. Kang and B. K. Kim, ‘‘Operating System Support for Adaptive Distributed Real-Time Systems in Dragon Slayer’’, *ACM Operating Systems Review* **Special Issue**, pp. 126-140 (1989).

84. H. S. Wilf, *Algorithms and Complexity*, Prentice-Hall International (1986).

85. J. Xu and D.L. Parnas, ‘‘Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations’’, *IEEE Transactions on Software Engineering* **16**(3), pp. 360-369 (March 1990).

86. W. Zhao and K. Ramamritham, ‘‘Distributed Scheduling Using Bidding and Focussed Addressing’’, pp. 103-111 in *Proceedings 6th IEEE Real-Time Systems Symposium* (December 1985).

87. W. Zhao, ‘‘A Heuristic Approach to Scheduling Hard Real-Time Tasks with Resource Requirements in Distributed Systems’’, PhD Thesis, Laboratory for Computer Science, MIT (1986).

88. W. Zhao, K. Ramamritham and J.A. Stankovic, ‘‘Preemptive Scheduling under Time and Resource Constraints’’, *IEEE Transactions on Computers* **38**(8), pp. 949-960 (August 1987).

89. W. Zhao, K. Ramamritham and J.A. Stankovic, ‘‘Scheduling Tasks with Resource Requirements in Hard Real-time Systems’’, *IEEE Transactions on Software Engineering* **SE-13**(5), pp. 564-577 (May 1987).