Flexible Scheduling of Hard Real-Time Systems

Neil C. Audsley

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

August 1993

Contents

Acknowledgements

I would like to thank my supervisor, Dr. Alan Burns, for his interest and involvement with this work and my research in general.
I am grateful to Rob Davis for reading earlier drafts of this dissertation. Also, I am grateful to all those people with whom I have had discussions upon various matters, especially Mike Richardson, Andy Wellings, Ken Tindell and Rob Davis.
Finally, my thanks go to Liz, who encouraged, fed and watered me whilst writing-up.

To Grandad, and his belief in education.

Declaration

Certain parts of this thesis have appeared in previously published papers, specifically the following references (marked * for principle author):

*N. C. Audsley, A. Burns, "Scheduling Real-Time Systems", YCS 134, Department of Computer Science, University of York, 1990.

*N. C. Audsley, "Deadline Monotonic Scheduling", YCS 146, Department of Computer Science, University of York, 1990.

*N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach", Proceedings of the IFAC/IFIP Workshop, Atlanta, Georgia, USA, 15-17 May 1991, ("IFAC Workshop Series, 1992 Number 1", pp127-132, Pergamon Press, 1992).

N. C. Audsley, A. Burns, K. Tindell, M. F. Richardson, A. J. Wellings, "The DrTee Architecture for Distributed Hard Real-Time Systems", Proceedings 10th IFAC Workshop on Distributed Control Systems, Semmering, Austria, pp49-54, 9-11 September 1991.

N. C. Audsley, A. Burns, K. Tindell, M. F. Richardson, A. J. Wellings, "The DrTee Architecture for Distributed Hard Real-Time Systems", Proceedings IEEE Workshop on Architecture Support for Real-time Systems, San Antonio, Texas, December, 1991.

*N. C. Audsley, "Resource Control For Hard Real-Time Systems: A Review", YCS 159, Department of Computer Science, University of York, 1991.

*N. C. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times", YCS 164, Department of Computer Science, University of York, 1991.

*N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, "Incorporating Unbounded Algorithms Into Predictable Real-Time Systems", YCS 171, Department of Computer Science, University of York, 1992.

N. C. Audsley, A. Burns, K. Tindell, M. F. Richardson, A. J. Wellings, "The DrTee Architecture for Distributed Hard Real-Time Systems", Proceedings

9th IEEE Workshop on Real-Time Operating Systems and Software, Pittsburgh, pp57-61, May 1992.

*N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, "Deadline Monotonic Scheduling Theory", Proceedings IFAC/IFIP International Workshop on Real-time Programming, WRTP'92, Bruges, pp55-60, June 1992.

*N. C. Audsley, A. Burns, A. J. Wellings, "Unbounded Algorithms, Predictable Real-Time Systems and Ada 9X", Proceedings IEEE Workshop on Imprecise and Approximate Computation, Phoenix, pp11-15, December 1992.

*N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, "Incorporating Unbounded Algorithms Into Predictable Real-Time Systems", Computer Systems Science and Engineering, 8(3), pp80-89, April 1993.

*N. C. Audsley, A. Burns, A. J. Wellings, "Deadline Monotonic Scheduling Theory and Application", Control Engineering Practice, 1(1), pp71-78, 1993.

N. C. Audsley, K. Tindell, A. Burns, "The End of the Line for Static Cyclic Scheduling", Proceedings of 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland, pp36-41, 1993.

N. C. Audsley, A. Burns, K. Tindell, M. F. Richardson, A. J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling", Software Engineering Journal, pp284-292, September 1993.

Abstract

The design and implementation of increasingly complex hard real-time computer applications has been limited by the severe restrictions imposed by run-time support systems, in particular the scheduler. The restrictions arise from the assumptions required to afford 100% guarantees to processes with hard deadlines. This constrains the application to conform to the model required by the scheduler. For example, rate-monotonic scheduling, as originally proposed, restricts systems to periodic independent processes, with no shared resources.

This thesis examines the trade-offs between the constraints required to enable offline feasibility analysis and the demand for increased flexibility of the next generation of hard real-time systems. Initially, feasibility analysis is developed for a flexible process model. Tests are derived for static priority pre-emptive scheduling which relaxes the common restriction that the period of a periodic process must be equal to its deadline. One result of relaxing this constraint is that sporadic processes can be accommodated directly. The feasibility analysis is extended to permit processes that have their first execution offset from the initial start of system execution. This enables process sets with arbitrary precedence constraints to be expressed. An optimal priority assignment algorithm for processes that have offsets is given.

Guaranteeing process deadlines using pessimistic estimations of processor and resource requirements implies that at run-time, an under-utilisation of system resources will occur. An approach is proposed to identify this spare system capacity as soon as possible within the execution of a process. Conventionally, this spare capacity is used for the executions of processes without hard deadlines. This thesis presents an approach that enables spare capacity to be allocated to processes with hard deadlines so that system utility can be increased.

This thesis contends that the adoption of such an approach imposes no real restrictions upon the application engineer, and greatly increases the flexibility of hard real-time systems, enabling many of the requirements of the next generation of hard real-time applications to be met.

Chapter 1.
Introduction

Increasingly, society is placing trust in computers to perform tasks safely and reliably. The "fly-by-wire" flight systems on some of todays passenger aircraft place almost total reliance in computers to control the aircraft in flight, to affect a change of course at a pre-determined time, or prevent the pilot from performing manoeuvres beyond the design limitations of the airframe. An example of such a system is the space shuttle, which relies heavily upon computers for safe mission accomplishment [Carlow84]. The role of the computer in such systems is vital, as they are

"supporting, or necessary to, life"

[*vital*: Chambers 20th Century Dictionary]

Given the importance of the computer in these systems, and the inherent trust that is placed in them, we need to be absolutely sure about the practical and theoretical principles used to construct such systems.

The control systems described above are often termed real-time, in that the actual absolute time at which the computer performs an application process is significant. Consider the embedded control system in a washing machine. This must perform specific functions at specific times and dynamically react to changes in the environment (i.e. water temperature) in order to wash and dry its load. When the consequence of failing to meet timing requirements is potentially catastrophic, the systems are termed hard real-time. In a nuclear power station, cooling rods must be inserted into the pile within a time-limit else the reactor may go to a critical state, with obvious consequences. The potential for catastrophic failure of systems that do not perform within pre-defined functional and timing requirements has made hard real-time systems a major area of research. It is these systems that will be studied in this thesis.

The fundamental requirement of hard real-time systems is *predictable* behaviour at run-time. In this context, predictability can be seen from two main viewpoints: *functionality* and *timeliness*. Functionality refers to the logical execution. This must be defined by requirements specification, given the possibility of hardware component failure within the system, and of transient

errors occurring within a component (either software or hardware). Timeliness refers to the time at which the logical execution occurs. Any timing bounds on the execution must be adhered to. Conventionally, this equates to ensuring a process (or task) receives sufficient resource to complete its logical execution between a given start time and a given deadline. As intimated above, a failure in either the functional or timing domains can have catastrophic results in hard real-time systems.

The desire for predictability in hard real-time systems must be reflected in all phases of the software life-cycle, from inception within requirements capture, to the realisation of a running system, the latter being composed of application software, kernel and/or run-time support software, and hardware. Clearly, predictability must be maintained across the broad range of complex interactions between these three basic constituent parts of a running system.

1.1 Scheduling in Hard Real-Time Systems

To maintain timing predictability at run-time, application functions must execute within given time bounds. The problem of interleaving the execution of application computations, or separate processes performing those computations, is conventionally performed by a *scheduler* [Lister84]. Since the predictability of a hard real-time system must be determined before it is run, offline feasibility analysis is required which can decide whether a proposed scheduling technique will ensure that all timing requirements of all processes are met at run-time.

Operational research approaches, such as queuing theory or job-shop scheduling, are not applicable for hard real-time systems. Such approaches are concerned with average performance: this is of secondary importance to the predictable meeting of hard deadlines. For the same reasons, the scheduling approaches of time-sharing systems, which are dedicated to a metric of fairness amongst competing processes, are not appropriate.

Many applicable scheduling techniques have been proposed for hard real-time systems, together with associated analyses that determine whether a set of processes will meet all deadlines at run-time: that is determine whether a process set is feasible (hence *feasibility test*). The scheduling techniques fall into different categories according to the nature of the decision made at run-time as to which process or application computation to execute at any time. Two commonly used approaches are cyclic scheduling and static priority scheduling.

Cyclic scheduling defines a schedule off-line, in the form of a list of times to execute each process. Often the schedule is hand-crafted in an *ad hoc* manner, needing to be re-calculated for even the slightest change in system requirements or implementation. Hence systems employing this scheduling approach can be brittle and difficult to maintain. Another problem with cyclic scheduling is lack of flexibility, given the fixed nature of the executions of processes.

Static priority scheduling dictates that priorities are assigned to processes, with the highest priority runnable process scheduled for execution at any point in time. Many different analyses have been proposed, each providing a sufficient decision regarding whether timing requirements will be met. The differences in the analyses lie in their inherent assumptions and restrictions. For example, two differing analyses cater for independent processes and synchronising processes respectively. The former analysis is simpler than the latter, although the latter permits more general process systems. This trade-off between analysability and boundability on the one hand and generality and flexibility on the other, is fundamental to scheduling theory: fewer restrictions upon processes equates to more general and flexible systems; however, the analysis is more difficult to formulate and more complex to perform.

With any scheduling approach for hard real-time systems, guarantees regarding the timing requirements are made, from necessity, using worst-case (or maximal) evaluations of individual process requirements for resources (e.g. processor, shared memory, databases). Often, analyses for worst-case execution time and resource requirements are overly pessimistic: the expected worst-case does not in fact occur. This is illustrated by the control-flow analysis to calculate worst-case execution time: the conditional execution of one block of code may also imply the non-execution of another conditional block; worst-case analysis may conclude that both blocks will be executed. The effect on system feasibility of pessimistic worst-case estimations is apparent in fewer systems being declared feasible; systems that may well, at run-time, meet all timing requirements. Reduction in pessimism, both in terms of worst-case analysis and feasibility analysis is required.

Even if the pessimism of worst-case estimations and feasibility are reduced, the actual utilisation of system resources at run-time will still, in general, be less than off-line worst-case estimations. At run-time, unused execution time and resources, guaranteed to be available off-line, can be exploited to increase the utility of the system. In present systems, this is

achieved by letting other processes complete earlier than anticipated. Other, perhaps more useful approaches, include the use of spare time and resources by other processes to perform extra, unguaranteed execution, to increase the utility of the system.

1.2 Thesis Goals

This thesis will explore the trade-offs evident between boundability of system timing constraints and the flexibility provided for application implementations. Specifically, the trade-off between analysability and flexibility will be examined in terms of flexible scheduling techniques and associated analyses for hard real-time systems, in particular, static priority scheduling. Two forms of flexibility are introduced: offline and online.

Offline flexibility reflects the restrictions imposed upon the application by the scheduler and associated analysis. An increase in flexibility is achieved by relaxing these restrictions. Additionally, a method for efficient optimal priority assignment is established for process systems with arbitrary timing constraints (previous optimal assignments rely upon all processes having, at some point in time during system execution, an identical start time). Then, sufficient feasibility tests are developed for such process models.

Online flexibility relates to the ability of the running system to fully utilise system resources. Given the availability of spare time and resources at run-time, a framework for its constructive dynamic re-use is proposed by which processes may increase their utility or value.

The goal of the research undertaken, and described within, is to increase flexibility in hard real-time systems, permitting applications to be designed in a less constrained manner. Therefore, the objective of this thesis is to test the hypothesis:

> "the flexibility of hard real-time systems can be improved by weakening the constraints placed upon applications by the choice of scheduling approach and its associated feasibility analyses; and by the detection and re-use of spare capacity at run-time."

1.3 Structure of Thesis

The remaining sections in this chapter provide definitions and a nomenclature for the rest of the thesis. Chapter 2 forms a review of scheduling literature. In

Chapter 3, a framework enabling the development of flexible scheduling for hard real-time systems is outlined. Chapter 4 examines increasing offline flexibility using deadline monotonic scheduling and associated feasibility analysis. This form of scheduling permits process deadlines to be less than their periods. Further offline flexibility improvements are given in Chapter 5, which provides an optimal priority assignment and feasibility analysis for processes with arbitrary start times. The effects of inter-process interaction are also considered. In Chapter 6, methods for increasing online flexibility are considered. In particular, an efficient approach for identifying spare system capacity at run-time is developed, with Chapter 7 examining issues related to the re-use of such spare capacity. Finally, Chapter 8 provides conclusions and identifies areas requiring further work.

1.4 Definitions

A *computer system* consists of one or more *nodes*, each node forming an autonomous physical computing resource. A node can consist of a single processor, in which case it is termed *uniprocessor*; or many processors, termed *multiprocessor*. Resources in the system are either physical, for example devices, or logical, for example shared data. All physical resources (apart from the processor) are mapped onto logical resource representations (e.g. software device drivers).

Each processor executes a number of *processes*. These are the logical units of concurrency within the system; they interact to achieve the common goal of the system. Processes execute code which is either shared (with other processes, for example a common procedure or function) or non-shared.

Processes whose progress is not dependent upon the progress of other processes are termed *independent*. This definition discounts competition for processor time. *Interdependent* processes interact in three main ways: competition for shared logical resources, precedence-relationships between processes and non-pre-emptability. The first reflects the possibility that a resource may not be available when requested by a process. In this case, the process becomes *blocked*. A process may also *suspend* itself, by execution of a "delay" statement. Precedence-constraints (also called precedence-relationships) define a partial or total order on the execution sequences of processes. That is, one process may be prevented from executing until the completion of a number of predecessor processes. Non-pre-emptable code

requires that any process executing it cannot be interrupted or pre-empted. Processes may execute a mixture of pre-emptable and non-pre-emptable code.

Processes are generally partitioned into two groups according to the nature of their invocations or *releases*. If a process is released at regular, pre-defined intervals, it is termed *periodic*. Other processes are termed *non-periodic*. These later processes are further subdivided: *sporadic* processes are those which have a minimum time between successive releases; *aperiodic* processes have unconstrained release patterns, with the possibility of an unbounded number of instances of the process requiring execution at any time.

In a real-time system, the time by which a process must complete execution maximum is termed the *deadline* (relative to the start time). This is derived from application requirements. The actual time at which the process completes is termed the *response time*. Between the invocation of a process and its deadline the process requires a given amount of *computation time*. For processes with hard deadlines the computation time must be boundable: the maximum time required is termed the *worst-case execution time* (WCET). Likewise, the minimum execution time is termed the *best-case execution time* (BCET). The precise amount of execution time required for a release of a process is termed the actual execution time (ACET).

A *scheduler* consists of an algorithm or *policy* which produces a *schedule* defining the execution order of processes on a processor. A *feasibility test* (also termed *schedulability test*) determines if a schedule meets a given pre-condition. A typical pre-condition for hard real-time systems is that all processes with hard deadlines always meet those deadlines. If the test is passed, the schedule is termed *feasible*. An *optimal* scheduler is able to produce a feasible schedule for all feasible process sets.

A scheduler is termed *offline* if all scheduling decisions are made prior to system execution. For example, cyclic scheduling is an offline approach: a table is generated that contains all scheduling decisions for use during run-time. An *online* scheduler makes decisions at run-time. For example, static priority scheduling requires that the highest priority runnable process is executed at all times. In essence, the decisions made by the online scheduler are based upon process characteristics.

Schedulers may also have the attribute *deterministic*, whereby the decision made at any point in time can be pre-determined offline given the system state at that time. For example, static priority scheduling is deterministic since at any time the highest priority runnable process is executed.

1.5 Nomenclature

A process set Δ contains n processes, namely $\{\tau_1,..,\ \tau_i,..,\ \tau_n\}$, where the priority of τ_i is i with 1 being the highest priority, and n the lowest. When processes have not been assigned priorities, they are labelled $\{\tau_A,..\}$ (i.e. upper-case subscript) with the process set still having cardinality n.

The timing characteristics of a process are given by O_i,C_i,D_i,T_i (or O_A,C_A,D_A,T_A) representing the start time (relative to system start time at time 0), worst-case computation time, deadline and period of τ_i (or τ_A). Periodic processes are released initially at O_i (or O_A) and subsequently every T_i (or T_A) time units. The deadline of a process is relative to its release. For sporadic processes, T_i (or T_A) defines the minimum inter-arrival time between successive releases.

Chapter 2.
Scheduling Hard Real-Time Systems: A Survey

Many papers have been published in the field of real-time scheduling, including several general surveys [Gonzalez77, Casavant88] and those concentrating on hard real-time systems [Cheng87, Audsley90, Burns91a]. Many varied aspects of scheduling are worthy of note in any survey on the subject, from the theoretical complexity of the problem through to specific scheduling approaches themselves. The latter depend largely upon the exact characteristics and assumptions inherent in the target system; from uniprocessor systems with periodic processes, to distributed systems containing periodic and sporadic processes which may share resources. Therefore, to provide background context for this thesis, the following survey concentrates upon:

(i) computational complexity of scheduling;
(ii) scheduling with simplistic assumptions;
(iii) scheduling with realistic assumptions.

Primarily, the above are considered in terms of uniprocessor systems; although key results are given for multiprocessor systems also.

2.1 Complexity Results

Complexity can be examined in three main ways:

(i) complexity of finding a schedule;
(ii) complexity of testing the schedule;
(iii) run-time scheduling complexity.

Clearly, exponentially complex online scheduling schemes are not ideal for hard real-time systems - their impact upon the processor time available for application software is extreme. The following sections review scheduling complexity work in the literature.

2.1.1 Computability and Decidability

Two separate considerations are necessary: computability and decidability. These parameters are illustrated using the travelling salesman problem [Wilf86]. Whilst deciding if a solution to the problem exists (i.e. a route with cost less than a given value) is not possible (in general) in polynomial time (and is in fact NP-complete); computing whether a particular route has a cost below a given value is trivial. Computability and decidability for scheduling an arbitrary process set are described as follows:

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

Finding a feasible schedule consists, essentially, of an exhaustive search amongst all possible schedules. Computing whether a schedule is feasible could involve running the schedule to its natural conclusion, or to a point where it repeats.

2.1.2 Important Uniprocessor Results

Important results for uniprocessor scheduling stem from the work of Garey and Johnson [Garey75, Garey77, Garey78] summarised in Table 1.1. Concluding from the results in the table, it is clear that for uniprocessor scheduling, all combinations of desirable factors for flexible hard real-time systems (e.g. arbitrary computation times and any number of shared resources) result in NP-complete complexity for deciding whether a feasible schedule exists.

Other important results include those given by Mok [Mok83]. The most important relates to computing whether a feasible schedule exists when the processes use mutual exclusion primitives:

> *"The problem of deciding whether it is possible to schedule a set of periodic processes which use semaphores only to enforce mutual exclusion is NP-hard."* [Mok83]

The above is proved by reduction to the 3-PARTITION problem which is known to be NP-hard [Garey78]. The proof is based upon partitioning non-preemptable processes into the intervals between successive executions of another process, the latter having to be executed at exact points in time.

Number of Processors	Number of Resources	Precedence Constraints	Process Execution Times	Other Conditions	Complexity
≥ 1	0	none	$\leq$ polynomial expression in the number of processes		polynomial
≥ 0	≥ 0 (but finite)	none	all unit time		polynomial
≤ 2	0	none	all unit time		polynomial
1	0	any	all unit time		polynomial
1	0	none	arbitrary		NP-complete
1	0	none	all unit time	deadline for the process set	polynomial
1	0	none	either 1 or 2	deadline for the process set	NP-complete
1	0	none	arbitrary	deadline for the process set	NP-complete
1	0	none	all unit time	minimise total tardiness for process set	NP-complete
2	1	limited	all unit time		NP-complete
≥ 3	1	none	all unit time		NP-complete
arbitrary	0	limited	all unit time		Polynomial
arbitrary	0	arbitrary	all unit time		NP-complete

Table 1.1: Complexity Results for Finding a Feasible Schedule.

We now turn to the problem of computing whether a schedule is feasible. The schedule takes one of two forms:

(i) a list of exact times at which processes are executed;

(ii) a set of processes and their timing characteristics, together with an online scheduling algorithm.

To test schedules of form (i), we must simulate it up to some time limit. This time must be bounded, since the schedule list must have been defined up to this point. It is entirely possible that the checking of the schedule occurs as it is created. Schedules of form (ii) require a feasibility test. This incorporates the characteristics of the online scheduling algorithm, together with the assumptions inherent in the timing characteristics of the processes. For example, whether or not processes may be pre-empted, or have deadlines less than their periods.

One important result has shown that a necessary and sufficient test for a single processor and an arbitrary process set (no constraints on period, deadline, computation time etc.) is NP-hard [Leung80]. Factors that can reduce the complexity include:

- constraining the characteristics of the process set;
- defining a sufficient but not necessary test.

Constraints could include unit length processes (see Table 1.1). However, this would make system design difficult. Defining a less complex test could result in a test that fails every process set presented to it. This is sufficient and not necessary, but practically useless.

2.1.3 Important Multiprocessor Results

The results of multiprocessor complexity work build upon that of uniprocessor studies. Generally, the addition of processors increases the complexity of both finding if a valid schedule exists, and validating a given schedule. One cause of this is process allocation. For example, a set of independent processes is to execute on two processors. These processes require a total of $2b$ units of execution time. An optimal schedule would be one in which the processes were split into two groups, each group having a total execution time of b units. This splitting is an NP-complete problem, having been shown to reduce to the 2-PARTITION problem [Garey77].

Some results of scheduling complexities for multiprocessor systems are given in Table 1.1. The simple case (two processors, zero resources, no precedence constraints, unit execution time of processes) has polynomial complexity. The following constraints could be weakened:

(i) greater than zero resources, OR

(ii) precedence constraints permitted, OR

(iii) non-unit execution times of processes.

Weakening any of the above constraints results in NP-complete complexity.

Deciding whether a given schedule is feasible is at least as complex as for uniprocessor systems, due to the extra dimension of allocation. Indeed, Leung and Merrill have shown that for an arbitrary process system and multiprocessors, the decision is NP-hard [Leung80].

2.1.4 Summary

Deciding whether a valid schedule exists is NP-hard in all but trivially simple cases. Hard real-time systems have processes that are likely to have:

- non-unit computation times;
- resources that need to be accessed in a mutually exclusive manner (e.g. devices, communications media etc.);
- complex interactions between processes (e.g. precedence constraints).

From the above discussion, it is apparent that for these characteristics of hard real-time systems determining the existence of a valid schedule is NP-hard.

Computing whether a given schedule is feasible is achieved via a sufficient feasibility test. A sufficient and necessary test is NP-hard in the general case. However, less complex sufficient and not necessary tests exist such that it is still possible for process sets to pass the test, whilst permitting some of the characteristics of process sets that increase the complexity of determining feasibility. Hence, a trade-off between the complexity of the test and its accuracy can be observed, that is a trade-off between the complexity of the feasibility test and the constraints imposed upon the processes.

2.2 Simple Uniprocessor Scheduling

This section discusses the current state of scheduling for a hard real-time system consisting of a single uniprocessor node with extreme constraints upon resources and process characteristics.

The scheduling approaches considered are sub-divided into three categories:

(i) offline;

(ii) online static priority;

(iii) online dynamic priority.

The first category refers to those approaches that create a schedule offline. Mostly, they create a list of processes (or parts of processes) and assign them to exact points in time for their execution. The list is bounded, enabling it to be continually repeated. Hence, cyclic executives are formed, with the approaches commonly known as *cyclic scheduling*. The run-time scheduler for these approaches is trivial.

Online approaches are split into those which assign static priorities to processes (category (ii)), and those varying process priorities at run-time (category (iii)). At run-time, the highest priority runnable process is executed at any time. Dynamic priority approaches allow process priorities to change at run-time.

After a brief statement of assumptions for the simple uniprocessor model in the following section, proposed scheduling schemes for the above three categories are reviewed.

2.2.1 Restrictions and Assumptions

The following common restrictions are made [Liu73, Cheng87]:

(i) computation times for a given process are constant;
(ii) all processes are periodic;
(iii) no precedence relations exist between processes;
(iv) no inter-process communication or synchronisation is permitted;
(v) no process resource requirements are considered;
(vi) system overheads (e.g. context switches) have zero cost;
(vii) processes are not permitted to voluntarily suspend themselves.

We assume that all processes $\tau_i \in \Delta$ have $C_i \leq D_i$ and $C_i \leq T_i$.

2.2.2 Offline Scheduling

Offline approaches are based upon an exhaustive search amongst all possible schedules for one that is feasible [Wilf86]. An optimal search would always find a feasible schedule if one exists. Such a search has, in general, NP-complete complexity (except for even more restricted models). Therefore, proposed approaches are sub-optimal, limiting the search-space examined.

Two main methods could be used to limit search space: *approximate* and *heuristic*. An approximate method could stop when a sufficiently "good" schedule is found. One difficulty with this approach is that it relies upon a metric to evaluate the relationship between the current and optimal solutions. That is, how to define relative quality of different solutions. This may not be possible until a feasible solution is found.

Heuristic approaches define rules which restrict and guide the path taken through the search space. The rules are often intuitive, encapsulating some observation which has been seen to aid improvement from an infeasible schedule towards one that is feasible. It is the heuristic approach that has been adopted for many offline scheduling approaches.

Stankovic *et al* begin with an empty schedule [Stankovic87a]. 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.

Fohler *et al* [Fohler89] use a similar scheme for generating offline schedules for the Mars system [Damm89, Kopetz85, Kopetz89]. A schedule is divided into fixed time slots and then filled with processes. Again search space is restricted by use of heuristic functions.

2.2.3 Static Priority Scheduling

The second form of simple uniprocessor scheduling, static priority, incorporates three aspects:

(i) priority assignment;

(ii) feasibility test;

(iii) online priority pre-emptive dispatcher.

Proposed approaches for (i) and (ii) are given in the following sections.

Assignment of Static Priorities to Processes

Proposed priority assignment techniques depend upon the exact timing characteristics of processes. When all processes $\tau_i \in \Delta$ have $C_i \leq D_i = T_i$ and $O_i = 0$, rate-monotonic priority assignment is known to be optimal [Liu73]. Here, the process with the shortest period is assigned the highest priority; the process with the second shortest period is assigned the second highest priority. Finally, the longest period process is assigned the lowest priority. This assignment is optimal in the sense that if a priority assignment exists such that all processes will meet their deadlines at run-time, a rate-monotonic priority assignment will also ensure that process deadlines are met. It is noted that if two processes have equal priority, ties may be broken arbitrarily without affecting optimality of priority assignment.

A similar priority assignment, deadline monotonic, has been proposed for process sets where all $\tau_i \in \Delta$ have $C_i \leq D_i \leq T_i$ and $O_i = 0$ [Leung82]. Priorities are assigned in a similar manner to rate-monotonic: the shortest deadline process is assigned the highest priority; processes with successively longer deadlines are assigned successively lower priorities. We note that deadline-monotonic priority assignment is equivalent to rate-monotonic priority

assignment when, for all processes $\tau_i \in \Delta, D_i = T_i$. Deadline-monotonic priority assignment is optimal in a similar manner to rate-monotonic: if there exists a feasible priority ordering over a set of processes, a deadline-monotonic priority ordering over those processes will also be feasible.

Both rate-monotonic and deadline-monotonic priority assignments are no-longer optimal if either

(i) process deadlines are permitted to exceed their periods, i.e. $D_i > T_i$, [Lehoczky90] or

(ii) processes have $O_i \neq 0$ [Leung82].

In Chapter 5 an optimal priority assignment is presented which does not suffer from the above problems.

Feasibility Testing of Static Priority Process Sets

Feasibility tests for static priority systems are not, in general, dependent upon the exact priority assignment approach used; rather upon the exact process timing constraints. Thus feasibility tests for process sets where all processes $\tau_i \in \Delta$ have $C_i \leq D_i = T_i$ are applicable for any priority ordering, including rate-monotonic; likewise feasibility tests applicable for processes with deadlines less than their periods will be applicable to any priority ordering, including deadline-monotonic.

The fundamental result regarding feasibility of static priority process sets when all processes $\tau_i \in \Delta$ have $O_i = 0$ is that only the first deadline of each τ_i (at D_i) need be checked. The point in time at which all processes are released simultaneously is termed a *critical instant* [Liu73]. If the deadline of a process is met for a release commencing at a critical instant, all subsequent deadlines will be met. The result is based upon the observation that at a critical instant, the work-load on the processor is at a maximum. Thus, the demand of higher priority processes $\tau_i..\tau_{i-1}$ in [0, D_i) is at a maximum, creating the hardest situation for τ_i to meet its deadline at D_i.

The $D_i = T_i$ Feasibility Tests

For processes assigned priorities according to the rate-monotonic approach, Liu and Layland proposed a test based upon processor utilisation [Liu73]:

$$n(2^{1/n} - 1) \leq \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

This implies, for a set of two processes, if the combined utilisation of those processes is no greater than 82.84%, the process set is feasible. As the

cardinality of the process set approaches infinity, the permissible utilisation approaches 69.31%. This test is sufficient but not necessary, as process sets with greater utilisation than the level given by equation (2.1) may still be feasible. We note that the complexity of this test is O(n) in the number of processes.

A sufficient and necessary test has been identified by Lehoczky *et al* [Lehoczky89]. This test is applicable to any arbitrary priority assignment. It formulated as follows. The work-load $W_j(t)$ on the processor at any time t due to processes of equal or higher priority than j is given by:

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

This equates to the sum of the computation times of all releases of processes with priority equal or greater to j in the interval [0, t]. Process τ_j is feasible if and only if the following condition holds:

$$\min_{(0<t\leq T_j)}\left(\frac{W_j(t)}{t}\right)\leq 1$$

The $D_i \leq T_i$ Feasibility Tests

When some or all processes have deadlines less than their periods, the tests described for $D_i=T_i$ are, in general, not applicable. The simplest method of determining feasibility is to create a schedule for the interval [0, $max(D_{i=1..n})$) [Leung80]. If all deadlines are met in this interval, the processes are feasible. This approach is computationally expensive, although sufficient and necessary.

Separate sufficient and necessary tests have been proposed by Joseph and Pandya [Joseph86], Audsley *et al* [Audsley91c, Audsley91d] and Nassor and Bres [Nassor91]. The latter test extends the $D_i=T_i$ test defined by Lehoczky *et al* [Lehoczky89]. Further feasibility tests (and comparisons) are presented in Chapter 4.

The $D_i > T_i$ Feasibility Tests

For the case where process deadlines are permitted to exceed periods, Lehoczky has proposed two sufficient and not necessary feasibility tests [Lehoczky90]. Both tests are based upon utilisation, extending the $D_i=T_i$ utilisation test of Liu and Layland [Liu73]. The tests restrict all processes $\tau_i \in \Delta$ to have $D_i=kT_i$, where k is constant across all processes. One test restricts k to be an integer, the other does not. The tests are sufficiently complex to omit reproduction here.

Tindell has also proposed a feasibility test for processes with deadlines greater than periods [Tindell92], based upon a static priority feasibility test for processes with deadlines no greater than periods [Audsley91c].

2.2.4 Dynamic Priority Scheduling

The third form of simple uniprocessor scheduling, dynamic scheduling, prescribes that all scheduling decisions are made online as the system executes. The three main forms of dynamic scheduling are now discussed. All are described in terms of priority pre-emptive dispatching.

Earliest Deadline

Earliest deadline scheduling assigns the runnable process closest to its deadline the highest priority at any time [Liu73]. This process then executes until a point in time when it either completes execution, or another runnable process has a closer deadline. The feasibility of process sets under the earliest deadline regime is given by:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.0 \qquad (2.2)$$

The above assumes that all processes $\tau_i \in \Delta$ have $D_i = T_i$.

Given the constraints of the simple model, Liu and Layland have shown that earliest deadline is an optimal form of dynamic scheduling:

> *"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."* [Liu73]

Least Laxity

Least laxity scheduling assigns the process with the smallest laxity (time remaining before deadline minus remaining computational requirement), the highest priority at any time [Dertouzos89]. The executing process has constant laxity, whilst the laxities of the other processes decrease as the former executes. Eventually, one of the latter processes may have the smallest laxity, so becoming the highest priority process, pre-empting the executing process.

A problem arises with this scheme when two processes have similar laxities. One process will run for a short while and then get pre-empted by the other and *vice versa*. Thus, many context switches occur in the lifetime of the processes. This can result in "thrashing", a term used in operating systems to

indicate that the processor is spending more time performing context switches than useful work [Lister84].

The feasibility constraint for least laxity is exactly that stated for earliest deadline scheduling (see equation (2.2) above). The least laxity heuristic is optimal in the same way as the earliest deadline approach [Dertouzos89].

Value Functions

Whilst static priority assignment fixes the real-time importance of processes, value functions enable the varying importance of a process throughout the system lifetime to be described. Specifically, the value function of a process defines the benefit to the system of completing that process at a given time. For example, Figure 2.1 shows a process which is considered damaging if it executes before its start time (i.e. negative value); has constant positive value (i.e. benefit) if it completes before its deadline; with deteriorating value for completion after the deadline.

Locke has shown that the maximum value can be obtained across all processes in the system if, at any time, the process with the highest value density is assigned the highest priority [Locke86]. Value density is defined as the constant completion value (i.e. the value of completing the process in Figure 2.1 between its start time and deadline) divided by its remaining computation time.

The system overhead for value function scheduling becomes increasingly intrusive on system performance as the value functions become increasingly complex. For example, if constant completion values are not evident (e.g. increased value for completing a process as near to its deadline as possible), value functions would have to be continually evaluated.

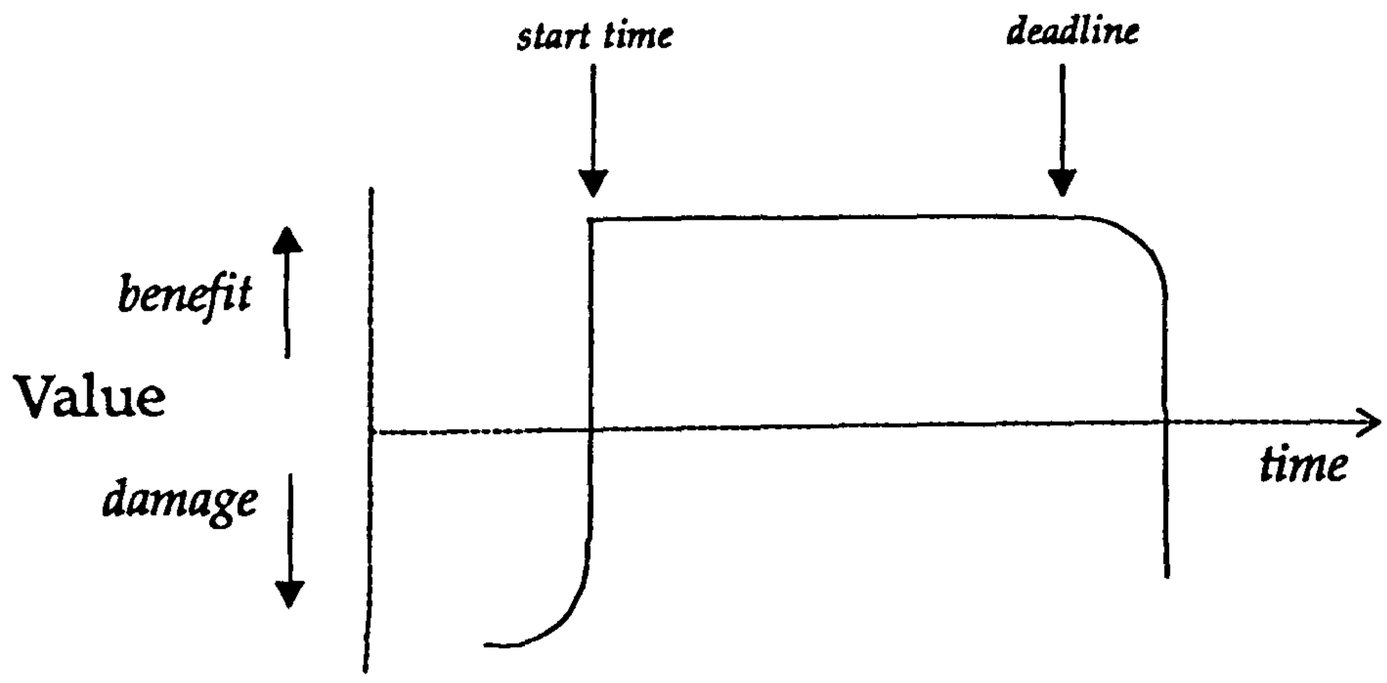


Figure 2.1: Example Value Function.

2.2.5 Summary

The three proposed forms of scheduling simplistic uniprocessor systems have been discussed. The offline costs of the approaches vary from high, in the case of graph-based scheduling; to low for dynamic scheduling. In contrast, the run-time costs are highest for dynamic priority scheduling, and lowest for graph-based and static priority scheduling.

All the scheduling approaches examined have associated feasibility tests, varying in accuracy and complexity. We observe that the sufficient and necessary tests are more accurate, but are also more computationally expensive than sufficient and not necessary tests.

2.3 Scheduling Realistic Uniprocessor Systems

When more realistic systems are considered, for example those that permit resource sharing amongst processes, additional problems are introduced. The following sections present a realistic system model, the problems introduced into scheduling by such a model, and the development of the scheduling approaches for simple uniprocessor systems.

2.3.1 Realistic Model

We extend the simple system model to permit:

(i) process computation times to be variable between a pre-calculable minimum and maximum (best-case and worst-case respectively);

(ii) sporadic processes with hard deadlines;

(iii) processes to access (potentially) shared resources in a mutually exclusive manner;

(iv) precedence relationships may be defined between processes, such that the execution of one process must wholly precede the execution of another process;

(v) I/O effects and other system overheads to be considered.

2.3.2 Problems Associated with the Realistic Model

The increased flexibility of the realistic model over the simple model increases scheduling complexity, both in terms of run-time support and offline feasibility. Two aspects of this more complex scheduling problem are now outlined.

Sporadic Processes

Hard real-time systems are

> *"inherently non-deterministic in nature."* [Jensen91]

Events do not necessarily occur periodically. To monitor such events requires either:

(i) periodic polling processes, or

(ii) sporadic processes.

The first option constrains application processes to view all events as periodic. This is clearly inefficient: to achieve a sufficiently short response time to events requires a polling process with small period, perhaps half the shortest inter-arrival time between successive events. This detracts from the feasibility of the system due to the extra computations of the polling process. Whilst sporadic processes provide a more natural method for providing fast response to sporadic events in an efficient manner, the problem of incorporating sporadic processes into the feasibility analysis is introduced.

Resources

When processes are permitted to access shared resources in a mutually exclusive manner, the potential for blocking is introduced: a process wishing to access a resource can be prevented from doing so by another process which has already locked that resource. The feasibility of such interacting processes depends upon the ability to bound potential blocking times of processes, which in turn depends upon the exact protocol used to control access to resources. Also, deadlock must be avoided. All these issues must be addressed when permitting processes to share resources.

Whilst many techniques have been proposed to cope with or avoid blocking, few are applicable to hard real-time systems. Some approaches are outlined in the following sections, whilst a more extensive survey of some of these approaches has been undertaken elsewhere [Audsley91a].

2.3.3 Developments of Offline Scheduling

Stankovic *et al* extend their graph-based approach to incorporate resources and precedence constraints between processes [Stankovic87]. Now, processes are only inserted into a slot in the schedule if all the resources they require are available for that slot, and if precedence constraints involving that process are not violated. The graph based approach of Koza and Fohler has been extended in a similar manner [Fohler89].

Xu and Parnas describe a successive approximation approach for generating static schedules [Xu90]. Initially a schedule is created based upon an earliest deadline ordering. Processes, or parts of processes, are then shuffled until all precedence constraints between processes are met, and no blocking can occur.

The developments of offline scheduling do not include sporadic processes or take account of I/O effects. The static nature of schedules created offline implies an implicit polling approach for sporadic events.

2.3.4 Developments of Static Priority Scheduling

For simple uniprocessor systems, it was noted that several optimal priority assignment schemes were available, depending upon the exact relationship between process deadlines and periods. When sporadic processes are introduced, this optimality is no longer apparent, except when sporadic processes execute at their maximum frequency. If this is not the case, an optimal priority assignment can only be achieved if it is known *a priori* the exact future release times of sporadic processes. Such clairvoyance is not usually apparent.

Feasibility is also affected by the incorporation of sporadic processes and resources. For the former, either the sporadic processes must be modelled to fit existing feasibility tests, or the feasibility test must be extended. Similarly, for resources, the tests must be extended to take account of potential process blocking.

The following sections review developments of static priority scheduling for sporadic processes, resources and I/O effects.

Sporadic Processes

To avoid the need for periodic polling processes various "bandwidth preserving" algorithms have been proposed. These specify a periodic server process to deal with sporadic events, with the server being able to preserve, within certain constraints, its unused computation time, if no events need servicing.

The priority exchange approach declares a periodic server for sporadic events [Lehoczky87, Sprunt88, Sprunt90]. When the server's period commences, it only executes if there are any outstanding sporadic events requiring servicing. If no such requests exist, priority exchange allows the high priority server to swap priorities with a lower priority periodic process. In this way, the server's priority decreases whilst maintaining execution time reserved

for sporadic events. In contrast, the deferrable server maintains server priority across its period, responding to sporadic events at a constant priority [Lehoczky87, Sha89]. Under both approaches, the computation time allowance for the server is replenished at the start of its period.

Whilst the priority at which sporadic events are serviced decreases within the server period under the priority exchange approach, the size of a priority exchange server (i.e. the computation time that can be assigned to it in each period) is greater than a deferrable server for a comparable system. Thus, more sporadic events can be serviced by the former.

The sporadic server combines the server size of the priority exchange approach with the constant priority of the deferrable server approach [Sha89]. The replenishment strategy of the sporadic server is as follows. Assume capacity c is consumed in $[t, t+c)$ then replenishment of c occurs at t'+T, where T is the period of the server, and t' is the latest time prior to t at which a process of lower priority than the server was executing or the processor was idle. In this way, the sporadic server increases the quantity of sporadic processes that can be serviced without lowering server priority.

The above approaches have a common problem, that of increasing system overheads, due to the existence of additional server processes. It is noted that the approaches were proposed to guarantee a minimum computation time for servicing aperiodic events. Hence, their motivation lies more with providing aperiodic (unguaranteed) processes with processing time, rather than guaranteeing sporadic process deadlines.

Resources

The requirement for shared resources to be accessed in a mutually exclusive manner creates some interesting problems. For example, consider two periodic processes which share a resource. Within a static priority system, the situation could arise where the *low* priority process has locked the resource and is pre-empted by a *high* priority process. The latter attempts to access the resource. *High* is now blocked by the lower priority process. A *medium* priority process pre-empts *low*. This is a form of *priority inversion* [Sha90]: the *high* priority process has to wait for *medium* to complete execution and *low* to finish its critical section before it can lock the resource and continue execution.

The problem of priority inversion can be avoided by the use of the Priority Inheritance Protocol (PIP) [Sha87a, Sha90]. This prescribes that if a higher priority process becomes blocked by a low priority process, the priority

of the former is inherited by the latter (allowing it to execute immediately) until the lock on the requested resource is released.

The PIP bounds the blocking time of process τ_i for each execution to a maximum of *min(j,k)* critical regions of lower priority processes, where *k* is the number of lower priority processes which are able to block τ_i, and *j* is the number of resources used by lower priority processes that can block τ_i. Effectively, the blocking is equal to the sum of the longest critical regions of each lower priority process.

The major disadvantage of the PIP is that deadlock can occur. For example, let τ_i require `resource1` and `resource2`. It obtains the lock on `resource1`, but before it can obtain the lock on `resource2`, it is pre-empted by τ_j which locks `resource2` and now requests `resource1`. Deadlock has occurred. The problem arises because a process is able to lock a free resource at any instant, irrespective of its priority relationship with other processes that will require that resource.

The Priority Ceiling Protocol (PCP) addresses the deadlock problem inherent in the PIP [Sha90]. This is achieved by ensuring that a strict ordering of critical region execution is maintained. The notion underpinning PCP is as follows. If one or more resources in the system are already locked, τ_i can only lock a resource if that resource, or any other locked resource in the system, is not accessed by a process with higher priority than τ_i. Thus, the priority of a process holding a resource is guaranteed to be higher than can be inherited by any pre-empted process.

The PCP can be summarised as:

- a priority ceiling is assigned to each resource equal to the highest priority of all processes that could lock it;
- a resource is allocated if the priority of the requesting process is strictly greater than the ceilings of all currently held resources. If the resource is not allocated, the requesting process becomes blocked upon that resource;
- a process executes at its assigned priority unless it blocks a higher priority process at which time it inherits the priority of the blocked process for the duration of the current critical region.

The maximum priority that a process can inherit whilst holding a resource is equal to the ceiling of that resource.

Deadlock avoidance is inherent in the PCP due to the strict priority ordering of critical region executions. A formal proof of this has been developed by Pilling *et al* [Pilling90].

The maximum blocking time τ_i is bounded to the longest critical region of a lower priority process that shares a resource with a process of equal or higher priority than τ_i. This occurs if a high priority process can be blocked for the entire duration of the critical region of a lower priority process. Effectively, the low priority process must lock a resource momentarily before the higher priority process becomes runnable. Clearly, in many cases, this will not occur, implying that worst-case blocking times are, in general, pessimistic.

Two main variations on the PCP have been proposed, the Semaphore Control [Rajkumar88a] and the Ceiling Semaphore Protocol [Rajkumar89]. The former provides a sufficient and necessary approach to resource allocation (with respect to approaches based upon the PIP) by ensuring that resources accesses denied by the PCP for reasons of possible deadlock prevention, are only denied if they will definitely lead to deadlock. The latter ensures that any blocking that a process receives will be at the beginning of its execution. Thus, once the process has actually commenced execution, it will run to completion without becoming blocked.

All the above approaches permit a process executing in a critical region to be pre-empted. The kernelised monitor prohibits such pre-emption [Dertouzos89]. However, the length of critical regions is required to be small as the blocking time that any process can endure is limited to the maximum length of any critical region. This approach requires programming and design discipline to keep critical regions small. If critical regions become large, then blocking times increase with associated loss of system feasibility. The kernelised monitor is valuable in systems where pre-emption costs are high compared to critical region execution times since system overheads can be minimised.

An alternative approach to allowing processes to block is to adopt a non-blocking run-time resource management scheme. One such approach is the Four Slot Mechanism, which prevents the reader and writer of a shared resource ever interfering with each other [Simpson90]. Two pairs of slots are provided for the shared data in the resource, one each for the reader and writer processes. They are accessed in such a manner that the writer process will never write to a slot currently being read; likewise, the reader will never read from a slot being updated by the writer process.

Obviously, this approach is deadlock free, with zero blocking times for processes. However, two problems exist. Firstly, time coherence of data may be violated. Secondly, a resource may only have a single writer and a single reader process at any one instance.

The non-blocking approach requires no extension to existing feasibility tests. However, the blocking approaches require slight modification of the tests, to guarantee the additional blocking time of a process before its deadline. Thus, the test must ensure that for all $\tau_i \in \Delta$, C_i+B_i (where B_i is the blocking time) be guaranteed before D_i. This applies to both $D_i=T_i$ and $D_i \leq T_i$ systems. Such extensions to the $D_i=T_i$ feasibility tests have been proposed by Sha *et al* [Sha90].

I/O Effects

Studies performed to determine how I/O affects static priority scheduling draw the following conclusions [Sha87b, Rajkumar87, Davari92]:

(i) FIFO I/O schemes are not applicable to real-time systems due to the potential of high priority process I/O being blocked by low priority I/O so forming a priority inversion;

(ii) dynamically ordered queues can lead to fairness problems;

(iii) DMA can steal bus cycles from the processor and therefore from application processes.

2.3.5 Developments of Dynamic Priority Scheduling

The earliest deadline approach has been developed to provide time for sporadic and aperiodic processes. Chetto *et al* note that the normal earliest deadline formulation runs periodic processes as soon as possible [Chetto89]. This has the effect of postponing idle time for as long as possible - it is this idle time that is used for servicing sporadic events. When these occur, the scheduler switches to a variation of the earliest deadline approach which runs guaranteed periodic processes as late as possible (the latest start time of each process has been pre-calculated offline). This is proved to provide the maximum time for processing sporadic processes [Chetto89].

Whilst the non-blocking approach to resources outlined by Simpson [Simpson90] is applicable to dynamic priority scheduling, two proposed variations of the Priority Ceiling Protocol have been proposed that provide solutions to any potential blocking. The Dynamic Priority Ceiling Protocol re-defines the Priority Ceiling Protocol in terms of dynamic priorities [Chen90].

