Chapter 10

Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach

Alan Burns

Scheduling theories for fixed-priority scheduling are now sufficiently mature that a genuine engineering approach to the construction of hard real-time systems is possible. In this chapter we review recent advances. A flexible computational model is adopted that can accommodate periodic and sporadic activities, different levels of criticality, process interaction and blocking, cooperative scheduling (deferred preemption), release jitter, precedence constrained processes, arbitrary deadlines, deadlines associated with specific events (rather than the end of a task's execution), and offsets. Scheduling tests for these different application characteristics are described. This model can be supported by structured, object-oriented, or formal development methods. The chapter also considers the issues involved in producing safe and predictable kernels to support this computational model.

10.1 Introduction

Recent developments in the analysis of fixed-priority preemptive scheduling have made significant enhancements to the models introduced by Lui and Layland in their seminal 1973 paper [33]. These developments, taken together, now represent a body of analysis that forms the basis for an engineering approach to the design, verification, and implementation of hard real-time systems. In this chapter we review much of this analysis in order to support the thesis that safety critical real-time systems can, and should, be built using these techniques.

Preemptive priority-based scheduling prescribes a run-time environment in which tasks, with a priority attribute, are dispatched in priority order. Priorities are, essentially, static. Processes are either runnable, in which case they are held on a notional (priority-ordered) run queue; delayed, in which case they are held

on a notional delay queue; or suspended, in which case they are awaiting an event which may be triggered externally (via an interrupt) or internally (from some other task).

Most existing hard real-time systems are implemented using a static table-driven schedule (often called a *cyclic executive*). Priority-based scheduling has many advantages over this static approach (see Locke [35] for a detailed discussion of this issue). In essence these advantages all relate to one theme—increased flexibility. However, in order to challenge the role of static scheduling as the premier implementation model, priority-based scheduling must:

- provide the same level of predictability (of temporal behavior)
- allow a wide range of application characteristics to be accommodated
- enable dependable (safe) implementations to be supported.

All of these issues are addressed in this review, which is organized as follows. The remainder of this introduction outlines a simple model of task attributes and shows how worst-case response times can be calculated. Section 10.2 considers the necessary run-time kernel and shows how its temporal characteristics can be accommodated and its implementation can be made safe. Section 10.3 extends the simple model to include a number of important application characteristics. One criticism that is often made about scheduling work is that it is not well integrated with other aspects of software production; in Section 10.4 we outline a computational model that is amenable to timing analysis and software production. A structured, an object-oriented, and a formal instantiation of this computational model are described. Finally, in Section 10.5 we address another criticism of priority scheduling: namely that it is too static. Methods of integrating soft and best-effort scheduling into the framework provided by the static priority-based model are considered. Section 10.6 presents our conclusions.

10.1.1 Calculating Response Times

We restrict our considerations to single-processor systems. The techniques are, however, applicable in a distributed environment with static allocation [52]. The processor must support a bounded, fixed number of tasks, N. The general approach is to assign (optimally) unique priorities to these tasks and then to calculate the worst-case response time, R, for each task. These values can then be compared, trivially, with each task's deadline, D. This approach is illustrated with the derivation of appropriate analysis for a simple computational model.

Each of the N tasks is assumed to consist of an infinite number of invocation requests, each separated by a minimum time T. For periodic tasks this value T defines its period, for sporadic tasks T is the minimum inter-arrival time for the event that releases the task. Each invocation of the task requires C computation time (worst case). During this time the task does not suspend itself. Tasks are independent of each other apart from their use of shared protected data. To bound priority inversion a ceiling priority protocol is assumed (for access to the protected

data) [10, 42]. This gives rise to a maximum blocking time of B (i.e., B_i is the maximum time task i can be blocked waiting for a lower-priority task to complete its use of protected data). Our simple model has the restriction that each task's deadline must be less than, or equal to, its inter-arrival time (i.e. $D_i \leq T_i$ for all i). We also assume that context switches, and so on, take no time (this optimistic assumption is removed in Section 10.2). Each task has a unique priority, P.

For this simple model optimal priority assignment is easily obtained. Leung and Whitehead [32] showed that deadline monotonic assignment is optimal, i.e. the shorter a task's deadline, the higher its priority ($D_i < D_j \implies P_i > P_j$).

The worst-case response time for the highest-priority task (assuming task 1 has the highest priority) is given by:

$$R_1 = C_1 + B_1.$$

For the other tasks it is necessary to calculate the worst- case interference suffered by the task. Interference results from higher-priority tasks executing while the task of interest is preempted. It can be shown, for this simple computational model, that maximum interference occurs when all higher-priority tasks are released at the same time as the task under consideration—this time is known as the *critical instant*. This leads to the following relation:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of tasks of higher priority then task i.

As R_i appears on both sides of this equation a simple solution is not possible [27]. Rather, an iterative (recurrent) process is derived. Let w be a window (time interval) into which we attempt to insert the computation time of the task. We expand w until all of C_i can be accommodated:

$$w_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j. \tag{10.1}$$

The iteration can start with $w_i^o = C_i$ (although more optimal start values can be found). It is trivial to show that $w_i^{n+1} \geq w_i^n$. If $w_i^n > D_i$, then task i cannot be guaranteed to meet its deadline. However, if $w_i^n = w_i^{n+1}$, then the interaction process has terminated and $R_i = w_i^n$.

The derivation of this result together with examples of its use can be found in a number of publications [3, 4, 7, 27]. Note that w_i is referred to by Lehoczky [28] as the $i - level$ busy period since the "priority" of the processor does not fall below that of task i during this period. The following simple example shows how response times are calculated using Equation (10.1). Consider the simple three-task set given in Table 10.1. Let the blocking time be 2 units of computation for Task_1 and Task_2.