This is achieved by re-evaluating the inherent priority ordering between processes (i.e. the process with the closest deadline is assigned the highest priority at any time) and also the ceiling priorities of all resources. This approach is relatively expensive, requiring resource ceiling priorities to be re-evaluated on every process release or completion.

The Stack Resource Policy defines a dynamic priority and static pre-emption level for each process [Baker90]. The latter is a measure of how processes can pre-empt each other. For example, a process with a low pre-emption level may not pre-empt a process with a high pre-emption level. For earliest deadline scheduling, pre-emption levels are assigned according to the deadline of the process: the process with the shortest deadline is assigned the highest pre-emption level; the process with the longest deadline is assigned the lowest level. The ceiling priorities of resources are defined in terms of these pre-emption levels and are therefore static. Hence, this approach does not suffer from the run-time overhead of ceiling priority re-evaluation incurred by the Dynamic Priority Ceiling Protocol. It is noted that both the Dynamic Priority Ceiling Protocol and the Stack Resource Policy assume that all processes $\tau_i \in \Delta$ have $D_i=T_i$.

2.3.6 Summary

Due to the additional complexity of the realistic model, sub-optimal scheduling schemes must be used. Such schemes include developments in offline, static priority and dynamic priority scheduling. All three have been extended to include shared resources. This requires expanded feasibility tests for each scheduling approach and the provision of resource allocation protocols for controlling access to shared resources at run-time.

Sporadic processes are not explicitly catered for by the offline scheduling techniques examined, although they could be incorporated via a periodic polling process. This is less efficient than the servers developed for static priority scheduling, although the latter incur a run-time overhead due to their complexity. Dynamic priority scheduling can incorporate sporadic processes by running periodic processes as late as possible when the former process needs to execute. However, the analysis for this approach does not guarantee sporadic process deadlines.

Whilst the literature indicates how to include system overheads (i.e. context switch) into the static priority feasibility analysis, no such theory has been proposed for offline or dynamic priority scheduling.

2.4 Scheduling Multiprocessor Realistic Systems

Whilst multiprocessor and distributed system scheduling is beyond the scope of this thesis, for completeness, a number of key results are given.

In the previous section, proposed approaches for offline scheduling were outlined. These have been further extended for multiprocessor systems. The MARS project initialise one empty schedule per processor [Kopetz85, Damm89, Kopetz89]. Successively, individual processes are placed into one of the schedules such that deadline, resource and precedence constraints are not violated. A similar approach is taken by the Spring project [Stankovic87b, Stankovic89, Stankovic91]. An extension to the approach permits a process to move to another processor if its deadline can be guaranteed at the destination processor [Cheng85].

Both the above offline approaches treat the allocation of processes to processors as part of the offline creation of schedules. An alternative strategy is adopted by static priority scheduling. Here, allocation is viewed as a separate issue, such that the processes are partitioned into sets, one per processor, with each set then tested individually using the feasibility tests developed for uniprocessor systems. This approach is taken by the ARTs kernel [Tokuda89, Tokuda91] and the DrTee kernel [Audsley91b].

Whilst offline approaches place processes into a schedule so that resource blocking is avoided, the static priority scheduling approach must account for local and remote blocking. The latter occurs when a process requests a locked remote resource, or a local resource that has been locked by a remote process. Unfortunately, the Priority Ceiling Protocol does not translate easily to a multiprocessor environment. The Multiprocessor Priority Ceiling Protocol bounds blocking time to a function of the critical regions of other processes by forbidding any process to execute outside a critical region whilst other processes on the processor are blocked [Rajkumar88b]. Also, all resources that are shared between processes resident on different processors are allocated to a single synchronisation processor where all critical regions associated with those resources are executed. The Generalised Priority Ceiling Protocol multiprocessor [Rajkumar88b] is a further development that allows shared resources to be resident on any processor.

2.5 Availability of Run-time Flexibility

The scheduling approaches outlined in the previous sections have enabled the deadlines of processes in realistic systems to be guaranteed offline, via

associated feasibility tests. Whilst this provides 100% predictability at run-time, it is clear that at run-time, system resources will be under-utilised due to processes executing for less than their worst-case execution times; or due to sporadic processes not being released at their maximum frequency. Potentially, this spare time could be used to provide additional flexibility.

The identification of spare capacity has been discussed by Haban *et al* [Haban89, Haban90]. The code of individual processes is decomposed into a chain of basic code blocks, such that a block has a single entry and a single exit point. Software triggers are inserted between blocks so that a hardware monitor can measure precise execution time. Hence, at trigger points, the spare capacity generated by the previous block can be calculated. The spare capacity is used to enable the schedule to be revised to reduce the number of deadlines missed.

A similar approach has been proposed by Dix *et al* [Dix89]. The *milestone* is a software trigger inserted by the programmer to signal the scheduler when it reaches a point in its computation such that the process is certain of its remaining computation requirement. This enables improved scheduling. For example, a high priority process that has declared that it requires 1 unit of CPU time before its deadline in 2 units, can be stopped to allow a lower priority process requiring 1 unit before a deadline in 1 unit. Thus, both deadlines can be met. This approach is non-systematic, relying upon the programmer to insert milestones.

Neither of the above approaches consider either sporadic processes occurring at less than their worst-case arrival rate, unrequired blocking time or unrequired resources. Unrequired resources have been considered by Shen *et al* within the context of a multiprocessor architecture with shared memory between processors [Shen89]. All resources can be accessed on each processor with all processors sharing a common list of processes to run: an idle processor runs the next runnable process on the list. A process becomes runnable if all its required resources are available. Resources are reclaimed from running processes if no longer required. The problem considered by Shen is that of processes executing early due to resources becoming free earlier than anticipated by offline analysis. This can actually cause deadline failure (by Graham's anomalies [Graham69]). A number of conditions are developed to prevent this. The problems solved by this work are only encountered on multiprocessor architectures with a central shared queue of processes to schedule.

The literature does not address the lower-level problems of re-assigning spare capacity generated by one process to another, without inducing the possibility of a deadline being missed. Also, the effective assignment of spare capacity by the scheduler to multiple requesting processes is not addressed. These issues are considered in Chapters 6 and 7.

2.6 Summary

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 shared resources, process precedence constraints and arbitrary process timing constraints were not initially considered.

Due to these complexity considerations, sub-optimal scheduling schemes have been proposed for the feasibility analysis of realistic hard real-time systems, including the development of both cyclic and static priority scheduling. Together, these indicate the possibility of:

- guaranteeing both periodic and sporadic hard real-time processes on the same processor;
- utilisation of spare time by non-critical processes;
- process blocking permitted;
- precedence constraints between processes.

Some areas of scheduling theory were identified as pessimistic. For example, worst-case blocking calculations for the family of resource control protocols derived from priority inheritance. Also, existing analyses for basic realistic feasibility are sufficient but not necessary. Increasing the accuracy of these analyses will increase the number of process systems declared feasible. However, it was seen that reducing forms of feasibility pessimism, in general, increases the complexity of the feasibility test, and therefore the time required to perform that test. Increasing the efficiency of feasibility tests is therefore also important.

The detection of spare capacity at run-time occurs at the completion of a process's execution or at points in a process's computation where the actual computation time may be less than the worst-case computation time (from the start of the process). In the literature, the re-use of spare capacity is constrained to allowing the execution of soft real-time processes to occur.

Chapter 3.
An Approach For Obtaining Flexibility in Hard Real-Time Systems

For hard real-time systems, where failure is costly in terms of life and other resources, it is imperative to show that the system is predictably safe within the scope of a given failure model. Arguments regarding predictability must be made offline: there is little merit in the post-mortem approach of showing a system to be unpredictable after failure has occurred. In hard real-time systems, the meeting of the timing requirements of processes is a major factor in determining the predictability of the system. Whilst recognising that meeting these timing requirements embodies the complete software engineering life-cycle from specification to implementation, it is the latter stage that actually determines whether they will be met during the lifetime of the system. Within the implementation, the role of the scheduler is crucial. The decisions made at run-time regarding the sequencing and interleaving of process executions directly affect whether the system will meet timing requirements and thus achieve desired predictability.

In Chapter 2 it was seen that current scheduling theory places many constraints upon the form of process, and the interactions between processes at run-time. Also feasibility theory was observed to be pessimistic for such process models. Thus, applications whose processes may actually meet all deadlines at run-time could be declared infeasible by offline analysis.

The next generation of hard real-time systems are expected to include long lifetime applications which are required to adapt dynamically to failure, overload and re-configuration [Stankovic88]. Processes may be required to execute in degraded states in order to occupy less processor capacity in the event of total processor capacity being reduced, for example by failure or overload. Applications may require a process to be guaranteed to reach a pre-defined minimum accuracy of result, but allow additional computation to use spare processor capacity to improve that result. The implementation of such systems requires more flexibility than is available using current scheduling theory. Any increase in flexibility must itself be constrained to ensure that guarantees afforded offline to processes with hard deadlines are not

compromised: a trade-off is observed between increasing system flexibility and decreasing the predictability of the resultant system.

In the following sections, the flexibility required for the next generation of hard real-time systems is examined and compared with that supported by current scheduling theory. From this discussion, a twin layered model for flexible scheduling is introduced, consisting of offline guarantees and run-time re-use of spare capacity. The provision of such flexibility needs to be balanced against the predictability and analysability of the resultant system. Such trade-offs are examined.

3.1 Characteristics of Next Generation Hard Real-Time Systems

The typical applications for today's real time systems are in the command and control domain, e.g. flight control, aircraft avionics, industrial plants. The next generation of real-time systems will be more complex, although broadly in the same command and control area, e.g. space station, undersea exploration, intelligent manufacturing. The additional complexity of these systems arises from the need to incorporate [Stankovic88]:

- distribution;
- dynamic and adaptive behaviour;
- long lifetime components;
- critical components.

The motivating factors behind moving towards next generation real-time systems are the advance in hardware, particularly in distributed systems, and the need for adaptive and intelligent run-time behaviour. However, theory, in particular feasibility analysis, has not kept pace with these developments (see Chapter 2). For example, incorporation of application components with unbounded timing characteristics (e.g. AI), to improve the results of critical processes, is not possible using cyclic scheduling alone.

Given the additional complexity of these systems, it has been argued that only a small proportion of the processes will be safety critical, requiring 100% predictability in terms of meeting their deadlines [Stankovic90]. This permits offline guarantees to be afforded to a small number of process deadlines, with run-time scheduling concentrating mainly upon increasing the number of other processes meeting their deadlines (via run-time scheduling heuristics). In contrast, Burns and Wellings argue that mission critical and other essential processes need to be afforded 100% predictability [Burns91b].

Thus, the proportion of guaranteed (crucial) processes is larger. The Integrated Modular Avionics system being developed for civil aircraft flight control is given as a supporting example [AEEC91]. This view is also supported by Xu and Parnas [Xu91] citing the U.S. Navy's A-7E aircraft flight control software as an example [Faulk88].

The provision of adaptive process behaviour is key to the next generation. This can be viewed in many forms: processes changing operating modes between distinct phases in the lifetime of an application; processes providing degraded but guaranteed functionality during overload or failure; processes using different (possibly unbounded) additional components to improve their benefit to the system. The overriding requirement is that spare processor capacity should not be exhausted by the executions of soft real-time processes, as in many scheduling schemes (e.g. see [Tokuda89]), but also to increase the benefit of guaranteed processes to the system:

> *"where the system is predominantly concerned with crucial [i.e. guaranteed] activity then it is these services that should benefit from this extra [processor] capacity."* [Burns91b]

This viewpoint is supported by the Imprecise Computation model [Lin87, Chung90], in which spare processor capacity is used to improve the accuracy of result of a guaranteed basic service.

Specific requirements placed upon feasibility theory by the next generation of hard real-time systems are now examined.

Periodic and Sporadic Processes

The relative proportion of periodic and sporadic processes is unclear, varying greatly between applications. In general, the need to provide end-to-end deadline guarantees across interdependent processes (possibly distributed), some of which are initiated by external events, requires that both periodic and sporadic processes are supported. Given stringent timing constraints upon processes, it is unlikely that polling for sporadic environment events is efficient enough. Hence, direct support is required for sporadic processes.

Process Deadlines Unequal To Their Periods

Permitting process deadlines to be less than their respective periods represents a more natural method for representing real-time processes. For example, processes often take the form:

```
read input; calculate response; output result
```

The release of the process occurs when the input data is available to be read. The deadline of the process is naturally identified with the completion of the output. A finite amount of time is left between the output phase of one execution, and the input phase of the next. This ensures that, for example, hardware activators have been set to their updated value.

This process structure is also observed when messages are passed between nodes in a distributed system. Here, the deadline of the process may represent the point at which a process sends a message to a process on another node, with the interval between the deadline and the period of the process used for message transmission. The same effect is seen in precedence-constrained processes where one process must complete before its successor commences. The deadline of the former process is set to be less than its period in order to let the successor process execute (assuming both processes share a common period).

Arbitrary Process Start Times

The ability to have arbitrary process start times, relaxing the restriction that all processes have (during the lifetime of the system) a common release time, enables a more natural representation of application problems. Consider a system which has two networks (for fault-tolerance purposes) using a token passing protocol. To even out the work load on a node, the arrival times of the tokens from the two networks are phased so that they never arrive simultaneously (assuming constant token rotation time). This is naturally represented by two network device driver processes, one for each network, with equal periods and deadlines, but whose release times are offset from each other. Another example concerns (non-trivial) device accesses. One process may request data from a device, with a second process, offset from the first, retrieving the data. The offset between the two processes relates to the response time of the device to the request.

A precedence constrained set of processes can be modelled by assigning arbitrary start times to processes. Later processes in the precedence constraint are given an offset such that earlier processes are guaranteed to finish before the later process is released. For example, processes of the form:

```
read input; calculate response; output result
```

can be split into three precedence constrained processes (one for each phase), where the offset of the input phase is less than the offset of the computation

phase, which is in turn less than the offset of the output phase (the three phases have identical periods).

Process Interaction Via Shared Resources and Precedence Constraints

Increasing flexibility of inter-process interaction can be achieved in many ways. At a scheduling level, most approaches can be supported by the provision of shared resources which may be accessed in either a non-blocking or blocking manner (the latter for mutually exclusive access); and precedence constraints between processes.

Although non-blocking access may be provided to some resources, others must be accessed in a mutually exclusive manner. For example, access to some low-level devices must be mutually exclusive to ensure non-corruption of data returned from the device. Also, if the context switch time in a system is significant, this will form a blocking factor for all processes.

Incorporation of precedence constraints leads to greater flexibility in the process model: multi-deadline processes can be represented. These are used when processes have several time constrained goals to meet within a single execution. For feasibility purposes, the process may be decomposed to represent its multiple goals.

Re-Use of Spare System Capacity

Offline feasibility analysis considers worst-case estimations of processor (and other resource) requirements. Therefore, at run-time, a degree of spare system capacity will become evident. Such spare capacity can be re-used to enhance the functionality of system processes. For example, spare capacity could be utilised to enable unbounded software components to be executed at run-time, that is components which could not be afforded offline guarantees [Audsley93a].

3.1.1 Summary

In summary, the demands of next generation hard real-time systems on feasibility theory include:

- large proportion of crucial processes;
- guaranteed periodic and sporadic processes;
- process deadlines being not necessarily equal to periods;

- arbitrary process start times;
- shared resources accessed in a mutually exclusive or non-blocking manner;
- precedence constraints between processes;
- run-time ability for guaranteed processes to re-use any available spare processor capacity.

Note that issues of distribution are largely ignored within this thesis.

It is clear from the review in Chapter 2 that many of the scheduling requirements of next generation hard real-time systems are not supported by current scheduling theory. Whilst cyclic, static priority and dynamic priority scheduling can all afford 100% predictability to limited forms of process, little attention is given to run-time use of spare capacity by guaranteed processes.

3.2 An Approach For Introducing Additional Flexibility

In the previous section the requirements for the next generation of hard real-time systems were outlined. From the observations made in Chapter 2, it is clear that current scheduling approaches are too inflexible for such systems. Therefore, in this section, a two-tiered approach is introduced to increase flexibility:

(i) basic offline guarantee of all crucial process deadlines;

(ii) run-time re-use of spare processor capacity to increase system utility.

Tier (i) is the most significant, since 100% crucial process predictability is of primary importance. Tier (ii) provides the ability to cope with process models that permit crucial (and other) application processes to contain optional components for increasing overall system utility.

3.2.1 Increasing Offline Flexibility

To increase offline flexibility, feasibility tests need to cope with all (or as many as possible) of the requirements of complex hard real-time systems outlined in the previous sections: the coverage of feasibility tests must be improved. Also, the efficiency of algorithms used to test feasibility should be improved.

Additionally, feasibility tests must be made more accurate, to reduce pessimism in worst-case bounds on process execution and blocking times; and to reduce the number of process sets declared infeasible, which would actually meet all deadlines at run-time. For example, the conventional approach towards finding worst-case execution times scans the control-flow graph of the

process noting the longest path (in terms of execution time) [Puschner89]. As a by-product, the required resources on each path are found, with the worst-case blocking requirement of processes now calculated. However, the worst-case path through code with respect to execution time and resource usage may be different: it may be impossible for both worst-cases to occur within a single execution of the process. Thus, the worst-case estimations are pessimistic.

To form the basis for increasing offline flexibility, static priority pre-emptive scheduling is chosen. The reasons for this choice include:

(i) much feasibility theory already exists;
(ii) the theory is extensible and adaptable;
(iii) it has simple run-time requirements;
(iv) it is predictable in overload situations;
(v) it is easy to incorporate detection and re-use of spare processor capacity.

The availability of static priority feasibility theory for basic timing characteristics (such as process period equal to deadline) and resource sharing (via semaphores) provides an adequate basis for the development of feasibility theory. Such analysis can be extended without having to start from first principles.

Static priority pre-emptive scheduling is simple to implement at run-time, even allowing for the complexity of resource allocation protocols (such as priority ceiling protocol). This is due to not having to re-calculate priorities dynamically. The simple static allocation of priorities enables acceptable behaviour during a transient overload. For example, during an overload, processes will miss their deadlines in a predicable manner: in general, lower priority processes will miss their deadlines before higher priority processes.

Dynamic priority scheduling, for example earliest deadline, is discounted since although it often leads to (potentially) greater processor utilisation, it is unpredictable in overload situations: the process that misses its deadline first during an overload cannot, in general, be determined offline.

Cyclic processor scheduling is not employed because it leads to systems that are inflexible at run-time. Xu and Parnas argue that

> *"For satisfying timing constraints in hard real-time systems, predictability of the system's behaviour is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system."* [Xu91]

However, by extending the coverage of static priority feasibility tests, it is possible to match or exceed the coverage of feasibility tests for cyclic systems [Audsley93b]. We note that, even with the same coverage, a cyclic schedule may exist for a process set declared infeasible by a static priority test: in this case, the cyclic test is more accurate. However, in the general case, the flexibility that is lost, particularly at run-time, accounts for any gain in accuracy.

Other disadvantages of cyclic scheduling compared to static priority scheduling have been articulated by Locke [Locke92], who intimates that the key disadvantage is fragility: it is difficult to change process code or other timing characteristics during design and/or maintenance, without having to re-engineer a schedule (c.f. static priority scheduling which only requires a re-calculation of feasibility). We note that simple run-time implementation is often cited as an advantage of cyclic scheduling over static priority. Both schemes require an interrupt handler for a clock, and an ordered list of processes to execute. However, the cost of explicitly handling resource allocation at run-time adds to the cost of static priority scheduling. In reality, the run-time overhead of static priority scheduling is not significantly greater than cyclic scheduling.

Extending offline flexibility is considered further in Chapters 4 and 5.

3.2.2 Increasing Run-Time Flexibility

Guaranteeing process deadlines offline leads to an under utilisation of resources at run-time. This arises from [Audsley93a]:

(i) software components not taking their worst-case execution time;

(ii) hardware behaving better than expected (due to pipelines, caches etc);

(iii) sporadic processes not executing at their maximum rate;

(iv) non-execution of error handling software (i.e. recovery blocks, exception handlers);

(v) spare time incorporated by feasibility analysis to guarantee hard deadlines.

At run-time, this under utilisation is evident as spare capacity. In some systems, this capacity is not explicitly detected, with processes merely completing earlier than anticipated (e.g. time share operating systems). Other systems allow soft real-time (or non-real time) processes to utilise the spare capacity [Tokuda89]. In section 3.1 it was argued that crucial processes should

be permitted to use some (or all) of the spare capacity to increase their benefit to the system. As indicated in section 2.5, little work is evident in the literature regarding the re-use of spare capacity. Any approach must:

(i) characterise available spare capacity when it occurs;

(ii) dynamically identify spare capacity at run-time;

(iii) consider viable policies for re-assigning spare capacity.

Characterising spare capacity reflects the need to distinguish between its various forms. For example, between guaranteed execution time not required by a process, and that time not guaranteed to any process offline. Also, mechanisms are required to enable automatic detection of spare capacity without creating unnecessary overheads. Finally, re-assignment of spare capacity must not violate already guaranteed deadlines. This could occur, for example, if spare capacity were assigned to a process which proceeded to lock a resource guaranteed to be available during another guaranteed process's execution.

It has been argued that the requirement to provide 100% predictability regarding crucial process deadlines implies the use of a simplistic run-time scheduler, as alluded to by Stankovic and Ramamritham [Stankovic90] and Damm *et al* [Damm89]. This argument would preclude the detection and re-use of spare processor capacity in hard real-time systems as run-time complexity and overheads would be increased. However, if 100% predictability can still be afforded to crucial processes in spite of this additional complexity, the flexibility gained out-weighs the loss of simplicity.

The approach adopted in this thesis is to ensure that whilst detection of spare processor capacity is an overhead slightly extending the computation time of a process, any algorithm employed as part of the run-time scheduler to re-allocate spare capacity is itself performed using spare capacity. This illustrates the two-tied approach: if no spare capacity exists, the run-time scheduling is simple static priority pre-emptive; if spare capacity does exist, potentially complex algorithms may be executed using part of it to allocate the remaining spare capacity.

The detection and re-use of spare capacity is considered further in Chapters 6 and 7.

3.2.3 Summary

A two-tiered approach for increasing hard real-time system flexibility, with respect to feasibility theory, has been identified. Offline flexibility can be

improved by extending the coverage, accuracy and efficiency of feasibility analysis. Run-time flexibility can be increased by firstly detecting spare processor capacity, and then employing spare capacity re-assignment policies that do not violate offline guarantees regarding process deadlines.

Essentially, any extensions of feasibility theory, that permit more flexible use of the processor, must adhere to the fundamental 100% predictability requirement of crucial processes in hard real-time systems.

3.3 The Complexity / Flexibility Trade-Off

Increasing the flexibility of scheduling to reflect the requirements of next generation hard real-time systems has profound effects on the complexities of both the offline feasibility test required and the scheduling algorithm employed at run-time. As offline flexibility increases, so does the complexity of the required feasibility test. Hence, a trade-off exists between flexibility and ease of feasibility testing. A trade-off is also observed between the complexity of run-time scheduling and the overheads incurred: the amount of spare processor capacity that could be re-used decreases as overheads increase due to the additional complexity.

In the following sections, the trade-offs outlined above are discussed.

3.3.1 The Offline Complexity / Flexibility Trade-Off

In general, as the flexibility of the process model is increased, the ease of determining feasibility decreases since the complexity of the required feasibility test increases (see Chapter 2). As optimal solutions are now NP-complete (or NP-hard), sub-optimal (i.e. sufficient and not necessary) feasibility tests are employed. The coverage of such tests needs to reflect the requirements of next generation hard real-time systems whilst remaining efficient (i.e. tractable). The tests must have enough accuracy to detect an adequate proportion of feasible process sets.

The trade-off between accuracy and coverage (i.e. flexibility) on the one hand and the ease of determining feasibility on the other is illustrated in Figure 3.1.

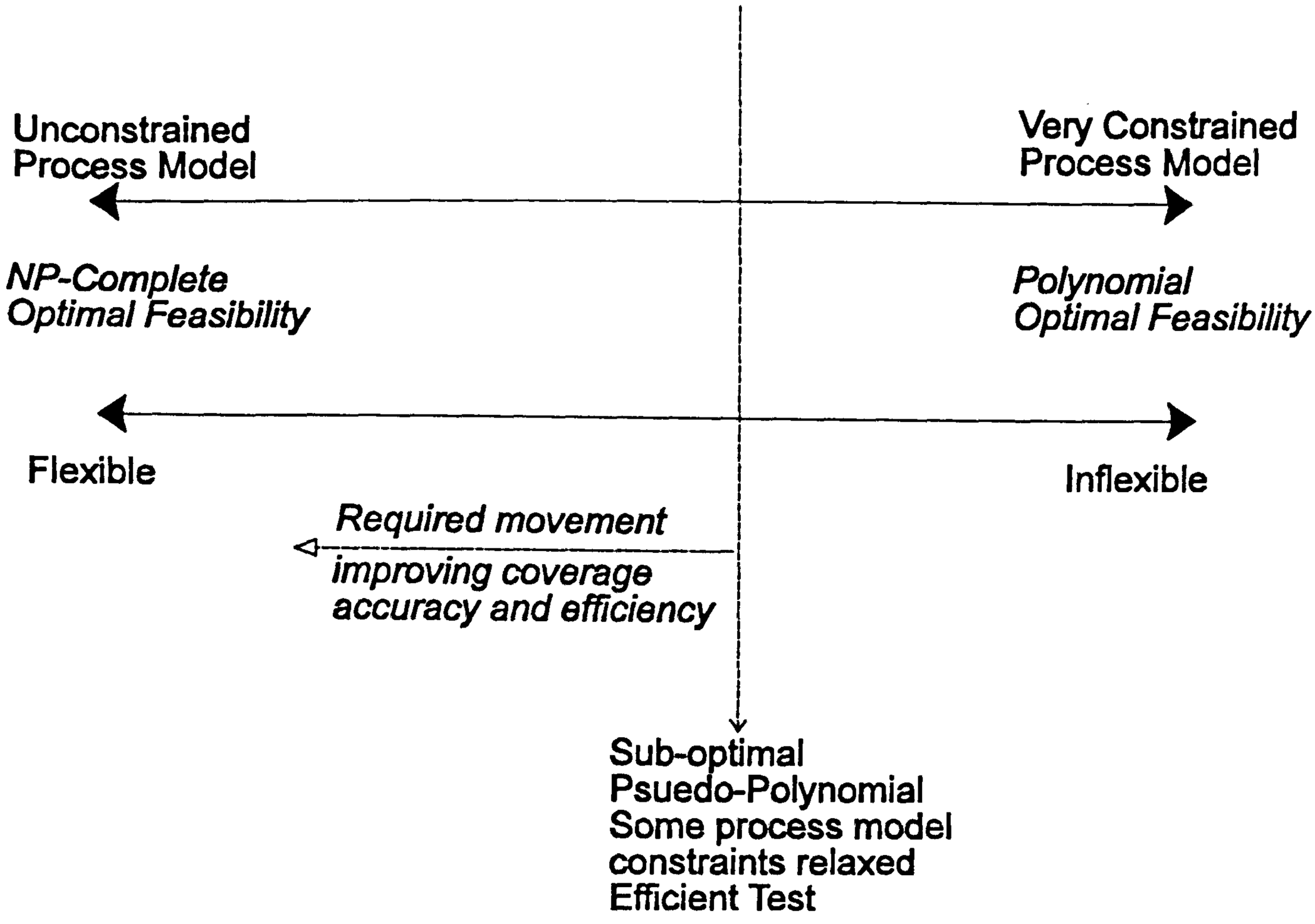


Figure 3.1: The Offline Complexity / Flexibility Trade-Off.

Flexibility is increased as the constraints upon the process model are relaxed and/or the accuracy of the feasibility test increases. The requirement is to make applicable sub-optimal tests more flexible by increasing coverage, accuracy and efficiency (whilst not losing any process sets previously declared feasible).

This trade-off is observed when the feasibility of realistic applications is considered. Assumptions made regarding zero length context switches (and other system overheads), and no inter-process interaction (usually some blocking when accessing kernel) are often not adequate. Therefore, feasibility theory should be extended (more coverage) whilst maintaining current accuracy as much as possible (i.e. attempt to ensure that process sets that passed previously will still pass).

3.3.2 The Run-Time Complexity / Flexibility Trade-Off

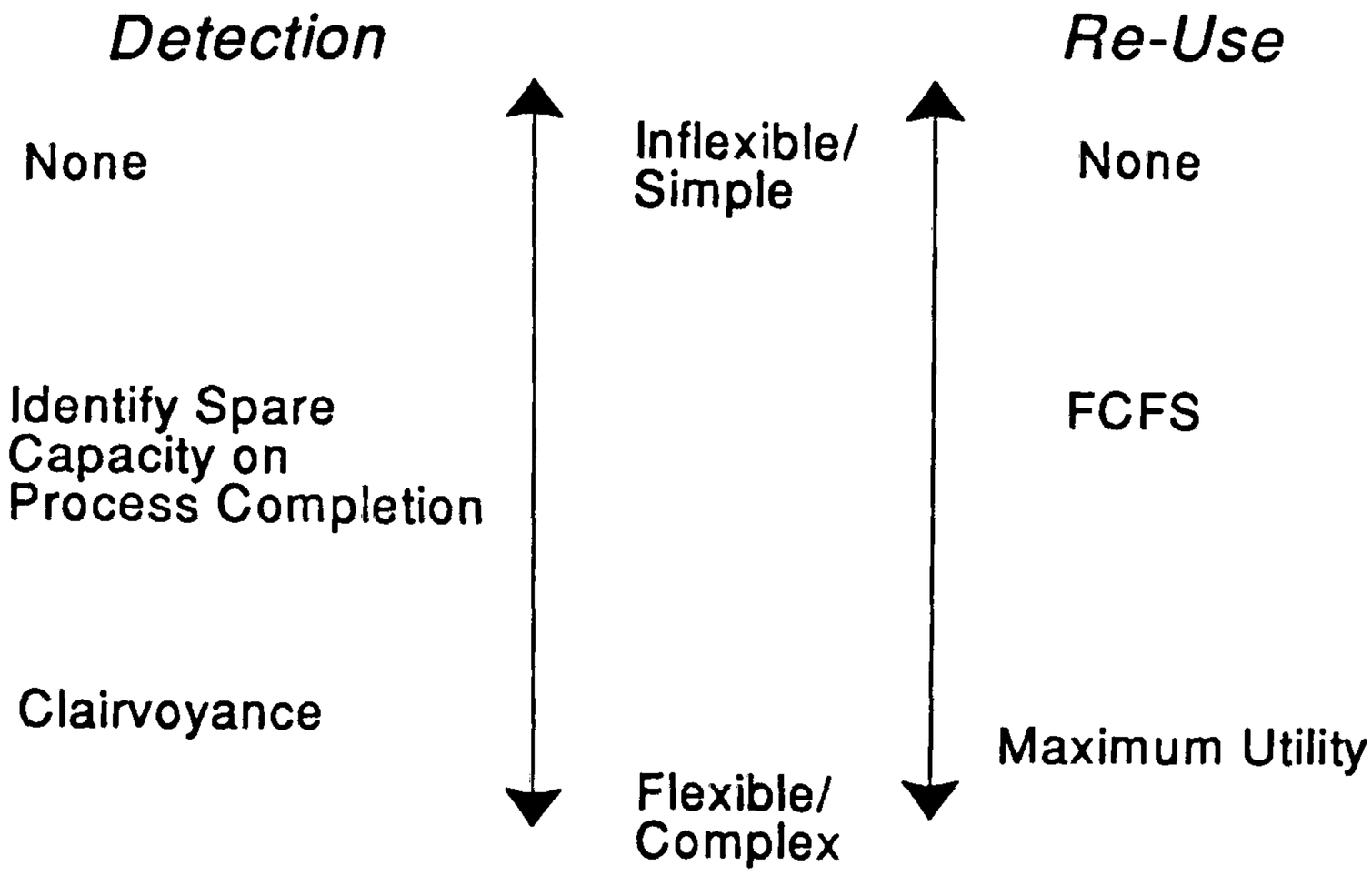


Figure 3.2: The Run-Time Complexity / Flexibility Trade-Off.

The amount of spare processor capacity that is available at run-time is dependent upon the exact run-time scheduling scheme assumed by the feasibility test (and any further assumptions therein). For example, different feasibility tests may require processes to be executed in different orders at run-time: since the actual computation time of the processes may be dependent upon current (real) time, the amount of spare processor capacity may vary in quantity under different scheduling schemes. Also, the actual time at which any spare processor capacity becomes apparent may differ. Therefore, the comparison of different run-time schemes for detecting and re-using spare capacity should be within the context of the same offline feasibility test and assumed basic run-time scheduling regime (e.g. static priority pre-emptive).

Assuming that no explicit detection of spare capacity at run-time has zero complexity (i.e. is simple) and incurs no system overheads, we may observe that as detection complexity increases, so does that amount of spare capacity detected. Also, the associated run-time overheads increase. Hence, as complexity increases, a point may be reached whereby additional complexity implies that less spare capacity will be detected, due to the size of overheads incurred. This is illustrated in Figure 3.2. As complexity of detection increases, for example from no explicit detection to identifying spare capacity on process completion, flexibility and overheads increase.

For re-use of spare processor capacity, the null run-time policy (i.e. do nothing) has no complexity and incurs no system overheads. As the run-time

re-use policy becomes more complex, more flexibility is introduced, although at increased run-time cost. For example, assigning spare capacity on a First-Come-First-Served basis amongst requesting processes is a more flexible policy than no re-use at all, although it incurs increased overheads. As the policy approaches the complexity of attempting to gain the most utility (e.g. via Value-Functions [Locke86]) from available spare capacity, complexity and overheads increase further. These trade-offs are illustrated in Figure 3.2.

We note that detection and re-use are interdependent in that the ability to re-use spare capacity is entirely dependent upon its previous detection: it is little use to employ complex run-time re-use policies if detection is very poor.

Since schemes for detection and re-assignment add to system overheads at run-time, the feasibility of crucial processes could be affected due to the fall in available processor utilisation for guaranteeing deadlines. Therefore, if possible, the detection and re-use policies should occupy spare processor capacity themselves.

3.4 Summary

Existing feasibility theory is not sufficiently flexible to cope with the requirements of the next generation of hard real-time systems. In particular, inadequacies are highlighted by the need for more relaxed process timing characteristics and adaptive run-time behaviour.

Increased flexibility can be achieved in two areas. Offline feasibility analysis can be extended, so that the coverage, accuracy and efficiency of any test is adequate for the features of next generation systems. The coverage of feasibility tests must reflect the requirements of such systems, whilst being sufficiently accurate to detect a large proportion of feasible process sets.

Run-time flexibility can be improved by the efficient determination and re-use of spare capacity by crucial (and other) application processes, to improve the utility of the system.

These observations lead to a two-tiered approach toward increasing the flexibility of hard real-time systems. Primarily, offline guarantees for crucial process deadlines are provided using static priority pre-emptive scheduling. Then, at run-time, spare processor capacity is re-used to increase system utility. This is achieved by accurate detection of spare capacity (based upon an offline static code analysis), combined with efficient policies for allocation to crucial processes.

Chapter 4.
Extending Offline Flexibility Via Deadline Monotonic Feasibility Analysis

In the initial chapters of this thesis it has been observed that current feasibility analysis for static priority systems lacked the coverage required for the next generation of hard real-time systems. Also, the available feasibility analysis is overly pessimistic, often assuming worst-case scenarios that can never occur at run-time. Thus system flexibility is constrained.

The aim of this chapter is to examine the relationship between flexibility and complexity with respect to offline feasibility analysis. Additional flexibility is introduced by extending the coverage of feasibility analysis to encompass processes whose deadlines are no greater than their periods.

In the literature, few sufficient and not necessary feasibility tests for processes with $D_i \leq T_i$ have been articulated. This chapter develops several such tests, with differing accuracies.

As observed in section 2.2.3, sufficient and necessary feasibility tests are available for processes with $D_i = T_i$. For example, for the purpose of determining feasibility only, all process periods could be reduced to be equal to the deadline (establishing $D_i = T_i$). This assumes a higher workload on the processor than is actually encountered at run-time. Under these circumstances, a sufficient and necessary $D_i = T_i$ feasibility test would become, in general, sufficient and not necessary for $D_i \leq T_i$. Exact feasibility tests derived for $D_i \leq T_i$ processes include:

- construction of a schedule over the interval $\left[0, \max_{1 \leq i \leq n}(D_i)\right)$ [Leung80];
- extension of the $D_i = T_i$ test given by Lehoczky *et al* [Lehoczky89] for $D_i \leq T_i$ processes [Nassor91];
- test based upon interval mathematics [Joseph86].

In general, these tests all attempt to determine the existence of a point between a process's release and deadline such that all its computational requirement has been met. The tests vary in their derivation, and also in

efficiency when implemented. The tests consider independent periodic processes only: issues such as the incorporation of sporadic processes and shared resources are not considered. All three tests assume deadline monotonic priority assignment (optimal for $D_i \leq T_i$ processes) [Leung82].

Within this chapter, more efficient sufficient and necessary feasibility tests are derived that also facilitate the introduction of sporadic processes and process blocking.

Initially, some assumptions are made:

(i) all processes are periodic;

(ii) process computation times are bounded and known offline;

(iii) all processes have a common start time (i.e. for all processes τ_i, $O_i = 0$).

(iv) processes do not interact (via shared resources or precedence constraints);

(v) processes cannot voluntarily suspend, or become blocked by an external event (e.g. reception of data from an external source).

During the course of the chapter, several of the above restrictions will be lifted.

The following section introduces background analysis of the run-time behaviour of $D_i \leq T_i$ processes. Section 4.2 develops several sufficient and not necessary feasibility tests of differing accuracy and complexities. Section 4.3 develops sufficient and necessary feasibility tests. Sections 4.4 and 4.5 extend the feasibility tests for sporadic processes and process that may block on shared resources respectively. Section 4.6 discusses issues related with determining infeasibility, presenting appropriate tests. Section 4.7 examines the accuracy and efficiency of the developed tests using randomly generated process sets. Finally, a summary of the chapter is given in section 4.8.

4.1 Feasibility Analysis of $D_i \leq T_i$ Processes: Background

Initially, it is observed that under static priority scheduling, at run-time, only processes with higher priority may affect the execution of a process by pre-emption. Also, the worst-case for meeting the deadline of a process is for a release starting at a *critical instant*, when it is released simultaneously with all processes of higher priority [Layland73]: if a process meets its deadline for a release starting at a critical instant it will always meet its deadline (i.e. it is feasible).

Consider the process set in Table 4.1 (see section 1.5 for nomenclature).

Process	C	D	T
τ_1	2	3	5
τ_2	2	5	6
τ_3	1	8	9
τ_4	1	18	20

Table 4.1: Example Process Set 1.

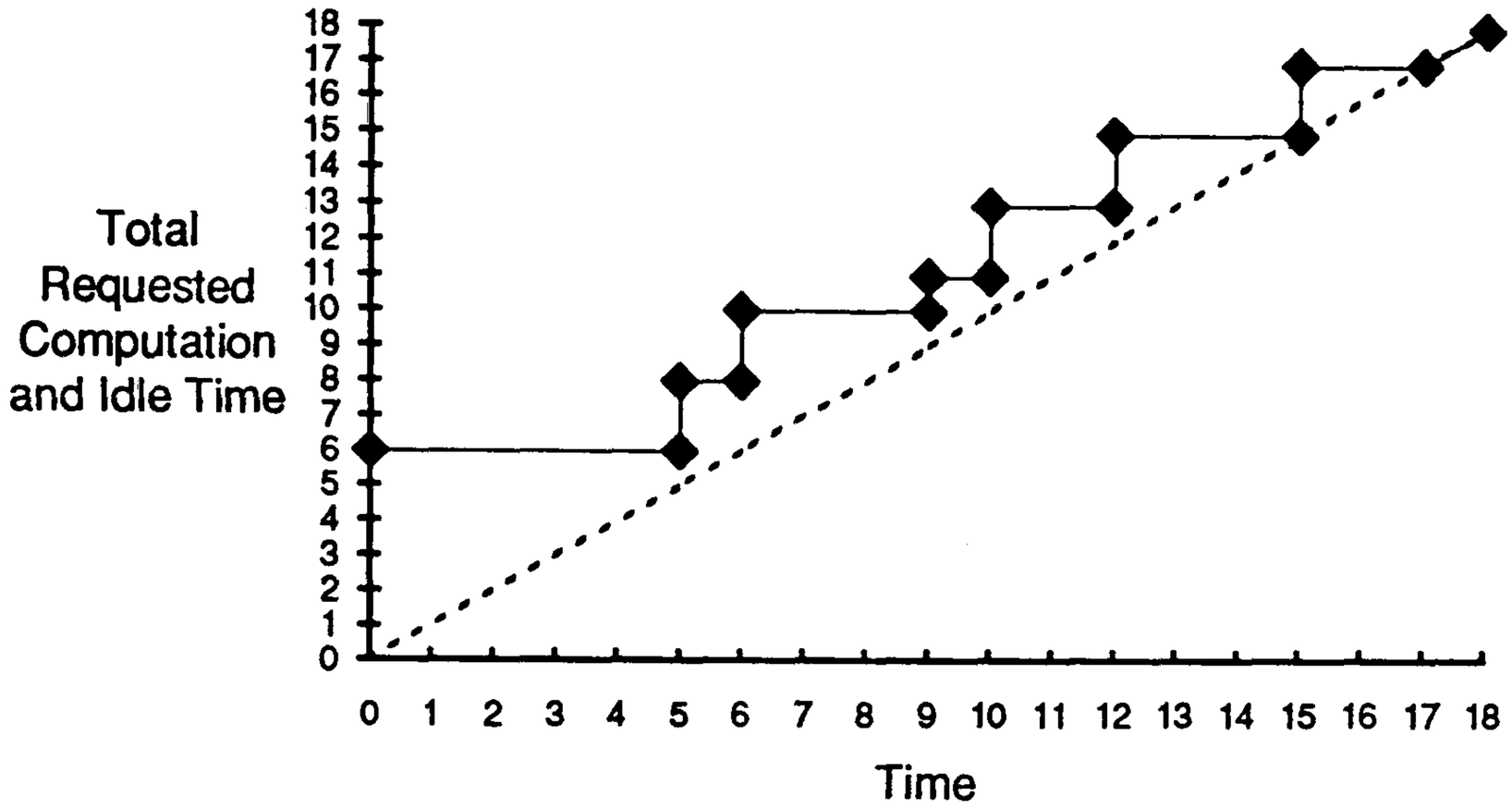


Figure 4.1: TRCG of Example Process Set 1.

We can illustrate the behaviour of the processes using the Total Requested Computation Graph (TRCG) in Figure 4.1. This plots the sum of the total requested computation requirement of processes $\tau_1..\tau_4$ and the processor idle time against time (solid line). The $x=y$ line (dotted) represents the maximum amount of computation (plus idle time) that the processor could have serviced at a given time. Three observations can be made:

(i) when outstanding computation exists at time t, the solid line remains above the dotted line;

(ii) when the outstanding computation reaches zero at time t, the solid line touches the dotted line;

(iii) whilst the processor is idle, due to zero outstanding computation, the solid line and the dotted line are coincident.

For example in Figure 4.1, when $t \in [0,15)$, the total requested computation is greater than the amount of computation serviced by the processor in the interval. At t=15, all requested computation has been serviced. This includes all computation requested by τ_4, implying the process to be feasible (as all computation of processes $\tau_1..\tau_4$ has been met and lower priority processes

cannot pre-empt $\tau_1..\tau_4$). During the interval [17,18) the processor is idle with an interval of idle time identified.

It is noted that if processes do not actually require all their WCET at run-time, τ_4 may meet its deadline earlier. However, the worst-case must be assumed to enable 100% predictability (with respect to meeting deadlines) to be established.

At any time, the total requested computation can be split into that of τ_4 and that of higher priority processes (i.e. $\tau_1..\tau_3$). The latter represents that amount of time that higher priority processes interfere with the execution of τ_4.

In general, we form the following definition:

Definition 4.1:

I_i, the interference on τ_i, represents the total computation of higher priority processes between the release of τ_i at a critical instant and its deadline.

Formalising the feasibility constraint observed in the TRCG (Figure 4.1):

$$C_i + I_i \leq D_i \tag{4.1}$$

Since C_i and D_i are known constants, we conclude that it is the determination of the exact value of I_i that ensures that the complexity of the feasibility test is, in general, NP-hard (see Chapter 2). This becomes apparent by noting that the calculation of the exact value of I_i must consider all higher priority process releases in [0, D_i) (this is shown in Figure 4.1 for τ_4). Thus the complexity is due to the values of process periods, not upon the number of processes. Hence, in general, the complexity of determining (sufficient and necessary) feasibility cannot be bounded by a polynomial function in the number of processes.

Although determination of a precise value of I_i results in NP-complete complexity, pessimistic (i.e. high) estimations can be obtained in polynomial time, although feasibility tests based on such a value will be sufficient and not necessary. This point is illustrated by a pessimistic estimation of I_4 given for the process set in Table 4.1. The pessimistic estimation of total requested computation is given by the dashed line in the TRCG in Figure 4.2. The estimation is constant over [0,18]. This implies that the I_4 component of the requested computation needs evaluation only once, at time 0. Assuming that the evaluation of I_4 has polynomial complexity, the overall complexity of the feasibility test becomes polynomial. Hence, the complexity of the problem has been reduced, but at the cost of decreased accuracy: if τ_4 has deadline 17 the

pessimistic estimation of total computation time would not meet the $x=y$ line until after the deadline, so declaring τ_4 to be infeasible, whilst the exact value for total requested computation time would still declare τ_4 feasible. This has illustrated the offline complexity-flexibility trade-off described in Chapter 3.

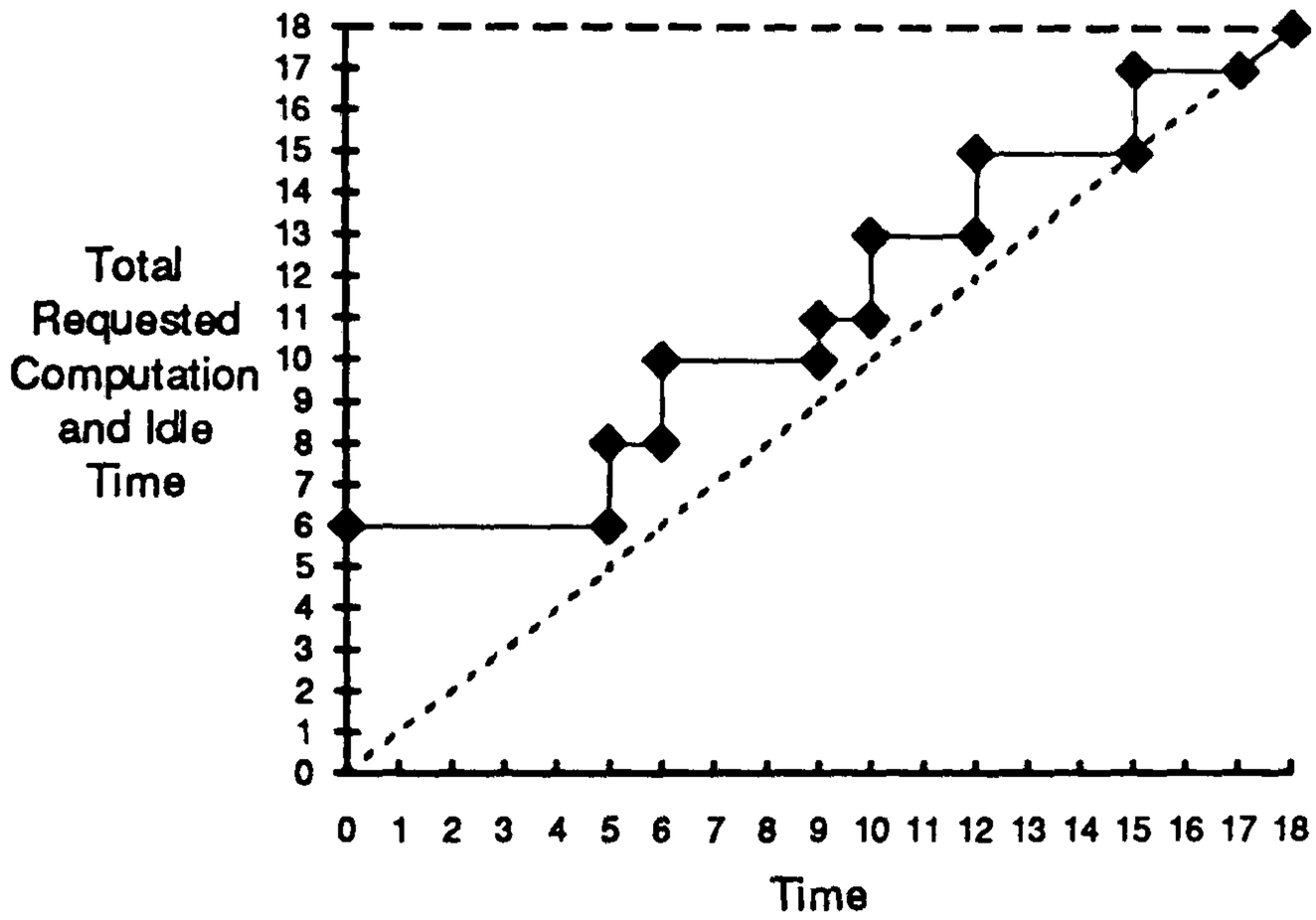


Figure 4.2: TRCG of Example Process Set 1 With Pessimistic I_4.

The relationship between estimations of I_i and the exact value of interference are formalised in the following theorems:

Theorem 4.1:

If the estimated interference I_i is equal to the exact interference I_i' the feasibility test given by equation (4.1) is sufficient and necessary.

Proof:

The proof is in two parts, proving sufficiency and necessity respectively.

Sufficiency

The proof is by contradiction. We assume there is a process set that passes the test but is not feasible, but show that if these processes are not feasible, they must fail the test. Consider a set of n processes. The feasibility of these processes is considered in order $\tau_1..\tau_i..\tau_n$. Let τ_i be the first process to pass the test but not actually meet its deadline. To pass the test, the following must hold:

$$C_i+I_i \leq D_i \tag{4.2}$$

Now, for τ_i not to be feasible, it must miss its deadline during an instance of the process starting at the critical instant of all processes

(i.e. time 0). At this point τ_i suffers its maximum interference, I_i', due to higher priority processes. For τ_i to miss its deadline at run-time, the following must hold:

$$C_i + I_i' > D_i \tag{4.3}$$

Since $I_i = I_i'$ a clear contradiction exists between equations (4.2) and (4.3).

Necessity

Again, the proof is by contradiction. Assume there is a process set that fails the feasibility test but meets all deadlines at run-time (assuming all processes always take their WCET). Consider a set of n processes. The feasibility of these processes is considered in order $\tau_1..\tau_i..\tau_n$. Let τ_i be the first process to fail the test, but actually meet all deadlines at run-time. To fail the test, the following condition must hold:

$$C_i + I_i > D_i \tag{4.4}$$

At run-time, for τ_i to meet its deadline, the following must hold:

$$C_i + I_i' \leq D_i \tag{4.5}$$

Since $I_i = I_i'$ a clear contradiction exists between equations (4.4) and (4.5).

Theorem 4.2:

If the estimated interference I_i is greater than the exact interference I_i' (i.e. is pessimistic) the feasibility test given by equation (4.1) is sufficient and not necessary.

Proof:

The proof of sufficiency follows from the first part of the proof of Theorem 4.1.

The time that may be guaranteed to τ_i by the feasibility test is given by $D_i - I_i$. Since the actual time that could be guaranteed to τ_i is $D_i - I_i'$, with $I_i' < I_i$, less time can be assigned to τ_i by this feasibility test. Therefore, from the proof of necessity in Theorem 4.1, we observe that this feasibility test is not necessary as processes may fail the test but actually meet their deadlines at run-time.

4.2 Sufficient And Not Necessary Feasibility Tests For $D_i \leq T_i$ Processes

As intimated in the previous section, the basic feasibility test for $D_i \leq T_i$ processes can be stated (assuming a critical instant):

$$\forall i: 1 \leq i \leq n : C_i + I_i \leq D_i \tag{4.6}$$

Differing estimations for I_i, together with the above equation, define different feasibility tests, varying in accuracy and complexity. As pessimistic estimations of I_i approach the exact value, their complexity increases, since to obtain increasing accuracy requires a more accurate representation of the actual points in time requests for computation of higher priority processes are made (i.e. higher priority process release times) and when that computation is actually performed by the processor. The following sections outline different estimations of I_i, forming sufficient and not necessary feasibility tests of differing accuracies and complexities. We assume that process sets have a critical instant and that all process offsets are zero.

4.2.1 Sufficient And Not Necessary Feasibility Test No. 1

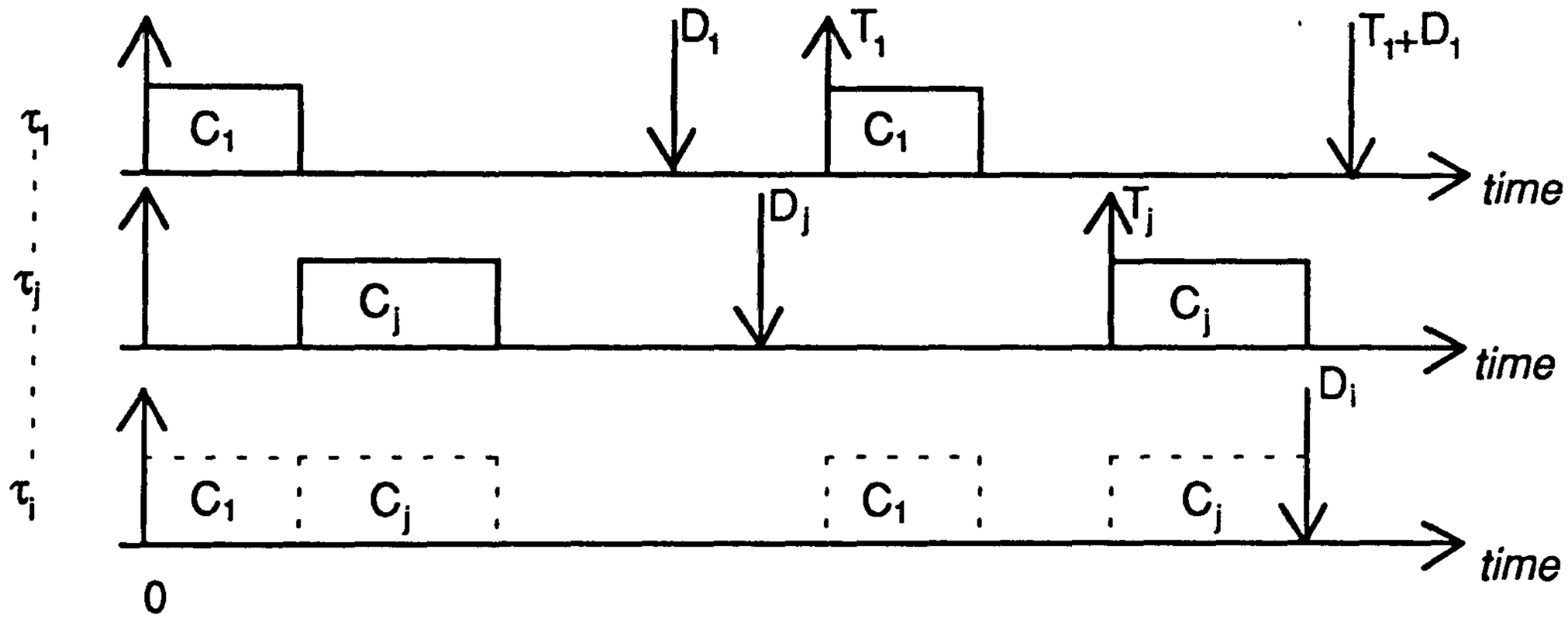


Figure 4.3: Estimating I_i (1).

Consider Figure 4.3. To estimate I_i we observe that the interference inflicted upon τ_i by all higher priority processes corresponds to the computation demands by those processes in the interval of time from the critical instant to the first deadline of τ_i, that is $[0, D_i)$. The computation of higher priority processes that forms part of I_i is shown in the Figure by dotted boxes on the timeline of τ_i. Consider the following theorem:

Theorem 4.3:

The maximum interference of τ_j on τ_i $(1 \le j < i \le n)$ is given by $\lceil D_i/T_j \rceil C_j$.

Proof:

The interference of a higher priority process τ_j is at a maximum when it executes in the first C_j time units after its release, and when its final release in $[0, D_i)$ is at least C_j units before D_i. Formally, the maximum interference of τ_j on τ_i occurs when τ_j executes in the interval $[t, t+C_j)$ where $t \in \{0, T_j, .., \lfloor D_i/T_j \rfloor T_j\}$ and where $\lfloor D_i/T_j \rfloor T_j + C_j \le D_i$. Thus each release of τ_j in $[0, D_i)$ creates C_j units of interference on τ_i. In total there are $\lceil D_i/T_j \rceil$ releases of τ_j in the interval, implying a maximum interference of $\lceil D_i/T_j \rceil C_j$.

The feasibility test is given by:

$$\forall i: 1 \le i \le n : C_i + I_i \le D_i$$

$$\text{where} \qquad I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \tag{4.7}$$

The complexity of this test is $O(n^2)$ in the number of processes.

Since the estimation of I_i is at least the actual interference (Theorem 4.3), the test is sufficient and not necessary (by Theorem 4.2). The pessimism in I_i is due to:

(i) processes are assumed to execute concurrently (i.e. only one process may actually execute at time 0, even though all are assumed to do so when deriving I_i);

(ii) execution of processes included in I_i may actually occur after D_i.

The non-necessity of the test is illustrated in the following example:

Example 4.1:

Process	C	D	T
τ_1	2	3	5
τ_2	2	5	6
τ_3	1	8	9
τ_4	1	15	20

Table 4.2: Example Process Set 2.

Consider the process set in Table 4.2. Processes $\tau_1..\tau_3$ are feasible (calculation omitted for brevity). Consider τ_4. The feasibility of this process is given by (from equation (4.7)):

$$C_4+\left\lceil\frac{D_4}{T_3}\right\rceil C_3+\left\lceil\frac{D_4}{T_2}\right\rceil C_2+\left\lceil\frac{D_4}{T_1}\right\rceil C_1 \le D_4$$

$$1+\left\lceil\frac{15}{9}\right\rceil 1+\left\lceil\frac{15}{6}\right\rceil 2+\left\lceil\frac{15}{5}\right\rceil 2 \le 15$$

$$15 = 15$$

Hence τ_4 is feasible. Let the deadline of τ_4 be increased to 16 - this should not affect the feasibility of the process. Re-consider the feasibility of τ_4:

$$1+\left\lceil\frac{16}{9}\right\rceil 1+\left\lceil\frac{16}{6}\right\rceil 2+\left\lceil\frac{16}{5}\right\rceil 2 > 16 \quad \text{i.e. } 17 > 16$$

Hence τ_4 is declared infeasible.

Since increasing a deadline cannot detract from the actual feasibility, the above example has shown the feasibility test to be sufficient and not necessary.

Under certain circumstances, the above test is also necessary:

Theorem 4.4:

When considering the feasibility of τ_i by the test defined by equation (4.7), if the following condition holds the test is necessary:

$$\forall j : 2 \le j < i : D_i \ge \left\lfloor\frac{D_i}{T_j}\right\rfloor T_j + D_j$$

Proof:

Trivially, the interference of τ_1 on any τ_i ($i > 1$) is always exact since τ_1 always executes for the initial C_1 time units after its release. Assume that processes $\tau_1..\tau_{i-1}$ have been declared feasible. The interference of τ_j ($1 \le j < i \le n$) upon τ_i in $[0, D_i)$ is due to releases of τ_j at times $t \in \{0, T_j, .., \lfloor D_i/T_j \rfloor T_j\}$. If the entire interval in which the computational requirement of τ_j can be honoured by the processor (i.e. the interval $[t, t+D_j)$), is contained in the interval $[0, D_i)$, all C_j will occur in $[0, D_i)$. Since the final release of τ_j in $[0, D_i)$ occurs at $\lfloor D_i/T_j \rfloor T_j$, if the deadline for this release also lies in $[0, D_i)$, all

computation for releases of τ_j in $[0, D_i)$ are honoured in $[0, D_i)$. Thus, if the condition in the theorem holds, the estimation of interference given in equation (4.7) is exact, ensuring that the feasibility test is necessary (by Theorem 4.1).

It is noted that other conditions may also exist which make the test necessary.

4.2.2 Sufficient And Not Necessary Feasibility Test No. 2

We may improve the accuracy of I_i given in the previous section (equation (4.7)) by ignoring computation of higher priority processes that must occur after D_i. Consider the interference of τ_j $(1 \le j < i \le n)$ upon τ_i in $[0, D_i)$. The final release of τ_j in the interval occurs at $\lfloor D_i/T_j \rfloor T_j$. If this release time is less than C_j before D_i, some execution must occur after D_i. Consider the following theorem:

Theorem 4.5:

The maximum interference of τ_j on τ_i $(1 \le j < i \le n)$ is given by:

$$\left\lfloor \frac{D_i}{T_j} \right\rfloor C_j + min\left(C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \right)$$

Proof:

The number of whole periods of τ_j in $[0, D_i)$ is given by $\lfloor D_i/T_j \rfloor$. Thus, $\lfloor D_i/T_j \rfloor C_j$ represents the interference due to executions of τ_j that are guaranteed to complete in the interval. The final release of τ_j in the interval occurs at $\lfloor D_i/T_j \rfloor T_j$. The maximum interference that can be imposed on τ_i by this release is bounded by:

$$D_i - \lfloor D_i/T_j \rfloor T_j$$

Thus, the maximum interference, i, that this release may impose on τ_j is given by:

$$i = \begin{cases} C_j & \text{if } C_j \le D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \\ D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j & \text{otherwise} \end{cases}$$

Hence, the theorem holds.

The feasibility test is given by:

$$\forall i{:}1 \le i \le n \; : C_i + I_i \le D_i$$

$$\text{where} \qquad I_i = \sum_{j=1}^{i-1} \left(\left\lfloor \frac{D_i}{T_j} \right\rfloor C_j + min\left(C_j,\, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \right) \right) \tag{4.8}$$

The complexity of this test is $O(n^2)$ in the number of processes, although in practice it will take longer to execute than the test given in the previous section.

Since the estimation of I_i is at least the actual interference (Theorem 4.5), the test is sufficient and not necessary (by Theorem 4.2).

Consider the following example which illustrates the improvement in accuracy of the above feasibility test compared to that of the previous section:

Example 4.2:

Process τ_4 in Table 4.2 was declared infeasible when its deadline was increased from 15 to 16 by the feasibility test given by equation (4.7) in the previous section. We now consider the feasibility of τ_4 as defined by equation (4.8):

$$C_4 + \left\lfloor \frac{D_4}{T_3} \right\rfloor T_3 + min\left(C_3, D_4 - \left\lfloor \frac{D_4}{T_3} \right\rfloor T_3 \right) + \left\lfloor \frac{D_4}{T_2} \right\rfloor T_2 + min\left(C_2, D_4 - \left\lfloor \frac{D_4}{T_2} \right\rfloor T_2 \right)$$
$$+ \left\lfloor \frac{D_4}{T_1} \right\rfloor T_1 + min\left(C_1, D_4 - \left\lfloor \frac{D_4}{T_1} \right\rfloor T_1 \right) \le D_4$$

$$1 + \left\lfloor \frac{16}{9} \right\rfloor 1 + min\left(1, 16 - \left\lfloor \frac{16}{9} \right\rfloor 9 \right) + \left\lfloor \frac{16}{6} \right\rfloor 2 + min\left(2, 16 - \left\lfloor \frac{16}{6} \right\rfloor 6 \right) + \left\lfloor \frac{16}{5} \right\rfloor 2$$
$$+ min\left(2, 16 - \left\lfloor \frac{16}{5} \right\rfloor 5 \right) \le 16$$

$16 = 16$

Hence τ_4 is feasible.

An example is now given that illustrates the not necessary nature of the feasibility test defined by equation (4.8).

Example 4.3:

Process	C	D	T
τ_1	2	3	5
τ_2	3	5	6
τ_3	2	25	30

Table 4.3: Example Process Set 3.

Consider the process set in Table 4.3. Processes τ_1 and τ_2 are feasible (calculation omitted for brevity). Consider τ_3. The feasibility of this process is given by (from equation (4.8)):

$$C_3+\left\lfloor\frac{D_3}{T_2}\right\rfloor C_2+min\left(C_2,D_3-\left\lfloor\frac{D_3}{T_2}\right\rfloor T_2\right)+\left\lfloor\frac{D_3}{T_1}\right\rfloor C_1+min\left(C_1,D_3-\left\lfloor\frac{D_3}{T_1}\right\rfloor T_1\right)\leq D_3$$

$$2+\left\lfloor\frac{25}{6}\right\rfloor 3+min\left(3,25-\left\lfloor\frac{25}{6}\right\rfloor 6\right)+\left\lfloor\frac{25}{5}\right\rfloor 2+min\left(2,25-\left\lfloor\frac{25}{5}\right\rfloor 5\right)=25$$

i.e. 25 = 25

Hence τ_3 is feasible. Let the deadline of τ_3 be increased to 26 - this should not affect the feasibility of the process:

$$2+\left\lfloor\frac{26}{6}\right\rfloor 3+min\left(3,26-\left\lfloor\frac{26}{6}\right\rfloor 6\right)+\left\lfloor\frac{26}{5}\right\rfloor 2+min\left(2,26-\left\lfloor\frac{26}{5}\right\rfloor 5\right)>26$$

i.e. 27 > 26

Hence τ_3 is declared infeasible, although would still meet its deadline.

Since increasing a deadline cannot detract from the actual feasibility, the above example has shown the feasibility test to be not necessary.

It is noted that the test is necessary under the conditions defined by Theorem 4.4.

4.2.3 Sufficient And Not Necessary Feasibility Test No. 3

The estimation of I_i given by equation (4.8) in the previous section is pessimistic as some (or all) of the final releases of τ_j $(1\leq j<i)$ are assumed to execute concurrently. Consider the interference on τ_i by processes τ_j and τ_k $(1\leq j<k<i)$. For shorthand purposes, we denote the final release of these processes in $[0, D_i)$ by t_j and t_k respectively, where $t_k < t_j$. That is:

$$t_j=\left\lfloor\frac{D_i}{T_j}\right\rfloor T_j \qquad\qquad t_k=\left\lfloor\frac{D_i}{T_k}\right\rfloor T_k$$

Interference estimation is now improved. If the following condition holds, concurrent execution will be assumed by equation (4.8):

$$D_i - t_j < C_j \ \wedge \ D_i - t_k < C_k \ \wedge \ t_j - t_k < C_k$$

Assume that the following condition holds true:

$$D_i - t_j \leq C_j \ \wedge \ D_i - t_k \leq C_k \ \wedge \ t_k < t_j$$

This is illustrated in Figure 4.4. Clearly the final release of τ_k in [0, D_i) may execute before t_j (since $t_k < t_j$). The remainder of its execution will begin at or after $t_j + C_j$. The computation of higher priority processes that forms part of the exact interference on τ_i is shown in the Figure by dotted boxes on the timeline of τ_i.

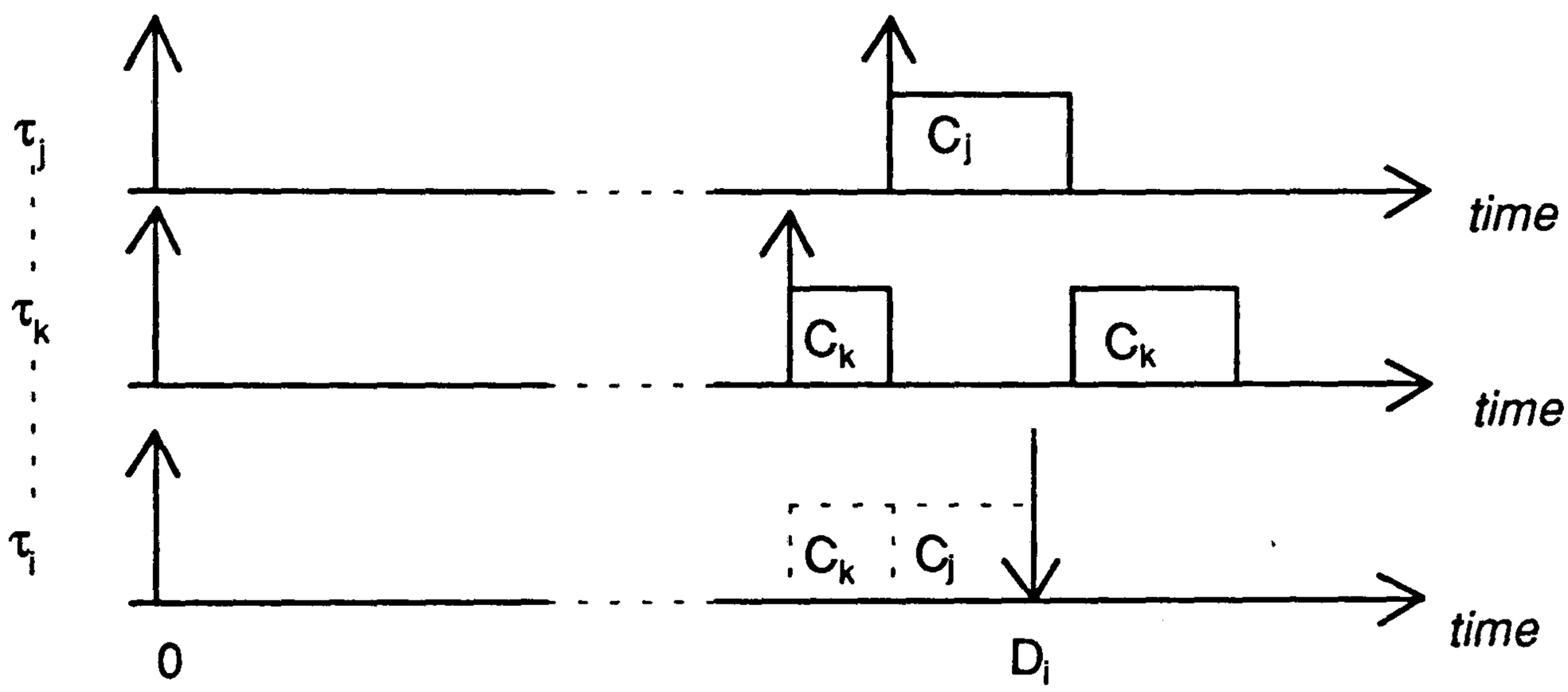


Figure 4.4 : Estimating I_i (2).

After calculation of the interference of τ_i upon τ_i, we may reduce D_i to t_j when considering the interference of τ_k on τ_i.

Definition 4.2:

The *effective deadline* of τ_i when considering the interference of τ_j upon τ_i is denoted d_j^i. The interval $[d_j^i, D_i)$ is occupied entirely by the executions of processes of higher priority than τ_j.

In the above example, $d_k^i = t_j$ when considering the interference of t_k on τ_i. Consider the following theorem:

Theorem 4.6:

The maximum interference of τ_j on τ_i $(1 \leq j < i \leq n)$ is given by:

$$\left\lfloor \frac{d_j^i}{T_j} \right\rfloor C_j + min\left(C_j, d_j^i - \left\lfloor \frac{d_j^i}{T_j} \right\rfloor T_j \right)$$

Proof:

The interference of τ_j upon τ_i can only be due to executions of τ_j in $[0, d_j^i)$, where $d_j^i \leq D_i$. Therefore, the quantification of interference follows from Theorem 4.5.

The feasibility test may be stated:

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

where

$$I_i = \sum_{j=1}^{i-1} \left(\left\lfloor \frac{d_j^i}{T_j} \right\rfloor C_j + min\left(C_j, d_j^i - \left\lfloor \frac{d_j^i}{T_j} \right\rfloor T_j \right) \right)$$

$$d_j^i = \begin{cases} D_i & \text{if } \; j = 1 \\ d_{j-1}^i & \text{if } \; j > 1 \wedge d_{j-1}^i - \left\lfloor \frac{d_{j-1}^i}{T_{j-1}} \right\rfloor T_{j-1} > C_{j-1} \\ \left\lfloor \frac{d_{j-1}^i}{T_{j-1}} \right\rfloor T_{j-1} & \text{otherwise} \end{cases} \tag{4.9}$$

The definition of d_j^i reduces the effective deadline by the length of the interval

$$\left[\left\lfloor \frac{d_{j-1}^i}{T_{j-1}} \right\rfloor T_{j-1}, d_{j-1}^i \right)$$

if and only if that length is not more than C_{j-1}. In this way, the interval is guaranteed to be occupied by the execution of τ_{j-1} (if it is not pre-empted by a process of higher priority than τ_{j-1}).

The feasibility test above is sufficient and not necessary by Theorem 4.2 as I_i is at least the exact value of interference (Theorem 4.6). The test is $O(n^2)$ in complexity, although in practice will take longer to execute than the test given in the previous section.

Consider the following example which illustrates the improvement in accuracy of the above feasibility test compared to that of the previous section:

Example 4.4:

According to the feasibility test given by equation (4.8), process τ_3 in Table 4.3 is feasible if D_3=25 but not if D_3=26. We now consider the feasibility of τ_3 as defined by equation (4.9):

Noting that $d_1^3 = D_3 = 26$ the interference upon τ_3 due to τ_1 is:

$$\left\lfloor \frac{d_1^3}{T_1} \right\rfloor C_1 + min\left(C_1, d_1^3 - \left\lfloor \frac{d_1^3}{T_1} \right\rfloor T_1 \right) = \left\lfloor \frac{26}{5} \right\rfloor 2 + min\left(2, 26 - \left\lfloor \frac{26}{5} \right\rfloor 5 \right) = 11$$

Now since $d_1^3 - \left\lfloor \frac{d_1^3}{T_1} \right\rfloor T_1 < C_1$ i.e. $1 < 2$ we have $d_2^3 = \left\lfloor \frac{d_1^3}{T_1} \right\rfloor T_1 = 25$

Interference due to τ_2 is:

$$\left\lfloor \frac{d_2^3}{T_2} \right\rfloor C_2 + min\left(C_2, d_2^3 - \left\lfloor \frac{d_2^3}{T_2} \right\rfloor T_2 \right) = \left\lfloor \frac{25}{6} \right\rfloor 3 + min\left(3, 25 - \left\lfloor \frac{25}{6} \right\rfloor 6 \right) = 13$$

Evaluating feasibility condition:

$$C_4 + I_4 \leq D_4$$

$$2 + 11 + 13 = 26$$

Hence τ_3 is feasible.

The value of I_i calculated by equation (4.9) is pessimistic, being at least the actual I_i. The main reason for this is that concurrent execution is still assumed between higher priority processes. For example, if τ_1 has a final release at t_1, with $t_1 + C_1 < D_1$, then $d_2^i = D_i$. Now, if the final release at t_2 is such that $t_2 < t_1 < t_2 + C_2$, overlapping execution between τ_1 and τ_2 is assumed within I_i. We illustrate the pessimism of the test with the following example:

Example 4.5:

Process	C	D	T
τ_1	2	5	6
τ_2	2	6	8
τ_3	3	12	18
τ_4	2	20	30

Table 4.4: Example Process Set 4.

Consider the process set in Table 4.4. Processes τ_1, τ_2 and τ_3 are feasible (calculation omitted for brevity). Consider τ_4. The feasibility of this process is given by (from equation (4.9)):

Noting that $d_1^4 = D_4 = 20$ the interference due to τ_1 is:

$$\left\lfloor \frac{d_1^4}{T_1} \right\rfloor C_1 + min\left(C_1, d_1^4 - \left\lfloor \frac{d_1^4}{T_1} \right\rfloor T_1 \right) = \left\lfloor \frac{20}{6} \right\rfloor 2 + min\left(2, 20 - \left\lfloor \frac{20}{6} \right\rfloor 6 \right) = 8$$

Now since $d_1^4 - \left\lfloor \frac{d_1^4}{T_1} \right\rfloor T_1 \leq C_1$ i.e. $2 = 2$ we have $d_2^4 = \left\lfloor \frac{d_1^4}{T_1} \right\rfloor T_1 = 18$

Interference due to τ_2 is:

$$\left\lfloor \frac{d_2^4}{T_2} \right\rfloor C_2 + min\left(C_2, d_2^4 - \left\lfloor \frac{d_2^4}{T_2} \right\rfloor T_2 \right) = \left\lfloor \frac{18}{8} \right\rfloor 2 + min\left(2, 18 - \left\lfloor \frac{18}{8} \right\rfloor 8 \right) = 6$$

Now since $d_2^4 - \left\lfloor \frac{d_2^4}{T_2} \right\rfloor T_2 \leq C_2$ i.e. 2=2 we have $d_3^4 = \left\lfloor \frac{d_2^4}{T_2} \right\rfloor T_2 = 16$

Interference due to τ_3 is:

$$\left\lfloor \frac{d_3^4}{T_3} \right\rfloor C_3 + min\left(C_3, d_3^4 - \left\lfloor \frac{d_3^4}{T_3} \right\rfloor T_3 \right) = \left\lfloor \frac{16}{18} \right\rfloor 3 + min\left(3, 16 - \left\lfloor \frac{16}{18} \right\rfloor 18 \right) = 3$$

Evaluating feasibility condition:

$$C_4 + I_4 \leq D_4$$

$$2 + 8 + 6 + 3 < 20$$

Hence τ_4 is feasible.

Let the deadline of τ_4 be increased to 21 - this should not affect the feasibility of the process:

Noting that $d_1^4 = D_4 = 21$ the interference due to τ_1 is:

$$\left\lfloor \frac{d_1^4}{T_1} \right\rfloor C_1 + min\left(C_1, d_1^4 - \left\lfloor \frac{d_1^4}{T_1} \right\rfloor T_1 \right) = \left\lfloor \frac{21}{6} \right\rfloor 2 + min\left(2, 21 - \left\lfloor \frac{21}{6} \right\rfloor 6 \right) = 8$$

Now since $d_1^4 - \left\lfloor \frac{d_1^4}{T_1} \right\rfloor T_1 > C_1$ i.e. 3>2 we have $d_2^4 = d_1^4 = 21$

Interference due to τ_2 is:

$$\left\lfloor \frac{d_2^4}{T_2} \right\rfloor C_2 + min\left(C_2, d_2^4 - \left\lfloor \frac{d_2^4}{T_2} \right\rfloor T_2 \right) = \left\lfloor \frac{21}{8} \right\rfloor 2 + min\left(2, 21 - \left\lfloor \frac{21}{8} \right\rfloor 8 \right) = 6$$

Now since $t_2^4 - \left\lfloor \frac{t_2^4}{T_2} \right\rfloor T_2 > C_2$ i.e. 5>2 we have $d_3^4 = d_2^4 = 21$

Interference due to τ_3 is:

$$\left\lfloor \frac{d_3^4}{T_3} \right\rfloor C_3 + min\left(C_3, d_3^4 - \left\lfloor \frac{d_3^4}{T_3} \right\rfloor T_3 \right) = \left\lfloor \frac{21}{18} \right\rfloor 3 + min\left(3, 21 - \left\lfloor \frac{21}{18} \right\rfloor 18 \right) = 6$$

Evaluating feasibility condition:

$$C_4 + I_4 \leq D_4$$

$$2 + 8 + 6 + 6 > 21$$

Hence τ_4 is declared infeasible, although it would still meet its deadlines at run-time.

Since increasing a process's deadline cannot detract from the actual feasibility, example 4.5 shows the test to be not necessary. The test is necessary under the assumptions given by Theorem 4.4, with d_j^i replacing D_i in the condition.

4.2.4 Sufficient And Not Necessary Feasibility Test No. 4

The estimation of I_i given by equation (4.9) may still assume concurrent execution of processes during their final release before D_i. We may improve on this estimation by separating the calculation of the effective deadline of τ_i for τ_j and the calculation of the interference of τ_j upon τ_i. Initially, the effective deadline of τ_i is calculated iteratively considering all processes $\tau_1..\tau_{i-1}$. Then, interference is calculated with all processes $\tau_1..\tau_{i-1}$ assuming the same effective deadline. This is in contrast to the approach adopted in the previous section where only $\tau_1..\tau_{j-1}$ contribute to the calculation of the effective deadline for τ_j (where $j<i$).

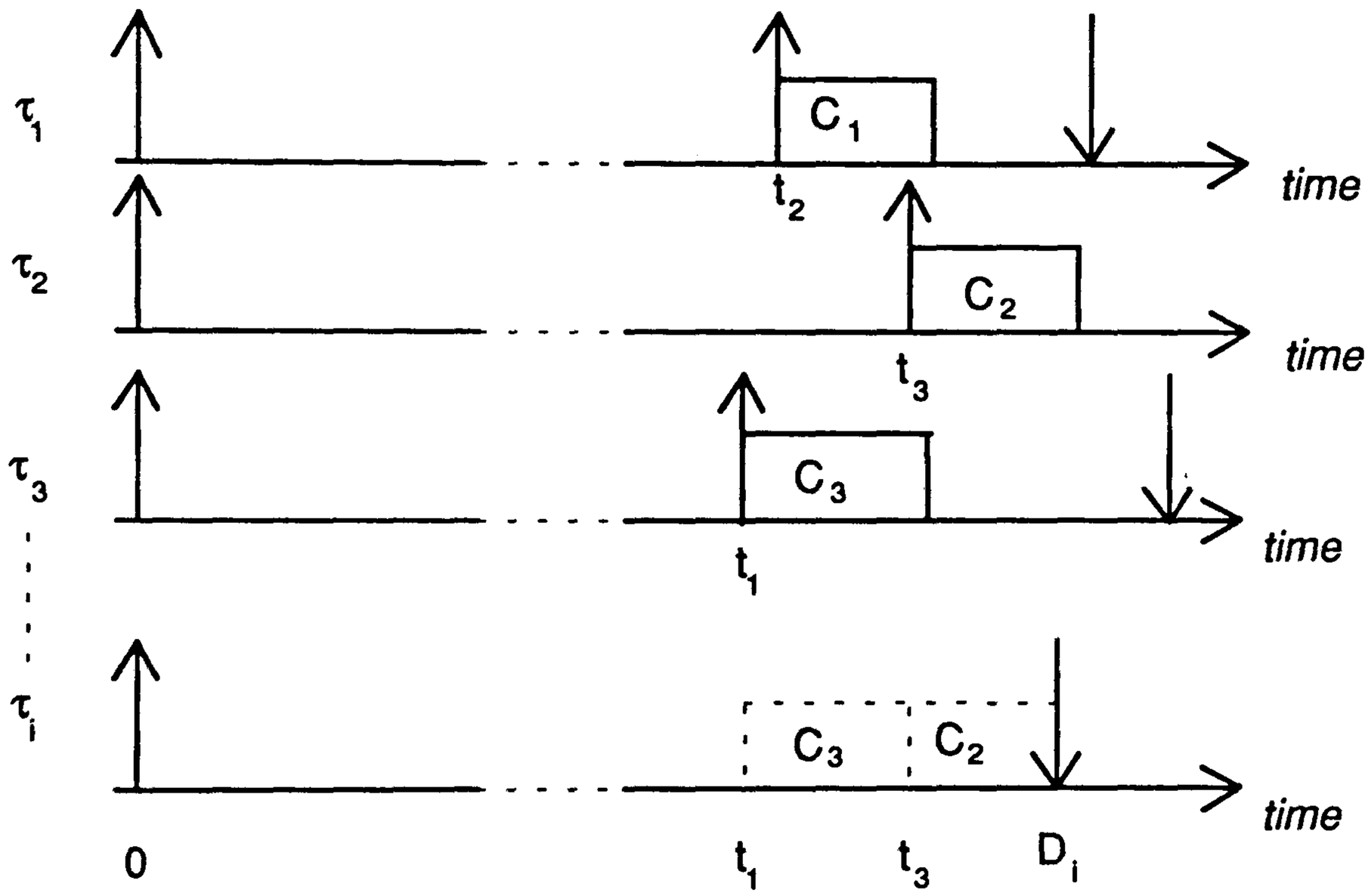


Figure 4.5: Estimating I_i (3).

Consider the interference of τ_1 upon τ_i in Figure 4.5. From the figure we observe that by the definition of d_j^i by equation (4.9) $d_1^i = d_2^i = D_i$ and $d_3^i = t_3$. Clearly, the executions of τ_1 at t_2, τ_2 at t_3 and τ_3 at t_1 should not all constitute part of I_i. From the figure we may observe that the interval $[t_1, D_i)$ is occupied by the executions of processes of higher priority than τ_i and that $D_i - t_1 < C_1 + C_2 + C_3$. Ignoring τ_1, the execution of τ_2 at t_3 will occupy $[t_3, D_i)$ with the execution of τ_3 at t_1 occupying $[t_1, t_3)$. Therefore, we may reduce the effective deadline when calculating the interference of $\tau_1..\tau_{i-1}$ upon τ_i to t_1. This is illustrated in Figure 4.5 where the executions of τ_2 and τ_3 form the

interference upon τ_i (dotted box on the timeline of τ_i). It is noted that τ_1 would actually run in $[t_2, t_2 + C_2)$ as it has a higher priority than τ_2 and τ_3, although the net effect on τ_i is equivalent: the interval $[t_1, D_i)$ is occupied by higher priority process executions.

Consider processes $\tau_1..\tau_{j-1}$ (in order) in terms of their final release in $[0, D_i)$. Let the initial effective deadline be $d=D_i$. If any processes are guaranteed to occupy $[t, d)$, where t is the final release of the process in $[0, d)$, we may decrease the effective deadline to t.

Definition 4.3:

The *effective deadline* of τ_i when considering the interference of τ_j upon τ_i is denoted $d^i_{j,k}$ where k represents the current iteration (from 0 upwards). The interval $[d^i_{j,k}, D_i)$ is occupied entirely by the executions of processes of higher priority than τ_i.

The definition assumes that $d^i_{j,k} \leq d^i_{j,k-1}$ for any $k>1$. Consider the following theorem:

Theorem 4.7:

The maximum interference of τ_j upon τ_i $(1 \leq j < i \leq n)$ is given by:

$$\left\lceil \frac{d^i_{i-1,k}}{T_j} \right\rceil C_j$$

where $d^i_{1,0} = d^i_{2,0} = \ldots = d^i_{i-1,0} = D_i$

and where the following condition holds:

$$\exists k : k > 0 \bullet \forall m : 1 \leq m < i : d^i_{m,k} = d^i_{m,k-1}$$

Proof:

After *k-1* iterations, each of which considering processes $\tau_1..\tau_{i-1}$ in order of descending priority, the effective deadline has been reduced to $d^i_{i-1,k-1}$ (since τ_{i-1} is considered last). Let all processes have a final release in $[0, d^i_{i-1,k-1})$ that is completed by $d^i_{i-1,k-1}$, that is the following condition holds:

$$\forall j : 1 \leq j < i : \left\lfloor \frac{t^i_{i-1,k-1}}{T_j} \right\rfloor T_j + C_j < t^i_{i-1,k-1}$$

Now, no processes can reduce the effective deadline on the k^{th} iteration, so fulfilling the condition in the theorem. Thus, the interval $[d^i_{i-1,k-1}, D_i)$ is occupied by the execution of processes of higher priority than τ_i. The total interference is the length of the

interval $[d^i_{i-1,k}, D_i)$ and the interference of each τ_j on τ_i in $[0, d^i_{i-1,k})$. The latter is given by extending Theorem 4.3:

$$\left\lceil \frac{d^i_{i-1,k}}{T_j} \right\rceil C_j \tag{4.10}$$

We note that since the final release of all τ_j $(1 \le j < i \le n)$ in $[0, d^i_{i-1,k})$ is at least C_j time units before $d^i_{i-1,k}$, there is no need to perform the minimum operation required in the feasibility tests given by equations (4.8) and (4.9).

Since the interval $[d^i_{i-1,k-1}, D_i)$ is entirely occupied by the executions of processes of higher priority than τ_i, the exact interference in this interval is

$$D_i - t^i_{i-1,k} \tag{4.11}$$

Therefore, total interference is a summation of equations (4.10) with (4.11) for each higher priority process. The feasibility test may be stated:

$$\forall i : 1 \le i \le n \; : C_i + I_i \le D_i$$

$$\text{where} \qquad I_i = D_i - d^i_{i-1,k} + \sum_{j=1}^{i-1} \left\lceil \frac{d^i_{i-1,k}}{T_j} \right\rceil C_j \tag{4.12}$$

$$d^i_{j,k} = \begin{cases} D_i & \text{if } (k=0) \vee (k=1 \wedge j=1) \\ \left\lfloor \dfrac{d^i_{\text{prev}^i_j, \text{iter}^j_k}}{T_{\text{prev}^i_j}} \right\rfloor T_{\text{prev}^i_j} & \text{if } 1 < j \le i-1 \wedge \; d^i_{\text{prev}^i_j, \text{iter}^j_k} - \left\lfloor \dfrac{d^i_{\text{prev}^i_j, \text{iter}^j_k}}{T_{\text{prev}^i_j}} \right\rfloor T_{\text{prev}^i_j} \le C_{\text{prev}^i_j} \\ d^i_{\text{prev}^i_j, \text{iter}^j_k} & \text{otherwise} \end{cases}$$

$$\text{prev}^i_j = \begin{cases} i-1 & \text{if } j=1 \\ j-1 & \text{otherwise} \end{cases} \qquad \text{iter}^j_k = \begin{cases} k-1 & \text{if } j=1 \\ k & \text{otherwise} \end{cases}$$

Assuming that at least 1 iteration is performed (i.e. the k=1 iteration), the definition of $d^i_{j,k}$ corresponds to that required by Theorem 4.7. The definition of $d^i_{j,0} = D_i$ for $1 \le j < i$ allows the condition in the theorem to be evaluated after the k=1 iteration. The functions prev and iter enable the effective deadline

calculated for τ_{i-1} to be used for τ_1 on the next iteration, that is when calculating $d^i_{1,k}$ the value $d^i_{i-1,k-1}$ is needed and is available.

The above test is sufficient and not necessary as I_i is at least the exact value of interference (Theorem 4.7).

The complexity of this test is due to both effective deadline calculation and the subsequent determination of feasibility. The former is dependent upon the number of iterations (k) and the calculation of revised effective deadlines during an iteration. In the worst-case this is $O(kn^2)$, when exactly one $\tau_j \in\{\tau_1,..,\tau_{i-1}\}$ reduces the effective deadline by one on each iteration. Feasibility determination has complexity $O(n^2)$. Thus the overall complexity is $O((k+1)n^2) \approx O(kn^2)$ where $k=\max_{1\le j<i}(D_j)$. Since k is not a polynomial function of n, the test is NP-complete (see Chapter 2). Such complexity occurs when very large numbers are involved (i.e. large periods and deadlines). In most cases, the test will not be applied to such numbers. Thus, the test has pseudo-polynomial complexity[1] for limited values of k, for example when $k=\max_{1\le i\le D_i}(D_i)$. The implication of this observation is that for most cases (excepting pathological cases) the test has effective polynomial complexity.

The complexity may be reduced by setting k to be a constant for a process set (or across all process sets). This has the effect of making the feasibility test less accurate if the value chosen is insufficient to find the actual final value of k, although this version of the test has polynomial complexity (for all values of process periods and deadlines).

Consider the following example which illustrates the improvement in accuracy of the above feasibility test compared to that of the previous section:

Example 4.6:

According to the feasibility test given by equation (4.9), process τ_4 in Table 4.4 is feasible if $D_4=20$ but not if $D_4=21$. We now consider the feasibility of τ_4 as defined by equation (4.12):

Calculate effective deadline:

Iteration $k=1$

$d^4_{1,1} = D_4 = 21$

Since $d^4_{1,1} - \left\lfloor \frac{d^4_{1,1}}{T_1} \right\rfloor T_1 > C_1$ i.e. $3>2$ we have $d^4_{2,1}=d^4_{1,1}=21$

---

[1] the definition of pseudo-polynomial complexity is drawn from [Garey79].

Since $d_{2,1}^4 - \left\lfloor \frac{d_{2,1}^4}{T_2} \right\rfloor T_2 > C_2$ i.e. 5>2 we have $d_{3,1}^4 = d_{2,1}^4 = 21$

Iteration k=2

Since $d_{3,1}^4 - \left\lfloor \frac{d_{3,1}^4}{T_3} \right\rfloor T_3 \leq C_3$ i.e. 3=3 we have $d_{1,2}^4 = \left\lfloor \frac{d_{3,1}^4}{T_3} \right\rfloor T_3 = 18$

Since $d_{1,2}^4 - \left\lfloor \frac{d_{1,2}^4}{T_1} \right\rfloor T_1 \leq C_1$ i.e. 0<2 we have $d_{2,2}^4 = \left\lfloor \frac{d_{1,2}^4}{T_1} \right\rfloor T_1 = 18$

Since $d_{2,2}^4 - \left\lfloor \frac{d_{2,2}^4}{T_2} \right\rfloor T_2 \leq C_2$ i.e. 2=2 we have $d_{3,2}^4 = \left\lfloor \frac{d_{2,2}^4}{T_2} \right\rfloor T_2 = 16$

Iteration k=3

$d_{1,3}^4 = d_{2,3}^4 = d_{3,3}^4 = 16$ (calculation omitted for brevity)

Thus, the iterative derivation of the effective deadline completes by the condition in Theorem 4.8 when k=3.

The feasibility of τ_4 is given by:

$$C_4 + D_4 - d_{3,3}^4 + \left\lceil \frac{d_{3,3}^4}{T_1} \right\rceil C_1 + \left\lceil \frac{d_{3,3}^4}{T_2} \right\rceil C_2 + \left\lceil \frac{d_{3,3}^4}{T_3} \right\rceil C_3 \leq D_4$$

$20 < 21$

Hence τ_4 is feasible.

In the previous example, k could be set to a constant value to ensure polynomial complexity. If k=*1* then the effective deadline is 21, which would cause the process set to be found infeasible (see example 4.5). If k=*2* or k=3 the effective deadline is 16 with the resulting feasibility test finding the process set feasible.

The feasibility test given by equation (4.12) is sufficient and not necessary as concurrent execution of processes is still assumed when calculating I_i.

The estimation of I_i remains pessimistic. Consider the following condition:

$$\forall j : 1 \leq j < i \leq n : \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j + C_j < D_i$$

When the condition holds the effective deadline of τ_i remains D_i. Concurrent execution of (some) or all higher priority processes is now assumed if the following condition holds:

$$\exists \leq j,k:1 \leq j,k < i \leq n:$$

$$j \neq k \wedge \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \leq \left\lfloor \frac{D_i}{T_k} \right\rfloor T_k < \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j + C_j$$

This observation is shown in the process set given in Table 4.5. The set is declared infeasible by the above test although all deadlines will be met at run-time (see example 4.7 section 4.3).

Process	C	D	T
τ_1	1	5	6
τ_2	3	6	8
τ_3	4	14	14
τ_4	1	20	30

Table 4.5: Example Process Set 5.

The feasibility test given by equation (4.12) is necessary under the same general conditions as described by Theorem 4.4, with $d^i_{i-1,k}$ replacing D_i in the condition. We note that the theorem also holds if constant values of k are used, since D_i-$d^i_{i-1,k}$ is always the exact interference of interval $[d^i_{i-1,k}, D_i)$ for any k.

4.2.5 Summary

This section has presented four sufficient and not necessary feasibility tests. The tests are not-necessary as they assume some concurrent execution of processes $\tau_1..\tau_{i-1}$ when calculating I_i (although under some circumstances the tests were found to be necessary). The concurrent execution, and therefore the pessimism, occurs in the final releases of the former processes before D_i. In successive tests, this inaccuracy has lessened, with the effect of increased complexity.

In general, more accurate sufficient and not necessary tests could be developed. The complexity of these tests would, in general, increase further.

4.3 Sufficient And Necessary Feasibility Tests For $D_i \leq T_i$ Processes

To form a sufficient and necessary feasibility test an exact evaluation of I_i is required (by Theorem 4.1). All releases of processes of higher priority than τ_i must be examined in [0, D_i) so that only executions actually occurring in the

interval form I_i. That is, the inherent assumption in the sufficient and not necessary tests that some concurrent execution may occur (in [0, D_i)) is removed.

Several approaches may be defined for determining sufficient and necessary feasibility, for example by the explicit construction of a schedule for the process set over $\left[0, \max_{1 \le i \le n}(D_i)\right)$ [Leung82]. This approach is computationally expensive. The remainder of this section develops two other more efficient approaches.

The first approach is to find the completion or response time of τ_i within [0, D_i). This requires an exact value for the interference of higher priority processes for the interval [0, R_i) where R_i is the response time of τ_i ($R_i \le D_i$ for the release of the process at a critical instant if the process is feasible). This approach is especially useful when considering end-to-end deadlines: we must determine whether a number of process, often arranged in a precedence-constrained manner, meet a collective deadline. Here, the worst-case response time of one process forms the latest start time of subsequent processes in the precedence constraint. This is more accurate than assuming processes complete at their deadlines.

The second approach is to find an exact value of I_i for the interval [0, D_i), then to apply the simple feasibility constraint given by equation (4.1). Such an approach is useful if the WCET of a process may be increased (by no more than D_i - I_i), for example to permit a more complex algorithm to execute (we return to this in Chapter 7).