Task_1 has the highest priority and has a worst-case response time of 4 (i.e., its own computation time plus the blocking time). Task_2 has an earliest possible

Table 10.1 Simple Task Set

	Period T	Computation Time, C	Priority P	Deadline D	Blocking Time, B
Task_1	8	2	1	6	2
Task_2	12	3	2	10	2
Task_3	20	7	3	20	0

response time of 3; putting this value into Equation (10.1) gives a right-hand-side value of 7, 7 then balances the equation:

$$7 \quad = \quad 3 \quad + \quad 2 \quad + \quad \left\lceil \frac{7}{8} \right\rceil 2.$$

For Task_3 the initial estimate is 7; the right-hand side of Equation (10.1) is thus (note the blocking factor is zero):

$$7 \quad + \quad \left\lceil \frac{7}{8} \right\rceil 2 \quad + \quad \left\lceil \frac{7}{12} \right\rceil 3.$$

This yields 12. Hence:

$$7 \quad + \quad \left\lceil \frac{12}{8} \right\rceil 2 \quad + \quad \left\lceil \frac{12}{12} \right\rceil 3.$$

This now yields a value of 14. A further iteration produces 17. Another iteration then gives a value of 19; this value is stable (i.e., actually causes Equation (10.1) to balance) and therefore the actual worst response time of Task_3 is 19. Hence all tasks will complete before their deadlines.

In Section 10.3 we show how this simple model can be extended. But first we must consider the implementation of preemptive priority-based scheduling.

10.2 Safe and Predictable Kernels

It is undoubtedly true that the support needed to implement preemptive priority-based dispatching is more complicated than static scheduling—although the difference is not as large as it would first appear. It should be noted that a full operating system is not required, only a micro-kernel with efficient context switching and an ample range of priorities [9].

The production of a correct kernel necessitates the development of a formal specification of the interface to the kernel and its behavior following calls to that interface. Formal notations such as Z [45] have been used to give precise definitions to such kernels [13, 46].

The notion of a safety kernel was introduced by Rushby [36] to imply a kernel that was not only built correctly but had a positive role in ensuring that various

negative behaviors (of the application) were inhibited. A prototype run-time support system for a restricted subset of Ada9X has been built along these lines [20]. It monitors all application tasks to make sure that they do not use more resources (in particular, CPU processing time) than was assigned to them during the scheduling analysis of the application. If a task attempts to run over its budget, it has an exception raised to enable it to "clear up" (the exception handler also has a budget defined).

In addition to engineering the kernel to an appropriate level of reliability, it is also critically important for the timing characteristics of the kernel to be obtainable. This is true both in terms of models of behavior and actual cost (i.e., how long each kernel routine takes to execute). The following sections address these issues.

10.2.1 Predicting Overheads

Simple scheduling models ignore kernel behavior. Context switch times and queue manipulations are, however, significant and cannot usually be assured to take negligible time unless a purpose-built processor is used. For example, the FASTCHART [49] processor can genuinely claim to have zero overheads.

Even if a dual processor is used to perform context switches (in parallel with the application/host processor), there will be some context switch overhead. When a software kernel is used, models of actual behavior are needed. Without these models excessively pessimistic overheads must be assumed. The interrupt handler for the clock will usually also manipulate the delay queue. For example (in one implementation [20]), when there are no tasks on the delay queue, then a cost of 16 μs may be experienced. If an application has 20 periodic tasks that all share a critical instant, then the cost of moving all 20 tasks from the delay queue to the run queue may take 590 μs—i.e., 37 times more.

Context switch times can be accounted for by adding their cost to the task that causes the context switch. For periodic tasks, the cost of placing itself on the delay queue (and switching back to the lower-priority task it preempted) is, however, not necessarily a constant. It may depend on the potential size of the delay queue (i.e., on the number of periodic tasks in the application).

To model adequately the delay queue manipulations that occur in the clock interrupt handler (i.e., at one of the top priority levels), it is necessary to address directly the overheads caused by each periodic task. It may be possible to model the clock interrupt handler using two parameters: C_{CLK} (the overheads occurring on each interrupt assuming that tasks are on the delay queue but that none are removed), and C_{PER} (the cost of moving one task from the delay queue to the run queue). Each periodic task now has a v*fictitious* task with the same period T but with computation time C_{PER}. Equation (10.1) thus becomes :

$$w_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j + \left\lceil \frac{w_i^n}{T_{CLK}} \right\rceil C_{CLK} + \sum_{f \in fpt} \left\lceil \frac{w_i^n}{T_f} \right\rceil C_{PER} \tag{10.2}$$

where fpt is the set of fictitious periodic tasks.

Our analysis of kernels indicates that this model is itself overly simplistic and hence too pessimistic. There is usually a cost saving when more than one task has been transferred between the queues. A three-parameter model would hence seem to be appropriate, (see Burns, Wellings and Hutcheon for a derivation of this model).

In addition to supporting periodic behavior, the kernel will also have to accommodate interrupt handling and the release of sporadic tasks following an interrupt. This again gives rise to parameters that must be established before full scheduling analysis can be undertaken [17].

10.2.2 Tick-Driven Scheduling

In all the above analysis, periodic tasks are assumed to have periods which are exact multiples of the clock period. They can thus be *released* (i.e., put on the run queue) as soon as they *arrive* (i.e., are potentially runnable). If the release time is not equal to the arrival time, then the task is said to suffer from *release jitter*. Although it would usually be a poor engineering decision to have release jitter, there are situations where it might be inevitable.

Sporadic tasks are also assumed to be released as soon as the event on which they are waiting has occurred. A full tick-driven scheduler will, however, poll for these events as part of the clock interrupt-handling routine. This has the advantage of clearly defining the times at which new tasks can become runnable. It also allows safety checks to be implemented that can ensure that sporadic tasks are not released too often. With this implementation scheme, sporadic tasks are bound to suffer release jitter.