The approaches outlined above are discussed in the following sections.

4.3.1 Response Time Sufficient and Necessary Feasibility Test

To find the response time R_i of τ_i requires that we examine the interference of higher priority processes over intervals within [0, D_i).

Definition 4.4:

The term I_i^t is the interference on τ_i due to releases of processes with higher priority than τ_i in the interval [0, t) where $0 \le t \le D_i$.

We note that the definition prescribes that requested computation due to releases of $\tau_1..\tau_{i-1}$ at 0,1,..,t-1 is included in I_i^t, not those at t. Hence, we may state the feasibility test to be:

$$\forall \tau_i : 1 \le i \le n \bullet$$
$$\exists t \in [0, D_i) \bullet \; C_i + I_i^t = t \tag{4.13}$$

Consider the behaviour of the process set in Table 4.6 as depicted in the TRCG of the set in Figure 4.6.

Process	C	D	T
τ_1	1	2	4
τ_2	2	4	6
τ_3	3	12	13
τ_4	1	14	20

Table 4.6: Example Process Set 6.

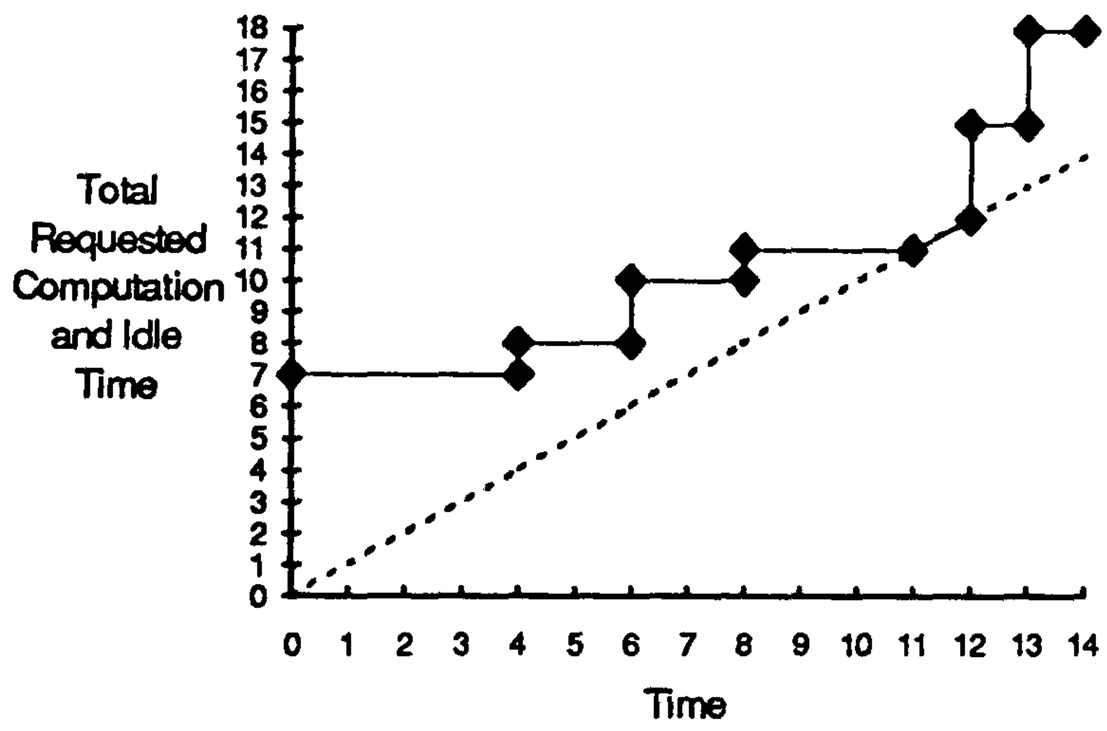


Figure 4.6: TRCG of Example Process Set 6.

From the discussion in section 4.1 we make the following observations regarding Figure 4.6:

(i) τ_4 meets its deadline, completing execution at time 11 (solid line first touches dotted line);

(ii) the exact value of $I_4 + C_4$ is given by the sum of lengths of (non-overlapping) intervals in [0, D_4], that is the amount of time where outstanding computation exists. In the graph this equates to the lengths of intervals where the solid line is above the dotted line:

$$I_4 + C_4 = length([0,11]) + length([12,14])$$
$$I_4 + C_4 = 13$$

(iii) the idle time is given by the sum of lengths of (non-overlapping) intervals in [0, D_4] where the solid line lies on top of the dotted line:

$$S_4 = length([11,12]) = 1$$

Given that τ_4 completes at time 11, we note that if D_4 is reduced to 11 then the sufficient and not necessary test estimation of interference given by equation (4.7) becomes sufficient and necessary (by Theorem 4.4).

Therefore, the maximum interference of a higher priority process τ_j upon τ_i $(i < j)$ in the interval $[0, t)$ is given by:

$$\left\lceil \frac{t}{T_j} \right\rceil C_j$$

When t represents the completion time of τ_i, the above estimate will be exact. Therefore, we may fully state the feasibility test:

$$\forall \tau_i : 1 \le i \le n \bullet$$
$$\exists t \in [0, D_i) \bullet \; C_i + I_i^t = t$$

$$\text{where} \qquad I_i^t = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \qquad (4.14)$$

The test is sufficient and necessary since the interference is exact.

One obvious implementation of this test would use values of t from 1 to D_i (since C_i is assumed to be strictly positive), testing to see if the length of $[0, t)$ (i.e. t) is sufficient to contain C_i and the execution due to releases of $\tau_1 .. \tau_{i-1}$ (i.e. I_i^t) in the interval. This would continue until a value for t was found satisfying equation (4.14). If no such value of t could be found, the process is infeasible. We note that many (i.e. D_i) points in time are examined to determine feasibility. This provides the non-polynomial complexity of the test since at each point in time a polynomial function is evaluated.

The non-polynomial complexity can be lessened by reducing the number of equations that need to be evaluated whilst determining the feasibility of τ_i. This is achieved by limiting the points in $[0, D_i)$ that are considered as possible solutions for t. It is noted that I_i^t is monotonically increasing over $[0, D_i)$. The points in time that the interference increases correspond to a release of a higher priority process in $[0, D_i)$. This is illustrated in Figure 4.6. In the figure there are three processes of higher priority than τ_4. Over the interval [0,11) the value of total requested computation time is given by $I_4^t + C_4$. The graph is stepped with plateaus representing intervals of time in which no higher priority processes are released. Only one value of t need be considered for each plateau as I_4^t does not change. For example, in Figure 4.6 $I_4^1 = I_4^2 = I_4^3 = I_4^4 = 6$, that is 6 units of computation are requested by $\tau_1 .. \tau_{i-1}$ in [0, 4). To maximise the time available for the execution of τ_i we let t be equal to the time at the

right-most point of the plateau (e.g. t=4 initially in for Figure 4.6). This reduces the number of equations required from 4 to 1 in this example.

The number of points in time we evaluate the feasibility of τ_i can be reduced further by considering the computation times of higher priority processes. Firstly, there is little point in considering any $t \in [0, C_i)$ as a solution. Secondly, since time 0 corresponds to a critical instant, the least possible value of t at which τ_i may complete is $R_{i-1}+C_i$ as [0,) is occupied entirely by the executions of $\tau_1..\tau_{i-1}$. We term the first possible value of t to be t_0, given by:

$$t_0 = R_{i-1} + C_i$$

If a release of a higher priority process occurs in $[0, t_0)$ the value of t_0 will not form a solution. The exact amount of interference in this interval is given by $I_i^{t_0}$. Hence the next point in time τ_i may complete is at:

$$t_1 = I_i^{t_0} + C_i$$

Again the constraint will fail if higher priority processes are released in $[t_0, t_1)$. Thus we may identify the points in time at which feasibility of τ_i must be tested in $[0, D_i)$:

$$t_0 = R_{i-1} + C_i \qquad \text{where } R_0 = 0$$
$$t_1 = I_i^{t_0} + C_i$$
$$t_2 = I_i^{t_1} + C_i$$
$$\ldots.$$
$$t_k = I_i^{k-1} + C_i$$

We note that all $t_0, t_1, \ldots, t_k \in [0, D_i]$. At each of these points in time t_k we evaluate (extending equation (4.14)):

$$C_i + I_i^{t_k} = t_k$$

where
$$I_i^{t_k} = \sum_{j=1}^{i-1} \left\lceil \frac{t_k}{T_j} \right\rceil C_j$$

If for any value of t_k the constraint holds, τ_i is feasible, otherwise τ_i is infeasible.

The complexity of sufficient and necessary testing is in general NP-complete (see Chapter 2). Indeed, the above approach has complexity $O(kn^2)$ where k represents the number of values of t required to determine feasibility (or infeasibility). We observe that for τ_i the maximum number of values of t required is D_i, giving $k = \max_{1 \le i \le n}(D_i)$. Although k is not a polynomial function of n, it is constant for a particular process set. Hence the complexity is pseudo-polynomial.

The feasibility test may be expressed by the algorithm in Figure 4.7. We note that the algorithm terminates since:

(i) $t_i > t_{i-1}$ for successive iterations unless $t_i = t_{i-1}$, in which case a solution has been found;

(ii) if $t_i > D_i$ the algorithm exits the main loop.

R_i records the response time for τ_i if that process has been found feasible. All R_i are assumed to be global variables. The calculation of I_i^t is according to equation (4.14). The function returns *n+1* if the process set is feasible; else the index of the process that first fails the test.

```
function response_test (Δ) return integer is
    feasible = TRUE      ;
    R0 = 0               ;
    i = 1                ;
begin
    while i ≤ n and feasible loop
        t = R(i-1) + Ci      ;
        while Ii^t + Ci > t and Ii^t + Ci ≤ Di loop
            t = Ii^t + Ci    ;
        end loop  ;
        if Ii^t + Ci > Di
            /* τi is infeasible force exit of loop */
            /* loop using feasible condition.      */
            feasible = FALSE    ;
        else
            /* τi is feasible note response time   */
            /* go to next process.                 */
            Ri = t      ;
            i = i + 1 ;
        end if    ;
end loop  ;
return i  ;
end response_test  ;
```

Figure 4.7: Algorithm for Response Time Feasibility Test.

Consider the following example, illustrating the behaviour of the test.

Example 4.7:

The feasibility of the process set in Table 4.6 as calculated by equation (4.12) is summarised in Table 4.7. We note that all processes are declared feasible.

	t	I_i^t	$I_i^t+C_i>t$	$I_i^t+C_i\leq D_i$	$I_i^t+C_i>t$ $\wedge$ $I_i^t+C_i\leq D_i$	Feasible	R_i
τ_1	1	0	$0+1=t$	$0+1<D_i$	False	✓	1
τ_2	3	1	$1+2=t$	$1+2<D_i$	False	✓	3
τ_3	6	4	$4+3>t$	$4+3<D_i$	True		
	7	6	$6+3>t$	$6+3<D_i$	True		
	9	7	$7+3>t$	$7+3<D_i$	True		
	10	7	$7+3=t$	$7+3<D_i$	False	✓	10
τ_4	14	13	$13+1=t$	$13+1<D_i$	False	✓	11

Table 4.7: Feasibility Summary for Process Set 6.

4.3.2 Exact Interference Sufficient And Necessary Feasibility Test

To calculate I_i we need to consider the releases of processes with higher priority than τ_i in [0, D_i). The pattern of processor use in the interval is as follows. Initially, an interval of execution of processes $\tau_1..\tau_{i-1}$ occurs. This is followed by an interval (possibly 0 length) of time when processes $\tau_1..\tau_{i-1}$ have no outstanding computation (although τ_i will execute if it has not completed). This is repeated until D_i is reached. The sum of the lengths of the intervals of execution of processes $\tau_1..\tau_{i-1}$ in [0, D_i) forms the exact interference I_i.

Trivially, the exact interference on τ_1 is 0. The exact interference of $\tau_2..\tau_n$ is now derived. Consider τ_i ($1<i\leq n$).

Definition 4.5:

The start of the j^{th} execution interval of $\tau_1..\tau_{i-1}$ in [0, D_i) is given by s_j^i (where j is 1 upwards).

Definition 4.6:

The end of the j^{th} execution interval of $\tau_1..\tau_{i-1}$ in $[0, D_i)$ is given by e_j^i (where j is 1 upwards).

Definition 4.7:

The term $I_i^{a,b}$ is the interference of processes with higher priority than τ_i in the interval $[a, b)$ where $0 \le a \le b \le D_i$:

$$I_i^{a,b} = \sum_{j=1}^{i-1} \left(\left\lceil \frac{b}{T_j} \right\rceil C_j - \left\lceil \frac{a}{T_j} \right\rceil C_j \right)$$

Thus, execution intervals of $\tau_1..\tau_{i-1}$ in $[0, D_i)$ are $[s_1^i, e_1^i), [s_2^i, e_2^i),...,[s_m^i, e_m^i)$ where $s_1^i < e_1^i \le ... \le s_m^i < e_m^i \le D_i$. We note that for a critical instant process set, $s_1^i = 0$ $(1 < i \le n)$. The last execution interval is constrained to end at or before D_i, since execution at or after D_i (i.e. in $[D_i, D_i+1)$ etc.) by $\tau_1..\tau_{i-1}$ is not part of I_i.

To calculate the bounds of the j^{th} execution interval, the following method is used. Let s_j^i be the start of the interval, with τ_a being one of the processes released at s_j^i $(1 \le a < i)$. The earliest point at which outstanding computation due to $\tau_1..\tau_{i-1}$ can reach 0, signalling the end of the execution interval, is given by:

$$t_0 = s_j^i + C_a$$

The interval of execution of higher priority processes ends at t_0 if no releases of $\tau_1..\tau_{i-1}$ have occurred in $[s_j^i, t_0)$. If such a release exists, the next point in time that the interval could end, t_1, is given by:

$$t_1 = s_j^i + I_i^{s_j^i, t_0}$$

Note that C_a forms part of $I_i^{s_j^i, t_0}$. A series of times can be found at which we check for the end of the execution interval:

$$t_0 = s_j^i + C_a$$

$$t_1 = s_j^i + I_i^{s_j^i, t_0}$$

$$t_2 = s_j^i + I_i^{s_j^i, t_1}$$

....

$$t_k = s_j^i + I_i^{s_j^i, t_{k-1}}$$

We note that all $t_0, t_1, ..., t_k \in [t_{r_j}^i, D_i]$. At each of these points in time t_k we evaluate

$$I_i^{s_j^i, t_k} + s_j^i = t_k \qquad (4.15)$$

Thus we check to see if the interval $[s_j^i, t_k)$ is long enough to contain all execution demanded by $\tau_1..\tau_{i-1}$ in $[s_j^i, t_k)$. If the constraint does not hold for all t_k, there is outstanding computation (due to $\tau_1..\tau_{i-1}$) at D_i, with the execution interval concluding after D_i. In this case, $e_j^i = D_i$. If the constraint does hold, $e_j^i = t_k$ (noting $t_k \leq D_i$). If $e_j^i < D_i$, the start of the next execution interval is given by:

$$s_{j+1}^i = \min_{1 \leq k < i} \left(\left\lceil \frac{e_j^i}{T_j} \right\rceil T_j \right)$$

If $s_{j+1}^i < D_i$ the end of this next execution interval is found, else no more execution intervals start before D_i. This progresses until the last execution interval $[s_m^i, e_m^i)$ is identified, where e_m^i is constrained to be no more than D_i.

The contribution of the j^{th} execution interval toward I_i is the length of the interval, i.e. $e_j^i - s_j^i$, giving:

$$I_i = \sum_{j=1}^{m} \left(e_j^i - s_j^i \right)$$

Now, feasibility of τ_i can be found by equation (4.1).

The above measure of I_i is exact, implying (by Theorem 4.1) that this approach for determining feasibility is sufficient and necessary.

The complexity of this approach is similar to the response time feasibility test given in section 4.3.1. Again, in the worst-case, the complexity is $O(kn^2)$ where

$$k = \max_{1 \leq i \leq n} \left(D_i \right)$$

Hence, the algorithm is pseudo-polynomial.

The test is illustrated by the algorithm in Figure 4.8. For each process, the execution intervals of higher priority processes are identified by the **while** `incomplete` **loop**. The function returns $n+1$ if all processes are feasible, else the priority of the first process to be found infeasible.

```
function interference_test (Δ) returns integer is
    feasible = TRUE      ;
    i = 1                ;
begin
    while i ≤ n and feasible loop
        incomplete = TRUE   ;
        t = 0       ;
```

```
I_i = 0      ;
while incomplete loop
      /* find next release of process     */
      /* τ_1..τ_{i-1} in [t, D_i)         */
      s = D_i      ;
      for j in 1..i-1 loop
            if ⌈t/T_j⌉T_j < s then
                  index = j ;
                  s = ⌈t/T_j⌉T_j
            end if    ;
      end loop  ;
      if s < D_i then
            /* find end of exec interval  */
            t = s + C_index    ;
            while I_i^{s,t} + s > t
                        and I_i^{s,t} + s ≤ D_i loop
                  t = I_i^{s,t} + s      ;
            end loop  ;
            if I_i^{s,t} + s > D_i then
                  /* end of exec interval at   */
                  /* or past D_i               */
                  I_i = I_i + (D_i - s)      ;
                  incomplete = FALSE   ;
            else /* end of exec interval       */
                  /* before D_i                */
                  I_i = I_i + (t - s)      ;
            end if    ;
      end if    ;
end loop  ;
if I_i + C_i ≤ D_i then  i = i + 1 ;
else feasible = FALSE    ;
      end if    ;
      end loop  ;
      return    i    ;
end interference_test    ;
```

Figure 4.8: Algorithm for Exact Interference Feasibility Test.

The test is illustrated in the following example.

Example 4.8:

Consider the process set in Table 4.8 (noting that τ_4 is declared infeasible by all sufficient not necessary tests in section 4.2).

Process	C	D	T
τ_1	2	3	5
τ_2	3	5	14
τ_3	2	10	15
τ_4	2	18	20

Table 4.8: Example Process Set 7.

In Table 4.9, the behaviour of the algorithm (Figure 4.8) is given for Process Set 7.

	s	t	$I_i^{s,t}+s>t$	$I_i^{s,t}+s\le D_i$	$I_i^{s,t}+s>t$ $\wedge$ $I_i^{s,t}+s\le D_i$	Interference of interval $\min(s-t, D_i-t)$	I_i	$I_i+C_i\le D_i$	Feasible	Idle $D_i-C_i-I_i$
τ_1	-	-	-	-	-	0	0	$0+2<D_i$	✓	1
τ_2	0	2	$2+0=t$	$2+0<D_i$	False	2				
	5	7	$2+5=t$	$2+5>D_i$	False	0	2	$2+3=D_i$	✓	0
τ_3	0	2	$5+0>t$	$5+0<D_i$	True					
		5	$5+0=t$	$5+0<D_i$	False	5				
	5	7	$2+5=t$	$2+7<D_i$	False	2	7	$7+2<D_i$	✓	1
τ_4	0	2	$7+0>t$	$7+0<D_i$	True					
		7	$9+0=t$	$9+0<D_i$	True					
		9	$9+0=t$	$9+0<D_i$	False	9				
	10	12	$2+10=t$	$2+10<D_i$	False	2				
	14	17	$7+14>t$	$7+14>D_i$	False	4	15	$15+2<D_i$	✓	1

Table 4.9: Feasibility Summary for Process Set 7.

The table shows that all processes are found feasible, with exact interference found. Consider τ_4 in the table: I_i=15 and 1 unit of idle time exists in [0, D_i) (time not required by $\tau_1..\tau_4$). This is confirmed by the TRCG for the interval and all four processes given in Figure 4.9. The idle time occurs in [13,14).

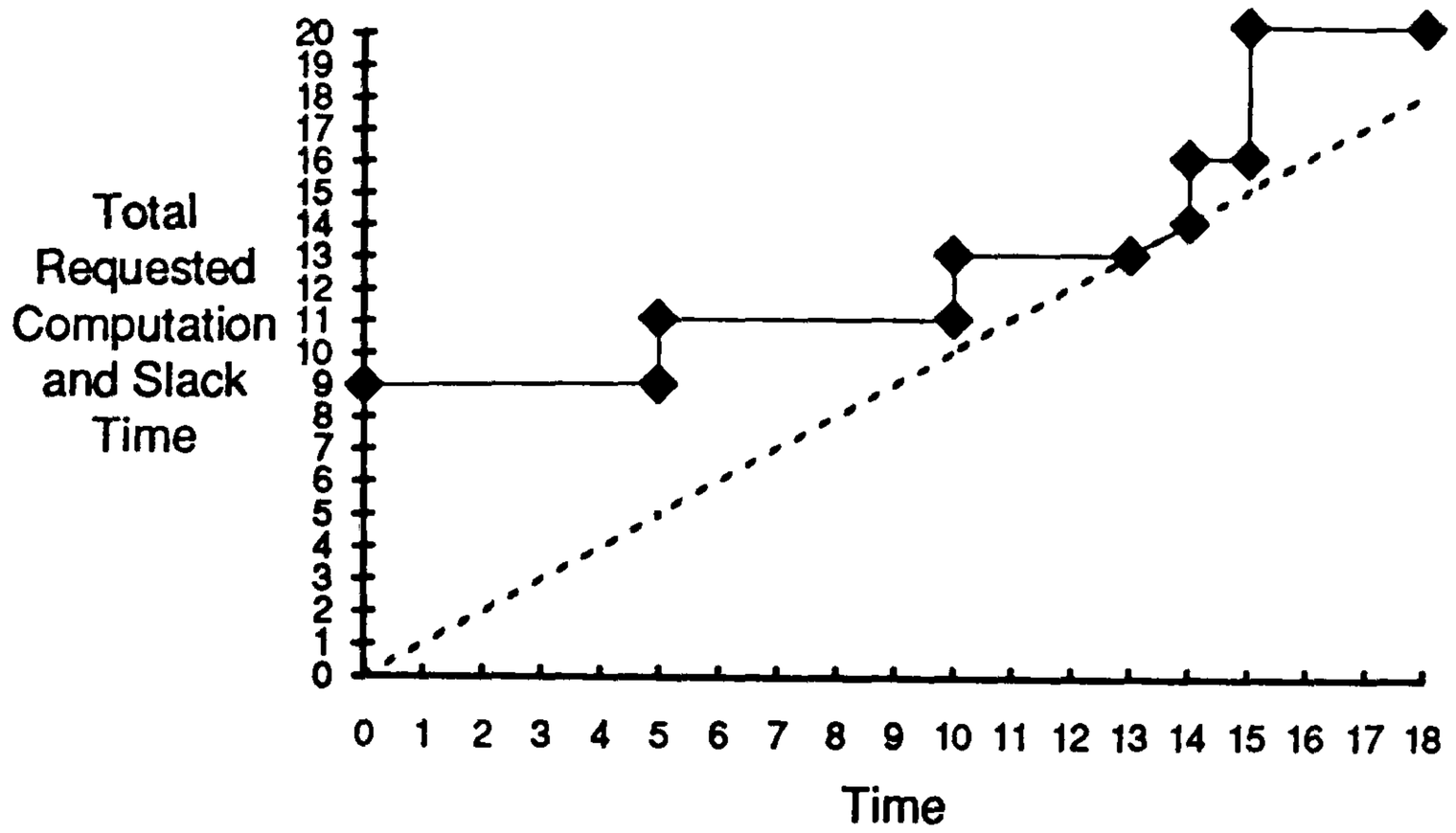


Figure 4.9: TRCG of Example Process Set 7.

4.3.3 Hybrid Sufficient And Necessary Feasibility Test

In some circumstances it is useful to know both the response time of τ_i and the exact interference on τ_i. Whilst this could be achieved by using both the response time and exact interference feasibility tests (sections 4.3.1 and 4.3.2 respectively), greater efficiency can be achieved by combining the tests. Firstly, the response test is used to find all R_i. Then, the mechanisms for identifying higher priority process execution intervals and slack intervals are used in $[R_i, D_i)$. We note that this test has NP-complete complexity (see sections 4.3.1 and 4.3.2). The test is summarised by the algorithm in Figure 4.10.

```
function hybrid_test (Δ) is
    feasible = TRUE      ;
    R_0 = 0              ;
    i = 1                ;
begin
    i = response_test (Δ)    ;
    if i ≤ n then
while i ≤ n and feasible loop
        incomplete = TRUE   ;
        t = R_i      ;
        I_i = I_i'      ;
        while incomplete loop
            /* find the next release of   */
```

```
/* process τ1..τi-1 in [t, Di)      */
s = Di      ;
for j in 1..i-1 loop
     if ⌈t/Tj⌉Tj < tr then
          index = j ;
          s = ⌈t/Tj⌉Tj
     end if     ;
end loop  ;
if s < Di then
     /* find end of exec interval */
     t = s + Cindex     ;
     while I_i^{s,t} + s > t
               and I_i^{s,t} + s ≤ Di loop
          t = I_i^{s,t} + s      ;
     end loop  ;
     if I_i^{s,t} + s > Di then
          /* end of exec interval */
          /* at or past Di          */
          Ii = Ii + (Di - s)     ;
          incomplete = FALSE  ;
     else /* end of exec interval */
          /* before Di              */
          Ii = Ii + (t - s)      ;
     end if     ;
   end if     ;
  end loop  ;
  i = i + 1 ;
 end loop  ;
end if     ;
return     i     ;
end hybrid_test     ;
```

Figure 4.10: Algorithm for Hybrid Feasibility Test.

4.3.4 Summary

This section has introduced two forms of sufficient and necessary feasibility test. The first calculates maximum response time, the second exact interference. Motivation for the actual form (and implementation) of the tests is efficiency. This is achieved by reducing the (non-polynomial) number of time points considered when determining feasibility. The two tests were also combined to produce a hybrid test calculating both response time and exact interference.

Further improvements in efficiency could be made by incorporating the effective deadline calculation of sections 4.2.3 and 4.2.4.

4.4 Feasibility Of Sporadic Processes

Processes whose releases are not periodic in nature are useful for responding to non-periodic environmental events, e.g. alarms raised. The general characteristic of these events is a short deadline, by which time it is necessary to complete some given computation, and a relatively long time between successive events. Non-periodic processes can be placed into two categories according to the nature of their release times [Burns91a]: aperiodic and sporadic. Aperiodic processes are those whose release frequency is unbounded. In the extreme, this could lead to an arbitrarily large number of simultaneously active processes. Sporadic processes are those that have a maximum frequency with the implication that a finite bound can be placed upon the number of instances of a sporadic process active at any time.

When a static scheduling algorithm is employed, it is difficult to introduce non-periodic process executions into the schedule: it is not known before the system is run when non-periodic processes will be released. More difficulties arise when attempting to guarantee the deadlines of these processes. It is clearly impossible to guarantee the deadlines of aperiodic processes as there could be an arbitrarily large number of them active at any time. Sporadic processes deadlines can be guaranteed since it is possible, by means of the maximum release frequency, to define the maximum workload they place upon the system.

One approach is to use static periodic polling processes to provide sporadic processes with execution time. This approach is reviewed in section 4.4.1. Section 4.4.2 illustrates how the properties of sporadic processes enable their deadlines to be guaranteed by the scheduling approach and feasibility

tests outlined in this Chapter. Finally, section 4.4.3 discusses provision of processor time to service aperiodic processes.

4.4.1 Sporadic Processes: the Polling Approach

To allow sporadic processes to execute within the confines of a static schedule (e.g. static priority pre-emptive scheduling) computation time must be reserved within that schedule. An intuitive solution is to set up a periodic process which polls for sporadic processes. Strict polling reduces the bandwidth of processing as:

- processing time that is embodied in an execution of the polling process is wasted if no sporadic process is active when the polling process becomes runnable;
- sporadic processes occurring after the polling process's computation time in one period has been exhausted have to wait until the next period for service.

A number of bandwidth preserving algorithms have been proposed for use with the rate-monotonic scheduling algorithm [Lehoczky87, Sprunt88, Sha89, Sprunt90]. These algorithms are founded upon a periodic server process being allotted a number of units of computation time per period. When an aperiodic process is released, it uses the computation time guaranteed to the server for execution. If no aperiodic process requires time when the server is released, its computation time is preserved whilst permitting other periodic processes to execute. The computation time for the server is replenished at the start of its period.

Problems arise when sporadic processes require deadlines to be guaranteed. It is difficult to accommodate these within periodic server processes due to the rigidly defined points in time at which the server computation time is replenished. The sporadic server [Sprunt88] provides a solution to this problem. The replenishment times are related to when the sporadic uses computation time rather than merely at the period of the server process. However, this approach still requires additional processes with obvious extra overheads.

4.4.2 Sporadic Processes: the Deadline Monotonic Scheduling Approach

Consider the timing characteristics of a crucial sporadic process τ_s. The demand for computation time is illustrated in Figure 4.11. The minimum time difference between successive releases of τ_s is the minimum inter-arrival time m. This occurs between the first two releases of τ_s. At this point, τ_s is behaving exactly like a periodic process with period m [Mok83]: the sporadic is being released at its maximum frequency and so is imposing its maximum workload. When its releases do not occur at the maximum rate (between the second and third releases in Figure 4.11) τ_s behaves like a periodic process that is intermittently activated and then laid dormant. The workload imposed by the sporadic is at a maximum when the process is released at its maximum rate, but falls when the next release occurs after greater than m time units have elapsed.

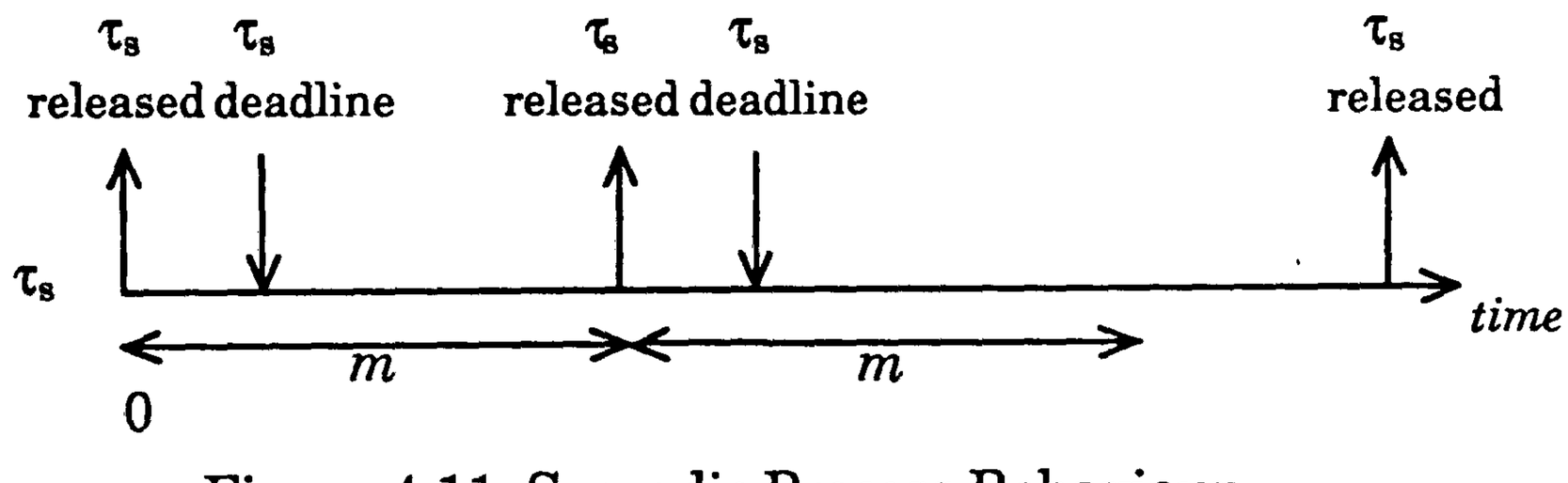


Figure 4.11: Sporadic Process Behaviour

In the worst-case the τ_s behaves exactly like a periodic process with period m and deadline D_s where $D_s \leq m$. The characteristic of this behaviour is that a maximum of one release of the process can occur in any interval $[t, t+m)$ where release time t is at least m time units after the previous release of the process. This implies that to guarantee the deadline of the sporadic process the computation time must be available within the interval $[t, t+D_s)$ noting that the deadline will be at least m after the previous deadline of the sporadic. This is exactly the guarantee given by the feasibility tests in sections 4.2 and 4.3.

For feasibility purposes only, we can describe the sporadic process as a periodic process whose period is equal to m. However, we note that since the process is sporadic, the actual release times of the process will not be periodic, but successive releases will be separated by no less than m time units.

For the feasibility tests given in sections 4.2 and 4.3 to be applicable for process sets containing both periodic and sporadic processes, it is assumed that at some instant all processes are released simultaneously (i.e. a critical instant). If the deadline of the sporadic can be guaranteed for the release at a critical instant then all subsequent deadlines are guaranteed. No limitations on the combination of periodic and sporadic processes are imposed by this scheme. Indeed, the approach is optimal for a fixed priority scheduling since sporadic processes are treated in exactly the same manner as periodic processes. All feasibility tests outlined in sections 4.2 and 4.3 are suitable for use with sporadic processes. To improve the responsiveness of sporadic processes their deadlines can be reduced to the point at which the system becomes infeasible.

4.4.3 Aperiodic Processes

Periodic server processes that provide a limited processor resource for sporadic process executions, as described in Chapter 2 and section 4.4.1 above, were originally defined to allow aperiodic processes a guaranteed proportion of processor utilisation. In particular, the priority exchange and deferrable server approaches [Lehoczky87, Sprunt88, Sprunt90], together with the extended priority exchange algorithm [Sprunt88] and the sporadic server [Sha89, Sprunt90] enable greatly improved aperiodic process response times compared to background processing (i.e. service aperiodic processes when the processor is idle) and strict periodic polling. Of these approaches the sporadic server has been shown to provide the best service for aperiodic processes [Sha89, Sprunt90].

Consider the following theorem:

> *"Theorem 2: A periodic [process] set that is [feasible] with a [process] τ_i, is also [feasible] if τ_i is replaced by a sporadic server with the same period and execution time."* [Sha89]

Hence, no extensions to the feasibility analysis (and tests) given in this chapter are necessary if sporadic servers are incorporated to provide some processor utilisation for aperiodic processes.

4.5 Process Blocking

In realistic hard real-time systems, processes interact in order to satisfy system-wide requirements and to share resources. Two forms of interaction are simple synchronisation and mutual-exclusion style protection of a shared resource. Both forms cause process blocking (see Chapter 2) with the additional problem of priority inversion [Sha90] occurring under static priority process scheduling. Priority inversion can be avoided by use of one of the family of priority inheritance protocols [Sha90] (see section 2.3.5). The effect on feasibility analysis by the adoption of any of these protocols is identical. Essentially, the blocking that any one process may receive can be bounded to B_i.

Now, the basic feasibility constraint (equation (4.1)) becomes:

$$C_i + I_i + B_i \leq D_i \tag{4.16}$$

Thus, for each process τ_i we guarantee that it can be blocked for its worst-case blocking time and still meet its deadline.

The sufficient and not necessary feasibility tests detailed in section 4.2 are easily extended to incorporate this blocking. For example, the test in section 4.2.1 can be restated:

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

$$\text{where} \qquad I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

The sufficient and necessary tests given in section 4.3 can also be extended, noting that the response time of τ_i includes B_i implying that the actual earliest completion time, t_0, of τ_{i+1} is given by:

$$t_0 = R_i - B_i + C_{i+1} + B_{i+1}$$

We may restate the feasibility test in section 4.3.1:

$$C_i + I_i^{t_k} + B_i = t_k$$

$$\text{where} \qquad I_i^{t_k} = \sum_{j=1}^{i-1} \left\lceil \frac{t_k}{T_j} \right\rceil C_j$$

As noted in Chapter 2, the estimation of B_i given by Sha *et al* [Sha90] is pessimistic: the assumptions inherent in the analysis allow for situations that can never occur. The following section examines how process timing characteristics can be used to eliminate certain forms of blocking pessimism.

4.5.1 Reducing B_i Pessimism By Consideration Of Timing Characteristics

The calculation of the worst-case blocking time, B_i, of process τ_i is the same for the priority ceiling protocol and any of its derivatives. B_i is the length of the longest critical region of any lower priority process, where that critical region corresponds to the locking and unlocking of a resource shared with a process of equal or higher priority τ_i [Sha90]. The reason behind this estimation is that a low priority process may lock a resource momentarily before τ_i is released, with τ_i assumed to require that resource. This is illustrated in Figure 4.12.

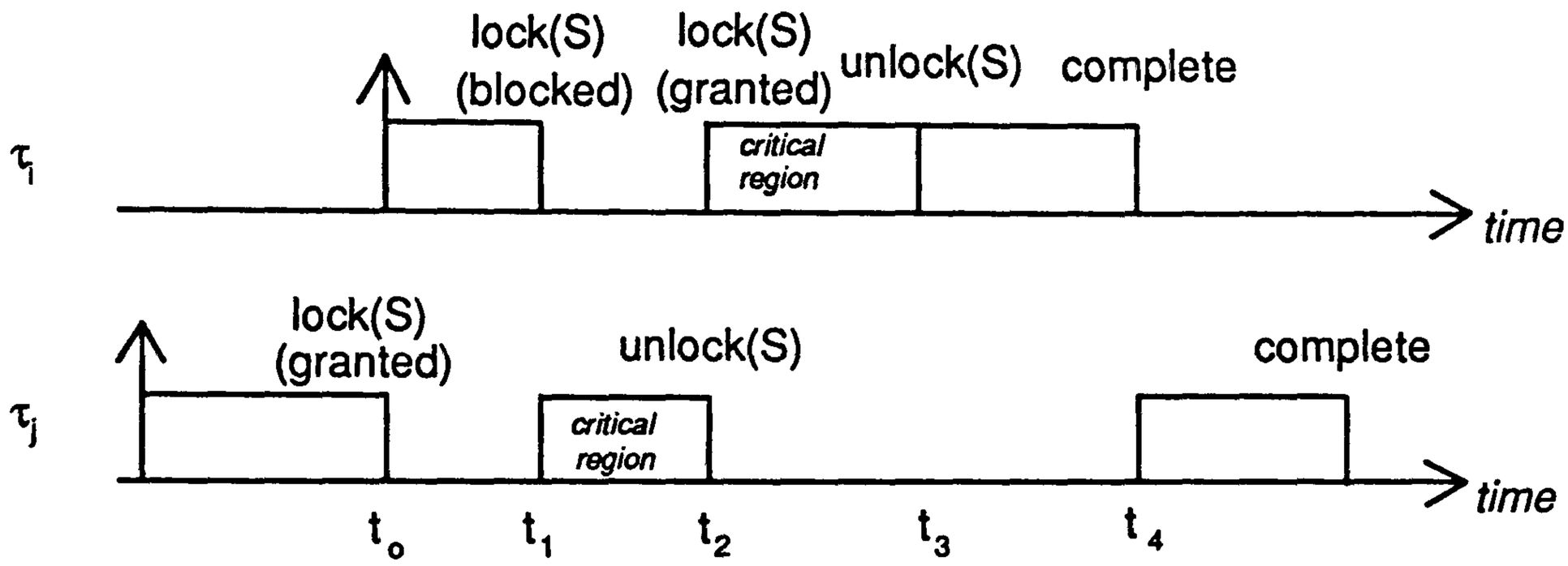


Figure 4.12: Worst-Case Process Blocking.

Process τ_j is of lower priority than τ_i (i.e. $i < j$). At time t_0 τ_j locks semaphore S and is immediately pre-empted by τ_i. At t_1 τ_i attempts to lock S and becomes blocked. τ_j inherits the priority of τ_i and executes its critical region, unlocking S at t_2 and returning to its original priority. Now, τ_i pre-empts τ_j and executes, locking S, finishing its critical region at t_3. Process τ_i completes execution at t_4 at which point τ_j is free to execute.

Consider the following definition:

Definition 4.8:

LCR_j^i is the length of the longest critical region of τ_j which locks and holds a resource whose priority ceiling is higher than or equal to the priority of τ_i ($j < i$).

Hence, B_i as defined under the priority ceiling protocol is given by:

$$B_i = \max_{\tau_j \in \{\tau_{i+1}, \ldots, \tau_n\}} (LCR_j^i) \tag{4.17}$$

In general this method of determining B_i is pessimistic in that it accounts for situations that may never occur. For example, if a higher priority and low

priority process are initially released at time 0, with the period of one process a multiple of the other, the higher priority process could never be blocked by the lower priority process. We note that a release of τ_i at t can only be blocked by a release of τ_j at t' $(j > i)$ if the following conditions hold:

(i) $t' < t < t' + D_j$

(ii) τ_j has actually executed in $[t', t)$;

(iii) τ_j has locked a resource in $[t', t)$ whose priority ceiling is at least i (i.e. the priority ceiling of the resource is numerically no more than i) and holds the lock on the resource at t.

For a release of τ_i at t the value of process periods and deadlines can be used to determine which lower priority processes are released before t but have a deadline after t (i.e. they may be runnable at t). For example, if the following holds, τ_j fulfils condition (i) above:

$$\left\lfloor \frac{t}{T_j} \right\rfloor T_j < t < \left\lfloor \frac{t}{T_j} \right\rfloor T_j + D_j$$

We assume that if condition (i) holds, τ_j executes in the interval $[\lfloor t/T_j \rfloor T_j, t)$ locking all resources that it requires with locks still held at t. If τ_j requires a resource whose ceiling priority is at least the priority of τ_i it will be locked at t. Hence conditions (ii) and (iii) are met. Consider the following definitions:

Definition 4.9:

HPR_i is the maximum of the priority ceilings of resources locked by τ_i.

Definition 4.10:

LR_i^t is the set of processes of lower priority than τ_i that access a resource with a priority ceiling of greater or equal priority than τ_i and are potentially runnable at t:

$$LR_i^t = \left\{ \tau_j \in \Delta \mid HPR_j \leq i \wedge \left\lfloor \frac{t}{T_j} \right\rfloor T_j < t < \left\lfloor \frac{t}{T_j} \right\rfloor T_j + D_j \right\}$$

We note that the right-hand clause in the conjunct identifies those lower priority processes that could possibly execute at t. The definition precludes lower priority processes released exactly at t and those where t lies in the interval bounded by the deadline of one release and the next release time.

Consider the worst-case blocking time for each release of τ_i separately. The maximum of these blocking times forms B_i. We note that the phasing of process requests repeats at intervals defined by the least common multiple

(LCM) of all process periods. Let the LCM of the periods of processes in Δ be denoted L^{Δ}. Now, B_i may be defined:

$$B_i = \max_{t \in \{0, T_i, 2T_i, \ldots, L^{\Delta}\}} \left(\max_{\tau_j \in LR_i^t} (LCR_j^i) \right) \tag{4.18}$$

Whilst this definition is less pessimistic than originally defined for the PCP (equation (4.17)), it still caters for situations that can never occur. Let the maximum blocking time occur at a release of τ_i at t and be due to τ_j. If the following condition holds, the estimation of B_i above may include part of a critical region execution that could not possibly occupy $[t, \lfloor t/T_j \rfloor T_j + D_j)$:

$$\left\lfloor \frac{t}{T_j} \right\rfloor T_j + D_j - t \leq C_j$$

Therefore, we may improve the estimation of B_i:

$$B_i = \max_{t \in \{0, T_i, 2T_i, \ldots, L^{\Delta}\}} \left(\max_{\tau_j \in LR_i^t} \left(\min \left(LCR_j^i, \left\lfloor \frac{t}{T_j} \right\rfloor T_j + D_j - t \right) \right) \right) \tag{4.19}$$

The inherent assumption is that all processes have computation time no greater than deadline.

Example 4.9:

Process	C	D	T
τ_1	1	3	10
τ_2	3	4	8

Table 4.10: Example Process Set 8.

Consider the process set in Table 4.8. Processes τ_1 and τ_2 share a resource, with the length of the critical region of τ_2 being 3. Thus, according to the priority ceiling protocol estimation of blocking (equation (4.17)) $B_1 = 3$, with the consequence that τ_1 would be declared infeasible as $B_1 + C_1 + I_1 > D_1$. Consider the evaluation of B_1 by equation (4.19):

Given $L^{\Delta} = 40$, we have $t \in \{0,10,20,30,40\}$

We note that $LR_1^0 = LR_1^{20} = LR_1^{30} = LR_1^{40} = \{\}$ and $LR_1^{10} = \{\tau_2\}$

Hence:

$$B_1 = \min \left(LCR_2^1, \left\lfloor \frac{10}{8} \right\rfloor 8 + 4 - 10 \right) = \min(3,2) = 2$$

Using equation (4.16), we observe that τ_1 is feasible as $B_1 + C_1 + I_1 = D_1$

In the above discussion the blocking time of (any) τ_j upon τ_i is determined in isolation: the effects of higher priority process executions are not

considered. This is pessimistic since a higher priority process may prevent τ_j from execution in $[\lfloor t/T_j \rfloor T_j, t)$ so breaking condition (ii) (i.e. τ_j cannot block τ_i in this release). Also, the analysis assumes that if τ_j executes in $[\lfloor t/T_j \rfloor T_j, t)$ it locks all resources it as its last action whilst executing in the interval, so that it has still to execute its critical region when pre-empted by τ_i at t.

To counter these problems, blocking time analysis could be made more complex. Consider a release of τ_i at t and a release of τ_j at t' ($j > i$), where $t' < t < t' + D_j$. To determine whether τ_j could block τ_i requires an evaluation of the schedule over the interval $[0, t)$ to see if τ_j actually executes in $[\lfloor t/T_j \rfloor T_j, t)$. Whilst evaluating the schedule minimum execution times are assumed for all processes to enable τ_j to have the maximum possibility of executing in the interval. Assume that τ_j can execute in the interval, having a minimum of t_{min} execution time (in the interval). For each resource access made by τ_j, there is a minimum time within the execution of τ_j before that access is made (determined during worst-case execution time analysis). If the minimum time before τ_j could access a resource is at least t_{min}, that resource access could not block τ_i (during its release at t).

This approach would be computationally expensive (similar to the sufficient and necessary tests in section 4.3). It would only be required if processes were infeasible assuming the assessments of blocking given by equations (4.17), (4.18) or (4.19). We note that the schedule would have to be evaluated once only, over the interval $[0, L^{\Delta})$.

Consider systems consisting of a mixture of sporadic and periodic processes. The worst-case blocking upon a periodic process is the maximum of the worst-case blocking due to sporadic processes (by equation (4.17)) and that due to periodic processes (by equation (4.19)). The worst-case blocking upon a sporadic process is calculated by equation (4.17).

4.6 Infeasibility Analysis

The accuracy/complexity trade-off described in Chapter 3 has been illustrated by the feasibility tests of sections 4.2 and 4.3. Sufficient and not necessary tests were seen to have polynomial complexity. In contrast, the more accurate sufficient and necessary tests have pseudo-polynomial complexity. One approach to determining feasibility of a process set using these tests is to use a

sufficient and not necessary test initially. If the process set fails this test, an exact feasibility test may be used.

An alternative approach is to use sufficient and not necessary infeasibility tests. Such tests have been given little coverage in the literature. Consider Figure 4.13. A sufficient and not necessary infeasibility test identifies some infeasible process sets in the same manner as a sufficient and not necessary feasibility test identifies some feasible sets. Now, we may initially try a sufficient and not necessary feasibility test. If the process set fails this test, sufficient and not necessary infeasibility can be used. If this test is also failed, the process set may still be feasible, so a computationally expensive sufficient and necessary test may be employed.

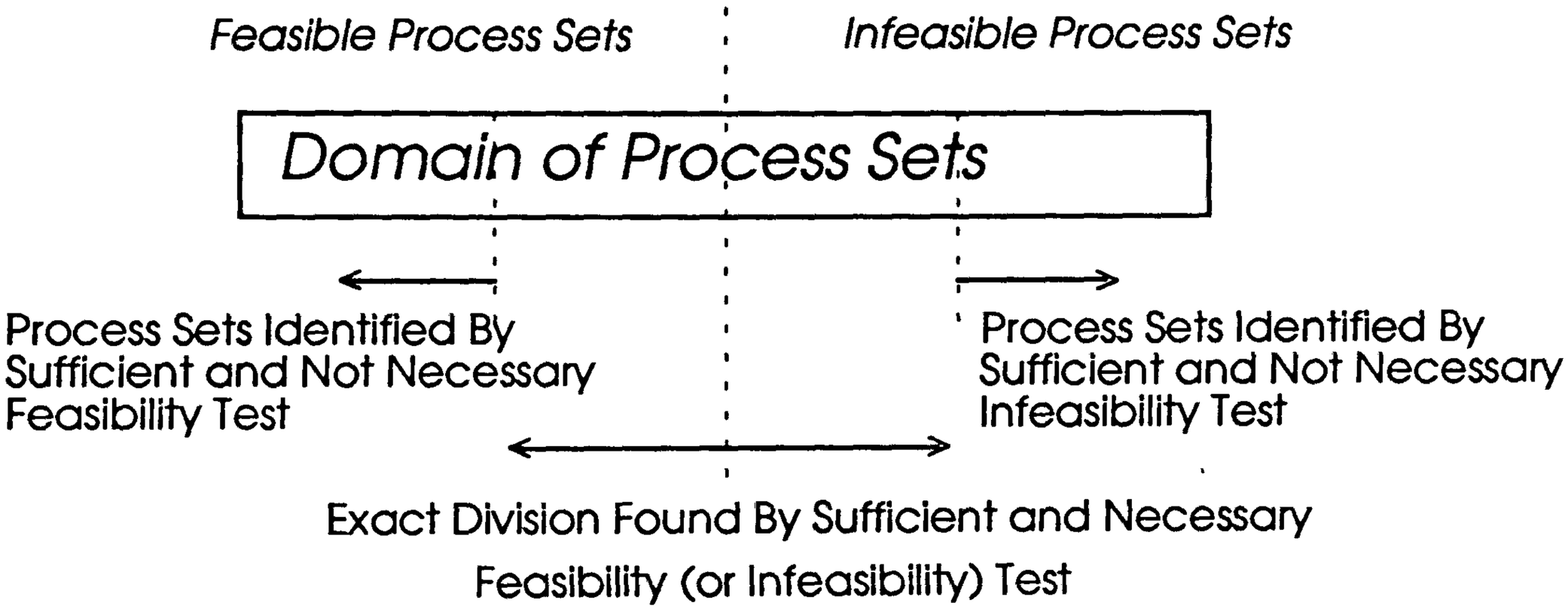


Figure 4.13: Feasibility and Infeasibility.

There are many approaches available for the development of sufficient and not necessary infeasibility tests. Essentially, a basic infeasibility test can be stated:

$$\forall \tau_i : 1 \le i \le n \bullet \quad \exists \tau_i \in \Delta \bullet \; C_i + I_i > D_i \tag{4.20}$$

This states that if one process is definitely infeasible, the entire process set is also infeasible. The value of I_i in the test must be no greater than the exact interference. Consider the following theorem:

Theorem 4.8:

If the estimated interference I_i is less than the exact interference $I_i^{'}$ the infeasibility test given by equation (4.20) is sufficient and not necessary.

Proof:

Converse of Theorem 4.2.

We note that if I_i is exact, the infeasibility test (equation (4.20)) is sufficient and necessary, and is therefore equivalent to a sufficient and necessary feasibility test. This observation is seen in Figure 4.12.

A basic estimation of a value of I_i that is less than the exact interference is 0. This reduces the infeasibility test to a check to see if the computation time of a process is greater than the deadline. Clearly, if such a process exists the process set is infeasible. Formally:

$$\exists \tau_i \in \Delta \bullet \; C_i > D_i \tag{4.21}$$

The test has complexity O(n) in the number of processes, although it is inaccurate: there are many process sets that fail the test that are infeasible.

The following sections introduces a more accurate infeasibility test. Then, the effects of sporadic processes and process blocking are considered.

4.6.1 Sufficient And Not Necessary Infeasibility Test

The accuracy of the infeasibility test given by equation (4.21) is now improved. A critical instant is assumed. The interference of τ_j on τ_i is exact for all releases except the last in $[0, D_i)$. Let the final release of τ_j in $[0, D_i)$ execute as late as possible.

Accuracy can be improved further by adopting the effective deadline strategy of sections 4.2.3 and 4.2.4. Now, $\left[d^i_{i-1,k}, D_i\right)$ is occupied by processes of higher priority than τ_i, with the exact interference for this interval being $D_i - d^i_{i-1,k}$. Now, it is assumed that the final release of τ_j in $\left[0, d^i_{i-1,k}\right)$ executes as late as possible. Therefore, the test may be stated:

$$\exists \tau_i \in \Delta \bullet \; C_i + I_i > D_i$$

where

$$I_i = D_i - d^i_{i-1,k} + \sum_{j=1}^{i-1} \left(\left\lfloor \frac{d^i_{i-1,k}}{T_j} \right\rfloor C_j + \min\left(C_j, \max\left(0, d^i_{i-1,k} - \left\lfloor \frac{d^i_{i-1,k}}{T_j} \right\rfloor T_j + D_j - C_j \right) \right) \right)$$

and where $d^i_{j,k}$ is defined by equation (4.12).

The test is sufficient and not necessary since the value of I_i is no greater than the exact interference (by Theorems 8 and 9).

The complexity of the test has increased to be pseudo-polynomial (see section 4.2.4) from the polynomial complexity of the test in the previous section, although greater accuracy has been achieved.

4.6.2 Process Blocking, Sporadic Processes and Infeasibility Tests

Extensions to permit sporadic processes to be incorporated within the infeasibility test could be achieved in a number of ways. When considering their interference on periodic processes, sporadic processes can be treated as not occurring or as periodic. The latter case presents a more accurate interference value. The interference of periodic (or sporadic) processes on sporadic processes can be calculated as if the latter processes are periodic.

Blocking can be included in infeasibility analysis in several ways. It could be assumed that no blocking occurs (i.e. $B_i = 0$). A more accurate alternative would be to calculate potential blocking in the manner outlined in section 4.5.

4.7 Comparison Of Feasibility Tests

The offline accuracy / complexity trade-off observed in the development of feasibility tests presented in this chapter is further explored in this section. Firstly relative efficiencies of the sufficient and necessary tests are considered. Secondly, the differing accuracies of the sufficient and not necessary tests are examined in conjunction with their differing complexities.

The feasibility (and infeasibility) tests maybe compared in four ways.

(i) *accuracy* - provides a measure of the number of process sets that are declared feasible by a given test.

(ii) *time points* - this measures the number of points in time that are examined by a test when determining the feasibility of a process set.

(iii) *sample complexity* - when a time point is considered, sample complexity is a measure of the complexity of the feasibility test at that point.

(iv) *overall complexity* - the combined complexity derived by multiplying the number of time points considered by sample complexity.

Accuracy is of interest when comparing sufficient and not necessary feasibility (and infeasibility) tests, assuming that sufficient and necessary tests have equivalent accuracies. The other three criteria are useful when discussing the complexity and efficiency of the tests. For sufficient and necessary tests, the relative efficiencies in determining feasibility can be examined by considering time points, sample and overall complexities. For example, the explicit construction of a schedule [Leung82] examines many time points at a minimal sample complexity (the overall complexity being mostly dependent upon the number of time points examined). In comparison, the tests described in section 4.3 limit the number of time points, at a cost of increased sample complexity.

In general, we may consider each of the above criteria in terms of both worst and average-case performance. The former reflects theoretical computational complexity (derived from the worst-case). The latter can be achieved by measuring performance of the tests with respect to randomly generated process sets. Essentially, four free variables exist when generating such a process set:

(i) number of processes;
(ii) process computation times;
(iii) process deadlines;
(iv) process periods.

Sporadic processes and process blocking were not considered as they do not affect the performance of the feasibility tests.

Within this section process sets were generated using normal and uniform distributions of process deadlines, periods and computation times (within given ranges). Some process sets had processes whose periods were related. This reflects the often observed characteristic of hard real-time systems that a process period is sometimes a multiple of another process's period, due to hardware constraints. Further details of random process set generation are given in Appendix A.

The following sections compare the efficiencies of sufficient and necessary tests, and examine the complexity / accuracy trade-off of sufficient and not necessary tests.

4.7.1 Comparison of The Efficiencies Of Sufficient and Necessary Feasibility Tests

Sufficient and necessary feasibility tests have equivalent accuracy, hence in this section only the relative efficiencies of tests for worst and average-cases

need to be examined. A number of algorithms have been proposed for determining sufficient and necessary feasibility for critical instant process sets (see Chapter 2 and section 4.3):

Leung's Algorithm (LA) [Leung82] : Construction of a schedule over $[0, D_n)$ for processes τ_i $(1 \leq i < n)$.

Lehoczky's Algorithm (LZA) [Lehoczky90] : For each τ_i each point $t \in [0, D_i)$ is examined to see if the computation demands of $\tau_i..\tau_i$ in $[0, t)$ is no greater than t.

Nassor and Bres's Algorithm (NBA) [Nassor91] : This is derived from the exact feasibility test derived for rate-monotonic analysis given in [Lehoczky89]. The algorithm maintains an event list of releases and deadlines of processes, checking each process deadline event against cumulative demands of released processes (of equal or higher priority).

Joseph's Algorithm (JA) [Joseph86] : This is based upon similar analysis to that given in section 4.3.1 considering releases of processes in $[0, D_i)$ when determining the feasibility of τ_i.

In addition to the above algorithms, the algorithm given in section 4.3.1 to find response time is considered, referred to as RA.

The worst-case performance of the algorithms is given by their theoretical complexity. This can be considered in terms of time points, sample and overall complexity. Table 4.11 summarises the above algorithms with respect to these three criteria.

Worst-Case Complexities	LA	LZA	NBA	JA	RA
Time-Point	$O(kn)$	$O(kn)$	$O(kn)$	$O(kn)$	$O(kn)$
Sample	$O(1)$	$O(n)$	$O(nlog_2n)$	$O(n)$	$O(n)$
Overall	$O(kn)$	$O(kn^2)$	$O(kn^2log_2n)$	$O(kn^2)$	$O(kn^2)$

Table 4.11: Worst-Case Complexities of Sufficient and Necessary Feasibility Tests.

In the table $k=\max_{1\leq i\leq n}(D_i)$ (for processes with deadline-monotonically assigned priorities $k=D_n$).

For LA, the worst-case time point complexity arises if when allocating the C_i^{th} slot to a release of τ_i, D_i slots need to be examined. The sample complexity is minimal, being the cost of examining a schedule slot. Thus the overall complexity is equivalent to the time point complexity.

The entries for LZA reflect that to determine the feasibility of τ_i each $t \in [0, D_i)$ may be considered. The sample cost at each time point reflects the need to add the total processor demands made by $\tau_1..\tau_i$ in $[0, t)$.

For NBA the worst-case time point complexity reflects the maximum number of events (i.e. releases and deadlines) of $\tau_1..\tau_i$ in $[0, D_i)$. This equates to $(i\text{-}1)D_i$ if all processes $\tau_1..\tau_{i-1}$ have period or deadlines set to 1. At each time point, an event is removed from an ordered (by time) queue (and one added). This has complexity $O(qlog_2q)$ in the number of items in the queue [Sedgewick83]. In this case, $q=n$ in the worst-case.

For JA and RA the time point complexity reflects the need to examine each $t \in [0, D_i)$ in the worst-case (although this ignores the limiting of time points to $D_i - R_{i-1}$ in RA). At each time point, the computational demand of each higher priority process is calculated.

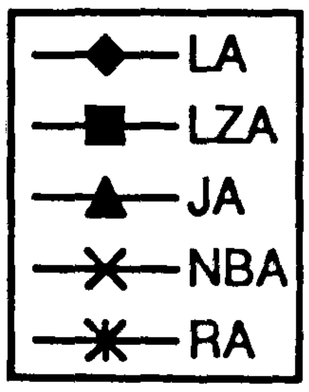
Although the overall complexities are similar, the worst-case complexity of tests is only realised in constrained and differing circumstances. For example, LA approaches its worst-case as process set utilisation approaches 100%. In contrast, RA has a worst-case if exactly one of $\tau_1..\tau_{i-1}$ is released at each $t \in [0, D_i)$, with the computation time of the released process being 1.

To gain greater insight into the respective performance of each algorithm, the average-case behaviour is examined. The average-case efficiencies of the algorithms were considered using approximately 2000 feasible process sets, randomly generated using both uniform and normal distributions of timing values (further details are given in Appendix A). Graphs 4.1 to 4.14 show the number of time points, overall complexity and actual time[1] taken by the algorithms plotted against the utilisation of the process set, for process sets of cardinality 10, 30, 50 and 100. Process sets with utilisation between 0 and 10% are plotted at y-axis co-ordinate 5; between 10% and 20% are plotted at y-axis co-ordinate 15 etc. Whilst the time point complexity of LZA is plotted, the overall complexity and actual time taken by

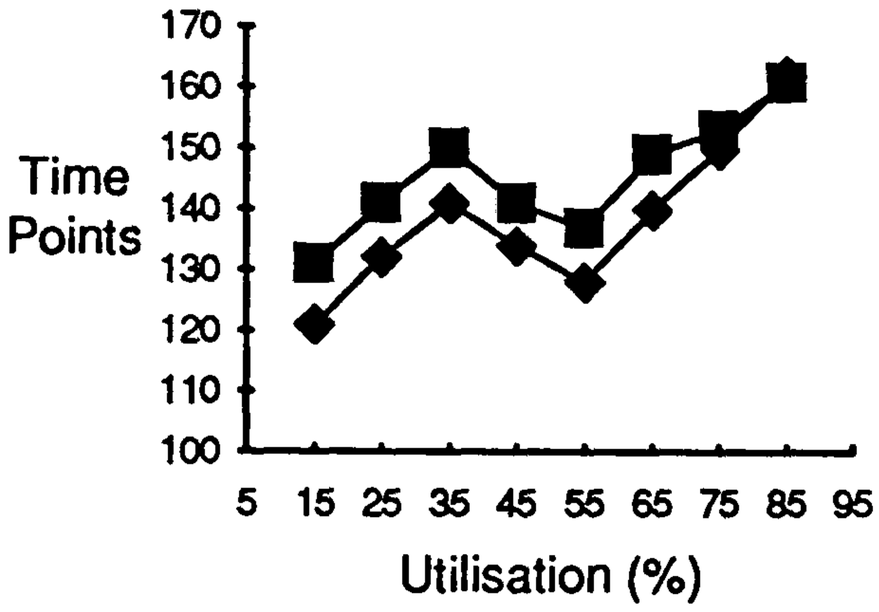
---

[1]Acutal time taken on an Intel 486 processor running at 33MHz.

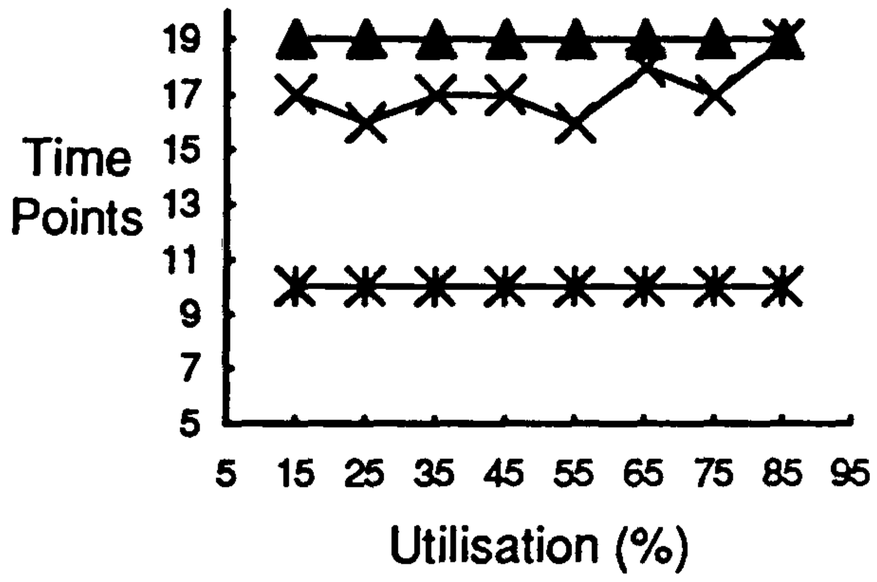
LZA is not plotted, being an order of magnitude greater than any of the other algorithms.



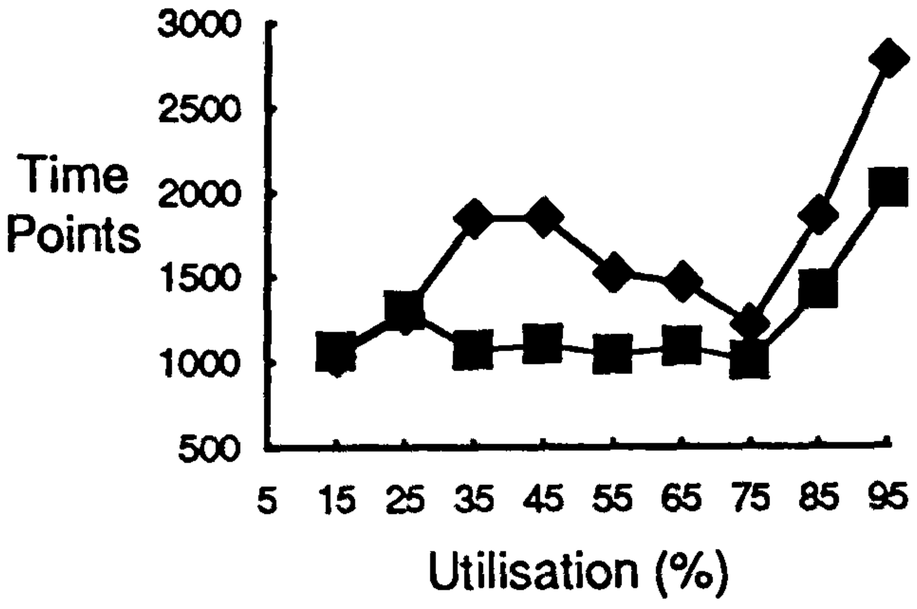
Key for Graphs 4.1 - 4.14.



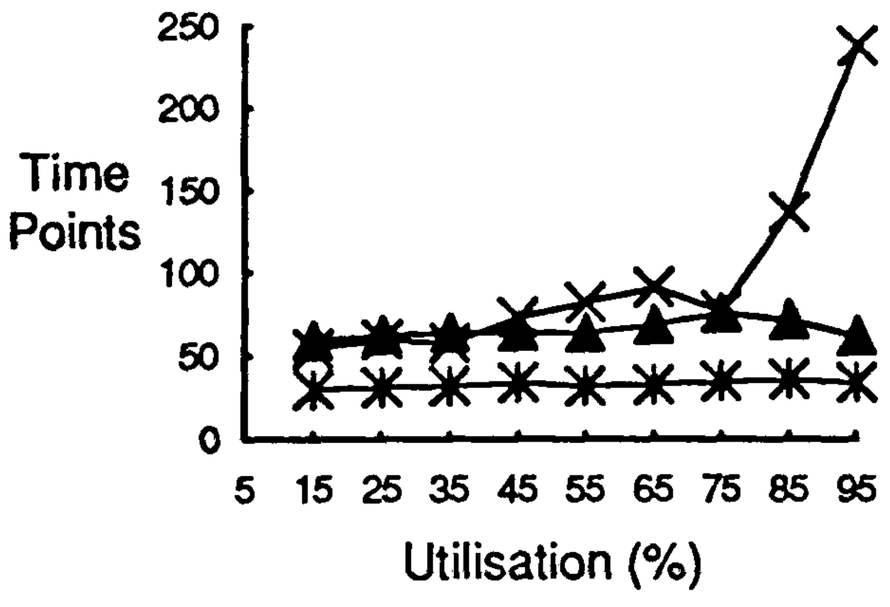
Graph 4.1: Time Points for 10 Processes (LA and LZA).



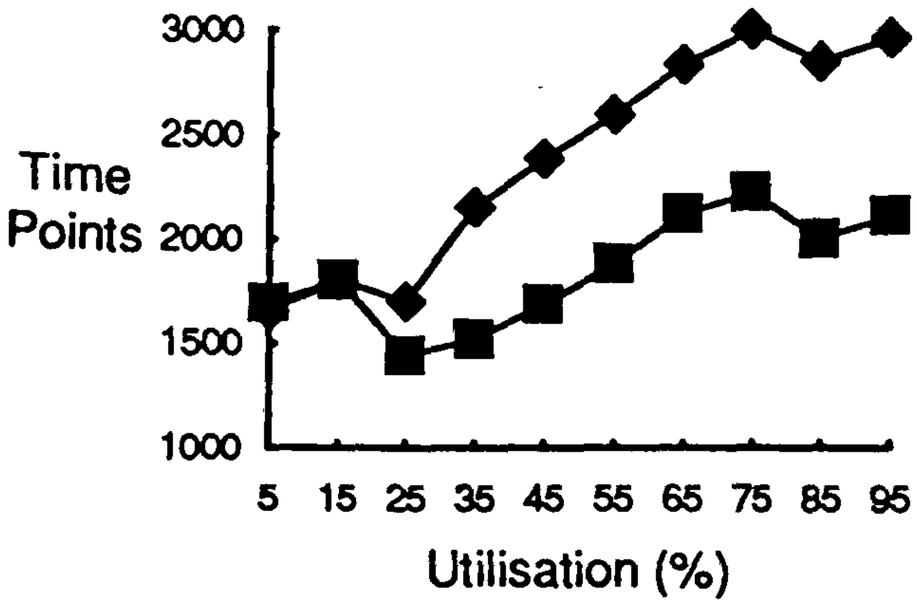
Graph 4.2: Time Points for 10 Processes (JA, NBA and RA).



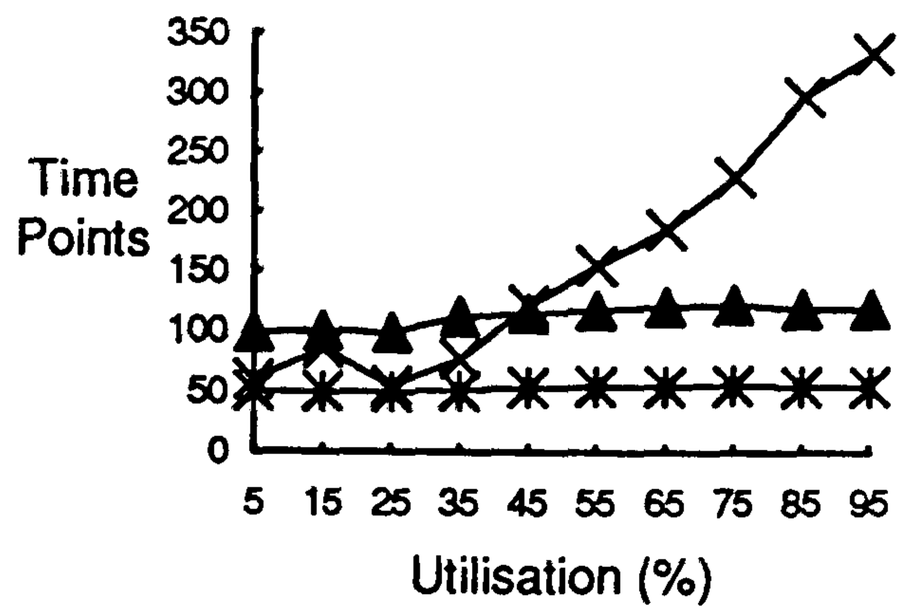
Graph 4.3: Time Points for 30 Processes (LA and LZA).



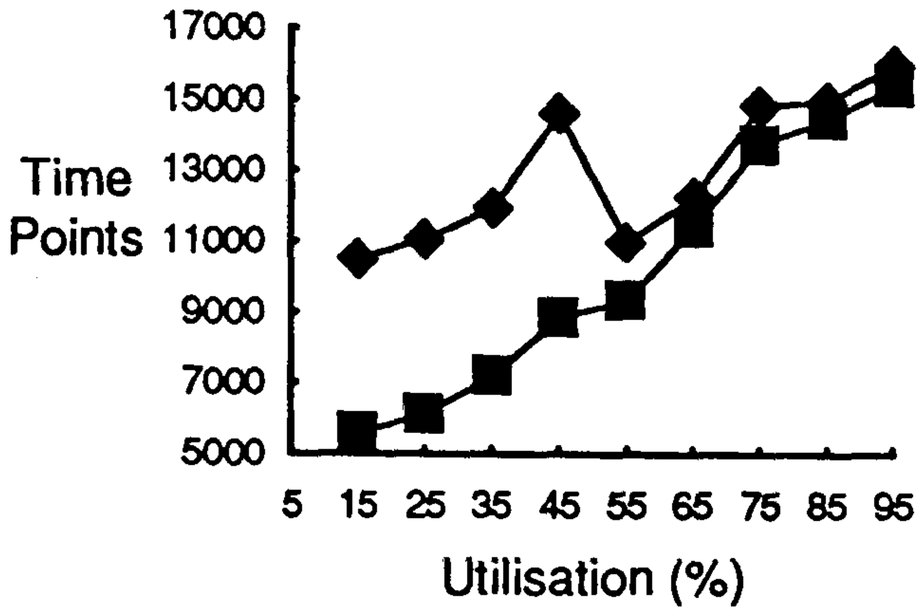
Graph 4.4: Time Points for 30 Processes (JA, NBA and RA).



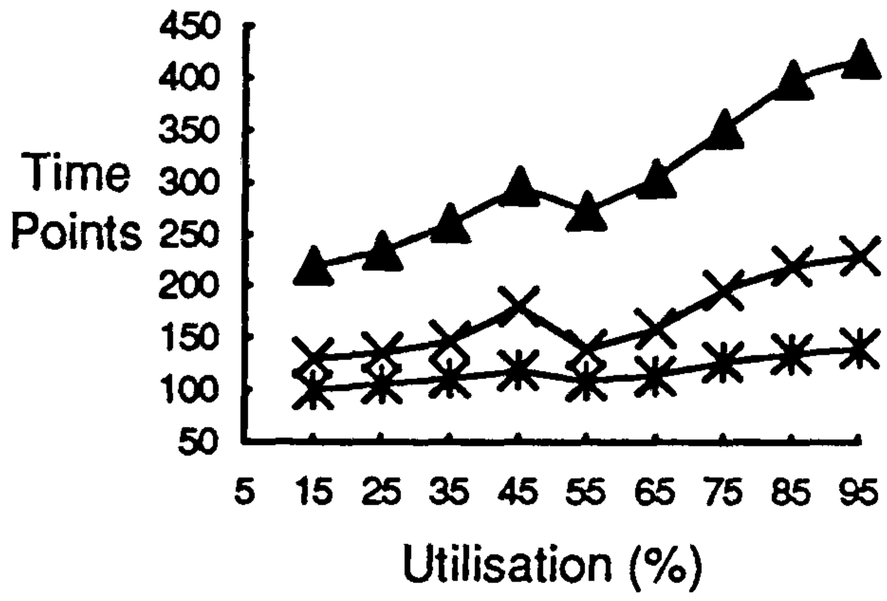
Graph 4.5: Time Points for 50 Processes (LA and LZA).



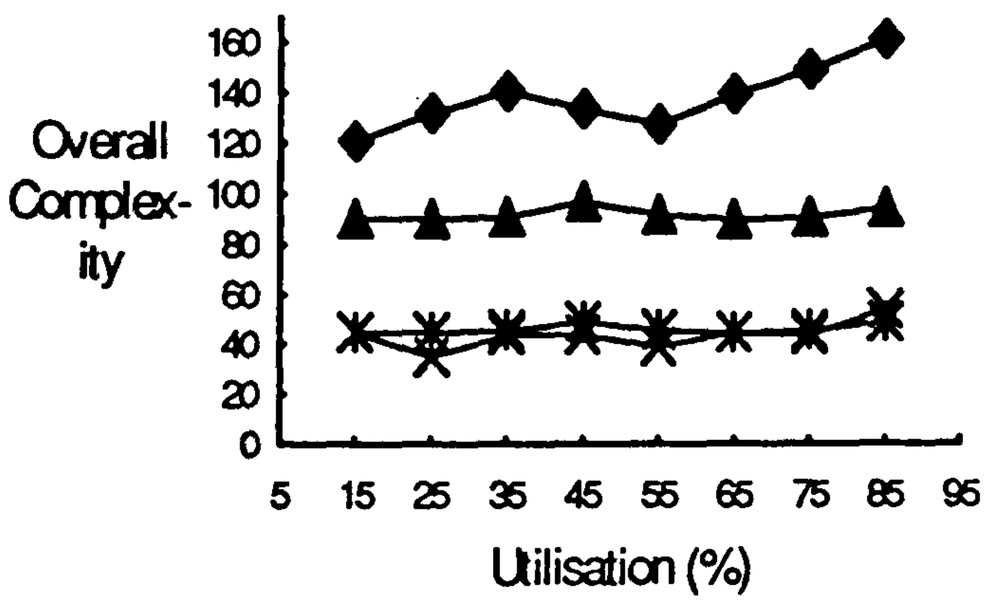
Graph 4.6: Time Points for 50 Processes (JA, NBA and RA).



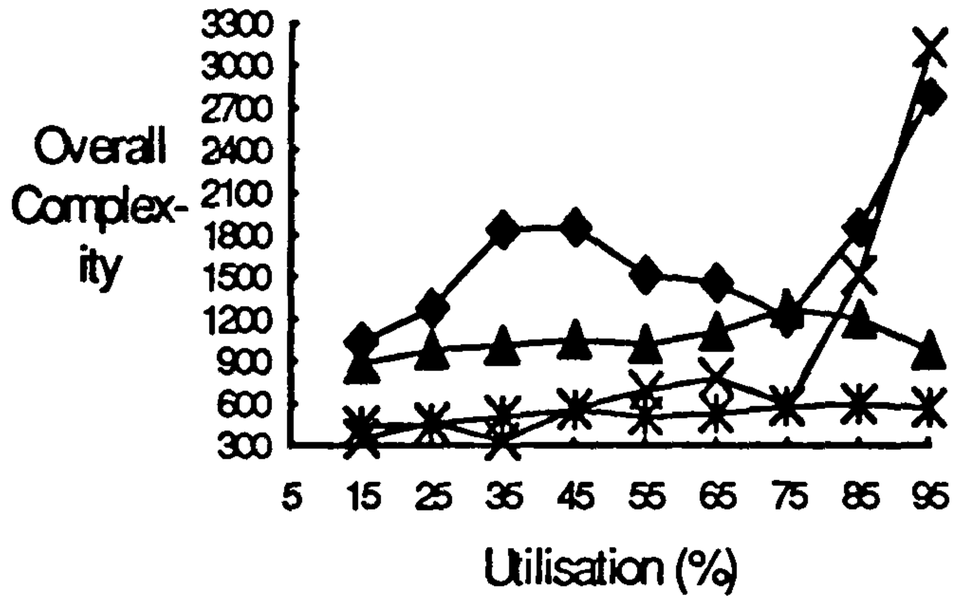
Graph 4.7: Time Points for 100 Processes (LA and LZA).



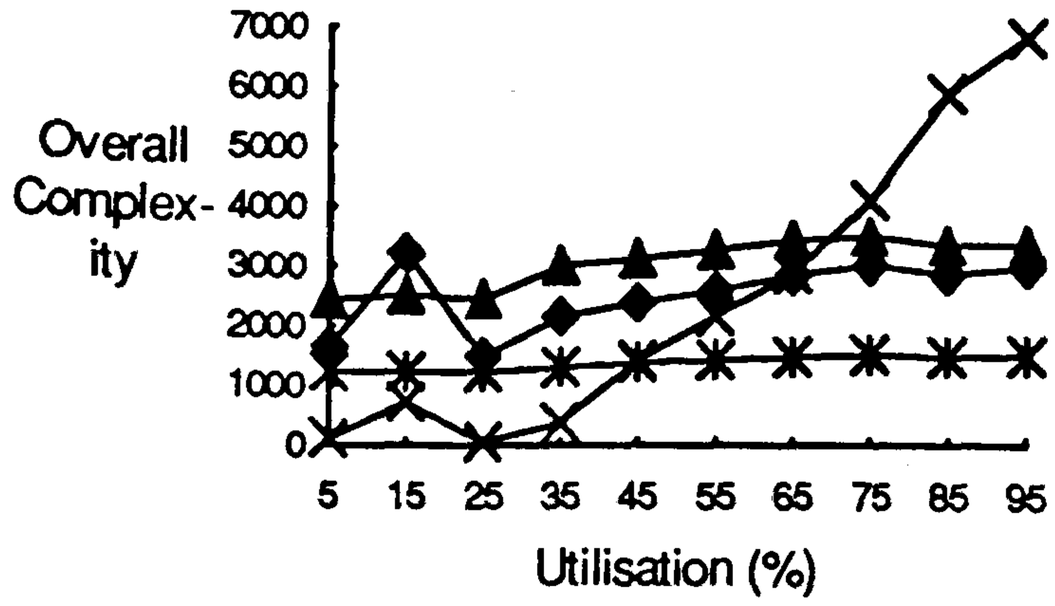
Graph 4.8: Time Points for 100 Processes (JA, NBA and RA).



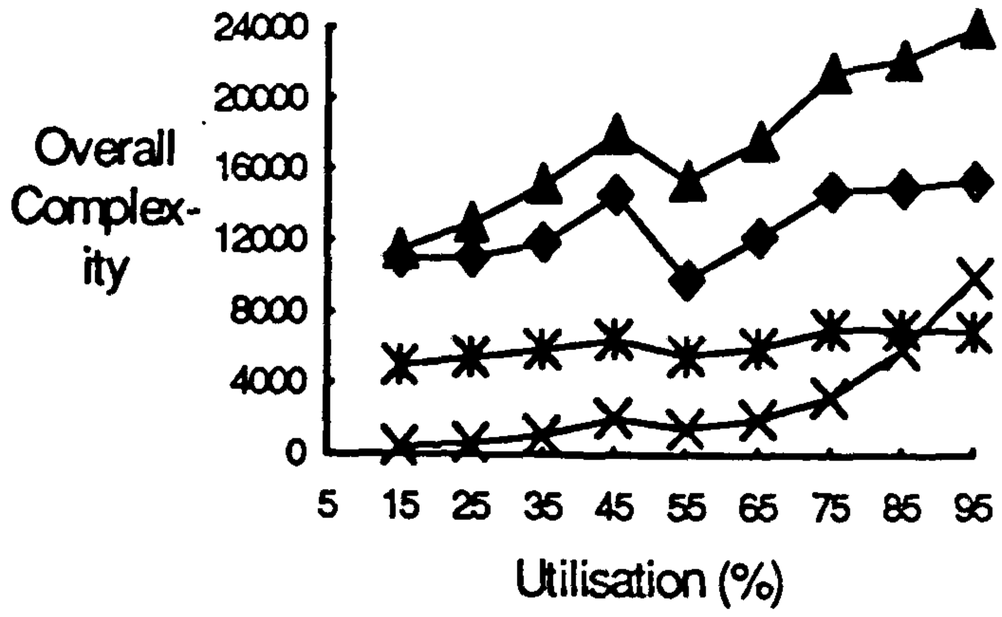
Graph 4.9: Overall Complexity for 10 Processes.



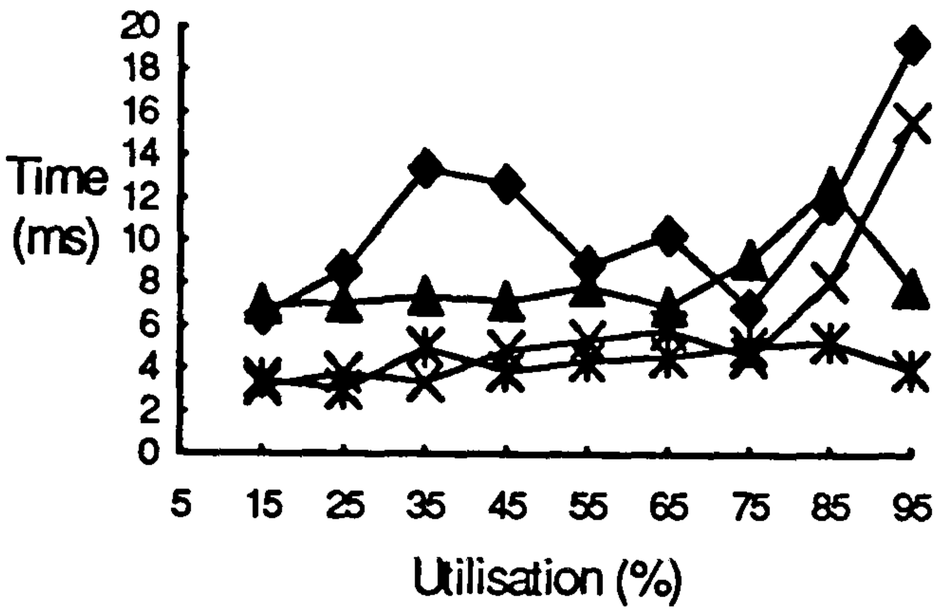
Graph 4.10: Overall Complexity for 30 Processes.



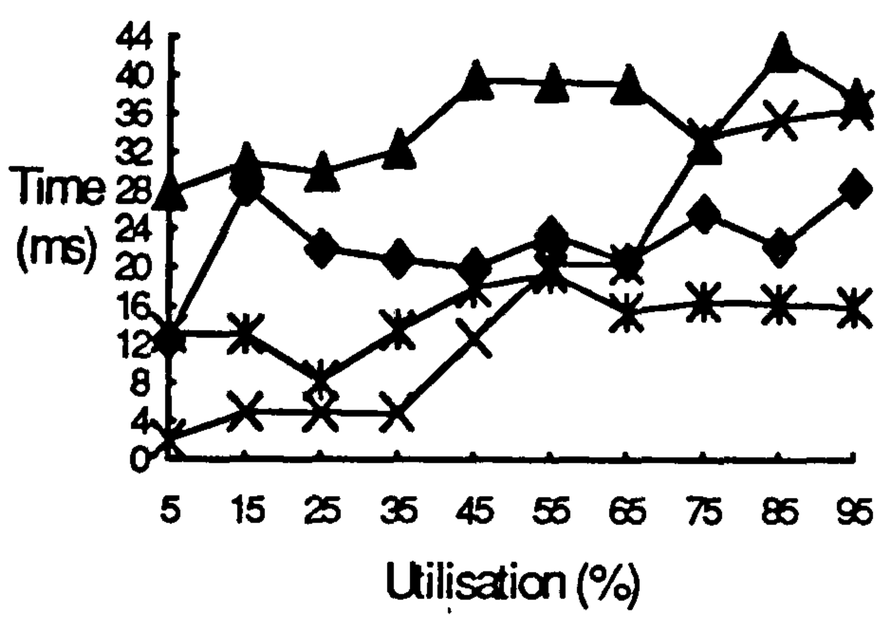
Graph 4.11: Overall Complexity for 50 Processes.



Graph 4.12: Overall Complexity for 100 Processes.



Graph 4.13: Actual Time for 30 Processes.



Graph 4.14: Actual Time for 50 Processes.

In general, graphs 4.1 to 4.8 show that RA visits less time points than the other algorithms, thus reducing the non-polynomial part of the feasibility test to a minimum (amongst these tests). The time point graphs for LA and LZA suggest that for 10 processes (Graph 4.1) the number of time points decreases from 30% to 60% process set utilisation. This is due to the difficulty of generating process sets with low utilisations: process periods tend to be relatively greater than those for higher utilisations, so reducing the number of time points. This is shown to a lesser extent by Graphs 4.3, 4.5 and 4.7. Since the same effect is not observed for JA, NBA and RA (Graphs 4.2, 4.4, 4.6 and 4.8), noting that the same process sets were used for all tests, the conclusion is drawn that the number of time points visited by LA and LZA is more data-dependent than the other three algorithms.

The overall complexities (graphs 4.9 - 4.12) of JA, NBA and RA were in most cases, more efficient than simple schedule construction (i.e. LA). The exceptions being for high utilisations (≥80%) and high numbers of processes,

where NBA was less efficient than the other three. For lower utilisations, NBA and RA were more efficient than JA, the former two tests having similar overall complexities. The actual execution times (graphs 4.13 - 4.14) of the algorithms reflected their relative overall complexities.

4.7.2 Comparison of Sufficient and Not Necessary Feasibility Tests

The worst-case time point, simple and overall complexities of the sufficient and not necessary feasibility tests given in section 4.2 are summarised in Table 4.12.

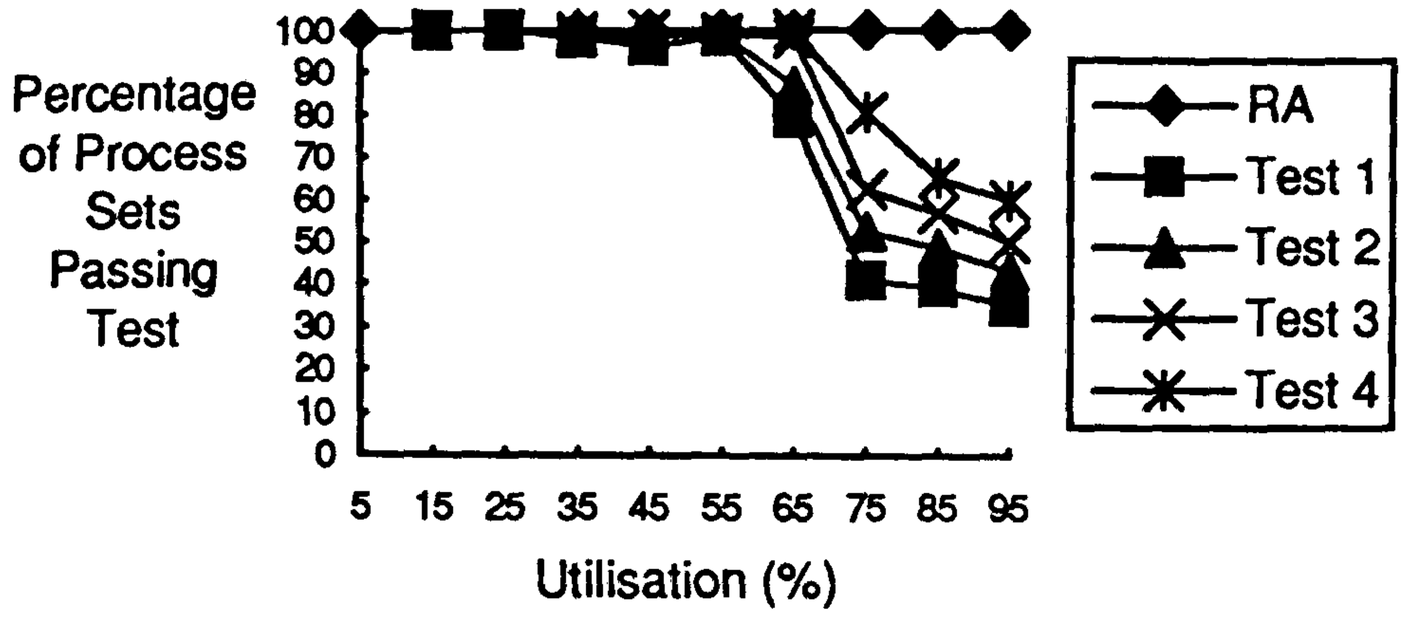
Worst-Case Complexities	Test 1	Test 2	Test 3	Test 4
Time-Point	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sample	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Overall	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(kn^2)$

Table 4.12: Worst-Case Complexities of Sufficient and Not Necessary Feasibility Tests.

We note that the average-case and theoretical worst-case performance are equivalent for tests 1, 2 and 3. The worst-case for test 4 is $O(n)$ time points, with $O(n)$ sample complexity. However, the overall complexity must include the calculation of the effective deadline. This has a worst-case when for each iteration, the effective deadline decreases by one. Thus, effective deadline calculation is of $O(kn^2)$, where $k=\max_{1\le i\le n}(D_i)$ giving an overall complexity of $O((k+1)n^2) \approx O(kn^2)$.

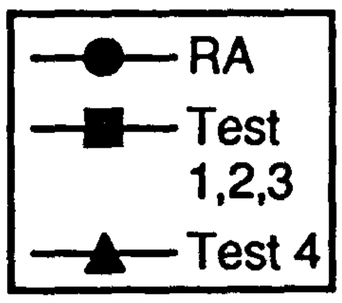
When considering sufficient and not necessary tests, the benefit, in terms of increased accuracy, for any increased cost, in terms of computational complexity, is of most interest. Therefore, the average-case performance of the tests was examined using the same randomly generated process sets used in the previous section. Initially, accuracy was considered. Graph 4.15 shows the percentage of feasible process sets passed by each of the tests plotted against process set utilisation. The results from a sufficient and necessary test (RA - see previous section) are also plotted to form a control. As expected, test 1 is the least accurate, with test 4 the most. The discrepancy between the accuracies of the sufficient and not necessary tests and the exact test widens as

utilisation increases: the effect of the assumed concurrent execution in the sufficient and not necessary tests becomes increasingly significant.

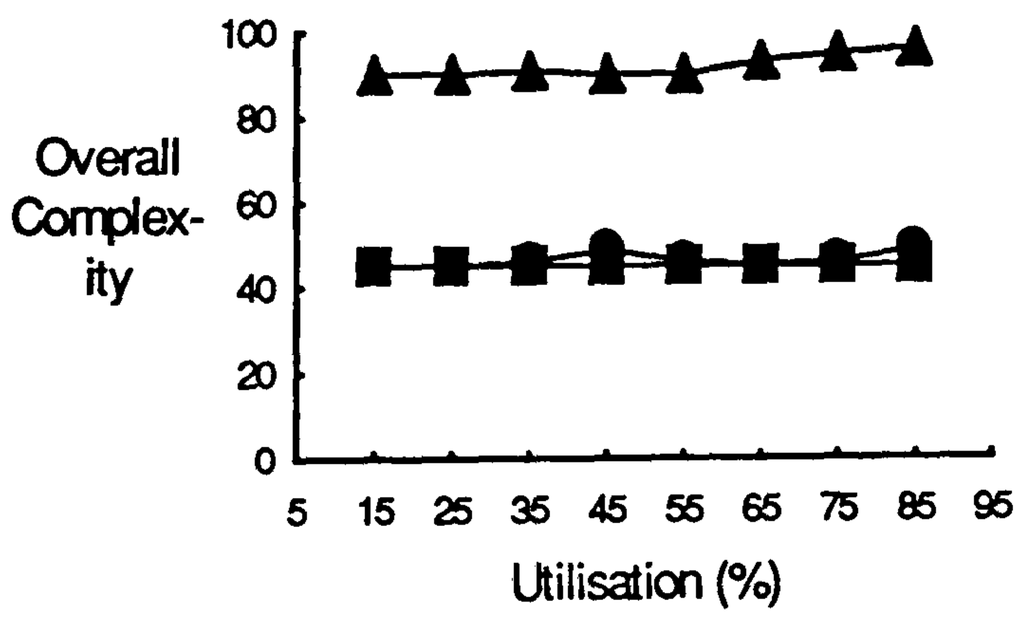


Graph 4.15: Accuracy For 10, 30, 50 and 100 Processes.

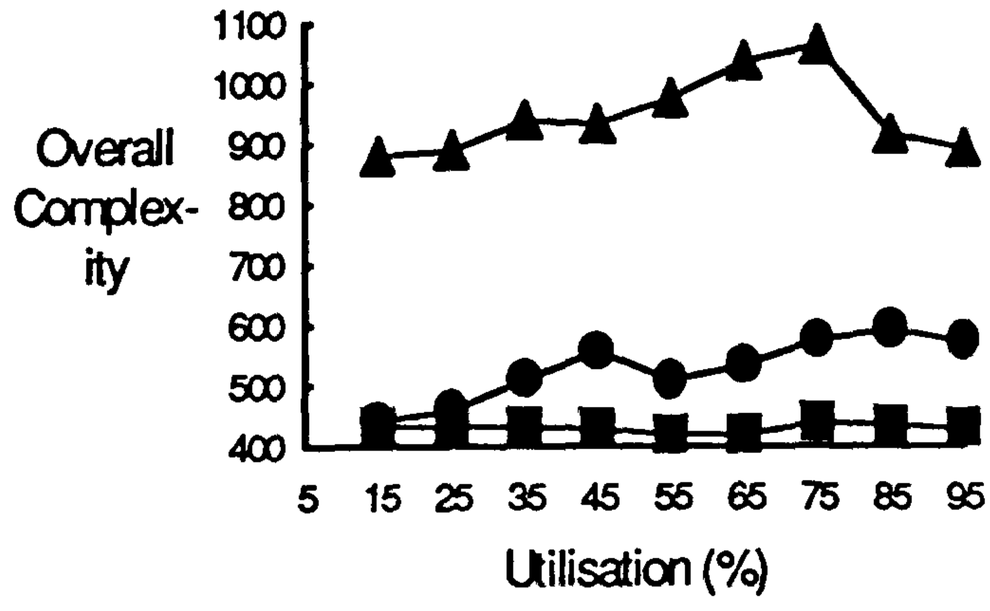
The cost of increased accuracy of the tests is now considered. It has been observed that as the accuracy of the feasibility tests increases, so does the complexity and hence the cost. The cost of tests 1, 2 and 3 is always n^2 if the process set is feasible (or the lowest priority process is the only infeasible process), whilst the cost of test 4 is higher, due to effective deadline calculation. However, the accuracy of this test is greater than tests 1, 2 and 3 (see graph 4.15). The overall complexity of tests 1, 2 and 3, and of test 4 (along with RA for a control) are plotted against process set utilisation in Graphs 4.16 - 4.19.