Let J_i represent the worst-case release jitter suffered by task i (i.e., the maximum time between task arrival and release). Two modifications of Equation (10.1) are now required. First, the calculated response time according to Equation (10.1) is from release, not arrival. The true (desired) maximum response time is measured from arrival:

$$R_i^{TRUE} = R_i + J_i.$$

Second, the interference that this task has on lower-priority tasks is increased. This is because two releases of the task can be closer together than the notional minimum T_j. If one arrival suffers maximum release jitter, but the next does not, then the two releases have a time gap of only $T_j - J_j$. The interference factor in Equation (10.1) must be modified to give [3]:

$$w_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j. \tag{10.3}$$

10.2.3 Cooperative Scheduling

The kernels described above have all implemented true preemptive dispatching. In this section an alternative scheme is outlined (the use of deferred preemption). This has a number of advantages but can still be analyzed by the scheduling technique

embodied in Equation (10.1). In Equation (10.1) there is a blocking term B that accounts for the time a lower-priority task may be executing while a higher-priority task is runnable. In the application domain this may be caused by the existence of data that is shared (under mutual exclusion) by tasks of different priority. Blocking can, however, also be caused by the kernel. Many systems will have the non-preemptable context switch as the longest blocking time (e.g., the release of a higher priority task being delayed by the time it takes to context switch to a lower priority task-even though an immediate context switch to the higher-priority task will then ensue).

One of the advantages of using the immediate ceiling priority protocol [10] (to calculate and bound B) is that blocking is not cumulative. A task cannot be blocked both by an application task and a kernel routine-only one could actually be happening when the higher-priority task is released.

Cooperative scheduling exploits this non-cumulative property by increasing the situations in which blocking can occur. Let B^{MAX} be the maximum blocking time in the system (using a convention approach). The application code is then split into non-preemptive blocks, the execution times of which are bounded by B^{MAX}. At the end of each of these blocks the application code offers a "de-scheduling" request to the kernel. If a high-priority task is now runnable, then the kernel will instigate a context switch; if not, the currently running task will continue into the next non-preemptive block.

Although this method requires the careful placement of de-scheduling calls, these could be inserted automatically by the worst-case execution-time analyzer which is itself undertaking a control flow analysis of the code.

The normal execution of the application code is thus totally cooperative. A task will continue to execute until it offers to de-schedule. To give some level of protection over corrupted (or incorrect) software, a safe kernel could use an interrupt mechanism to abort the application task if any non-preemptive block lasts longer than B^{MAX}. The use of cooperative scheduling is illustrated by the DIA architecture [44]. Here a kernel support chip deals with all interrupts and manages the run queue. The de-scheduling call is a single instruction and has negligible cost if no context switch is due.

The use of deferred preemption has two important advantages. It increases the schedulability of the system, and it can lead to lower values of C. In Equation (10.1), as the value of w is being extended, new releases of higher-priority tasks are possible that will further increase the value of w. With deferred preemption no interference can occur during the last block of execution. Let F_i be the execution time of the final block, such that when the task has consumed $C_i - F_i$, the last block has (just) started. Equation (10.1) is now solved for $C_i - F_i$ rather than C_i:

$$w_i^{n+1} = B_i + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j . \tag{10.4}$$

When this converges (i.e., $w_i^{n+1} = w_i^n$) the response time is given by:

$$R_i = w_i^n + F_i .$$

Table 10.2 Kernel Attributes

Notation	Description
C_{sw}^{P}	Cost of context switch away from a periodic task— may be a function of maximum size of delay queue
C_{sw}^{R}	Cost of context switch to a task currently on the run queue
C_{CLK}	Clock interrupt handler cost (no tasks being moved)
T_{CLK}	Clock interrupt handler period
C_{PER}	Cost of moving one task from delay queue to run queue
C_{sw}^{S}	Cost of context switch away from a sporadic task— when it suspends waiting for its next release
C_{SP}	Cost of releasing a sporadic (i.e., putting it on the run queue)
C_{INT}	Cost of an interrupt handler that just releases a sporadic task
B_K	Maximum length of non-preemption in the kernel

In effect the last block of the task has executed with a higher priority (the highest) than the rest of the task. Lehoczky has shown how increases in priority during the execution of a task can lead to better schedulability [24].

The other advantage of deferred preemption comes from predicting more accurately the execution times of a task's non-preemptable basic blocks. Modern processors have caches, prefetch queues, and pipelines that all significantly reduce the execution times of straight-line code. Typically, estimations of worst-case execution time are forced to ignore these advantages and obtain very pessimistic results because preemption will invalidate caches and pipelines. Knowledge of non-preemption can, however, be used to predict the speedup that will occur in practice. Zhang, Burns, and Nicholson have shown how a 20% reduction in worst-case execution time (C) can be obtained by modeling the prefetch queue directly [54]; Harmon, Baker, and Whalley have shown how the pipeline on a 68020 can be analyzed at the micro-code level [25]; and cache advantages can also be predicted. If modern processors are to be used in real-time systems, then this type of analysis is necessary.

Summary. A number of the parameters defined in the above discussion (for example, J and F) are, in reality, attributes of the application's task set. Others relate to the kernel itself. Table 10.2 summarizes those that have been introduced in this discussion. Their values are of key significance in determining the feasibility of any application running on top of the kernel. It follows that any kernel used for safety critical real-time systems will not only have to be instrumented but must also allow these parameters to be verified.

If interrupt handlers differ in terms of their cost, then a single C_{INT} value will not suffice.

10.3 An Extendible Model

Application requirements rarely (if ever) fit the simple model described in the introduction. An appropriate scheduling theory is one that can be extended to meet the particular needs of newer application requirements. In this section we consider a number of extensions to the basic model:

1. Variations in C and T
2. Precedence Relations and Multi-Deadline Tasks
3. Arbitrary Deadlines (i.e., $D > T$)
4. Internal Deadlines (i.e., not all C has to be completed by D)
5. Offsets and Phased Executions

We then consider how priorities can be assigned optimally when the simple rate monotonic or deadline monotonic policies do not apply.

10.3.1 Variation in Computation Time and Period

Where it can be shown that a task will not execute for its maximum time on each release, it is pessimistic to assume that it does. For example, a periodic task may do a small amount of data collection in each period but every, say, 10 periods analyses this data using a much more expensive algorithm. This behavior can simply be modeled, in Equation (10.1), as two tasks, one running every period (with a small C) and the other running every $10T$ (with a larger computation time).