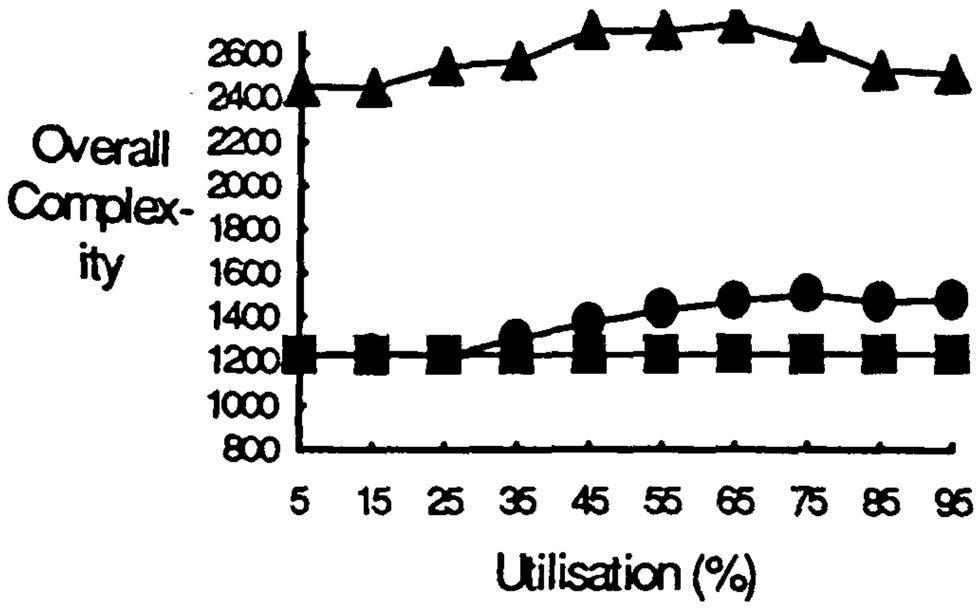
Key for Graphs 4.16 - 4.19.



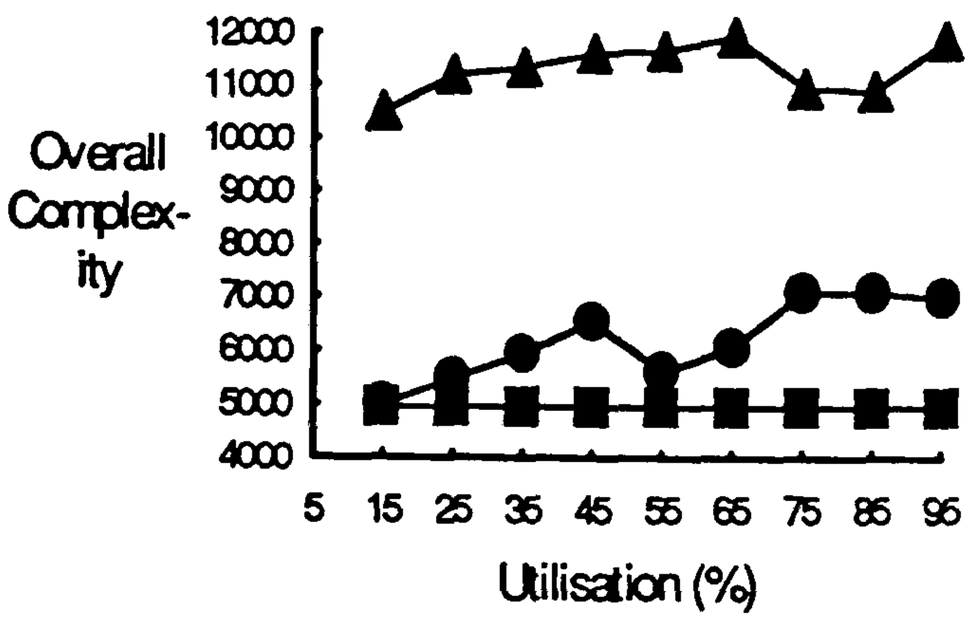
Graph 4.16: Overall Complexity for 10 Processes.



Graph 4.17: Overall Complexity for 30 Processes.

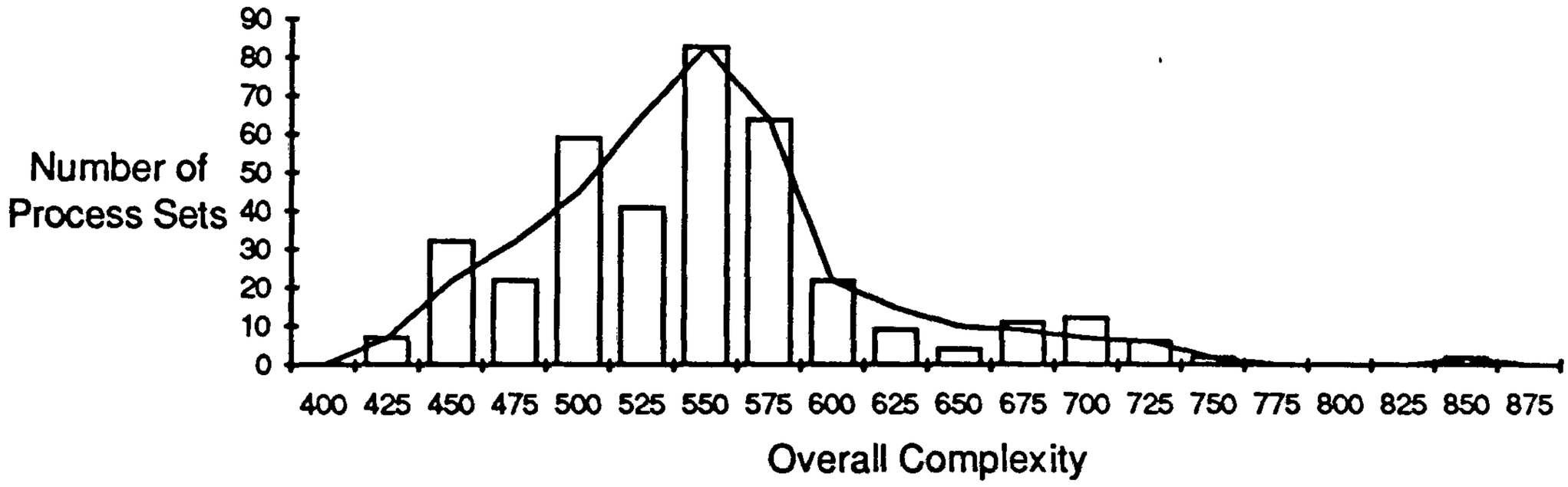


Graph 4.18: Overall Complexity for 50 Processes.

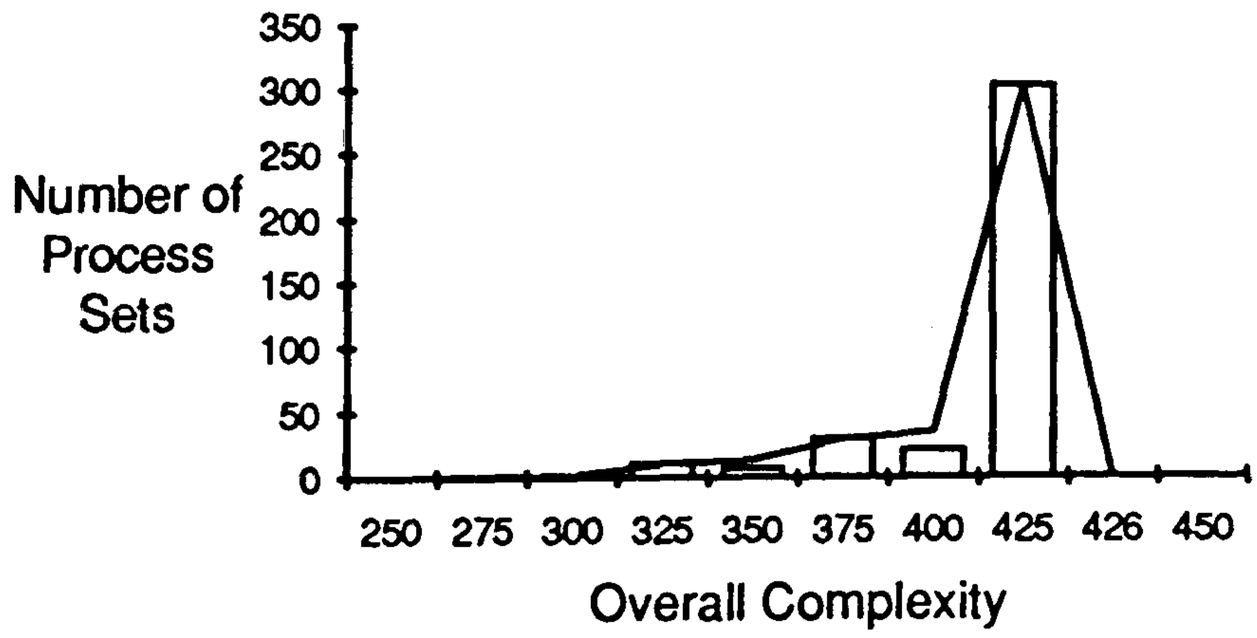


Graph 4.19: Overall Complexity for 100 Processes.

It is observed from the graphs 4.16 - 4.19 that the overall complexity of tests 1, 2 and 3 does not depend upon utilisation, only upon the cardinality of the process set. The overall complexity of test 4 is greater than tests 1, 2 and 3, essentially due to the non-polynomial property of effective deadline calculation in the test. In all cases, the overall complexity of test 4 is greater than RA, which is in turn has higher overall complexity than tests 1, 2 and 3. This indicates that the computation of effective deadline is expensive.



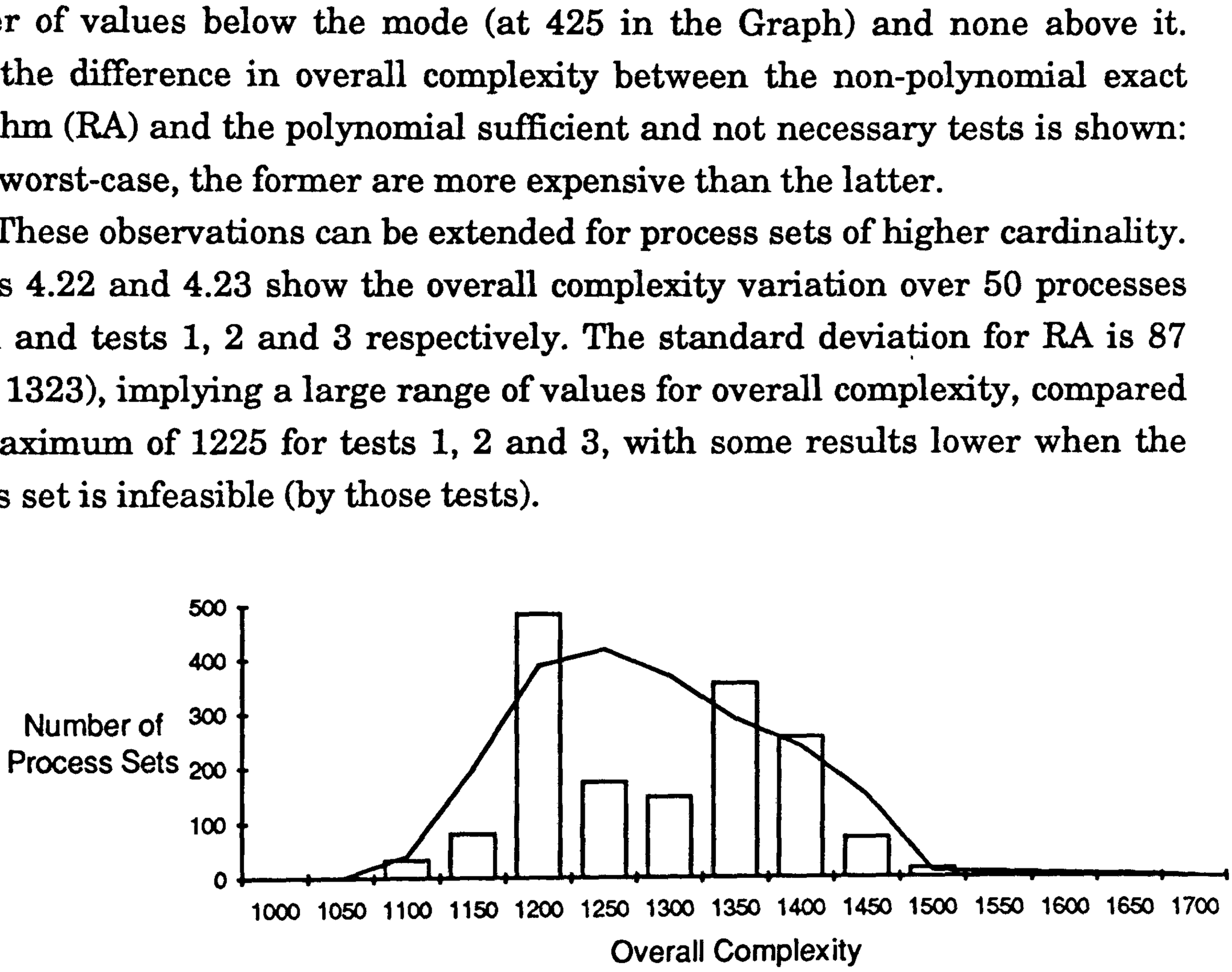
Graph 4.20: Overall Complexity of RA For 30 Processes.



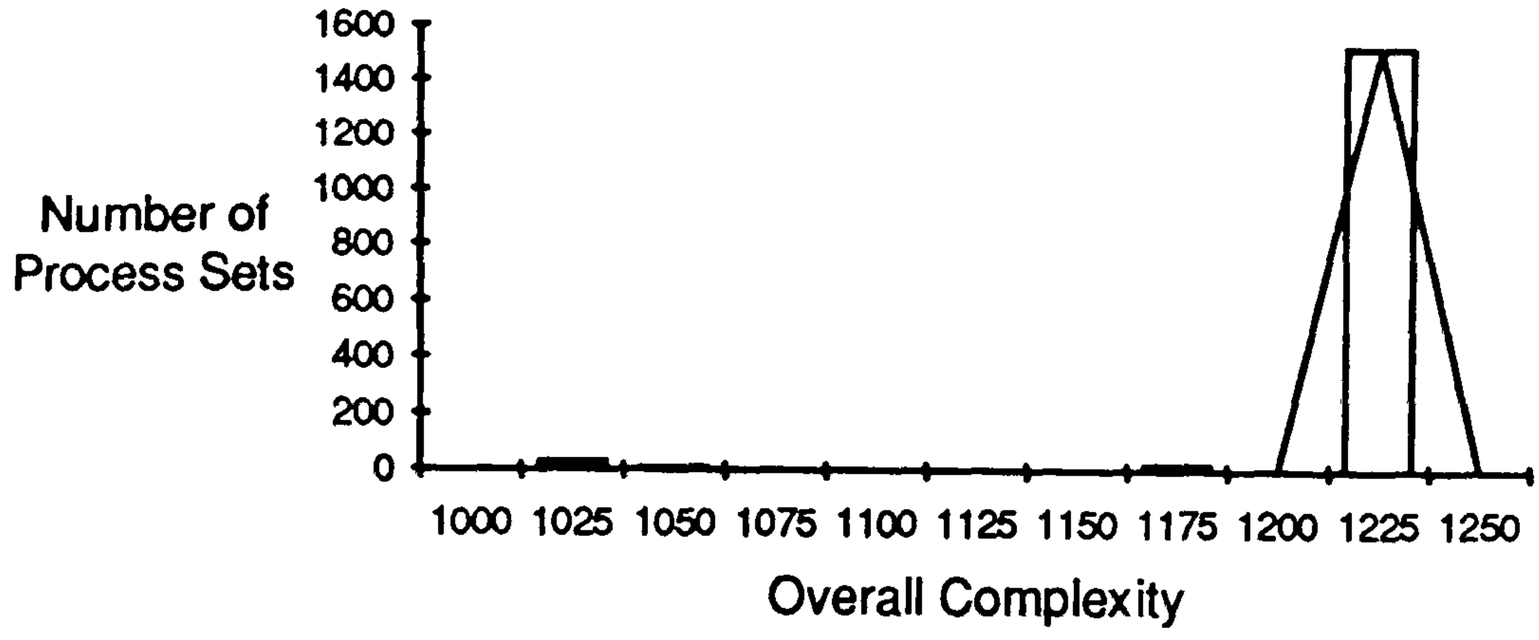
Graph 4.21: Overall Complexity of Tests 1, 2 and 3 For 30 Processes.

Although Graphs 4.16 - 4.19 indicate that the overall complexity of the sufficient and necessary test (RA) is similar to sufficient and not necessary tests 1, 2 and 3, it is observed that the latter tests have polynomial complexity in n, whilst RA has non-polynomial complexity. This is illustrated by the ranges of values for overall complexity of RA in Graph 4.20, considering process sets of cardinality 30. The number of process sets requiring a given number of time points is plotted against the number of time points. The distribution is normal, with standard deviation 68 (mean 558). It is observed that 98.84% of the values are within three standard deviations of the mean. In comparison, the values for tests 1, 2 and 3 (Graph 4.21) have constant overall complexity when a process set is feasible (by tests 1, 2 and 3), dropping below this constant value if the process set is infeasible. Hence the graph has a small number of values below the mode (at 425 in the Graph) and none above it. Thus, the difference in overall complexity between the non-polynomial exact algorithm (RA) and the polynomial sufficient and not necessary tests is shown: in the worst-case, the former are more expensive than the latter.

These observations can be extended for process sets of higher cardinality. Graphs 4.22 and 4.23 show the overall complexity variation over 50 processes for RA and tests 1, 2 and 3 respectively. The standard deviation for RA is 87 (mean 1323), implying a large range of values for overall complexity, compared to a maximum of 1225 for tests 1, 2 and 3, with some results lower when the process set is infeasible (by those tests).



Graph 4.22: Overall Complexity of RA For 50 Processes.



Graph 4.23: Overall Complexity of Tests 1, 2 and 3 For 50 Processes.

4.8 Summary

A representation of process execution has been presented which enabled the development of feasibility tests which removed some of the constraints imposed by current scheduling theory, therefore improving offline flexibility by increasing feasibility test coverage, accuracy and efficiency. Throughout the chapter, process deadlines were permitted to be no more than their periods.

Initially, a family of new sufficient and not necessary tests was outlined, with differing accuracies and complexities. It was observed that increasing test accuracy has the effect of increasing the complexity of analysis, and therefore the time and resources required to perform it.

Sufficient and not necessary tests were then developed. The first test provides a measure of worst-case response times of crucial processes, the second the maximum amount of interference it can endure whilst remaining feasible. These algorithms, together with methods proposed elsewhere for determining feasibility, were compared with a view to efficiency.

Efficiency is related to complexity, which can be viewed as a combination of the number of time points examined to determine the feasibility of a process, and the complexity of the feasibility test used at each time point. For sufficient and necessary tests, the latter is polynomial, with the number of time points exponential: this is the root of the NP-complete nature of the feasibility problem. The sufficient and necessary tests proposed in this chapter minimise the number of time points (compared to other proposed tests), that is the non-polynomial part of the problem, whilst maintaining a polynomial sample complexity. It is noted that the tests described in this chapter are applicable for any priority assignment, not merely deadline-monotonic priority assignment.

Also, the sufficient and necessary tests were compared with the family of sufficient and not necessary tests. The average-case behaviour of the former

tests indicates that in many cases, these tests do not incur significant additional execution cost over that required by the sufficient and not necessary tests. This emphasises the observation that the tests have pseudo-polynomial complexity: in many cases their actual behaviour is polynomial, like that of the sufficient and not necessary tests.

The coverage of the tests developed within the chapter was expanded to enable sporadic processes and process blocking to be incorporated. The former fit naturally within the deadline less than (or equal to) period process model, with no extension to the theory required. The latter required a minimal extension to each test. The resultant accuracy of the tests was improved by reducing the pessimism inherent in the worst-case blocking calculations of the Priority Inheritance Protocol (and its derivatives).

It is noted that this chapter has improved the flexibility of offline scheduling by increasing the coverage, accuracy and efficiency of feasibility tests. In the next chapter, further improvements are made.

Chapter 5.
Extending Offline Flexibility Via Optimal Priority Assignment

In Chapter 4 it was observed that as the coverage and accuracy of an offline feasibility test increases, so does its complexity. In the limit, sufficient and necessary tests for arbitrary process timing and interaction characteristics are NP-complete. In the previous chapter this was illustrated, with various feasibility (and infeasibility) tests of differing accuracies, efficiencies and complexities for processes with static priorities. This chapter extends this work by examining both priority assignment and feasibility of processes with arbitrary timing constraints.

One of the assumptions of the feasibility tests in Chapter 4 was that all processes have a common release time (i.e. critical instant). This assumption simplifies priority assignment (and subsequent feasibility testing). When process deadlines are equal to their periods, rate-monotonic priority ordering is optimal, with deadline monotonic priority assignment optimal for processes whose deadlines are no greater than their periods [Leung80]. If this assumption is relaxed, processes are permitted to have arbitrary initial start times, with this start time termed the *offset*. Thus, if the offset of τ_i is given by O_i the releases of τ_i are at O_i, O_i+T_i, O_i+2T_i,... etc.

Processes with offsets provide more flexibility in that more applications can now be represented (see section 3.1). For example, processes which have a high utilisation between a release and subsequent deadline (i.e. C_i close to D_i) can now be phased, such that they will never be active simultaneously: assuming a critical instant between such processes would almost inevitably result in the process set being declared infeasible. Also, application requirements may wish to place access to particular resources, either to reduce potential process blocking, or because the resource needs time between accesses (e.g. hardware devices).

Process offsets also allow a natural representation of precedence constrained processes. For example, consider the real-time control of a physical device. Periodically, a request is made by a process for data from the device.

Assume that the time for the device to respond is non-trivial. A second process, whose release is offset from the first process, collects (and processes) the data. This representation is an alternative to a single process which blocks whilst the device responds, although this could be modelled for feasibility purposes as two offset processes.

A process set containing processes with arbitrary offsets may still have a critical instant, with the consequence that deadline-monotonic (or rate-monotonic) priority assignment remains optimal. However, if a critical instant does not occur, the known priority assignment strategies are, in general, no longer optimal [Leung82]. This is illustrated by example.

Process	O	C	D	T	Rate-Monotonic Priority Assignment
τ_A	0	3	8	8	1
τ_B	10	1	12	12	2/3
τ_C	0	6	12	12	3/2

Table 5.1: Example Process Set 1.

Consider the process set in Table 5.1. Notionally we may assign priorities in a rate-monotonic manner. Process τ_A is assigned the highest priority. The assignment of priorities for τ_B and τ_C cannot be performed in the arbitrary manner suggested by Lui and Layland [Layland73] for processes with equal periods. If τ_B is assigned a higher priority than τ_C, the latter process will miss its first deadline. However, if τ_C is assigned a higher priority than τ_B all process deadlines are met.

Process	O	C	D	T	Deadline-Monotonic Priority Assignment
τ_A	2	2	3	4	1
τ_B	0	3	4	8	2

Table 5.2: Example Process Set 2.

Now, consider the process set in Table 5.2. Intuitively, deadline-monotonic priority assignment is applicable, with τ_A assigned a higher priority than τ_B. However, this leads to the deadline of the latter process to be missed

at time 4 (and then successively at 12, 20, 28,...) When priorities are reversed (contrary to deadline-monotonic priority assignment) process deadlines are met.

The above examples show that rate-monotonic and deadline-monotonic priority assignments are not optimal for processes with arbitrary offsets. Indeed, according to Leung, who describes processes with arbitrary offsets as asynchronous:

> *"At the present time no priority assignment has been found which is optimal for an arbitrary asynchronous system."* [Leung82]

An additional issue is that of feasibility testing. Tests developed assuming a critical instant between processes remain applicable (given that they do not assume a specific priority assignment rule). However, sufficient and necessary critical instant tests become, in general, sufficient and not necessary for processes with arbitrary offsets. For example, the exact feasibility tests developed in section 4.2 are sufficient and not necessary for such process sets; the tests in section 4.3 remain sufficient and not necessary.

Leung *et al* have proved that the problem of determining the feasibility of a process set that has no critical instant is NP-hard (Theorem 3.8 in [Leung82]). However,

> *"We also note that Theorem 3.8 does not imply that the problem of finding an optimal priority assignment is NP-hard, for the apparent difficulties in determining whether or not a [process set] is [feasible] on one processor may entirely be due to the difficulties in deciding whether or not the schedule produced by a particular priority assignment is valid."* [Leung82]

This statement is proved within this chapter, where it is shown that to find an optimal priority assignment requires the examination of a polynomial number of possible priority assignments.: the NP-hard complexity is due to determining the feasibility of a given priority assignment.

The main focus of this chapter is the identification of priority assignments and sufficient and necessary feasibility tests for processes with arbitrary offsets. To this end the following key issues are addressed:

- determining whether tasks with arbitrary start times are ever, within the system lifetime, released simultaneously;
- providing an optimal priority assignment mechanism;

- determining, in a sufficient and necessary manner, the feasibility of a process set with arbitrary start times.

Subsequently, the theory is expanded to allow processes with arbitrary interaction requirements, via precedence constraints and shared resources, to be analysed.

In the above discussion, the assumption is made that process offsets are either fixed (like period and deadline), or a property of the application requirements. Consider process sets that assume a critical instant, with all process offsets set to zero. If such a process set is declared infeasible by a sufficient and necessary test, we could assume that application timing requirements need to be re-considered and/or processes re-coded. Alternatively, we could assign offsets to processes to improve their feasibility, possibly enabling the process set to be declared feasible. Even if the original process set were declared feasible, it may be beneficial to introduce offsets, for example to reduce potential process blocking. Issues surrounding the assignment of offsets to processes is the final contribution of this chapter.

Initially, some assumptions are made:

(i) all processes are periodic;

(ii) worst-case process computation times are bounded;

(iii) processes do not interact (via shared resources or precedence constraints);

(iv) processes cannot voluntarily suspend, or become blocked by an external event (e.g. reception of data from an external source).

During the course of the chapter, restriction (iii) is removed.

The chapter is arranged as follows. The next section discusses the detection of critical instants in process sets with arbitrary process offsets. Section 5.2 describes a method for achieving optimal priority assignment of process sets without critical instants. Section 5.3 describes the time interval required for feasibility testing, with sections 5.4 and 5.5 developing sufficient and necessary and sufficient and not necessary feasibility tests respectively. Sections 5.6 and 5.7 extend the optimal priority assignment and feasibility tests for precedence constraints and blocking respectively. Section 5.8 discusses the assignment of offsets to processes. Finally, a summary is given in section 5.9.

5.1 Critical Instants

When processes have arbitrary offsets, it is difficult, by inspection of process timing characteristics alone, to determine if a critical instant will occur during system execution. The problem of identifying a time when a critical instant between all processes occurs can be solved using the Generalised Chinese Remainder Theorem (GCRT). This can be stated [Knuth68] [1]:

> Let $m_1, m_2, \ldots, m_r$ be positive integers. Let m be the least common multiple of $m_1, m_2, \ldots, m_r$, and let $a, u_1, u_2, \ldots, u_r$ be any integers. There is exactly one integer u which satisfies the conditions $a \le u < a+m$ and $u \equiv u_j \pmod{m_j}$ for all $1 \le j \le r$ if and only if $u_i \equiv u_j \pmod{(\gcd(m_i, m_j))}$ for all $1 \le i < j \le r$ where $\gcd(x,y)$ denotes the greatest common divisor of x and y.

Within this context we may set

a to be the maximum offset in the process set, i.e. $\max_{1 \le i \le n}(O_i)$

$m_1, m_2, \ldots, m_r$ are equivalent to $T_1, T_2, \ldots, T_n$ where $r=n$

$u_1, u_2, \ldots, u_r$ are equivalent to $O_1, O_2, \ldots, O_n$ where $r=n$

Thus for each pair of processes $\tau_i, \tau_j \in \Delta \ (i \neq j)$, the GCRT states that those processes have a simultaneous release if and only if:

$$|O_i - O_j| = h \gcd(T_i, T_j)$$

where h is a non-negative integer.

Example 5.1:

Consider the process set in Table 5.3.

Process	O	T
τ_A	5	10
τ_B	4	9
τ_C	10	24

Table 5.3 Example Process Set 1.

According to the GCRT, each pair of processes needs to be combined in turn to find if the process set has a critical instant.

τ_A, τ_B : $\quad |5 - 4| = h \gcd(10, 9)$

$$1 = 1h$$

The processes share a critical instant (integer solution for h).

---

1. $a \equiv b \bmod c$ is equivalent to $b \bmod c = a \bmod c$

τ_A, τ_C : $| 5 - 10 | = h \gcd(10, 24)$

$$5 = 2h$$

The processes do not share a critical instant (no integer solution for h).

τ_B, τ_C : $| 4 - 10 | = h \gcd(9, 24)$

$$3 = 1h$$

The processes share a critical instant (integer solution for h).

The process set does not have a critical instant since there is one pair of processes that do not share a critical instant.

5.2 Optimal Priority Assignment

The process set (Δ) is denoted Δ^* if the member processes never have a simultaneous release. We term Δ^* a *non-critical-instant process set*. For any Δ^* the initial discussion at the start of the chapter indicated that neither rate-monotonic or deadline-monotonic priority assignments were optimal. A feasible priority assignment could be found by searching through all *n!* distinct priority assignments over Δ^*. This is inefficient. We now develop an efficient optimal priority assignment for processes with arbitrary start times.

Consider $\Delta^* = \{\tau_A, \tau_B, \tau_C, \ldots\}$ of cardinality n. A priority assignment function maps each process onto a different priority level. For Δ^* there are *n!* distinct priority assignments over the process set, hence the set of distinct priority assignment functions has cardinality *n!*. This is denoted by $\Phi = \{\Phi_1, \Phi_2, \ldots, \Phi_{n!}\}$. Under a given priority assignment function, the mapping of a process onto a priority level is given by:

$$\Phi_i(\tau_A) = j$$

where the i^{th} priority assignment function maps τ_A onto priority level j. The mapping of priority level to process, is denoted:

$$\Phi_i^{-1}(j) = \tau_A$$

When the priority ordering over Δ^* specified by a priority ordering function is feasible, we term that function a feasible priority assignment function.

In general, τ_A is feasible if and only if (see section 4.1):

$$C_A + I_A \leq D_A$$

where I_A represents the execution requirement (interference) of higher priority processes in the interval defined by the release and deadline of τ_A.

If τ_A is not feasible, and the process timing characteristics cannot be changed (i.e. C_A cannot be decreased and D_A cannot be increased), the only way to make τ_A feasible is to decrease I_A (assuming that processes do not block - this restriction is lifted in section 5.7). This is achieved by changing the priority ordering over Δ^* by using a priority assignment function that reduces the priority of a higher priority process to be lower than τ_A. (Note that we could then promote a lower priority process to be higher than τ_A as long as the new I_A is less than the original.)

Let us now consider the effect on the feasibility of Δ^* (cardinality n) for $\Phi_x \in \Phi$ such that $\Phi_x(\tau_A)=n$. The following theorems discuss this assignment.

Theorem 5.1:

If τ_A is assigned the lowest priority, n, and is infeasible, no priority assignment function that assigns τ_A priority level n produces a feasible assignment.

Proof:

Amongst the $n!$ distinct priority assignment functions, $(n-1)!$ produce an assignment with τ_A at priority level n. For all such assignments, the interference due to processes of higher priority than τ_A is equal, as the same set of processes is of higher priority than τ_A in each ordering. Thus if τ_A is infeasible as the lowest priority process by one assignment function, it will be infeasible under the priority ordering of any other function assigning it the lowest priority.

Theorem 5.2:

If τ_A is assigned the lowest priority, n, and is feasible, then if a feasible priority ordering for Δ^* exists, an ordering with τ_A assigned the lowest priority exists.

Proof:

Let us assume that an assignment function Φ_y produces the feasible assignment:

$$\Phi_y(\tau_B)=1,\Phi_y(\tau_C)=2,\ldots,\Phi_y(\tau_A)=i,\Phi_y(\tau_D)=i+1,\ldots,\Phi_y(\tau_E)=n$$

We note that τ_A is feasible at priority level $i<n$. Within Φ there exists another priority assignment Φ_x:

$$\Phi_x(\tau_B)=1,\Phi_x(\tau_C)=2,\ldots,\Phi_x(\tau_D)=i,\ldots,\Phi_x(\tau_E)=n-1,\Phi_x(\tau_A)=n$$

Noting that by the theorem τ_A is feasible if assigned priority level n. The processes assigned priority levels i+1..n in Φ_y are promoted 1 place under Φ_x (i.e. the process at priority level i+1 is now assigned priority i). Clearly, the processes assigned levels 1..i-1 under Φ_x are feasible since they were feasible under Φ_y when assigned the same priority levels. The processes assigned priority levels i..n-1 under Φ_x are also feasible: the interference on these processes is less under Φ_x than Φ_y. Therefore, Φ_x is a feasible priority ordering: at least one feasible priority assignment exists with τ_A as the lowest priority process. The theorem is proved.

The above theorems limit considerations to priority level n. We now extend the theorems to consider assignment of arbitrary priorities to processes, rather than merely priority n.

Theorem 5.3:

Let the processes assigned priority levels $i,i+1,...,n$ by assignment function Φ_x be feasible under that priority ordering. If there exists a feasible priority ordering for Δ^*, there exists a feasible priority ordering that assigns the same processes to levels i..n as Φ_x.

Proof:

The theorem is proved by showing that a feasible priority assignment function Φ_y can be transformed to assign the same processes to priority levels $i,i+1,...,n$ as Φ_x, whilst preserving the feasibility of Φ_y. The proof is by induction: Φ_y is transformed successively by moving processes $\Phi_x^{-1}(n)$, $\Phi_x^{-1}(n-1)$,..., $\Phi_x^{-1}(i)$ to priority levels n,n-1,...,i respectively under Φ_y.

Base

Let $\Phi_x^{-1}(n)=\tau_A$ and $\Phi_y(\tau_A)=m$, where $m \leq n$. By Theorem 5.2 we can move τ_A to the assigned level (n) under Φ_y with Φ_y remaining a feasible priority assignment.

Inductive Hypothesis

Assume that the processes assigned to priority levels *n-1,n-2,...,i+1* under Φ_x are moved to levels *n-1,n-2,...,i+1* under Φ_y with Φ_y remaining a feasible priority assignment.

Inductive Step

Let $\Phi_x^{-1}(i)=\tau_B$ and $\Phi_y(\tau_B)=m$, where $m \leq i$ (since the reassignment of priority levels *n-1,n-2,...,i+1* has promoted τ_B to have a priority of between 1 and *i*). Under both orderings, the processes assigned to priority levels *i+1..n* are identical. Process τ_B is reassigned in Φ_y to level *i*. We know (by Φ_x) that τ_B is feasible at this level (assuming that processes assigned to levels *i+1..n* are identical under Φ_x and Φ_y). After the reassignment, processes at levels *1..i-1* remain feasible, as their respective interferences are no greater than before the reassignment and therefore must remain feasible. This proves the theorem.

5.2.1 Optimal Bottom-Up Priority Assignment

The above theorems are now used to develop an optimal static priority assignment scheme which assigns processes to priority levels *n,n-1,...,1* in order. Only if a feasible assignment can be made to priority level *i* do we proceed to priority level *i-1*.

Consider the assignment to level *i* ($1<i\leq n$). We assume that priority levels *n..i+1* have been assigned such that the processes assigned to those levels are feasible. Let the process assigned to priority level *j* be given by $\Psi(j)$. We note that $\Psi(j)$ is only defined for $i<j\leq n$. Let the set Δ^{i+1} be composed of those processes in Δ^* that have been assigned priority levels *n,n-1,...,i+1* (cardinality of Δ^{i+1} is *n-i*). For each process τ_A in $\Delta^*-\Delta^{i+1}$ (i.e. the set of unassigned processes) we select a single $\Phi_k \in \Phi$ such that

$$\forall l: i < l \leq n: \Phi_k^{-1}(l) = \Psi(l)$$

The set of such Φ_k for priority level *i* is termed Φ^i. Set Φ^i contains *i* elements of Φ, each of which assigns priority levels *n* to *i+1* identically, differing in their assignment of priority level *i* (each Φ_k assigns a different $\tau_r \in \Delta^*-\Delta^{i+1}$ to level *i*).

For each $\Phi_k \in \Phi^i$ we check the feasibility of the process assigned priority level *i* (by virtue of reaching the assignment of priority level *i* we know that processes assigned to levels *n,n-1,...,i+1* are feasible). Two cases are identified:

(i) all processes are infeasible when assigned priority level *i*;

(ii) one or more processes are feasible when assigned priority level *i*.

In the first case, we know by Theorem 5.2 that no feasible priority assignment exists for Δ^* and so the process set is infeasible. In the second case, we may

arbitrarily select one of the feasible processes noting by Theorem 5.3 that if a feasible priority assignment for Δ^* exists, one will exist with the selected task assigned priority level i. Thus, $\Psi(i)$ is defined. We proceed to the assignment of priority level i-1.

Eventually, we reach the assignment for level 1. This is trivial since at this stage only one process remains to be assigned (i.e. the cardinality of $\Delta^* - \Delta^2$ is 1) leaving no choice for priority level 1.

5.2.2 Algorithmic Implementation

An algorithm implementing the optimal bottom-up priority assignment method is given in Figure 5.1. The parameter passed to the function is the process set under consideration, with the function returning `TRUE` or `FALSE` depending whether a priority assignment could be found or not. It is assumed that the function `feasible()` exists, determining whether a process τ_A is feasible when assigned priority `j` within the process set Δ (where all other processes have defined priorities).

```
function priority_assignment (Δ*) returns boolean is
    begin
            Δ = Δ* ;
        for j in n..1 loop          -- priority level j
            unassigned = TRUE    ;
            for τA in Δ loop
                if unassigned = TRUE then
                    -- see if τA is feasible at
                    -- priority level j
                    if feasible(τA, j, Δ) = TRUE
                    then
                        Ψ(j)=τA; -- assign τA to j
                        Δ=Δ-τA ;
                        unassigned = FALSE ;
                    end if     ;
                exit when unassigned = FALSE ;
            end if     ;
        end loop  ;
        if unassigned = TRUE then
            return FALSE;
```

```
                    -- no feasible priority
                    -- assignment exists
                end if ;
            end loop  ;
            return TRUE ;
        end priority_assignment ;
```

Figure 5.1: Optimal Priority Assignment Algorithm

Firstly, we attempt to find a process τ_A that is feasible at priority level $j=n$. If one is found, then by Theorem 5.2 if a feasible priority assignment function exists, one also exists with τ_A assigned priority level n, i.e. $\Psi(j)=\tau_n$. Next, priority level $j=n-1$ is now considered. If a process can be found (amongst the $n-1$ processes that have not yet been assigned a priority level) that is feasible at priority level $n-1$, then by Theorem 5.3 we know that if a feasible priority assignment function exists, a feasible priority one also exists with this process assigned priority level $n-1$. Successively, processes are found that are feasible at priority levels $n..1$. If, for any priority level j a feasible process cannot be found, no feasible priority assignment function exists.

Consider the following example.

Example 5.2.

We return to the process set given by Table 5.2 at the beginning of the chapter, re-stated below:

Process	O	C	D	T	Deadline-Monotonic Priority Assignment
τ_A	2	2	3	4	1
τ_B	0	3	4	8	2

The process set is infeasible with priorities assigned in a deadline-monotonic manner.

Initially we attempt to find a process that is feasible when assigned priority level 2. Let τ_A be assigned priority level 2. This priority assignment is found to be feasible (see following sections for details).

Trivially, τ_B is feasible when assigned priority level 1. Thus a feasible priority assignment has been found.

5.2.3 Discussion

The priority assignment scheme detailed in this section is optimal in the sense that if a feasible priority ordering exists for a process set, it will be found by this method. The proof of this assertion lies in Theorems 5.1, 5.2 and 5.3.

The complexity of the priority assignment scheme is dependent upon the number of process feasibility tests performed. This is more readily described by examining the behaviour of the algorithm in Figure 5.1. To find a process that is feasible at priority level n involves testing the feasibility of a maximum of n processes. In general, to find a process which is feasible when assigned priority level $i \leq n$ requires testing the feasibility of a maximum of i processes (that is the i processes that have yet to be assigned a priority). Therefore, across all priority levels, the number of process feasibility tests required is given by:

$$= n + (n-1) + \ldots + (n-(n-1)) = \frac{1}{2}(n^2 + n)$$

This is polynomial in n and as such is exponentially more efficient than examining the feasibility of all possible $n!$ priority orderings. The overall complexity is $O\left(\frac{n^2+n}{2}E\right)$ where E represents the complexity of the feasibility test required.

5.3 Feasibility Interval

Feasibility testing of a process set requires the definition of an interval over which that testing needs to occur. We term this the *feasibility interval*. As seen in Chapter 4, the feasibility of processes that share a critical instant can be determined by examining the first deadline of each process after a critical instant. Hence the feasibility interval for each τ_i is $[t, t+D_i)$ where t corresponds to a critical instant. When arbitrary process offsets are permitted, or more specifically when the processes form a non-critical-instant process set, the feasibility interval needs to be re-defined.

For processes with offsets, Leung *et al* [Leung80] established that the interval

$$\left[\max_{1 \leq i \leq n}(O_i), \max_{1 \leq i \leq n}(O_i) + 2 * \operatorname{lcm}_{1 \leq i \leq n}(T_i)\right)$$

is sufficient for establishing feasibility: if all process deadlines are met in the interval, they will always be met. This interval was established by considering dynamic priority scheduling schemes, such as Earliest Deadline. For static priority schemes we now show that, in general, a smaller interval is sufficient.

It is observed that the feasibility of an individual process in a static priority pre-emptive scheduling scheme depends only upon itself and other processes of higher priority than τ_i. Therefore, when determining the feasibility interval for $\tau_i \in \Delta^*$ we consider τ_i and those processes in Δ^* of higher priority. Considering the optimal priority assignment method introduced in section 5.3, all priority levels 1..i-1 are unassigned. Therefore, without loss of generality, we arbitrarily assign currently unassigned processes to those levels. When determining the feasibility interval, the specific assignments of levels 1..i-1 is unimportant, only that all unassigned processes have priority greater than τ_i.

Definition 5.1:

P_i is the least common multiple of the periods of processes $\tau_1..\tau_i$, i.e.

$$P_i = \underset{1 \le j \le i}{lcm}\left(T_j\right)$$

Definition 5.2:

The initial stabilisation time S_i of process τ_i is the time after which the execution of the process set repeats exactly every P_i with respect to $\tau_1..\tau_i$.

Consider the following theorem, providing a precise value for S_i:

Theorem 5.4:

For all processes τ_i $(1 \le i \le n)$ the initial stabilisation time S_i is defined by:

$$S_i = \left\lceil \frac{\max\limits_{1 \le j \le i}\left(O_j\right)}{T_i} \right\rceil T_i$$

Proof:

Initially, consider a critical instant process set (with a critical instant at time 0). Assuming all processes always take their WCET on each release (and no blocking occurs) we observe that if a process τ_i executes at time t it will also execute at time $t+P_i$. Formally:

$$execute(\tau_i, t) \Rightarrow execute(\tau_i, t+P_i)$$

This is an oft-stated property of static priority process sets (see for example Lemma 2 in [Leung80]).

We now extend this observation to processes with arbitrary offsets. Consider the behaviour of the highest priority process τ_1. It executes for the first C_1 time units in every interval

$$[O_1 + kT_1, O_1 + kT_1 + D_1)$$

where k is an integer such that $0 \leq k \leq \infty$. The behaviour of τ_1 is static: if it executes at t it will also execute at $t+P_i$.

The behaviour of τ_2 can be expressed in a similar manner: it executes for the first C_2 units in every interval

$$[O_2 + kT_2, O_2 + kT_2 + D_2)$$

not used by τ_1. If the initial release of τ_1 is after O_2, i.e. $O_1 > O_2$ then

$$execute(\tau_2, t) \not\Rightarrow execute(\tau_2, t+P_2)$$

since τ_2 may execute earlier in a release at $t<O_1$ than at the release $t+kP_2>O_1$. If we assume that the initial release of τ_1 is no later than that of τ_2, i.e. $O_1 \leq O_2$, we may assert

$$execute(\tau_2, t) \Rightarrow execute(\tau_2, t+P_2)$$

The argument can be continued until τ_i is reached. This process will reserve the first C_i units of computation time that are not required by processes $\tau_1..\tau_{i-1}$. Again, assuming that all $\tau_1..\tau_{i-1}$ have initial releases at or before O_i we may assert:

$$execute(\tau_i, t) \Rightarrow execute(\tau_i, t+P_i)$$

Therefore, we have built up the static execution requirements of all processes, assuming all higher priority processes have been released.

We may now state that S_i corresponds to $\max_{1 \leq j \leq i}(O_j)$. Since for S_i we are concerned with releases of τ_i, we may advance S_i to be the first release of τ_i at or after $\max_{1 \leq j \leq i}(O_j)$. That is:

$$S_i = \left\lceil \frac{\max_{1 \leq j \leq i}(O_j)}{T_i} \right\rceil T_i$$

The theorem is proved.

The definition of the feasibility interval for τ_i is completed by the following theorem:

Theorem 5.5:

Process τ_i is feasible if and only if the deadlines corresponding to releases of the process in $[S_i, S_i+P_i)$ are met.

Proof:

By Theorem 5.4 any τ_j $(1 \leq j \leq i)$ that executes at time $t \in [S_i, S_i+P_i)$ will also execute at $t+P_i$. Therefore the schedules in the following intervals will be identical (with respect to $\tau_1..\tau_i$):

$[S_i, S_i+P_i)$

$[S_i+P_i, S_i+2P_i)$

....

$[S_i+mP_i, S_i+(m+1)P_i)$

where m is a positive integer. If a process deadline is missed at d from the beginning of one interval, it will be missed at d from the beginning of all intervals above. Therefore, it is sufficient to check the deadlines of one interval only, so proving the theorem.

The offsets assumed in the above discussion could be arbitrarily large. To aid discussion in subsequent sections we may refine offsets without altering relative process phasing, such that $O_i < T_i$ $(1 \leq i \leq n)$. Also, we may ensure that when considering the feasibility of τ_i, O_i is zero. The following definition introduces modified offsets, with the subsequent theorem defining them.

Definition 5.3:

The modified offset of τ_j when considering the feasibility of τ_i is given by O_j^i.

Theorem 5.6:

The timing characteristics of processes $\tau_1..\tau_i$ can be modified without altering the relative phasing of those process releases such that:

(a) $$O_i^i = \min_{1 \leq j \leq i}(O_j) = 0$$

(b) $$\forall \tau_j \in \{\tau_1, \ldots, \tau_i\} \bullet O_i^i \in [0, T_j)$$

Proof:

We identify two cases:

(i) $O_j < O_i$ - O_j^i needs to be increased to be at least O_i;

(ii) $O_j > O_i$ - O_j^i needs to be reduced as much as possible whilst remaining $\geq O_i$.

To ensure that relative phasing of process releases is not affected, any increases or decreases to O_j are of the form lT_j where l is a positive integer (or zero). In both cases above, O_j^i must equal the least element of

$$\{O_j, O_j + T_j, O_j + 2T_j, \ldots\}$$

which is at least O_i. For case (i), if $O_j^i = O_j + lT_j$ we may define

$$l = \left\lceil \frac{O_j - O_i}{T_j} \right\rceil$$

For case (ii), if $O_j^i = O_j - lT_j$ we may define

$$l = \left\lfloor \frac{O_i - O_j}{T_j} \right\rfloor$$

An amount O_i may also be substituted from each O_j^i $(1 \leq j \leq i)$ without altering phasing of process releases. Thus, a full definition of O_j^i is given by:

$$O_j^i = \begin{cases} O_j + \left\lceil \frac{O_i - O_j}{T_j} \right\rceil T_j - O_i & \text{if } O_j < O_i \\ O_j - \left\lfloor \frac{O_j - O_i}{T_j} \right\rfloor T_j - O_i & \text{if } O_j \geq O_i \end{cases} \qquad (5.1)$$

This satisfies conditions (a) and (b): the theorem holds.

The modification process, together with the calculation of feasibility interval is illustrated by the following example.

Example 5.3:

Consider the process set in Table 5.4

Process	O	T
τ_1	50	10
τ_2	7	12
τ_3	26	20

Table 5.4 Example Process Set 4.

We note that no critical instant exists between the processes. Process releases occur at:

τ_1 : 50, 60, 70, 80, 90,...