Variations in period are also possible. Bursts of activity involving a number of short periods are following by inactivity. Sporadic tasks released by interrupts can behave in this manner. For example, a sporadic could have a worst-case (minimum) arrival interval of 1 ms but have the restriction that no more than 5 releases can occur within a 100 ms interval. If the worst-case arrival interval is very small, then it is acceptable to collapse the 5 releases into a single task (with period of 100 ms). However, a more accurate prediction of the interference this task will impose on lower-priority tasks, in the window w, can be derived [3]. Let T be the outer period (e.g., 100 ms in the above example) and t be the smaller period (e.g., 1 ms). Also let n be the number of releases in the outer period (e.g., 5). Task j will have an interference on lower-priority tasks (I_i^j) as follows:

$$I_i^j = \left\lfloor \frac{w_i^n}{T_j} \right\rfloor n_j C_j + min \left\{ \left\lceil \frac{w_i^n - \left\lfloor \frac{w_i^n}{T_j} \right\rfloor T_j}{t_j} \right\rceil , n_j \right\} C_j . \qquad (10.5)$$

This can then be incorporated into Equation (10.1). The first term in Equation (10.5) gives the cost of complete cycles (outer period) contained within w_i^n. The second term gives the additional cost of minor cycles, this is upper bounded by the cost of a complete burst, $n_j C_j$.

Table 10.3 An Example Task Set

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

Table 10.4 Transformed Task Set

	C	D	T	P
L	2	5	20	1
Q^T	2	9	20	2
S^T	4	16	20	3

10.3.2 Precedence Relationships and Multi-deadline Tasks

A common paradigm for structuring real-time software is as a set of tasks linked via precedence relations (i.e., task B cannot start until task A has completed). Data is often passed along these precedence links, but as the tasks involved never execute together, mutual exclusion over this data need not be enforced.

For illustration, consider a simple straight-line "transaction" involving three tasks: L, which must run before Q, which runs before S. Table 10.3 contains the given timing attributes for these tasks. Note that the periods of the three tasks are identical and that the overall deadline is 16.

A naive application of, say, deadline monotonic analysis will assign priorities (P) as given in the table. The schedulability test will then assume that all tasks are released at the same time and deem the task set to be unschedulable.

The critical instant assumption (i.e., all tasks released simultaneously) is clearly too pessimistic for precedence-constrained tasks. We know that they never wish to execute together. Both Q and S require an *offset*. That, is they cannot execute at the start of the period.

A simple transformation can be applied to tasks with offsets that share the same period. We relate the deadlines of all tasks not to their start times but to the start time of the transaction. This will not affect L but it will mean that Q and S have their deadlines stretched (we refer to the new tasks as Q^T and S^T). Table 10.4 now has the new deadlines and priorities for the task set.

The priority model will now ensure that L executed first, then Q^T, and then S^T. Moreover, the new task set is schedulable and would actually allow other tasks to be given priorities interleaved with this transaction. As the tasks share the same period, only one of them will experience a block. Note, however, that task L (and Q^T) must not undertake any external blocking, as this would free the processor to execute Q^T (or S^T) early.

This formulation results in tasks having lower priorities for later positions down the precedence relationship (i.e., S lower than L). As indicated earlier, Har-

bour, Klein, and Lehoczky have shown that by increasing the priority (and imposing some mechanism to stop the later tasks starting too early) can result in greater schedulability [24].

Finally, it should be noted that precedence relations can be implemented with real offsets (i.e., Q not being released until time 5). This technique is considered in Section 10.3.5.

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

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

10.3.3 Arbitrary Deadlines

To cater for situations where D_i (and hence potentially R_i) can be greater than T_i, we must adapt the analysis. The following outlines the approach of Tindell [50, 53]. When deadline is less than (or equal) to period, it is only necessary to consider a single release of each task. The critical instant, when all higher-priority tasks are released at the same time, represents the maximum interference, and hence the response time following a release at the critical instant must be the worst-case. However, when deadline is greater than period, a number of releases must be considered. We assume that the release of a task will be delayed until any previous releases of the same task have completed. For each potentially overlapping release we define a separate window $w(q)$, where q is just an integer identifying a particular window (i.e., $q = 0, 1, 2, ...$). Equation (10.1) can be extended to have the following form:

$$w_i^{n+1}(q) \quad = \quad (q+1)C_i \; + \; B_i \; + \; \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j. \tag{10.6}$$

For example with q equal to 2, three releases of task i will occur in the window. For each value of q, a stable value of $w(q)$ can be found by iteration—as in Equation (10.1). The response time is then given as

$$R_i(q) \quad = \quad w_i^n(q) \; - \; qT_i \tag{10.7}$$

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

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

$$R_i(q) \quad \leq \quad T_i. \tag{10.8}$$

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

$$R_i \quad = \quad \max_{q=0,1,2,\ldots} R_i(q). \tag{10.9}$$

Note that for $D \leq T$ relation (10.8) is true for $q = 0$ (if the task can be guaranteed), in which case Equations (10.6) and (10.7) simplify back to the original equation.

10.3.4 Internal Deadlines

In a recent report, Gerber [23] argues that it is only meaningful to attach a deadline to the last observable event of a task. Moreover, this last observable event may not be at the end of the task's execution; i.e., there may be a number of internal actions after the last output event.

When the model for analysis is enhanced to include kernel overheads (as described in Section 10.2), it is necessary to "charge" to each task the cost of the context switch that allows it to preempt a lower-priority task plus the cost of the context switch back to the preempted task once the higher-priority task has completed. For realistic context switch times (i.e., not zero) it is meaningless to attach the "deadline" to the end of the context switch. Figure 10.1 gives a block representation of a task's execution (excluding preemptions for higher-priority tasks). Phase `a` is the initial context switch to the task, phase `b` is the task's actual execution time up to the last observable event, phase `c` represents the internal actions of the task following the last observable event, and phase `d` is the cost of the context switch away from the task. The real deadline of the task is at the end of phase `b`.

In the following we shall denote by C^D the computation time required by the real internal deadline (i.e., phases `a` + `b` only), and by C^T the total computation time of the task in each period (i.e., all four phases). Note that there is no requirement to complete C^T by T as long as C^D is completed by D. Hence an adaptation of the arbitrary deadline model (see the previous section) is required.

If we include the two phases of computation into Equation (10.5), we get:

$$w_i^{n+1}(q) \quad = \quad qC_i^T \; + \; C_i^D \; + \; B_i \; + \; \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j^T. \tag{10.10}$$

This when combined with (10.7), (10.8), and (10.9) allows the worst-case response time for C_i^D to be calculated (assuming maximum C_i^T, interference from early releases of itself). Equation (10.6) could be used directly to calculate the response time for C_i^T, but this value is not directly relevant to this formulation. It can be

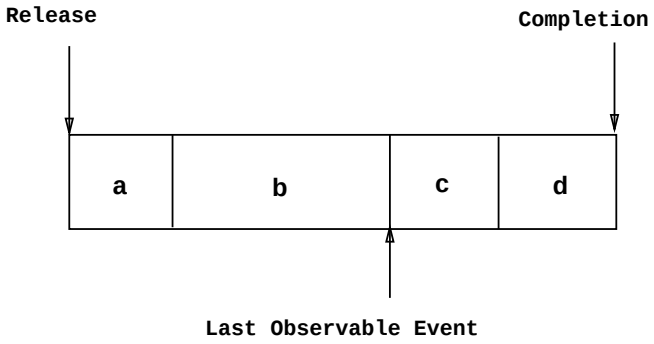

Figure 10.1 Four Phases of a Task's Execution

Table 10.5 Gerber's Task Set

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

shown, trivially, that for utilization less than 100% there exists bounded response times for all tasks.[1] What is important is that R_i^D is less than D_i.

The above analysis can be applied to the simple task set introduced and discussed by Gerber [23]. Table 10.5 shows the characterization of three tasks; note that $D = T$ for all entries, and that no task experiences blocking. With rate monotonic analysis, task 3 cannot be guaranteed. Gerber shows that by transforming this task, of the 653 units of computation only 493 are required to be completed by the deadline. He then shows how an implementation scheme can be used to guarantee task 3. However, the above analysis furnishes a value for R_3^D of 2493, which is just before the deadline (and period). Hence standard preemptive priority-based dispatching will satisfy the timing requirements of this task set. No transformation is needed. Note, for completeness, that the worst-case response time of R_3^T is 2653.

10.3.5 Offsets and Phased Executions

Perhaps the most extreme restriction of the basic model is that it assumes that all tasks could be released at the same time (the critical instant). This assumption simplifies the analysis but it is not applicable on many occasions. Cyclic execu-

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

Table 10.6 Three Offset Tasks

Task Name	Period	Computation Time	Offset	Priority	Deadline
Command actuators	200	2.13	50	20	14
Request DSS data	200	1.43	150	19	17
Request wheel speeds	200	1.43	0	18	22

Table 10.7 Combined Task

Task Name	Period	Computation Time	Offset	Priority	Deadline
Combined task	50	2.13	0	18	14

tives (static scheduling), for example, explicitly use offsets to order executions and obtain feasible schedules. Without offsets, priority-based systems are often too pessimistic; with offsets, equivalent behavior to cyclic executives can be obtained [8]. For example, a recent case study [18, 19] of the Olympus satellite AOCS (Attitude and Orbital Control System), containing some 30 tasks, was deemed unschedulable by the standard deadline monotonic test (i.e., Equation (10.1) modified to include kernel overheads). On inspection it contained three tasks of identical period that could not all be scheduled. Table 10.6 gives the details of these tasks.

The only requirements on these tasks were their periods (and deadlines); they did not have to be released together. By giving "Command Actuators" an offset of 50 ms and "Request DSS Data" an offset of 150 ms, their work was spread out. From an analysis point of view it was possible to replace these three tasks by just one (see Table 10.7). This task has a computation time requirement equal to the greatest of the original three, and a deadline which is the shortest. The task set (including this new one but not the originals) now passed the schedulability test.

Hence a simple transformation, that actually increases the overall load on the system (as it notionally executes every 50 ms) can increase schedulability by incorporating offsets.

In the more general case of arbitrary offset relationships it would be desirable to have an exact feasibility test. One way of testing feasibility is just to simulate the behavior of the system. Leung shows that the length of interval that should be examined is twice the LCM of the task set plus the largest offset (assuming tasks have been normalized to have offsets less than period) [31]. For task sets with periods that are relative primes this implies a computationally infeasible test.

Recently, Tindell [8, 51] has developed a feasible but inexact test, using the window approach outlined earlier for arbitrary deadlines. The resulting maximum window size is only marginally greater than the one that would be obtained for the full necessary and sufficient analysis. The derivation of this result is, however, beyond the scope of this review.

10.3.6 Priority Assignment

The formulations given in the last three sections (i.e., arbitrary deadlines, internal deadlines, and offsets) have the common property that no simple algorithms (such as rate or deadline monotonic) gives the optimal priority ordering. In this section we reproduce Audsley's algorithm for assigning priorities in these situations. Audsley [1] proves the following theorem:

Theorem 10.3.1 *If task τ is assigned the lowest priority and is feasible, then if a feasible priority ordering exists for the complete task set, an ordering exists with τ assigned the lowest priority.*

If a τ is found then a corollary of the theorem can be applied to the lowest-but-one priority, and so on; hence a complete priority ordering is obtained (if one exists).