τ_2 : 7, 19, 31, 42, 55,...

τ_3 : 26, 46, 66, 86, 106,...

Modified offsets, as given by equation (5.1) are:

$$O_1^3 = O_1 - \left\lfloor \frac{O_1 - O_3}{T_1} \right\rfloor T_1 - O_3 = 50 - \left\lfloor \frac{24}{10} \right\rfloor 10 - 26 = 4$$

$$O_2^3 = O_2 + \left\lceil \frac{O_3 - O_2}{T_2} \right\rceil T_2 - O_3 = 7 + \left\lceil \frac{19}{12} \right\rceil 12 - 26 = 5$$

$$O_3^3 = O_3 + \left\lceil \frac{O_3 - O_3}{T_3} \right\rceil T_3 - O_3 = 26 - 26 = 0$$

We note that both conditions of theorem 5.6 hold, and that process release phasings have been preserved. For example, originally the first release of τ_2 after O_3 occurs at 31, a difference of 31-26=5. With modified offsets, the first release of τ_2 after O_3 occurs at 5, again with a difference of 5 (noting that O_3^3=0). The feasibility interval may now be defined:

$$S_3 = \left\lceil \frac{\max(O_1^3, O_2^3)}{T_3} \right\rceil T_3 = \left\lceil \frac{14}{20} \right\rceil 20 = 20$$

Noting that P_3 = 60, the feasibility interval is given by [20, 80). This implies that three deadlines of τ_3 need to be checked, for releases at 20, 40 and 60.

5.3.1 Discussion

The feasibility intervals defined in the above sections are shorter than those proposed by Leung *et al* [Leung80, Leung82]. The latter interval was designed for testing earliest deadline feasibility, although it is also valid for static priority processes.

Clearly, any interval which is at least as long as the *lcm* of process periods is exponential in length. However, in practice, the interval defined above for static priority processes is less than the interval defined for dynamic priority processes. In general, the length of either interval is at a maximum when process periods are co-prime.

5.4 Sufficient And Necessary Feasibility

The optimal priority assignment method developed in section 5.2 requires a feasibility test. The test needs to be able to determine the feasibility of a process assigned a given priority level i ($1 \leq i \leq n$) assuming that priority levels $1..n$ have been assigned. We note that assignment of priority levels $1..i$-1 is arbitrary amongst processes not assigned priority levels $i..n$ and found feasible at those levels. Also, we assume that when considering the feasibility of τ_i, the offsets of $\tau_1..\tau_i$ have been modified according to Theorem 5.6 (section 5.3), that is O_j^i ($1 \leq j \leq i$) are set.

The remainder of this section is as follows. Sections 5.4.1 and 5.4.2 describe feasibility tests based upon schedule construction and the hybrid critical instant feasibility test of Chapter 4 respectively. Finally, section 5.4.3 provides a comparison of the tests.

5.4.1 Schedule Construction Sufficient And Necessary Feasibility Test

By Theorem 5.5 (section 5.3) the feasibility of a process with an arbitrary offset can be determined by checking that all deadlines in its feasibility interval are met. Notionally, the schedule needs to be constructed over the interval $[S_i, S_i+P_i)$. However, any process τ_j ($1 \leq j < i$) that has remaining computation at S_i must also be inserted into the schedule over $[S_i, S_i+P_i)$. Thus in general a schedule must be constructed over $[0, S_i+P_i)$. Since in the worst-case $S_i \approx P_i$ (actually $S_i+1=P_i$ in the worst-case), the length of the interval over which the schedule is constructed becomes $2P_i$. We note that the length of interval proposed by Leung *et al* [Leung80, Leung82] is always $2P_n$.

An empty schedule of length S_i+P_i is formed. Initially, τ_1 reserves the first C_1 slots in each interval $[t, t+D_1)$ where t represents a release of the process in $[0, S_i+P_i)$. Successively, $\tau_2..\tau_i$ also reserve slots. If sufficient slots cannot be allocated to a release of τ_i at $t \in [S_i, S_i+P_i)$ then within the context of the optimal priority assignment method, τ_i is not feasible at this priority level. In contrast, if all deadlines of τ_i are met, it is feasible at priority level i.

The complexity of this approach is dependent upon the length of the schedule which is not a polynomial function of n. Hence the complexity is NP-hard [Leung82]. We note that the size of data required (i.e. the length of the

schedule) is also non-polynomial. Thus, the schedule would be progressively more difficult to construct as the *lcm* of all process periods increases.

Example 5.4:

We extend example 5.3 (section 5.3). Values for process computation and deadline are added to Table 5.4 to generate the process set in Table 5.5.

Process	O_i	O_i^3	C_i	D_i	T_i
τ_1	50	4	3	5	10
τ_2	7	5	3	6	12
τ_3	26	0	2	8	20

Table 5.5 Example Process Set 5.

We note P_3=60 and S_3=20 (from example 5.2 section 5.3). Hence we construct a schedule over the interval [0,80). This is shown in Figure 5.2. The priority of the process executing in a slot is shown for each time slot in the schedule. The deadlines of τ_3 are shown by double bars (‖): all deadlines of τ_3 are met.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Process	3	3	-	-	1	1	1	2	2	2	-	-	-	-	1	1	1	2	2	2

Time	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Process	3	3	-	-	1	1	1	-	-	2	2	2	-	-	1	1	1	-	-	-

Time	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Process	3	2	2	2	1	1	1	3	-	-	-	-	-	2	1	1	1	2	2	-

Time	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Process	3	3	-	-	1	1	1	2	2	2	-	-	-	-	1	1	1	2	2	2

Figure 5.2: Schedule for Process Set 5.

5.4.2 Hybrid Sufficient And Necessary Feasibility Test

The previous section developed a schedule construction feasibility test. As Chapter 4 showed, this approach can be expensive. A more efficient approach is now developed.

To determine the feasibility of process τ_i we are required to check each deadline of τ_i corresponding to a release in $[S_i, S_i+P_i)$ (by Theorem 5.5). Consider a release of τ_i at $t \in [S_i, S_i+P_i)$. Within the interval $[t, t+D_i)$ τ_i is prevented from executing by:

(i) interference of $\tau_1 .. \tau_{i-1}$ due to releases in $[t, t+D_i)$;

(ii) interference of $\tau_1 .. \tau_{i-1}$ due to releases in $[0, t)$.

Case (i) corresponds to the definition of interference for critical instant processes in Chapter 4. Case (ii) recognises that at time t, there may exist some outstanding computation of $\tau_1 .. \tau_{i-1}$.

Any of the sufficient and necessary tests developed in Chapter 4 for critical instant processes sets could be developed for non-critical instant process sets. The hybrid test (section 4.3.3) is chosen as both the response time and exact interference is calculated.

The following definition is introduced:

Definition 5.4:

J_i^t represents the outstanding computation of $\tau_1 .. \tau_{i-1}$ at time t.

The above definition enables the feasibility constraint for a release of τ_i at $t \in [S_i, S_i+P_i)$ to be stated:

$$\exists t' \in [t, t+D_i) \bullet \quad C_i + J_i^t + I_i^{t,t'} = t' - t \tag{5.2}$$

We note that the hybrid feasibility test calculates response time and exact interference. Therefore, the following definitions are introduced:

Definition 5.5:

$I_i^{t,exact}$ represents the exact interference on τ_i by $\tau_1 .. \tau_{i-1}$ in $[t, t+D_i)$.

Definition 5.6:

R_i^t represents the response time of τ_i for a release at $t \in [S_i, S_i+P_i)$.

Extending the logic of the hybrid critical instant test (see section 4.3.3), for each release of τ_i at $t \in [S_i, S_i+P_i)$ the value J_i^t needs to be established. Then, R_i^t can be calculated, followed by $I_i^{t,exact}$. Initially, a method for calculating J_i^t for any $t \in [0, S_i+P_i)$ is presented. This is used for determining R_i^t and $I_i^{t,exact}$.

Calculation of J_i^t

The intuitive method for calculating J_i^t is to find all (non-overlapping) execution intervals of $\tau_1..\tau_{i-1}$ in $[0, J_i^t)$. However, if values of $J_i^{t_0}$ $(t_0 < t)$ have already been found, this result can be used when evaluating J_i^t. If more than one such $J_i^{t_0}$ exists, we select the one with the greatest value of t_0. At least one previous value of $J_i^{t_0}$ is available, when $t_0=0$, giving $J_i^0=0$.

Let $J_i^{t_0}=0$. Now, only executions of $\tau_1..\tau_{i-1}$ in $[t_0, t)$ need be considered. This can be achieved by the method of section 4.3.2, which finds the execution intervals of $\tau_1..\tau_{i-1}$ in a given interval. We note that there may be no execution intervals (of $\tau_1..\tau_{i-1}$) starting in the interval (c.f. the critical instant exact interference test (section 4.3.2) where there is guaranteed to be at least one execution interval). Let the final interval identified be $[s_m^i, e_m^i)$ where s_m^i and e_m^i are defined in section 4.3.2. If no intervals exists, $s_m^i=t$. The value of J_i^t is given by:

$$J_i^t = \max\left(0, I_i^{s_m^i,t} - \left(t - s_m^i\right)\right)$$

If $J_i^{t_0}>0$ this remaining computation must be accounted for when calculating J_i^t (assuming that if no execution intervals were found in $[t_0, t)$ then $s_m^i=t$). The remaining computation at t_0 may affect the computation of interference in $[s_m^i, t)$ if the following condition holds:

$$J_i^{t_0} + I_i^{t_0,s_m^i} > s_m^i - t_0$$

Thus we may state:

$$J_i^t = \max\left(0, I_i^{s_m^i,t} - \left(t - s_m^i\right)\right) + \max\left(0, J_i^{t_0} + I_i^{t_0,s_m^i} - \left(s_m^i - t_0\right)\right)$$

We note that at least one such $J_i^{t_0}$ $(t_0 \le t)$ is always available, as $J_i^0=0$ (no process is released before time 0).

The above calculation is illustrated by the algorithm in Figure 5.3. The function calculates J_i^{end} having been passed `i` as a parameter. The `rem` parameter defines J_i^{start} where `start` ≤ `end`.

```
function J (i, Δ*, start, end, rem) returns integer is
begin
    t = start ;
    incomplete = TRUE   ;
    while incomplete loop
        /* find the next release of process      */
```

```
/* τ1..τi-1 in [t, Di)                                          */
s = end    ;
for j in 1..i-1 loop
     if start ≤ Oji + ⌈(t − Oji)/Tj⌉ Tj < s then
          index = j ;
          s = Oji + ⌈(t − Oji)/Tj⌉ Tj
     end if     ;
end loop  ;
if s < end then
     /* find end of execution interval  */
     t = s + Cindex     ;
     while Ii^s,t + s > t and Ii^s,t + s ≤ end loop
          t = Ii^s,t + s;
     end loop  ;
     if Ii^s,t + s > end then
          /* end of execution interval  */
          /* at or past end             */
          incomplete = FALSE  ;
          end if     ;
     else incomplete = FALSE ;
     end if     ;
end loop  ;
/* last execution starts at tr */
return     max(0, Ii^s,end − (end − s)) + max(0, rem + Ii^start,s − (s − start))  ;
end J     ;
```

Figure 5.3: Algorithm for Calculating J_i^t.

An alternative method for calculating J_i^t ($t \in \{S_i, S_i+T_i, .., S_i+P_i\}$) can be formulated by finding the latest time at which there is no outstanding computation in [0, t). This can be achieved without finding all execution intervals of processes $\tau_1..\tau_{i-1}$ as in the method above. Let $t_l \in [0, t)$ be such a time. We may state:

$$J_i^t = \max(0, I_i^{t_l,t} - (t - t_l))$$

In general, the computation of t_l is non-trivial. In some constrained cases t_l can be found by inspection. For example, if for all processes of higher priority than τ_i, time t falls between the deadline of one release and the next release of τ_i, we observe $t_l=t$ and $J_i^t=0$.

Feasibility Test

Given that J_i^t can be evaluated, we now develop the feasibility test. Consider the release of τ_i at $t=S_i$. J_i^t is found by the call:

$$J_i^t = \mathtt{J}\ (\ \mathtt{i},\ \Delta^*,\ 0,\ S_i,\ 0\ )$$

Now the feasibility of this release of τ_i can be determined by finding R_i^t. This is achieved in a similar manner to section 4.3.1, with the series of time points at which we check the feasibility of τ_i given by:

$$t_0 = t + J_i^t + C_i$$
$$t_1 = t + J_i^t + I_i^{t,t_0} + C_i$$
$$t_2 = t + J_i^t + I_i^{t,t_1} + C_i$$
$$\ldots.$$
$$t_k = t + J_i^t + I_i^{t,t_{k-1}} + C_i$$

At each of these points in time we evaluate:

$$t + J_i^t + C_i + I_i^{t,t_k} = t_k$$

If for any value of $t_k \in [t, t+D_i)$ the constraint holds, τ_i is feasible for the release at t., with t_k identified as the point in time where τ_i completes in $[t, t+D_i)$. Hence the response time is given by:

$$R_i^t = t_k - t$$

Now, we proceed to calculate $I_i^{t,exact}$. Initially, we determine the outstanding computation of $\tau_1..\tau_{i-1}$ at $t+D_i$ by the call (noting that no outstanding computation due to $\tau_1..\tau_i$ exists at R_i^t+t):

$$J_i^{t+D_i} = \mathtt{J}\ (\ \mathtt{i},\ \Delta^*,\ R_i^t+t,\ t+D_i,\ 0\ )$$

Now, the exact interference on τ_i in $[t, t+D_i)$ can be stated:

$$I_i^{t,exact} = I_i^{t,t+D_i} + J_i^t - J_i^{t+D_i}$$

By Theorem 5.5 we need to repeat for all releases of τ_i at $t \in [S_i, S_i+P_i)$ to determine feasibility. We note that $J_i^{t+D_i}$ forms the seed required for determining the feasibility of the release of τ_i at $t+T_i$.

In section 4.3.3 it was noted that the hybrid test for critical instant process sets had NP-complete complexity. Also, Leung *et al* note that

determining the feasibility of a process set without a critical instant is NP-hard [Leung80]. The complexity of the method given in this section is $O(jkn^2)$ where k represents the maximum number of time points that are examined when determining the feasibility of one release of a process; j represents the number of deadlines of a process that need to be checked. In the worst-case:

$$k = \max_{1 \le i \le n}(T_i)$$

$$j = 2P_n$$

The value of k recognises that testing the feasibility of τ_i requires $\max_{1 \le i \le n}(D_i)$ time points with the calculation of J_i^t requiring an additional $\max_{1 \le i \le n}(T_i - D_i)$ time points. The value of j recognises that in the worst-case the interval $[0, 2P_n)$ can have $2P_n$ releases of τ_i. We note that the product jk is a constant for a process set (being dependent upon the fixed timing characteristics) implying that the complexity becomes pseudo-polynomial.

The feasibility test is illustrated by the algorithm in Figure 5.4. The function tests the feasibility of the process assigned priority level i in Δ^*, checking all deadlines for releases in [start, end). The function returns $S_i + P_i$ if the process is feasible, else the time of the first infeasible release.

```
function sn_offset_test ( i, Δ*, start, end )
                                        return integer is
    feasible = TRUE      ;
    release = start      ;
begin
    /* Calc the remaining interference at start  */
    Jit = J ( i, Δ*, 0, start, 0 ) ;

    /* For each release in [start, end) check τi  */
    /* deadline                                   */
    while start ≤ release < end and feasible loop
        /* calc the response time for this release */
        t = Jit + Ci + release    ;
        while release + Jit + I_i^release,t + Ci > t
                  and Jit + I_i^release,t + Ci ≤ Di loop
            t =  release + Jit + I_i^release,t + Ci      ;
        end loop  ;
        if I_i^release,t + Ci + Jit > Di
```

```
        feasible = FALSE ; /* τ_i infeasible */
    else        /* τ_i feasible - record R_i^release   */
        R_i^release = t - release ;
    end if    ;
    if feasible then
        /* calc remaining comp at release+D_i */
        rem = J ( i, Δ*, release + R_i^release ,
                                    release + D_i, 0)    ;
        /* calc exact interference for τ_i    */
        I_i^release,exact = I_i^release,release+D_i + Jit - rem    ;
                    /* calc remaining comp at release+T_i */
                    Jit = J ( i, Δ*, release + D_i,
                                release + T_i, rem )  ;
                    release = release + T_i          ;
            end if    ;
        end loop  ;
        return    release   ;
    end sn_offset_test ;
```

Figure 5.4: Algorithm For Sufficient and Necessary Feasibility Test For Processes With Arbitrary Offsets.

Example 5.5:

We return to the process set given in Table 5.5 (considered previously in example 5.4). Process τ_A is assigned priority level 2. According to Theorem 5.5 we may modify the process offsets such that O_A=0 and O_B=6. Thus, the feasibility interval is $[0, S_A + P_A)$ that is [0, 16). Hence, for the releases of τ_A at 0, 4, 8 and 12 we must ensure that the deadline is met. The test is summarised in Table 5.6.

Release at (t)	J_i^t	R_i^t	$t + D_i$	Feasible
0	0	2	3	✓
4	0	6	7	✓
8	1	11	11	✓
12	0	14	15	✓

Table 5.6: Summary Of Feasibility.

5.5 Sufficient And Not Necessary Feasibility

The optimal priority method detailed in section 5.3 relies upon the availability of a sufficient and necessary feasibility test. This was seen to be computationally expensive in section 5.4. This section examines the effectiveness of the optimal priority assignment method when used in conjunction with a sufficient and not necessary test. The advantage of using such a test is the decrease in complexity, although at a cost of decreased accuracy.

The optimal priority method is built upon the premise that the feasibility test employed has the following characteristics:

(i) the feasibility of a process can only improve if assigned a higher priority level;

(ii) the feasibility of a process can only decrease if assigned a lower priority level.

The sufficient and necessary tests in the previous section comply with the above criteria, where the only variable in determining feasibility is interference: interference on a process does not decrease if moved to a lower priority level, and does not increase if assigned a higher priority level.

Consider a sufficient and not necessary test which complies with the above criteria. Now, if a priority assignment exists over processes in Δ^* that could be declared feasible by the sufficient and not necessary test, then the priority assignment method would find it (in conjunction with the sufficient and not necessary test). If no feasible priority assignment is found (i.e. if for a particular priority level no feasible process can be found when assigned that priority level), then there still may exist a feasible priority assignment which could be found by either using a more accurate sufficient and not necessary test, or by a sufficient and necessary test.

The sufficient and not necessary tests developed for critical instant process sets in section 4.2 are not directly relevant: by assuming a critical instant, a deadline monotonic priority assignment may as well be used (since it is optimal in such circumstances) with the advantages of arbitrary offsets lost. To develop a sufficient and not necessary test to determine the feasibility of the process assigned priority level i (i.e. τ_i) requires:

(i) the determination of the release of τ_i in $[S_i, S_i+P_i)$ at which τ_i suffers its highest interference;

(ii) a sufficient evaluation of interference for a given release of τ_i.

If both can be achieved in polynomial time, a test of polynomial complexity is available. However, it is unclear how the worst-case release can be found in polynomial time amongst a potentially exponential number of releases of τ_i in its feasibility interval (c.f. critical instant feasibility when it is easily proved that the release of a process at a critical instant is the one which suffers greatest interference [Layland73]). Therefore, the feasibility of each release of τ_i is considered in turn. For each release, an approximation of interference is made that is at least the exact value, yielding a sufficient and not necessary test (by Theorem 4.2).

Definition 5.7:

$I_i^{t,release}$ is the interference of processes $\tau_1..\tau_{i-1}$ upon τ_i for a release of the latter process at $t \in [S_i, S_i+P_i)$.

With the above definition, the basic feasibility constraint becomes:

$$\forall t \in \{S_i, S_i+T_i, \ldots, S_i+P_i-T_i, S_i+P_i\} \bullet$$
$$I_i^{t,release} + C_i \leq D_i$$

To aid the discussion in subsequent sections, the following definitions are introduced:

Definition 5.8:

When considering the feasibility of τ_i, the last release of τ_j ($j<i$) at or prior to time t is denoted by:

$$p(i, j, t) = O_j^i + \left\lfloor \frac{t-O_j^i}{T_j} \right\rfloor T_j$$

Definition 5.9:

When considering the feasibility of τ_i, the next release of τ_j ($j<i$) after time t is denoted by:

$$n(i, j, t) = O_j^i + \left\lfloor \frac{t-O_j^i}{T_j} \right\rfloor T_j + T_j$$

Definition 5.10:

The notation $(x)_0$ is equivalent to $\max(0, x)$.

We note that definitions 5.8 and 5.9 assume that $t \geq 0$.

The following four sections extend the principles of the sufficient and not necessary tests in Chapter 4 for processes with arbitrary offsets.

5.5.1 Sufficient and Not Necessary Feasibility Test No. 1

For a release of τ_i at t, the interference of τ_j is composed of:

(i) releases of τ_j after t (before $t+D_i$), and

(ii) the release of τ_j at or prior to t.

This is shown in Figure 5.5 where the interference in case (i) is maximised by assuming that τ_j executes in intervals $[t_1, t_1+C_j)$ and $[t_2, t_2+C_j)$ for each of its releases in $[t, t+D_i)$ where $t \in [S_i, S_i+P_i)$. The interference in case (ii) is maximised by letting the release of τ_j at t_0 prior to t execute as much as possible in the interval $[t, t+C_j)$.

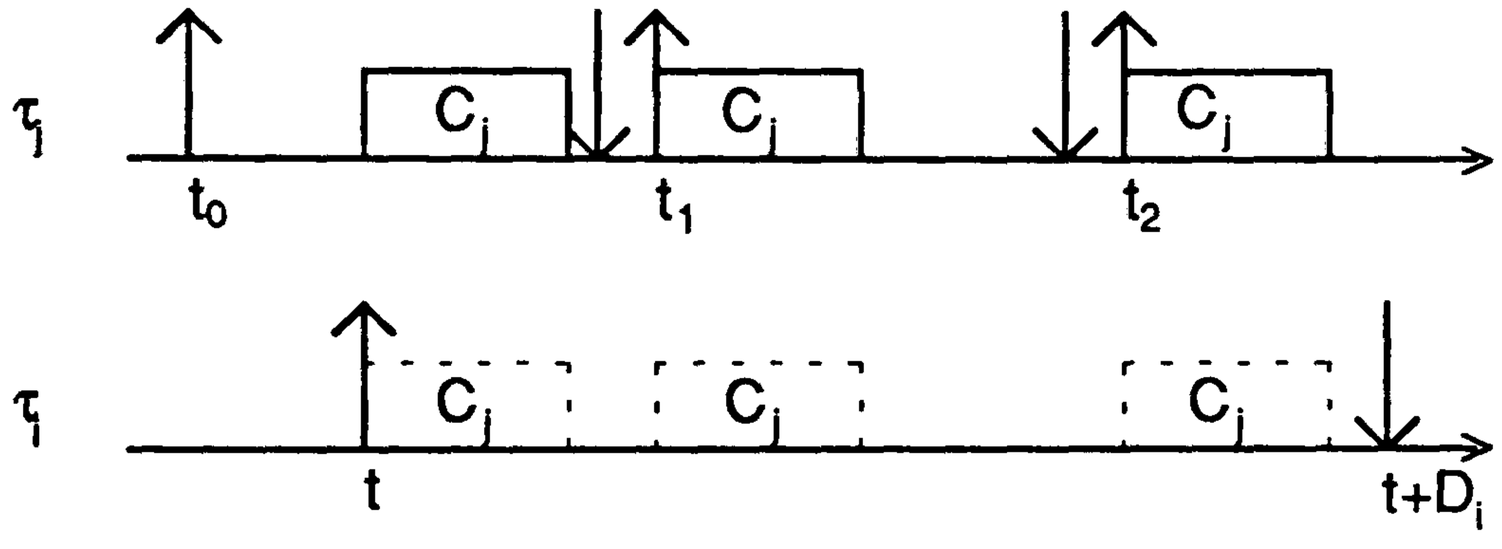


Figure 5.5: Estimating Interference.

Consider the following theorem.

Theorem 5.8:

The maximum interference of τ_j upon τ_i $(1 \le j < i \le n)$ for a release of τ_i at $t \in [S_i, S_i+P_i)$ is given by:

$$\min\left(\left(p(i,j,t)+D_j-t\right)_0, C_j\right)+\left\lceil\frac{\left(t+D_i-n(i,j,t)\right)_0}{T_j}\right\rceil C_j \quad (5.3)$$

Proof:

This is a simple extension of Theorem 4.3, noting that the last release of τ_j at or before t (i.e. in $[0, t]$) is given by $p(i,j,t)$. The left hand part of equation (5.3) limits the interference of this release on τ_i to be the minimum of C_j and the amount of time after t before the deadline of the release of τ_j.

The subsequent release of τ_j (i.e. the first release of τ_j in $(t, t+D_i)$) is given by $n(i,j,t)$. The right hand part of equation (5.3) limits the interference on τ_i to C_j for every release of τ_j in $(t, t+D_i)$. Note that if there is no release of τ_j in $(t, t+D_i)$ then the right hand

evaluates to 0. If a release of τ_j occurs at t, the interference of this release is calculated by the left hand part of equation (5.3) with releases strictly after t catered for by the right hand part.

We may state the feasibility test by:

$$\forall t \in \{S_i, S_i + T_i, \ldots, S_i + P_i - T_i, S_i + P_i\} \bullet$$
$$I_i^{t,release} + C_i \leq D_i$$

where

$$I_i^{t,release} = \sum_{j=1}^{i-1} \left(\min\left(\left(p(i,j,t) + D_j - t \right)_0, C_j \right) + \left\lceil \frac{\left(t + D_i - n(i,j,t) \right)_0}{T_j} \right\rceil C_j \right)$$

We note that by Theorem 4.2 this test is sufficient and not necessary since interference is at least the exact value (by Theorem 5.8). The estimation of interference has $O(n^2)$ complexity, although the overall complexity of the tests is non-polynomial due to having to check all releases of processes in their respective feasibility intervals.

Under constrained circumstances this test is also sufficient and necessary:

Theorem 5.9:

When considering the feasibility of τ_i by the test defined by equation (5.3), if the following condition holds the test is necessary:

$$\forall t \in \{S_i, S_i + T_i, \ldots, S_i + P_i - T_i, S_i + P_i\} \bullet$$
$$\forall j : 1 \leq j < i \bullet$$
$$p(i,j,t) + D_j \leq t \wedge p(i,j,t + D_i) + D_j \leq t + D_i$$

Proof:

We note that I_1^t is exact for any t as there are no higher priority processes. In general, the interference of τ_j $(1 \leq j < i \leq n)$ upon τ_i in $[t, t+D_i)$ is due to the last release of τ_j prior to the interval, and subsequent releases of τ_j within the interval. In the former case, if the deadline of the release is constrained to be before t, no interference will be due to that release. In the latter case, if the deadline of the final release of τ_i in the interval has a deadline at or before t the estimation of interference will be exact. Thus, if the condition in the theorem holds, the estimation of interference given in equation (5.3) is exact, ensuring that the feasibility test is necessary (by Theorem 4.1).

5.5.2 Sufficient and Not Necessary Feasibility Test No. 2

We note that the test in the previous section is inaccurate for similar reasons as the test in section 4.2.1: execution assumed to occur before $t+D_i$ (when calculating $I_i^{t,release}$) cannot actually occur until after $t+D_i$. For example, if the final release of τ_j $(1 \le j < i \le n)$ in $[t, t+D_i)$ is at t', with $t'+C_j > t+D_i$, the estimation of interference will exceed the exact value by (at least) $t'+C_j - (t+D_i)$.

Theorem 5.10:

The maximum interference of τ_j upon τ_i $(1 \le j < i \le n)$ for a release of τ_i at $t \in [S_i, S_i+P_i)$ is given by:

$$\min\left(\left(p(i,j,t)+D_j-t\right)_0, C_j\right) + \left\lfloor \frac{\left(t+D_i-p(i,j,t)\right)_0}{T_j} \right\rfloor C_j + \min\left(C_j, t+D_i-p\left(i,j,t+D_i\right)\right)$$

Proof:

Extending the proof of Theorem 5.8, we note that the first clause gives interference due to the last release of τ_j at or before t. The second clause determines interference for releases of τ_j at t' where $t < t' < t'+T_j \le t+D_i$. The third clause limits interference by a release of τ_j at $t' \in (t, t+D_i]$ where $t'+T_j > t+D_i$, to be no more than the length of the interval $[t', t+D_i)$.

The feasibility test may be stated by:

$$\forall t \in \left\{S_i, S_i+T_i, \ldots, S_i+P_i-T_i, S_i+P_i\right\} \bullet I_i^{t,release}+C_i \le D_i$$

where

$$I_i^{t,release} = \sum_{j=1}^{i-1} \left(\min\left(\left(p(i,j,t)+D_j-t\right)_0, C_j\right) + \left\lfloor \frac{\left(t+D_i-p(i,j,t)\right)_0}{T_j} \right\rfloor C_j + \min\left(C_j, t+D_i-p\left(i,j,t+D_i\right)\right) \right)$$

We note that the above test is sufficient and not necessary since interference is at least the exact value (by Theorem 5.10). The test has $O(n^2)$ complexity for interference determination, although its overall complexity is non-polynomial. The test is necessary under the same conditions as defined by Theorem 5.9.

5.5.3 Sufficient and Not Necessary Feasibility Test No. 3

The estimation of $I_i^{t,release}$ given in the previous section is pessimistic in a similar manner to the critical instant sufficient and not necessary test in section 4.2.2: concurrent execution is assumed between processes during their final release in $[t,t+D_i)$. Section 4.2.3 showed how this pessimism may be reduced by calculating the effective deadline of τ_i.

A second form of concurrent execution can be identified, peculiar to the calculation of interference for offset processes, where processes τ_j and τ_k ($1 \leq j < k < i \leq n$) are assumed to execute in $[t, t+C_j)$ and $[t, t+C_k)$ respectively. Clearly concurrent execution is in evidence. This pessimism can be reduced by calculating the *effective release* of τ_i. This is illustrated by considering τ_j and τ_k introduced above. If τ_j executes in $[t, t+C_j)$, the effective release of τ_i when considering τ_k is at $t' = t+C_j$ (for the purposes of feasibility determination only). Now, unless τ_k can execute for the entirety of $[t', t'+C_k)$, the estimation of interference will be reduced.

Definition 5.11:

The *effective deadline* of τ_i for a release at $t \in [S_i, S_i+P_i)$ when considering the interference of τ_j upon τ_i is denoted $d_j^{i,t}$ (relative to t). The interval $[t+d_j^{i,t}, t+D_i)$ is occupied entirely by the executions of processes of higher priority than τ_j.

Definition 5.12:

The *effective release* of τ_i for an actual release at $t \in [S_i, S_i+P_i)$ when considering the interference of τ_j upon τ_i is denoted $r_j^{i,t}$. In the worst-case, the interval $[t, r_j^{i,t})$ is occupied entirely by the executions of processes of higher priority than τ_j.

Consider the following theorem:

Theorem 5.11:

The maximum interference of τ_j upon τ_i $(1 \le j < i \le n)$ for a release of τ_i at $t \in [S_i, S_i+P_i)$ is given by:

$$\min\left(\left(p\left(i,j,r_j^{i,t}\right)+D_j-r_j^{i,t}\right)_0,C_j\right)+\left\lfloor\frac{\left(t+d_j^{i,t}-n\left(i,j,r_j^{i,t}\right)\right)_0}{T_j}\right\rfloor C_j$$
$$+\min\left(C_j,t+d_j^{i,t}-p\left(i,j,t+d_j^{i,t}\right)\right)$$

Proof:

Extending the proof of Theorem 4.6, we note that interference of τ_j upon τ_i can only be due to executions in $[r_j^{i,t}, t+d_j^{i,t})$ since $[t, r_j^{i,t})$ and $[t+d_j^{i,t}, t+D_i)$ are occupied by the execution of higher priority processes (i.e. $\tau_1..\tau_{i-1}$). We note that the exact interference of processes executing in the interval $[t, r_j^{i,t})$ (that is the length of the interval) may be less than $r_j^{i,t}-t$, but never more. Thus the interference is at least the exact amount.

The total interference of $\tau_1..\tau_{i-1}$ upon τ_i must include the lengths of the intervals $[t, r_{j-1}^{i,t})$ and $[d_{i-1}^{i,t}, D_i)$. The feasibility test may be stated by:

$$\forall t \in \{S_i, S_i+T_i, \ldots, S_i+P_i-T_i, S_i+P_i\} \bullet$$
$$I_i^{t,release} + C_i \le D_i$$

where

$$I_i^{t,release} = \left(r_{j-1}^{i,t}-t\right)+\left(D_i-d_{i-1}^{i,t}\right)$$
$$+\sum_{j=1}^{i-1}\left(\min\left(\left(p\left(i,j,r_j^{i,t}\right)+D_j-r_j^{i,t}\right)_0,C_j\right)+\left\lfloor\frac{\left(t+d_j^{i,t}-n\left(i,j,r_j^{i,t}\right)\right)_0}{T_j}\right\rfloor C_j\right.$$
$$\left.+\min\left(C_j,t+d_j^{i,t}-p\left(i,j,t+d_j^{i,t}\right)\right)\right)$$

$$d_j^{i,t}=\begin{cases} D_i & \text{if } j=1 \\ O_{j-1}^i+p\left(i,j-1,t+d_{j-1}^{i,t}\right)-t & \text{if } t+d_{j-1}^{i,t}-p\left(i,j-1,t+d_{j-1}^{i,t}\right)\le C_{j-1} \\ d_{j-1}^{i,t} & \text{otherwise} \end{cases}$$

$$r_j^{i,t} = \begin{cases} t & \text{if } j = 1 \\ r_{j-1}^{i,t} + \min\left(C_{j-1},\ p\left(i, j-1, r_{j-1}^{i,t}\right) + D_{j-1} - r_{j-1}^{i,t}\right) & \\ & \text{if } p\left(i, j-1, r_{j-1}^{i,t}\right) \le r_{j-1}^{i,t} < p\left(i, j-1, r_{j-1}^{i,t}\right) + D_{j-1} \\ r_{j-1}^{i,t} & \text{otherwise} \end{cases}$$

The definition of $d_j^{i,t}$ reduces the effective deadline of a release of τ_i at t for process τ_j by the length of:

$$\left[p\left(i, j-1, t + d_{j-1}^{i,t}\right), t + d_{j-1}^{i,t}\right)$$

if and only if that length is no greater than C_{j-1}. The definition of $r_j^{i,t}$ increases the effective release of τ_i when considering τ_j by the maximum amount of execution a release of τ_{j-1} prior to t can occupy in $\left[r_{j-1}^{i,t}, t + D_i\right)$. We note that these definitions are adequate with respect to Theorem 5.11.

The test defined in this section is sufficient and not necessary by Theorem 4.2 since the estimation of interference is at least the exact value (by Theorem 5.11). The complexity of the approach is $O(n^2)$ for interference estimation at each release of τ_i in its feasibility interval, although the overall test has non-polynomial complexity. The test is necessary under the same conditions as defined by Theorem 5.9.

5.5.4 Sufficient and Not Necessary Feasibility Test No. 4

The estimation of interference given in the previous section may still assume concurrent execution of processes during the final release prior to t and during their final release before $t+D_i$, due to the inaccuracy of effective release and deadline estimation. For example, if all processes τ_j $(1 \le j < i)$ have a final release at $t+D_i-C_j+1$, then the effective deadline will be equal to D_i, with concurrent execution assumed between the final releases of higher priority processes.

In a manner similar to section 4.2.4, the estimation of interference may be improved by separating calculation of effective deadlines and releases from the determination of $I_i^{t,release}$. Initially, the effective release and deadline are calculated iteratively, considering processes $\tau_1 .. \tau_{i-1}$ in turn. Then $I_i^{t,release}$ is determined.

Definition 5.13:

The *effective deadline* of τ_i for a release at $t \in [S_i,\ S_i+P_i)$ when considering the interference of τ_j upon τ_i is denoted $d_{j,k}^{i,t}$ (relative to t), where k represents the current iteration (from 0 upwards). The interval $[t+d_{j,k}^{i,t},\ t+D_i)$ is occupied entirely by the executions of processes of higher priority than τ_i.

Definition 5.14:

The *effective release* of τ_i for an actual release at $t \in [S_i,\ S_i+P_i)$ when considering the interference of τ_j upon τ_i is denoted $r_{j,k}^{i,t}$, where k represents the current iteration (from 0 upwards). In the worst-case, the interval $[t,\ r_{j,k}^{i,t})$ is occupied entirely by the executions of processes of higher priority than τ_i.

Initially, for a release of τ_i at $t \in \{S_i,\ S_i+T_i,...,\ S_i+P_i\}$, $r_1^{i,t} = r_2^{i,t} = ... = r_{i-1}^{i,t} = t$ and $d_1^{i,t} = d_2^{i,t} = ... = d_{i-1}^{i,t} = t+D_i$. For each iteration, i.e. k=1, k=2 etc., each process $\tau_1 .. \tau_{i-1}$ is considered in turn with a view to improving the effective release and deadline. If there exists a release of τ_j $(1 \le j < i)$ at t' where $t' < r_{i-1,k}^{i,t} < t'+D_j$ then the effective release could be improved. Likewise, if there exists a release of τ_j $(1 \le j < i)$ at t' where $t'+C_j \ge t+d_{i-1,k}^{i,t}$ then the effective deadline can be improved (i.e. decreased). When the effective deadline and release can no longer be improved, the interference of $\tau_1 .. \tau_{i-1}$ can be calculated.

Consider the following theorem:

Theorem 5.12:

The maximum interference of τ_j upon τ_i $(1 \le j < i \le n)$ in the interval $[r_{i-1,k}^{i,t},\ t+d_{i-1,k}^{i,t})$ for a release of τ_i at $t \in [S_i,\ S_i+P_i)$ is given by:

$$\left\lceil \frac{t+d_{i-1,k}^{i,t} - \min\left(t+d_{i-1,k}^{i,t},\ p\left(i,j,r_{i-1,k}^{i,t}\right)\right)}{T_j} \right\rceil C_j$$

when the following condition holds:

$$\exists k : k>0 \bullet$$
$$\forall m : 1 \le m < i :\ d_{m,k-1}^{i,t} = d_{m,k}^{i,t} \wedge r_{m,k-1}^{i,t} = r_{m,k}^{i,t}$$

Proof:

We note that for the condition to hold, no release of any process τ_j $(1 \le j < i)$ can improve the effective release (by increasing it) or the effective deadline (by decreasing it). Therefore, the interference of

τ_j upon τ_i is equal to C_j for each release of the process in $[r_{i-1,k}^{i,t}, t+d_{i-1,k}^{i,t})$.

The first release of τ_j at or after $r_{i-1,k}^{i,t}$ is at $p\left(i,j,r_{i-1,k}^{i,t}\right)$. However, this may also be at or after $t+d_{i-1,k}^{i,t}$. Therefore, we constrain the first release of τ_j at or after $r_{i-1,k}^{i,t}$ to be no later than $t+d_{i-1,k}^{i,t}$. This gives the first release at:

$$f = \min\left(t+d_{i-1,k}^{i,t}, p\left(i,j,r_{i-1,k}^{i,t}\right)\right)$$

Now, the number of releases of τ_j in $[r_{i-1,k}^{i,t}, t+d_{i-1,k}^{i,t})$ is given by:

$$\left\lceil \left(t+d_{i-1,k}^{i,t}-f\right)/T_j \right\rceil$$

For each of these releases, the interference created by τ_j is C_j. Thus, the theorem holds.

The feasibility test must include the assumed interference due to process executions in the intervals $[t, r_{i-1,k}^{i,t})$ and $[t+d_{i-1,k}^{i,t}, t+D_i)$:

$$\forall t \in \left\{S_i, S_i+T_i, \ldots, S_i+P_i-T_i, S_i+P_i\right\} \bullet$$
$$I_i^{t,release} + C_i \le D_i$$

where

$$I_i^{t,release} = D_i - d_{i-1,k}^{i,t} + r_{i-1,k}^{i,t} - t + \sum_{j=1}^{i-1} \left\lceil \frac{t+d_{i-1,k}^{i,t} - \min\left(t+d_{i-1,k}^{i,t}, p\left(i,j,r_{i-1,k}^{i,t}\right)\right)}{T_j} \right\rceil C_j$$

$$d_{j,k}^{i,t} = \begin{cases} D_i & \text{if } (k=0) \vee (k=1 \wedge j=1) \\ p\left(i, \text{prev}_j^i, t+d_{\text{prev}_j^i,\text{iter}_k^j}^{i,t}\right) - t & \text{if } 0 < t+d_{\text{prev}_j^i,\text{iter}_k^j}^{i,t} - p\left(i, \text{prev}_j^i, t+d_{\text{prev}_j^i,\text{iter}_k^j}^{i,t}\right) \le C_{\text{prev}_j^i} \\ d_{\text{prev}_j^i,\text{iter}_k^j}^{i,t} & \text{otherwise} \end{cases}$$

$$r_{j,k}^{i,t} = \begin{cases} t & \text{if } j = 1 \\ r_{\text{prev}_j^i,\text{iter}_k^j}^{i,t} + \min\left(C_{\text{prev}_j^i}, p\left(i, \text{prev}_j^i, r_{\text{prev}_j^i,\text{iter}_k^j}^{i,t}\right) + D_{\text{prev}_j^i} - r_{\text{prev}_j^i,\text{iter}_k^j}^{i,t}\right) & \text{if } p\left(i, \text{prev}_j^i, r_{\text{prev}_j^i,\text{iter}_k^j}^{i,t}\right) \le r_{\text{prev}_j^i,\text{iter}_k^j}^{i,t} < p\left(i, \text{prev}_j^i, r_{\text{prev}_j^i,\text{iter}_k^j}^{i,t}\right) + D_{\text{prev}_j^i} \\ r_{\text{prev}_j^i}^{i,t} & \text{otherwise} \end{cases}$$

$$\text{prev}_j^i = \begin{cases} i-1 & \text{if } j = 1 \\ j-1 & \text{otherwise} \end{cases} \qquad \text{iter}_k^j = \begin{cases} k-1 & \text{if } j = 1 \\ k & \text{otherwise} \end{cases}$$

The definition of prev_j^i returns the priority of the process immediately higher than τ_j, with wrap-around so that $\text{prev}_1^i = i\text{-}1$. The definition of iter_k^j returns the iteration index of the last calculated value of $r_{j,k}^{i,t}$ or $d_{j,k}^{i,t}$ noting that $\text{iter}_k^1 = k\text{-}1$ as the last process considered was τ_{i-1} on iteration k-1.

The definition of $r_{j,k}^{i,t}$ assumes sufficient iterations so that $r_{j,k}^{i,t} = r_{j,k-1}^{i,t}$ for all τ_j $(1 \le j < i)$. If this condition is not reached, a release of a τ_j may occur before $r_{j,k}^{i,t}$ that could contribute to interference on τ_i in $[r_{i-1,k}^{i,t}, t+d_{i-1,k}^{i,t})$. However, this release would not be counted in the interference calculation above. The estimation of $d_{j,k}^{i,t}$ is sufficient if at least one iteration is performed (i.e. the k=1 iteration), although successive iterations (until $d_{j,k}^{i,t} = d_{j,k-1}^{i,t}$ for all τ_j) improve the estimate, implying the estimation of interference will become more accurate (or at least will not become more inaccurate).

The calculation of interference has complexity $O(kn^2)$ where k represents the number of iterations required to calculate the effective deadline and release. In general, this is NP-complete since k is not a polynomial function of n. We observe that iterations can stop if $r_{i-1,b}^{i,t} = t+d_{i-1,b}^{i,t}$ (where b represents an iteration before the final iteration k) as τ_i is definitely infeasible for this release. Therefore, in the worst-case, k is equal to D_i. The overall test has non-polynomial complexity.

We note that the test given above remains pessimistic due to assumed concurrent execution in two places. Firstly, the final release of a higher priority process τ_j at t' before t may not actually execute at or after t even if $t'+D_j > t$. Thus, the execution of higher priority processes actually executing at t (if any)

is assumed to be concurrent to the execution of τ_j. The second form of concurrent execution occurs on the final release of τ_j $(1 \leq j<i)$ before $t+d^{i,t}_{i-1,k}$ which is assumed to complete before $t+d^{i,t}_{i-1,k}$. However, higher priority processes (i.e. $\tau_1..\tau_{j-1}$) may prevent τ_j from completing before $t+d^{i,t}_{i-1,k}$.

5.5.5 Summary

This section has extended the critical instant sufficient and not necessary feasibility tests (section 4.2) for use with the priority assignment method developed in section 5.2. With such tests, the priority assignment method is still applicable: if a priority assignment exists that is feasible by a given sufficient and not necessary test, it will be found. If no feasible priority assignment is found using a sufficient and not necessary test, a feasible ordering may be found by adopting a more accurate test (in the extreme a sufficient and necessary test). The tests presented increase in accuracy, at a cost of increased complexity.

5.6 Arbitrary Precedence Relations

In this section one of the restrictions upon processes, namely independence, is removed. Specifically, processes with precedence relationships between them are considered. These are especially important when considering systems that use intermediate processes to transform input data to output action.

Precedence relationships take many forms. The simplest form is a *chain*. Here, a single process reads input data. This is then passed to the next process in the chain which transforms it and passes modified data to the next process. Eventually, the output process is reached. A more complex and general form is the *arbitrary acyclic graph* (AAG) where nodes in the graph represent processes, arcs represent precedence relationships between processes. The acyclic constraint is imposed to reflect the bounded execution requirement of hard real-time systems. We note that the chain is a simple form of an AAG.

The assumption is made that all processes in an AAG have equal periods. Precedence constraints are imposed by the inherent timing constraints of processes, and the assignment of priority levels to processes. The highest priority levels are assigned to processes at the start of an AAG. Priority levels are lowered across the graph, with the lowest priority level assigned to processes at the end of the graph.

After initial description of the model, we show the bottom-up priority assignment technique to be applicable for this model of precedence constraints without complicating feasibility analysis.

5.6.1 Model

The non-critical instant process set Δ^* is split into precedence related process sets $\Delta_k \subseteq \Delta^*$ each containing exactly one AAG:

$$\text{(i)} \qquad \bigcup_{1 \le k \le m \le n} \Delta_k = \Delta^*$$

$$\text{(ii)} \qquad \forall \Delta_j, \Delta_k : \Delta_j \cap \Delta_k = \varnothing \quad (1 \le j < k \le m \le n)$$

where m represents the number of distinct AAGs in Δ^*. This can be a maximum of n when no precedence constraints exist: all processes are independent with one process per AAG.

Within a AAG, precedence relationships are defined by $\tau_A \rightarrow \tau_B$ meaning τ_A "immediately precedes" τ_B : τ_B cannot execute until τ_A has completed. For each $\tau_A \in \Delta_k$ the following definitions are made:

Definition 5.15:

pred (τ_A) - the set of processes that must have completed before τ_A may execute.

Definition 5.16:

succ (τ_A) - the set of processes that cannot be executed until at least τ_A has completed.