The following code in Ada implements the priority assignment algorithm; `set` is an array of tasks that is notionally ordered by priority, `set(1)` being the highest priority, `set(N)` being the lowest. The procedure `task_test` tests to see whether task `K` is feasible at that place in the array. The double loop works by first swapping tasks into the lowest position until a feasible result is found; this task is then fixed at that position. The next priority position is then considered. If at any time the inner loop fails to find a feasible task, the whole procedure is abandoned. Note that a concise algorithm is possible if an extra swap is undertaken.

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

If the test of feasibility is exact (necessary and sufficient), then the priority ordering is optimal. Thus for arbitrary deadlines and internal deadlines (without blocking), an optimal ordering is found. Where a non-exact test is used (for example, with the offset test), the priority ordering reflects the quality of the test.

10.3.7 Summary

This section has reviewed a number of recent results that have taken simple schedulability equations and extended them to cover a range of realistic and necessary application features. Many variations of the basic equations have been given, but

Table 10.8 Task Attributes

Notation	Description	Default
T	Minimum time between task releases (or burst releases)	
D	Deadline relative to start of any precedence relation	T
O	Release offset relative to start of a precedence relationship	0
J	Release jitter	0
n, T, t	Characteristics of a bursty task, n in time T with a minimum gap between inner cycles of t	$1, T, 0$
B	Blocking time	B_K
C or C^T	Computation time	
C^D	Computation time before last observable event	C
F	Final non-preemptive section of computation time	0
P	Priority (calculated)	
R	Response time (calculated)	

they can all be integrated together and implemented within some appropriate software tool. Table 10.8 gives the attributes that are needed for each task if the full analysis described in this review is to be applied; those attributes with a default value can be omitted (i.e., the default can be assumed).

The overheads due to implementing fixed-priority scheduling do reduce processor utilization, but the use of internal deadlines and offsets can move utilization close to 100%. A final technique is worth noting for some tasks sets that still cannot be scheduled by the fixed-priority approach. Even when 100% utilization is needed it is not necessary to move to a fully earliest deadline approach [21, 33]. It has been shown that a dual-priority scheme is adequate [15]. Here some low-priority tasks are given an intermediate deadline at which their priority is raised (if they still have work to do). This minimally dynamic scheme provides for optimal schedulability.

10.4 Computational Model

Scheduling work is often criticized for not addressing the broader problems of engineering real-time systems. It is clear that attempting to apply scheduling analysis to arbitrary software is doomed to failure. The interface between software development and scheduling is the computational model. This model must be amenable to analysis but also be a natural end product of the development process. Moreover, the computational model must be applicable to implementations on multi-processors and distributed systems.

The computational model implicit in the scheduling analysis reviewed in this chapter has the following properties:

- It consists of active entities (tasks) and protected shared data areas.
- The only communication between active entities is via the shared data areas; the only exception to this is when one active entity releases another for

execution.

- Precedence relationships between active entities are allowed.
- Active entities have temporal attributes defined (such as deadline, offset, period, minimal arrival rate, etc.).
- Shared data areas provide mutual exclusion but do not arbitrarily block clients.
- Entities are allocated to single processing units.
- The allowable remote actions (in a distributed system) are a remote write to a shared data area and the releasing of a remote active entity.

This simple model has sufficiently expressive power to allow systems to be designed, allocated to distributed hardware, and analyzed for realistic worst-case behavior. To support the view that it is sufficient for design work, three development methods will be reviewed briefly.

(a) A traditional approach — MASCOT

(b) A formal method — TAM

(c) An object-oriented approach — HRT-HOOD

It should be clear how the computational model leads to programs/systems that can be analyzed. Note that the desire to reduce blocking will dictate the use of simple shared data areas. The granularity of the active entities is also significant. All three design methods encourage the use of decomposition rules that lead to activities (modules) that are temporarily, as well as functionally, decoupled.

10.4.1 MASCOT

The MASCOT [11] method involves the production of a *real-time network*. Within this network there are activities and IDAs (intercommunication data areas). For hard real-time systems two forms of IDA are used : *pools* and *signals*. Pools provide non-destructive non-blocking read and destructive non-blocking write; signals have the same write characteristics, but the read is destructive and blocking. Hence pools are used for simple mutual exclusion, while signals are employed to release an activity that is waiting for data.

Recently, MASCOT has been extended to give full life-cycle support to the production of real-time systems. An interesting feature of this DORIS (Data-Oriented Requirements Implementation Scheme) technology is the use of algorithms that provide non-blocking mutual exclusion. If pools are single-writer, then pool I/O operations never block (for example, a read event will always return the most recent completely written data-even if a write-to operation is concurrently updating it). DORIS also advocates the use of the deferred preemption method described in Section 10.2.2. However, the main use of MASCOT is as a design method. It is used primarily in the safety-critical aerospace industry.

10.4.2 TAM

Many formal development methods have a very synchronous computational model that leads to difficult timing analysis. They often had to incorporate the extreme assumption of maximum parallelism and zero cost for many activities. By comparison, TAM (Temporal Agent Model) [39–41] is defined to support the development (via refinement) of systems that can be analyzed accurately.

TAM is a wide-spectrum language consisting of specification statements and concrete executable statements. As a system is being developed, specifications are refined into more concrete forms (a refinement calculus is defined for TAM). An executable program (i.e., one with no remaining specifications) consists of *agents* and *shunts*. Shunts are single-writer multiple-reader shared data areas. Agents can communicate only via shunts. All computations and communications take time, and data passing through a shunt is time stamped. Agents can also be released by the event of writing to a shunt.

First-order predicate logic has been extended (conservatively) to give the formal basis to TAM. A simple form of temporal logic (and the introduction of *timed variables*) is used to define period activities, deadlines and so on.

A number of case studies [2, 37, 38] have been written that indicate that real-time systems can be specified, refined, and analyzed using the TAM formulation.

10.4.3 HRT-HOOD

HRT-HOOD [14, 16] (Hard Real-Time HOOD) is an adaptation of HOOD (Hierarchical Object-Oriented Design). A system is decomposed into terminal objects that must be either cyclic, sporadic, or protected. Cyclic and sporadic objects contain a single thread of control. Protected objects are required to provide mutual exclusion (e.g., by ceiling priorities). Sporadic objects also have a single method used to release them for execution. Rules of decomposition and usage force the terminal system to match the computational model described earlier. Object attributes are used to hold the timing characteristics and derived properties such as priority and response time. HRT-HOOD is a structured method supporting a graphical representation and a textual equivalent syntactical form. It has been used, together with some of the scheduling analysis discussed in this chapter, on an extensive case study [18].

One of the interesting features of the HRT-HOOD method is that it contains systematic mapping from the object system to Ada 9X. This indicates that the computational model is realizable in that language.

10.5 Slack Scheduling

It is possible to compare scheduling approaches by considering the range of techniques that has at one extreme static scheduling (cyclic executives), and at the other, best-effort scheduling [12]. Fixed-priority scheduling falls in the middle of these extremes; and indeed is often criticized as being too static by the best-effort

lobby, and too dynamic by the cyclic executive supporters. The value of fixed-priority scheduling is that it does allow hard guarantees to be given, while allowing flexibility and various levels of non-determinism to be accommodated. This short section reviews the techniques that are available for allowing soft (non-guaranteed tasks) to be combined with the hard tasks that make up the safety critical subsystem being executed. The motivation of this section is to show that fixed priority scheduling can be extended into the realms of best-effort scheduling.

When there is no need for a hard task to be executing, the system is said to have *slack* available. This slack can be used to satisfy a number of application needs:

- The execution of soft aperiodic tasks
- The execution of background tasks
- The early completion of sporadic tasks
- The execution of components that enhance the utility of the hard task set.

The last entry can itself be subdivided into a number of techniques that are collectively known as *imprecise computation* [5, 6, 34, 43].

In general, best-effort scheduling [26] can be applied to collections of tasks running in slack time. The amount of slack available is, of course, dependent on the load exerted by the hard task set. This may vary in different modes of operation; so that, for example, a system that has lost processing resources may reduce its hard load (and increase its soft) so as to switch over to best-effort scheduling. In the extreme, a system could move to pure best-effort scheduling when the processing resource level is below that assumed for the static analysis undertaken as part of the fixed-priority approach.

More usually, there will be a mixture of hard and soft tasks to execute. Three implementation approaches can be identified:

- execute soft tasks at low priorities
- execute soft tasks using a hard server
- execute soft tasks using optimal slack scheduling

The motivation behind all three schemes is to execute soft tasks as early as possible (commensurate with all hard tasks meeting their deadlines by some appropriate safety margin). However, the schemes can also be compared by considering their overheads and the added complexity they impose on the kernel's design and behavior.

If all soft tasks are given priorities lower than any hard tasks, then no changes are needed in the kernel. Soft tasks are, however, executed only when the processor would otherwise be idle.

A number of different server schemes have been published [30, 47, 48] (e.g., polling server, priority exchange, deferrable server, extended priority exchange, and

sporadic server). Each attempts to define a capacity of work that can be assigned to soft tasks (even when there are runnable hard tasks) without jeopardizing the hard deadlines. As the servers all reserve enough capacity for the hard tasks, they are often called *bandwidth preserving*. They make differing demands on the kernel; all need task monitoring (i.e., CPU usage) and most require soft tasks to have quotas defined and enforced.

Optimal slack scheduling takes into account the phasing, and actual execution times, of tasks to calculate the maximum slack that can be made available at any moment in time (and at each priority level). For purely periodic ($D = T$) hard task sets, Lehoczky and Ramos-Thuel give an optimal scheme that can be calculated statically (i.e., off-line) [29]. For mixed periodic and sporadic task sets (and tasks with arbitrary deadlines, release jitter, etc.) Davis *et al.* have defined an optimal scheme that requires on-line analysis. The scheme would be optimal if it had zero cost and is executed frequently [22]. With realistic costs it is possible to define the frequency of execution for maximum effect.

10.6 Conclusions

In this chapter simple scheduling models have been extended to include realistic kernel features and necessary application requirements. The result is a flexible computational model supported by a rich set of analysis techniques. We can conclude that fixed-priority scheduling now represents an appropriate (and arguably, a mature) engineering approach. Although the many equations and relationships must be embedded in trusted tools, this is no different from many other engineering disciplines. The real-time systems designer now has the techniques available to engineer systems rather than just build them and then see if they meet their timing requirements during extensive (and expensive) testing.

Acknowledgments

The results presented in this chapter represent the work of many individuals within the Real-Time Systems Research Group at the University of York, UK. Thanks must particularly be given to Neil Audsley, Ken Tindell, and Andy Wellings.

References

[1] N.C. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical Report YCS 164, Department of Computer Science, University of York, December 1991.

[2] N.C. Audsley, A. Burns, M.F. Richardson, D.J Scholefield, A.J. Wellings, and H.S.M. Zedan. Bridging the Gap between Formal Methods and Scheduling

Theory. Technical Report YCS 195, Department of Computer Science, University of York, March 1993.

[3] N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Enginnering Journal*, September 1993.

[4] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, May 1991.

[5] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Incorporating Unbounded Algorithms into Predictable Real-Time Systems. *Computer Systems Science and Engineering*, 8(3):80–89, April 1993.

[6] N.C. Audsley, A. Burns, and A.J. Wellings. Unbounded Algorithms, Predictable Real-Time Systems and Ada 9X . In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pages 11–15, Phoenix, AZ, December 1992.

[7] N.C. Audsley, A. Burns, and A.J. Wellings. Deadline Monotonic Scheduling Theory and Application. *Control Engineering Practice*, 1(1), 1993.

[8] N.C. Audsley, K. Tindell, and A. Burns. The End of the Line for Static Cyclic Scheduling. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, pages 36–41, Oulu, Finland, June 1993. IEEE Computer Society Press, New York.

[9] N.C. Audsley, K. Tindell, A. Burns, M.F. Richardson, and A.J. Wellings. The DrTee Architecture for Distributed Hard Real-Time Systems. In *Proceedings of the 10th IFAC Workshop on Distributed Control Systems*, Semmering, Austria, September 1991.

[10] T.P. Baker. Stack-Based Scheduling of Realtime Processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[11] G. Bate. Mascot3: An Informal Introductory Tutorial. *Software Engineering Journal*, 1(3):95–102, 1986.

[12] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6(3):116–128, 1991.

[13] A. Burns and A.J. Wellings. Specifying an Ada Tasking Run-Time Support System. *Ada User*, 12(4):160–186, December 1991.

[14] A. Burns and A.J. Wellings. Designing Hard Real-time Systems. In *Ada: Moving Towards 2000, Proceedings of the 11th Ada-Europe Conference, LNCS 603*, pages 116–127. Springer-Verlag, New York, 1992.

[15] A. Burns and A.J. Wellings. Dual Priority Assignment: A Practical Method for Increasing Processor Utilization. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, pages 48–55, Oulu, Finland, June 1993. IEEE Computer Society Press, New York.

[16] A. Burns and A.J. Wellings. HRT-HOOD: A Design Method for Hard Real-Time Ada. *Real-Time Systems*, 6(1):73–114, 1994.

[17] A. Burns and A.J. Wellings. Implementing Analysable Hard Real-Time Sporadic Tasks in Ada 9X. *Ada Letters*, 14(1):38–49, 1994.

[18] A. Burns, A.J. Wellings, C.M. Bailey, and E. Fyfe. The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation. Technical Report YCS 190, Department of Computer Science, University of York, 1993.

[19] A. Burns, A.J. Wellings, C.M. Bailey, and E. Fyfe. The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation. In *Ada sans frontieres, Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science*. Springer-Verlag, New York, 1993.

[20] A. Burns, A.J. Wellings, and A.D. Hutcheon. The Impact of an Ada Runtime System's Performance Characteristics on Scheduling Models. In *Ada sans frontieres Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science 688*, pages 240–248. Springer-Verlag, New York, 1993.

[21] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[22] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *Proceedings of the Real-Time Systems Symposium*, pages 222–231, December 1993.

[23] R. Gerber and S. Hong. Semantic-Based Compiler Transformations for Enhanced Schedulability. In *Proceedings of the Real-Time Systems Symposium*, pages 232–243, December 1993.

[24] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, San Antonio, TX, December 1991.

[25] M.G. Harmon, T.P. Baker, and D.B. Whalley. A Retargetable Technique for Predicting Execution Time. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 68–77. IEEE Press, New York, December 1992.

[26] E.D. Jenson, C.D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.

[27] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *BCS Computer Journal*, 29(5):390–395, October 1986.

[28] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, Lake Buena Vista, FL, December 1990.

[29] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks Fixed-Priority. In *Proceedings of the Real-Time Systems Symposium*, pages 110–123, December 1992.

[30] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhancing Aperiodic Responsiveness in Hard Real-Time Environment. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, San Jose, CA, December 1987.

[31] J.Y.T. Leung and M.L. Merrill. A Note on Preemptive Scheduling of Periodic Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[32] J.Y.T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, December 1982.

[33] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[34] J.W.S. Liu, K.J. Lin, W.K. Shih, A.C.S. Yu, J.Y. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computations. *IEEE Computer*, pages 58–68, May 1991.

[35] C.D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Journal of Real-Time Systems*, 4(1):37–53, March 1992.

[36] J. Rushby. Kernels for Safety? In *Safe and Secure Computing Systems*, pages 310–320. Blackwell Scientific, Cambridge, MA, 1987.

[37] D. J. Scholefield. *A Refinement Calculus for Real-Time Systems*. Department of Computer Science, University of York, 1992.

[38] D. J. Scholefield and H.S.M. Zedan. The Temporal Agent Model: Theory and Practice. Technical Report YCS 163, Department of Computer Science, University of York, 1991.

[39] D. J. Scholefield and H.S.M. Zedan. A Standard for Finite TAM. Technical Report YCS 206, Department of Computer Science, University of York, September 1993.

[40] D. J. Scholefield and H.S.M. Zedan. Real-Time Refinement: Semantics and Application. In *Proceedings of MFCS '93*, Gdansk *(LNCS 711)*. Springer-Verlag, New York, 1993.

[41] D. J. Scholefield and H.S.M. Zedan. A Specification Oriented Semantics for Refinement of Rea-Time Systems. *Theoretical Computer Science*, 130, 1994.

[42] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[43] W.K. Shih, J.W.S. Liu, and J.Y. Chung. Algorithms for Scheduling Imprecise Computations with Timing Constraints. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, December 1989.

[44] H.R. Simpson. A Data Interactive Architecture (DIA) for Real-Time Embedded Multi-processor Systems. In *Computing Techniques in Guided Flight RAe Conference*, April 1990.

[45] M. Spivey. *The Z Notation: A Reference Manual*, 1989.

[46] M. Spivey. Specifying a Real-time Kernel. *IEEE Software*, 7(5):21–28, September 1990.

[47] B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 251–258, December 1988.

[48] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Journal of Real-Time Systems*, 1:27–69, 1989.

[49] F. Stanischewski. FASTCHART: Performance, Benefits and Disadvantages of the Architecture. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, pages 246–250, Oulu, Finland, June 1993. IEEE Computer Society Press, New York.

[50] K. Tindell. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Technical Report YCS189, Department of Computer Science, University of York, December 1992.

[51] K. Tindell. Adding Time-Offsets to Schedulability Analysis. Technical Report YCS 221, Department of Computer Science, University of York, January 1994.

[52] K. Tindell, A. Burns, and A. Wellings. Allocating Real-Time Tasks (An NP-Hard Problem Made Easy). *Journal of Real-Time Systems*, 4(2):145–165, June 1992.

[53] K. Tindell, A. Burns, and A.J. Wellings. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[54] N. Zhang, A. Burns, and M. Nicholson. Pipelined Processors and Worst Case Execution Time. *Real-Time Systems*, 5(4):319–343, 1993.