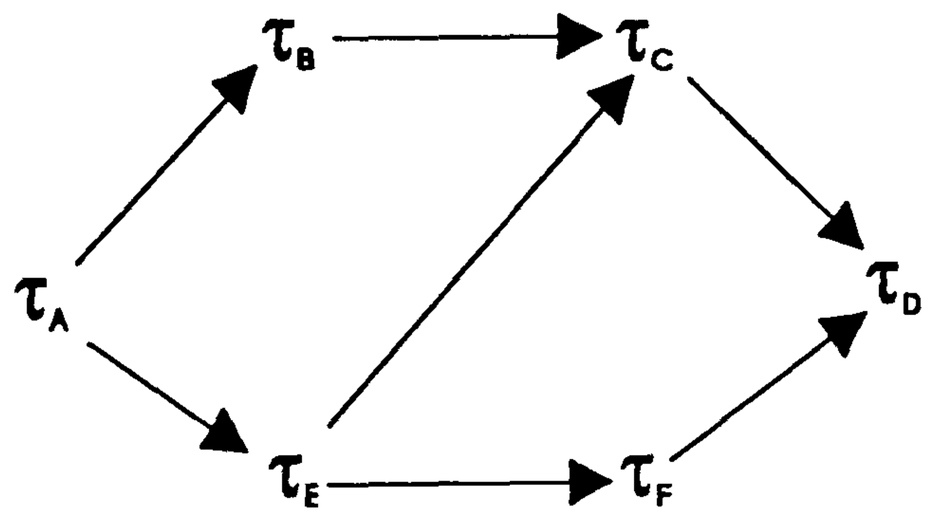


Figure 5.6: Example AAG

Consider the example AAG in Figure 5.6. The *pred* and *succ* sets for the processes are defined by:

$$pred(\tau_A) = \emptyset \qquad succ(\tau_A) = \{\tau_B, \tau_C, \tau_D, \tau_E, \tau_F\}$$
$$pred(\tau_B) = \{\tau_A\} \qquad succ(\tau_B) = \{\tau_C, \tau_D\}$$
$$pred(\tau_C) = \{\tau_A, \tau_B, \tau_E\} \qquad succ(\tau_C) = \{\tau_D\}$$
$$pred(\tau_D) = \{\tau_A, \tau_B, \tau_C, \tau_E, \tau_F\} \qquad succ(\tau_D) = \{\emptyset\}$$
$$pred(\tau_E) = \{\tau_A\} \qquad succ(\tau_E) = \{\tau_C, \tau_D, \tau_F\}$$
$$pred(\tau_F) = \{\tau_A, \tau_E\} \qquad succ(\tau_F) = \{\tau_D\}$$

5.6.2 Extending the Priority Assignment Technique

Initially, we observe that within any AAG a process will not execute until all its predecessors have completed. For arbitrary timing constraints, if $\tau_A \rightarrow \tau_B$ the situation may arise that $O_A > O_B$. Process τ_B cannot execute until at least O_A. This creates extra complexity for the feasibility test: it needs to account for τ_B not being runnable until O_A, even though the processor may be idle. Without loss of generality, offsets within an AAG can be re-arranged such that a process has an offset at least that of all its predecessors. The deadlines must keep the same relationship with the period: if O_A is increased by x, D_A must be decreased by x as D_A is relative to the release of τ_A. In general, for each process τ_A in each Δ_k (selected in order from start to finish of the AAG):

$$diff := \max_{\tau_J \in pred(\tau_A) \wedge O_J > O_A} (O_J - O_A)$$
$$O_A := O_A + diff$$
$$D_A := D_A - diff$$

Formally, priorities are assigned by Φ_i to processes in Δ^*. The processes within a Δ_k are assigned priorities such that:

$$\forall \tau_A \in \Delta_k :$$
$$\forall \tau_B \in pred(\tau_A) : \Phi_i(\tau_A) > \Phi_i(\tau_B) \wedge$$
$$\forall \tau_C \in succ(\tau_A) : \Phi_i(\tau_A) < \Phi_i(\tau_C)$$

This is a natural extension of the priority assignment technique. When assigning a process to a priority level, only those unassigned processes whose successor processes have all been assigned priorities are considered. Thus, the feasibility of τ_A can only be affected by predecessor processes and higher priority processes. Since all predecessor processes have higher priorities, the assumptions made in Theorems 5.1, 5.2 and 5.3 are not violated. Hence, the

optimal priority assignment technique is sufficient and necessary for this method of assigning priorities to processes with precedence constraints. In practise, the bottom-up priority assignment method may be adopted.

The offset and deadline re-arrangement, together with descending priorities, ensure that feasibility testing remains exactly equivalent to that employed for independent processes in section 5.2.

5.6.3 Algorithmic Implementation

The algorithmic implementation, given in Figure 5.7, is similar to that in Figure 5.5 for independent processes. Three sets are used: Δ^a, Δ^u and Δ. Respectively, these represent the processes that have been assigned priority levels (and are feasible); those that have not been assigned and cannot be considered for the current priority level; and those that have not been assigned but can be considered for the current priority level. At any time, $\Delta^a + \Delta^u + \Delta = \Delta^*$. At the beginning of the main loop, we determine if any more processes can be considered for the current priority level i.e. if any more processes have had all their predecessors assigned priority levels. The algorithm then proceeds as in Figure 5.4.

```
function priority_assignment (Δ*) returns boolean is
begin
      --  calculate pred succ sets
      Δ^u = Δ*;    -- unassigned processes
      Δ^a = ∅;     -- assigned processes
      Δ = ∅;       -- processes ready to be assigned

      for j in n..1 loop          -- priority level j
           unassigned = TRUE ;
           for τ_A in Δ^u loop
                if (succ (τ_A) ⊆ Δ^a) then
                     Δ^u = Δ^u - τ_A;
                     Δ = Δ + τ_A;
           end if ;
      end loop ;
      for τ_A in Δ loop
           if unassigned = TRUE then
                -- if τ_A is feasible at j
```

```
                    if feasible(τ_A, j, Δ) = TRUE
                    then
                        -- assign τ_A to pri j
                        Ψ(j)=τ_A;
                        Δ=Δ−τ_A ;
                        unassigned = FALSE ;
                    end if ;
                end if  ;
                exit when unassigned = FALSE  ;
            end loop ;
                if unassigned = TRUE then
                    return FALSE
                end if ;
            end loop ;
            return TRUE ;
        end priority_assignment ;
```

Figure 5.7: Priority Assignment With Precedence Constraints

5.6.4 Discussion

Whilst we observe that optimal priority assignment is achievable assuming descending priorities across an AAG, we note that the descending priority assignments for precedence constraints is not optimal amongst all priority assignment methods [Harbour91]. Consider the process set in Table 5.7, noting that $\tau_A \rightarrow \tau_B$.

Process	O	C	D	T
τ_A	0	4	5	10
τ_B	5	1	1	10
τ_C	0	1	2	5

Table 5.7 Example Process Set 6.

The only feasible priority assignment is τ_A - 3, τ_B - 1, τ_C - 2. Thus, ascending priorities are required along the precedence relation $\tau_A \rightarrow \tau_B$.

The complexity of the optimal priority assignment approach for descending priority precedence constraints is similar to that for independent

processes given in section 5.3. The worst-case of $(n^2+n)/2$ process feasibility tests occurs when processes are independent: exactly one process per AAG with n distinct AAGs. The worst-case overall complexity is $O(((n^2+n)/2)E)$ where E represents the complexity of the feasibility test (i.e. identical to that for independent processes).

The best case occurs when a single chain involving all processes in Δ^* exists. Now, there is exactly one process to choose from for each priority level assignment: the feasibility of n processes is determined.

We note that this method of handling precedence constraints imposes no extra burden on the feasibility test if offset and deadline re-arranging is performed: a sufficient and necessary test for independent processes is applicable.

5.7 Resources

In this section processes that share resources are considered. If these resources are not required in mutual exclusion, blocking cannot occur, with the consequence that the optimal priority assignment technique for independent processes can be used. When mutual exclusion is required, blocking may occur. The following discussion examines the impact of potential blocking on the optimal priority assignment technique, assuming that mutual exclusion is provided via locking and unlocking binary semaphores associated with a resource.

There are two main considerations for resources and the optimal priority assignment technique:

(i) bounded blocking times;

(ii) run-time behaviour of the resource allocation algorithm.

Potential blocking must be bounded, with worst case blocking times (WCBT) calculable *a priori* offline, to enable feasibility to be determined. Conventionally, this is achieved by adopting a run-time resource allocation protocol having the property of bounding all blocking. Unfortunately, WCBT estimations are pessimistic: it is difficult in the presence of arbitrary control-flow within process executions to predict *a priori* exactly when resources are locked and unlocked. Feasibility assessments based on such estimations are sufficient and not necessary. If clairvoyance were available to provide exact blocking times, sufficient and necessary feasibility may be possible.

This observation is considered further in the following sections. Initially, the validity of the optimal priority assignment method is considered when clairvoyant blocking times are available (that is the blocking times incurred by a release of a process is known before the system executes). This scenario is discussed in conjunction with both the Reservation Protocol of Babaoglu *et al* [Babaoglu90, Babaoglu93] and the Priority Inheritance Protocol defined by Sha *et al* [Sha90]. Then, the effects of removing clairvoyant blocking times and replacing them with pessimistic blocking times are considered.

5.7.1 Background Considerations

Essentially, Theorem 5.3 shows that given a feasible priority ordering with τ_A assigned priority level i, with τ_A also feasible at priority level $i+k$ (assuming the same processes assigned to priority levels $i+k+1..n$), a feasible priority ordering exists with τ_A assigned $i+k$. The proof is based on pair-wise swapping of process priority level assignments: i.e. priority levels i and $i+1$ are swapped, $i+1$ and $i+2$,..., $i+k-1$ and $i+k$. This assumes that the process with the higher priority before the swap finds it no easier to meet its deadline after the swap and the lower priority process finds it no harder. Also, the feasibility of other processes in Δ is not affected. Formally, for a release of τ_A at t (shown in the previous Theorems):

$$C_A + I_A^t(\Phi_j) \leq C_A + I_A^t(\Phi_k) \tag{5.4}$$

where Φ_j and Φ_k are the priority assignment functions before and after the swapping of τ_A from priority level i to $i+1$. The notation $I_A^t(\Phi_j)$ refers to the interference on τ_A for a release at t under priority ordering Φ_j.

With the inclusion of blocking, the estimation of WCBT needs to be included in the feasibility inequality:

$$C_A + I_A^t(\Phi_j) + B_A^t(\Phi_j) \leq C_A + I_A^t(\Phi_k) + B_A^t(\Phi_k) \tag{5.5}$$

where $B_A^t(\Phi_j)$ represents the WCBT for a release of τ_A at t under priority ordering Φ_j. For any calculation of blocking times and run-time resource allocation algorithm it must be shown that equation (5.5) holds.

5.7.2 Clairvoyant Blocking and the Reservation Protocol

Initially we examine the reservation protocol proposed by Babaoglu *et al* [Babaoglu90, Babaoglu93]. Each process reserves in advance the interval

during which it will hold a resource, with the highest priority process having first opportunity to reserve resources. A resource is only allocated to τ_A if it will not be required by a higher priority process during the interval requested by τ_A. Under this protocol, blocking occurs when a resource request is denied, although this is never due to a lower priority process holding the resource. The blocking time equates to the elapsed time before the resource is available. A block may occur on each resource access.

We assume the presence of clairvoyance for calculating blocking times: for any release of τ_A at time $t \in \{O_A, O_A+T_A, O_A+2T_A,...\}$ the exact length of block experienced by τ_A is known (B_A^t). The applicability of the priority assignment method with the reservation protocol assuming clairvoyant blocking is shown by the following theorem.

Theorem 5.13:

Consider two priority orderings Φ_j, $\Phi_k \in \Phi$ over Δ^*. Both assign the same process to priority levels $1..i\text{-}1, i+2..n$ with $\Phi_j(\tau_A) = i$, $\Phi_j(\tau_B) = i+1$; $\Phi_k(\tau_A) = i+1$, $\Phi_k(\tau_B) = i$. Let clairvoyant blocking calculations be available, with the reservation protocol employed for run-time resource allocation. The following conditions hold:

(1) $C_A + I_A^t(\Phi_j) + B_A^t(\Phi_j) \leq C_A + I_A^t(\Phi_k) + B_A^t(\Phi_k)$

(2) $C_B + I_B^t(\Phi_j) + B_B^t(\Phi_j) \geq C_B + I_B^t(\Phi_k) + B_B^t(\Phi_k)$

(3) the priority level exchange of τ_A and τ_B between and Φ_k does not affect the feasibility of any other process.

Proof:

The proof is in three parts, reflecting the clauses of the Theorem.

Proof of Clause (1)

We note that $I_A^t(\Phi_j) \leq I_A^t(\Phi_k)$ since the interference due to τ_B is present under Φ_k but non-existent under Φ_j. The blocking factor is the sum of all blocks endured during a release, one per resource request. Two cases of blocking are identified on τ_A:

(a) *due to τ_A sharing a resource with a process assigned one of priority levels 1..i-1 (i.e. not with τ_B)*

Under Φ_j let the request by τ_A for the resource occur at t_r with it allocated at t_g. For τ_A the interval $[t_r, t_g)$ is composed of blocking and interference. Under Φ_k the request time $t_{r'}$ may be later than

t_r due to the execution of τ_B pushing the request of τ_A back. However, t_g will remain the same as τ_A and τ_B do not share a resource: i.e. $t_r \leq t_{r'} \leq t_g$. If $t_{r'} > t_r$ we note that under both Φ_j and Φ_k the interval $[t_r , t_{r'})$ is occupied by the execution of τ_B. Hence, although blocking may decrease by an amount $t_{r'} > t_r$ under Φ_k the interference will increase by $t_{r'} > t_r$. Additionally, $I_A^t(\Phi_k)$ may be greater than $I_A^t(\Phi_j)$ due to the normal execution of τ_B.

(b) *due to τ_A sharing a resource with τ_B*

Let the resource be shared between τ_A and τ_B (and perhaps lower priority processes). Now, τ_A may receive a block under Φ_k not endured under Φ_j when requesting the shared resource (although this additional block may be of length 0). Let the resource be shared between τ_A, τ_B and a higher priority process. We observe that case (a) applies, with the possibility that the time at which the resource is granted under Φ_k may be greater than t_g (i.e. when it was granted under Φ_j) due to τ_B requiring the resource. Let the later time be $t_{g'}$. The interval $[t_g , t_{g'})$ contains some execution of τ_B (which will be part of $I_A^t(\Phi_k)$) but also may contain lower priority process executions. Hence blocking for this resource will be at least that under Φ_j.

Therefore clause (1) holds.

Proof of Clause (2)

As proof of clause (1).

Proof of Clause (3)

This holds by observation since a process cannot be directly blocked by a lower priority process holding a resource under the reservation protocol.

Given Theorem 5.13 (and the observations of section 5.7.1) the reservation protocol with clairvoyant blocking maintains the optimality of the priority assignment method of section 5.2.

5.7.3 Clairvoyant Blocking and the Priority Ceiling Protocol

A family of resource allocation protocols are available based upon priority inheritance (see section 2.3.5). One of these is the Priority Ceiling Protocol (PCP) [Sha90]. The PCP assigns a static priority ceiling to each resource, equal to the priority of the highest priority process that uses it. At run-time, a process is only granted the lock on a resource if it has a higher priority than all the ceiling priorities of all currently locked resources, or, if no other process currently holds a resource. When a process is blocked on a resource, the process holding that resource inherits the highest priority of the processes blocked on that resource until it releases the resource. The blocking that a process receives is experienced solely at the first resource access within a release.

The PCP is unsuitable for use with the optimal priority assignment technique, even assuming clairvoyant blocking, as the feasibility of higher priority processes may be degraded by two lower priority processes exchanging priority levels. Consider two priority orderings Φ_j, $\Phi_k \in \Phi$ over Δ^* with $\Phi_j(\tau_A) = i$, $\Phi_j(\tau_B) = i+1$; $\Phi_k(\tau_A) = i+1$, $\Phi_k(\tau_B) = i$; and $\Phi_j(\tau_C) = \Phi_k(\tau_C) = l$ where $1 \leq l < i$. Examine the execution of the tasks under Φ_j given in Figure 5.8. At time t_0, τ_B is released, executing up to t_1 when it is pre-empted by τ_A. The latter process executes, locking a resource shared with τ_C at t_2. At t_3 τ_C becomes runnable, executing up to t_4 when it becomes blocked requesting the resource held by τ_A, which inherits the priority of τ_C and completes its critical region at t_5. Now, τ_C executes completing at t_6, with τ_A and τ_B completing at t_7 and t_8 respectively.

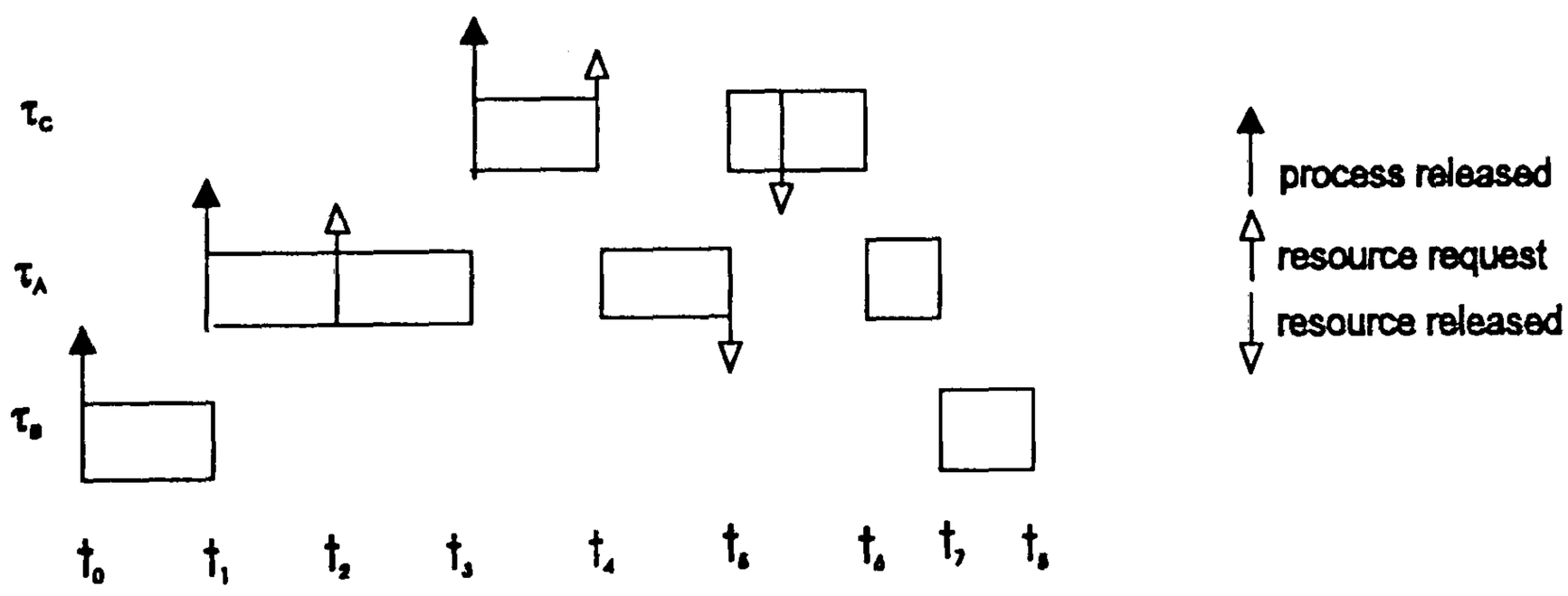


Figure 5.8: Execution Under Φ_j

Now consider the execution of the processes under Φ_k given in Figure 5.9. Firstly, τ_B executes to completion at t_1 with τ_A then executing and locking

the resource at t_2. Process τ_C pre-empts τ_A at t_3 becoming blocked at t_4 attempting to access the resource held by τ_A. The latter process now executes its critical region, completing at t_5 enabling τ_C to execute its critical region and complete at t_6, with τ_A completing at t_7.

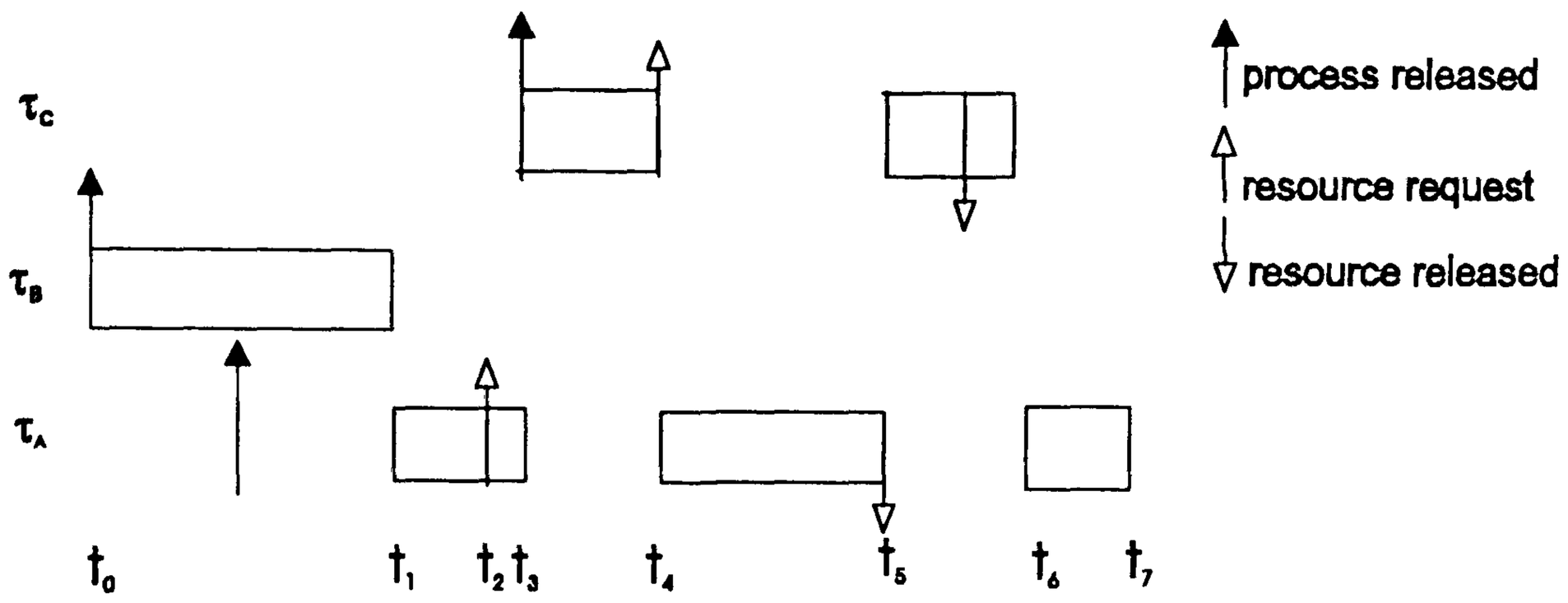


Figure 5.9: Execution Under Φ_k

Under Φ_k, process τ_A executes later than under Φ_j, so locking the resource later but still before the release of τ_C. Thus, more of τ_A's critical region is outstanding when τ_C becomes runnable. Therefore, the blocking that τ_C endures has increased, solely due to two lower priority processes swapping priority levels. This has degraded the feasibility of τ_C implying that using the PCP cannot always result in optimal priority orderings being found (even assuming clairvoyant blocking calculations).

We note that the PCP and the optimal priority assignment method is sufficient, in that when a process is proved feasible at priority level i, the feasibility of processes assigned priority levels i+1..n is unchanged. Hence, if the algorithm completes having assigned all priority levels, a feasible priority ordering will have been found.

The PCP and priority assignment method remains optimal in the constrained circumstances that all processes have equal WCBT. This could occur, for example, if the context switch time formed the longest block on each process. Now, no process endures increased blocking due to lower priority processes swapping priority levels.

5.7.4 Pessimistic Blocking

In practical systems, exact clairvoyant blocking factors are not available. Therefore, we now address the problems raised by the use of pessimistic blocking factors.

Consider the reservation protocol, seen to be applicable for use with the optimal priority assignment method with clairvoyant blocking. When calculating the WCBT offline for process τ_A, the exact ordering of higher priority processes is required, not merely the set of higher priority processes. Under the optimal priority assignment only the latter is available. Hence, for τ_A the worst possible priority assignment of the higher priority processes must be assumed for each resource access. The calculation of this has, in general, exponential complexity.

Assuming blocking factors are calculated in this way, the reservation protocol may still be used with the optimal priority assignment method. This combination is optimal: if a feasible priority ordering exists it will be found. The proof of this assertion lies in noting that Theorem 5.13 holds even assuming that clairvoyant blocking factors are not available. However, the effect of pessimistic blocking ensures that if no feasible priority assignment is found, the process set may still meet all deadlines: sufficient and not necessary feasibility is evident.

Consider the PCP. The blocking factor for τ_A assigned priority level i is equal to the longest critical region of any lower priority process which locks a resource with a process assigned one of priority levels 1..i [Sha90]. If a process swaps priority levels with another process with which it shares a resource, the sum of interference and blocking may actually decrease by lowering the priority level (i.e. i to $i+1$) if the increase in interference is not greater than the critical region length of the previously lower priority process. This can happen when the executions of the processes do not overlap.

For example, consider process τ_A assigned priority level i, and τ_B assigned priority level i+1, with their offsets and periods ensuring that their executions never overlap. The processes share a resource. The exact interference upon τ_A is I_A with worst-case blocking B_A (due to τ_B). Likewise, the interference upon τ_B is I_B (due to processes with priorities 1..i-1 since τ_A does not interfere with τ_B) with worst-case blocking 0. Let the two processes swap priority levels (i.e. τ_A is assigned priority level i+1 and τ_B is assigned i). The interference upon τ_A has not increased, but the blocking has fallen to 0.

Similarly, the interference on τ_B has not decreased, but its blocking has increased due to the critical region of τ_A. Thus, by lowering the priority of τ_A the likelihood that the process will be feasible has increased. Also, increasing the priority of τ_B has decreased the likelihood that the process will be feasible.

Hence, the PCP and the optimal priority assignment method will be sufficient and not necessary for the assignment of priorities: a feasible priority assignment may not be found even if one exists (assuming worst-case blocking times). Also, sufficient and not necessary feasibility is apparent due to the pessimistic blocking.

5.7.5 Discussion

A direct comparison between the two resource allocation protocols for use with optimal priority assignment is difficult. The nature of the reservation protocol dictates large blocking factors when compared with the PCP, although the combination provides optimal priority assignment. However, the loss in feasibility of the process set due to those blocking factors may override the lack of optimality available with the PCP.

It is noted that under certain circumstances the PCP with the optimal priority assignment method will indeed provide optimal priority assignments even with pessimistic blocking factors. For example, this occurs if all processes share a resource and are all subject to the same length maximum block (assuming the lowest priority process, for feasibility purposes, endures a block).

The main differences in the behaviours of the two protocols when combined with the optimal priority assignment method lies in the nature of their blocking characteristics. The reservation protocol does not permit a lower priority process to hold a resource at a point in time at which it is required by a higher priority process. Under the PCP this is not true. Therefore the precise relative ordering of lower priority processes affects the feasibility of higher priority processes under the PCP, but not the reservation protocol.

5.8 Increasing Feasibility

The relationship between the feasibility of process sets with a critical instant (i.e. all offsets are zero) and the same set without a critical instant (i.e. arbitrary offsets) is such that the latter is more likely to be feasible. This is observed by taking a process set with arbitrary offsets and no critical instant,

and setting all process offsets to be zero and employing a critical instant feasibility test. For example, consider the process set in Table 5.6.

Process	O	C	D	T
τ_A	0	5	5	10
τ_B	5	5	5	10

Table 5.8: Example Process Set 7.

Using a sufficient and necessary offset feasibility test, the processes are declared feasible. However, if offsets are assumed to be 0 (for feasibility purposes) the process set will be declared infeasible (by a sufficient and necessary critical instant feasibility test). The implication of this observation is that process sets that have a critical instant and are infeasible could be assigned offsets which ensure that a critical instant does not occur, leading to the possibility that the process set could be declared feasible by a sufficient and necessary offset test.

In broad terms, process offsets need to be defined such that a critical instant does not occur. That is the following condition holds (drawn from section 5.1):

$$\exists i, j: 1 \le i < j \le n \bullet \quad |O_i - O_j| \neq h \gcd(T_i, T_j) \tag{5.6}$$

where h is a non-negative integer. The number of different combinations of offset values which fulfil this condition is potentially large. For example, when choosing an offset for τ_i, values in [0, T_i) may be considered. In the extreme, this could lead to $T_1 * T_2 * \ldots * T_n$ different sets of values for process offsets. Consider τ_i and τ_j ($i \neq j$) where $g=\gcd(T_i, T_j)$. Only values of O_i and O_j need be considered such that $|O_i - O_j|$ is not a multiple of g and $O_i \in [0, D_i)$ and $O_j \in [0, D_j)$. Clearly this approach is impractical: even a small process set of cardinality 10, with all periods equal to 10, may require the consideration of $\approx 10^{10}$ different combinations of process offsets, each incurring the complexity of a subsequent optimal priority ordering method (including the sufficient and necessary feasibility test).

After all combinations of offsets have been considered, it is possible that the process set remains infeasible. For example, if all periods are mutually co-prime, no offset combination will result in a process set without a critical instant. Given that a number of combinations of offsets are available, each ensuring that no critical instant exists for the process set, it is an open

question as to whether (at least) one of these combinations is guaranteed to be feasible (assuming that the optimal priority assignment method is used to determine process priorities and feasibility).

A heuristic is now outlined that limits the number of offset combinations examined. Consider a critical instant process set that is infeasible and has utilisation no greater than 100%. Whilst attempting to determine feasibility, processes are considered in order $\tau_1..\tau_n$, i.e. high to low priority (see Chapter 4). Let the first process that fails the test be τ_i $(1 \leq i < n)$, where $\tau_1..\tau_{i-1}$ are feasible. Amongst the processes $\tau_1..\tau_{i-1}$ whose periods are not co-prime with respect to T_i, choose the process that has the highest interference on τ_i. Let this process be τ_j. Choose O_i such that the condition defined by equation (5.6) holds. Now, the feasibility of τ_i is re-considered. If τ_i remains infeasible, another offset may be chosen. We note a maximum of T_i-1 offsets for τ_i are tried. If an O_i is found such that τ_i is feasible, we proceed to τ_{i+1}

Since τ_{i+1} has a critical instant with $\tau_1..\tau_{i-1}$ then we test to see if it has a critical instant with τ_i. If it does, its feasibility is determined using the critical instant test, otherwise the sufficient and necessary offset test is employed. If τ_{i+1} is feasible, we proceed to τ_{i+2}. If τ_{i+1} is not feasible we select an offset for τ_{i+1} in the manner described for τ_i above.

The method examines a maximum of $\sum_{i=1}^{n} T_i$ offset combinations, less than the maximum possible number of combinations ($T_1 * T_2 * ... * T_n$).

Consider the following example.

Example 5.5:

Consider the process set in Table 5.9

Process	C	D	T	Priority
τ_A	1	4	5	1
τ_B	2	8	8	2
τ_C	2	6	12	3
τ_D	3	11	12	4

Table 5.9: Example Process Set 8.

When applying a sufficient and necessary critical instant feasibility test process τ_D is declared infeasible. Since only T_A (of all higher priority process periods) is co-prime with T_D, and the interference of τ_B

on τ_D is higher than that of τ_C, we choose an offset for τ_D such that it no-longer shares a common release time with τ_B. Given that gcd(τ_B, τ_D) = 4, then possible values for O_D are 1,2,3,5,6,7,9,10 or 11. Initially, let O_D = 1. By employing the sufficient and necessary offset test, it is found that τ_D is now feasible.

With minimal modification the approach can also be used for process sets that have no critical instant originally, and whose priorities are assigned using the optimal priority assignment approach given in this chapter. Now, the following approach is used if a process cannot be found that is feasible at a given priority level. Let such a priority level be i. One of the i processes that fail at this level is chosen, and its offset adjusted as indicated above. If no offset exists for the process, we choose another (of the i processes). If a process (with modified offset) can be found that is feasible at the priority level, we proceed to level i-1. This extension of the method examines a maximum of $n\sum_{i=1}^{n} T_i$ offset combinations.

Processes that have precedence constraints between them must maintain the same offset relationships. Thus, if O_i is changed where $\tau_i \rightarrow \tau_j$, then O_j is changed by the same amount (and so on for other processes in the AAG).

The method also applies if processes block since this only affects the determination of feasibility. More interestingly, even if a process set is feasible, we may use the above approach to reduce the amount of blocking that a process encounters. That is, if τ_i suffers a worst-case block of B_i, due to a critical region of τ_j ($i < j$) then τ_i (or τ_j) may be assigned an offset such that B_i is reduced (using the accurate blocking analysis in Chapter 4). Care must be taken, since by assigning such an offset, another process of priority greater than τ_j may now have increased blocking leading to it becoming infeasible.

5.9 Summary

This chapter has extended offline flexibility by considering and addressing several outstanding issues in static priority feasibility theory. Essentially, feasibility test coverage is increased by permitting processes to have arbitrary start times. This creates a number of problems, mainly that of priority assignment, since deadline-monotonic priority assignment (assumed in Chapter 4) is no longer optimal. The main results of the chapter are an

efficient method to assign optimal priorities to processes with arbitrary start times, and the provision of feasibility tests for such processes.

Leung *et al* observed that whilst the determination of the feasibility of a non-critical instant process set is NP-hard [Leung80], it was an open question as to whether this complexity was due to both priority assignment and feasibility testing, or merely feasibility testing [Leung82]. It has been shown in this chapter that optimal priority assignment can be achieved in polynomial time, implying that the NP-hard complexity is due to the determination of the feasibility of a given priority assignment over a process set.

The optimal priority assignment method given in the chapter relies upon the availability of a sufficient and necessary feasibility test. This was defined, together with the interval over which feasibility needs to be determined. Additionally, the sufficient and not necessary tests of Chapter 4 were extended for use with non-critical instant process sets.

The priority assignment approach was then extended to cater for processes with arbitrary precedence relations, remaining optimal when process priorities descend across the precedence relationship graph. Also, resource allocation protocols that permit processes to share resources were considered, in particular the reservation and priority ceiling protocols. Problems were encountered with the inclusion of resources since worst-case blocking times are inherently pessimistic. This has the consequence of sufficient and not necessary feasibility testing. However, the reservation protocol was seen to remain optimal with respect to priority assignment, with the priority ceiling protocol sufficient and not necessary.

Finally, it was shown how some infeasible critical instant process sets may be converted into feasible process sets by assigning offsets to processes, so that the process set no longer has a critical instant.

It is noted that further consideration of the behaviour of the optimal priority assignment method with different precedence constraint strategies and resource allocation policies is required.

Chapter 6.
Spare Capacity And Its Detection

Chapters 4 and 5 have examined methods of increasing the offline flexibility of hard real-time applications by improving the coverage, accuracy and efficiency of feasibility analysis. Such analysis provides the 100% deadline predictability required for crucial processes at run-time. The second level of potential additional flexibility identified in Chapter 3 occurs at run-time due to the detection and subsequent re-use of spare capacity by crucial (and other application) processes to improve the overall utility of the system. The focus of this chapter is the detection of spare capacity.

Rudimentary approaches for gain time detection have been proposed by Haban [Haban89, Haban90] and Dix *et al* [Dix89] based upon software triggers placed into application code (see Chapter 2). This enables gain time to be detected after it has been generated, although before process completion. The Extended Priority Exchange approach identifies gain time on process completion by comparing actual and worst-case execution time [Sprunt88].

In this chapter, a more powerful approach towards the detection of all spare capacity is developed. It is language based, using software triggers to declare when spare capacity is about to be, or has been, generated. The triggers are placed at strategic points within the code, decided by offline analysis of the control flow properties of the process code. The approach enables the detection of spare capacity as early as possible using offline control flow analysis of code only.

Given the need to provide 100% deadline predictability for crucial processes, it is inevitable that the processor and other resources, at run-time, will be under-utilised. This occurs for many reasons [Audsley93a], including the following:

(i) *worst-case execution time (WCET), resource usage and resource blocking time analyses are inherently pessimistic* [Puschner89]: at run-time, software components do not always require their worst-case execution time (due to the control-flow nature of application code); also, they may not require their worst-case requirement of resources.

(ii) *hardware behaving better than expected at run-time*:

for example, pipelines and caches speed-up the actual execution of processes, whilst their effects are not easily calculable offline, thus adding to the pessimism of worst-case execution time analysis.

(iii) *sporadics may not execute at their maximum frequency*:
the maximum frequency is assumed for feasibility analysis, although sporadic processes may not actually execute at such a frequency.

(iv) *non-execution of error handling software*:
For example, recovery blocks, exception handlers that are not required at run-time.

(v) *spare time incorporated by feasibility analysis to guarantee crucial process deadlines*:
Feasibility analysis often dictates that not all 100% of processor utilisation can be used for guaranteeing crucial processes. For example, in rate-monotonic scheduling [Layland73], process sets are only declared feasible if system utilisation is not greater than 69% (for large numbers of processes), effectively providing 31% of total processor utilisation as spare capacity. Earliest deadline scheduling [Layland73], in theory, allows utilisations to reach 100%. However, when resources are considered, this utilisation cannot be achieved, except in very contrived circumstances. Even scheduling approaches where static schedules are created offline (i.e. MARS, Spring) rarely produce schedules with 100% utilisation when resources are considered.

(vi) *the system may have an inherent utilisation of less than 100%*:
Even if feasibility permitted 100% utilisation, the process set itself may not require utilisation of that level.

When considering the types of spare capacity outlined above, three general forms may be identified:

Definition 6.1:

Slack Time - processor time that is not guaranteed to a process offline.

Definition 6.2:

Gain Time - processor time, guaranteed to a crucial process offline, but not required by that process at run-time.

Definition 6.3:

Spare Resources - those resources assumed to be required by a crucial process (by worst case resource usage analysis) but not actually required at run-time (including both logical and physical resources apart from the processor).

Gain time occurs due to processes executing for less than their worst-case execution time; hardware speed-ups; non-execution of guaranteed error-handling software. Slack time occurs due to the inherent utilisation of the system being less than 100%; in-built slack due to the feasibility test employed; and the execution of sporadic processes at less than their worst-case frequency.

One important difference between gain time and slack time lies in their potential assignment to processes. Since gain time has been guaranteed offline, as part of a crucial process's worst-case execution time, when re-assigning that gain time to another process, the latter inherits the guarantee afforded to the original process. The same guarantee cannot, in general, be afforded to processes inheriting slack time. Slack time occurs when no processes are runnable, i.e. after the completion of the currently running process, if that is the sole runnable process. Intuitively, slack time could be guaranteed, up to the next release of a crucial process, if that release time is known. When sporadic processes are present, the determination of this time is not possible (without aid of clairvoyance). However, a lower bound maybe placed upon it as the kernel must track the release times of sporadics to ensure that they do not occur at intervals less than their respective minimum inter-arrival times. Therefore, slack time can be generated up to the earliest possible release of a sporadic, but not after than time.

The following section introduces some assumptions regarding the language model used in the remainder of the chapter. Sections 6.2 and 6.3 describe informal and formal models of gain points, and a technique for detecting gain time. Section 6.4 introduces a possible implementation and discusses the trade-off between accuracy of gain time detection and overheads incurred. Extensions to the gain point technique are given in section 6.5, with a summary of the chapter given in 6.6.

6.1 Language Assumptions

Since the approach outlined in this chapter for the detection of spare capacity is at a language level, assumptions regarding the language are now considered.

Languages appropriate for hard real-time systems must be boundable and deterministic. This enables worst-case execution bounds to be placed upon code. Specifically, the language must not permit [Burns89]:

(i) unbounded loop iterations - a pre-declared constant is associated with each loop defining the maximum number of loop iterations;

(ii) unbounded recursion - both direct and indirect;

(iii) procedure and function parameters;

(iv) goto style operations;

(v) non-deterministic constructs (e.g. the guarded commands of Ada [Ada83]).

These limitations are seen in a number of experimental languages specifically developed for boundability, including the Ada83 derivative Spark Ada [Carre89]. When such languages are targeted at distributed applications the passing of pointers as parameters is often forbidden [Hutcheon87]: de-referencing may occur in the wrong physical address space.

A simple control-flow language is assumed with the following basic control-flow constructs:

- **IF** condition **THEN** ..code.. **ELSE** ..code.. **FI**
- **WHILE** condition **MAX** const_positive_int
  DO ..code.. **ENDDO**
- **FOR** loop_variable_expr **TO**
  [loop_limit_variable|constant]
  STEP const_positive_integer
  MAX const_positive_int
  DO ..code.. **ENDFOR**
- **CALL** procedure_name (parameters)

It is assumed that in a **FOR** loop, the loop variable increases by the step value in every iteration. If initially the loop limit variable (or constant) is less than the loop variable, the loop body will not be executed.

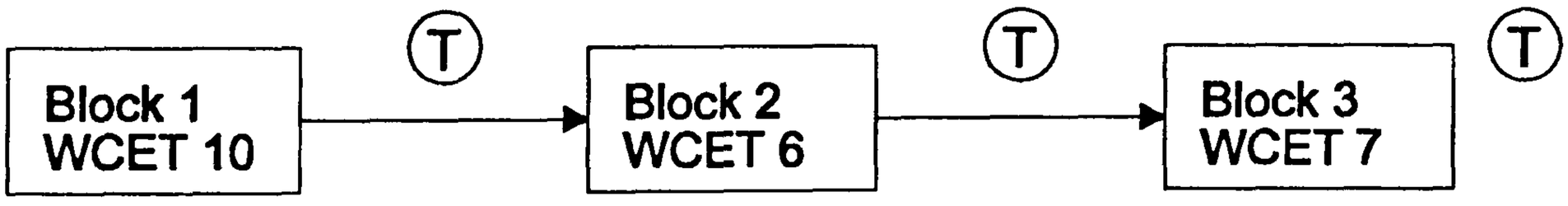
We assume conventional expression constructs (such as those defined for the Ada programming language [Ada83]). Note that a **CASE** statement is semantically equivalent to successive **IF** statements (although some run-time efficiency may be lost). Locking and unlocking resources is achieved via **GET** and **RELEASE** statements respectively. These will map onto kernel mechanisms

responsible for allocating resources (e.g. semaphores). A `DO..WHILE` construct is omitted as the same semantics can be achieved with a `WHILE..DO` loop, although some run-time efficiency may be lost. Within the context of this thesis, language support for fault-tolerance is not considered.

The language assumptions outlined above bound process execution times. This permits WCET analysis of processes. Conventionally, this is achieved by breaking code into *basic blocks*, each having a single entry point and a single exit point [Puschner89]. The worst-case path through the basic blocks, in terms of execution time, defines the WCET. Within this chapter basic blocks are further constrained to contain either a single control-flow statement, e.g. `IF`, `WHILE`, `CALL` etc, or many non-control-flow statements. Thus, the conditional statement in a `WHILE` statement is contained in a different basic block from the loop body. Initially, we preclude the use of semantic aids, such as markers to quantify exactly the number of loop iterations [Puschner89]; or source code annotations to define possible control-flow paths through the code at run-time [Park93]. These are discussed in section 6.5.

6.2 Gain Points: An Approach For The Detection Of Spare Capacity

Several approaches have been proposed to enable detection of gain time (see Chapter 2). Haban places software triggers into the code to enable actual execution time to be calculated [Haban90]. This is then compared with the worst-case execution times to identify gain time. Consider the application process code:



The process code has been broken into three basic blocks, with single control flow paths between them: block 1 is executed initially, then block 2 and finally block 3. Software triggers are inserted ("T" in a circle). At run-time, the triggers measure actual execution time for the three blocks as 6, 5 and 5 respectively. This is illustrated in Figure 6.1. The graph in Figure 6.1 plots the total accumulated gain time against the execution time of the process.

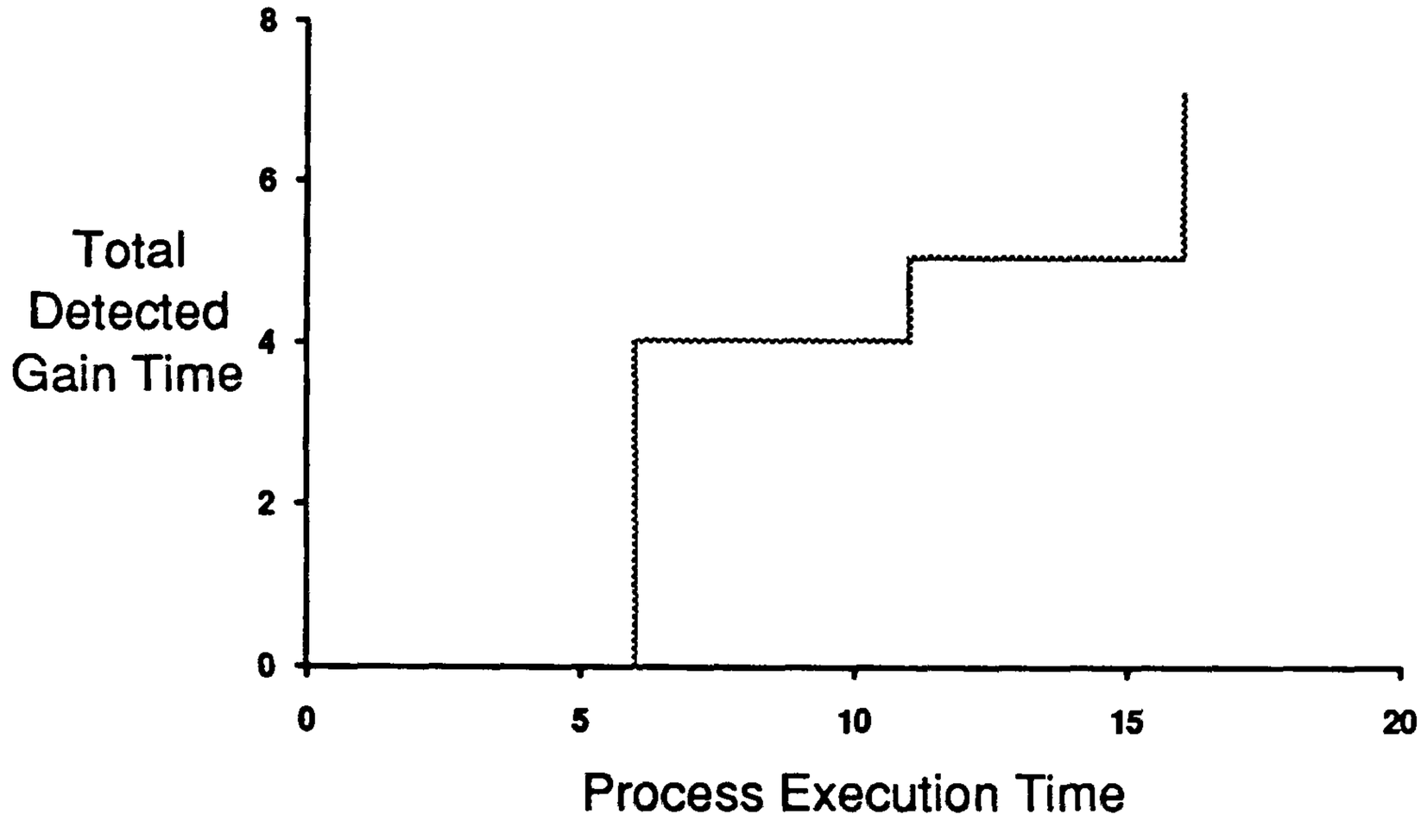
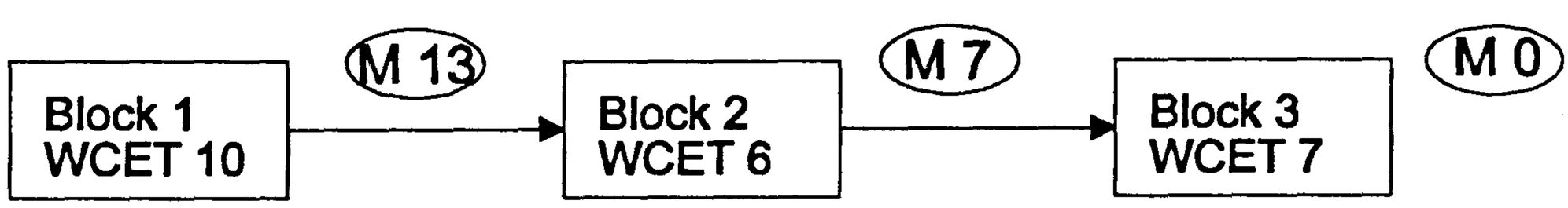


Figure 6.1: Gain Time Detection for Haban's Approach and Milestones

Dix *et al* insert software triggers termed "milestones" [Dix89]. The value of the milestone is the maximum remaining execution time of the process: effectively they declare that a process will finish at or before that time value. Consider the application process code:



Milestones ("M" in a circle with number denoting milestone value) are inserted within the code. Each milestone permits the identification of gain time from the previous block (hence the milestone of value 0 after the final block). Consider the same run-time behaviour as given above for Haban's model: blocks 1, 2 and 3 actually require 6, 5 and 5 time units respectively. At time 6, the first milestone is reached, declaring that a worst-case of 13 time units are required for the remainder of the process. Using the following formula, the gain time may be determined:

Total WCET of process - milestone value - actual execution time of block

Thus, after the first block a gain time of 23 - 13 - 6 = 4 is detected. The total gain time determined, together with the time at which it is detected is equivalent to that of Haban's approach (i.e. the graph in Figure 6.1).

The central observation is that both the above approaches determine gain time after it has been generated. In general, we wish to be aware of spare capacity as soon as possible: the sooner it can be determined, the sooner it can be usefully utilised (the assignment of detected spare capacity to processes is considered in Chapter 7).

It is noted that neither Haban's or Dix's approach cater for spare resources; non-execution of error handling code; sporadics not released at a maximum frequency; inherent under-utilisation of the processor; spare time required by feasibility analysis.

An approach for the earlier detection of spare capacity is now described. Consider the following process code fragment:

```
IF condition == TRUE
THEN  ... code ...
ELSE  ... code ...
FI
```

This can be analysed to determine the WCET of both the **THEN** and the **ELSE** clauses (the WCET of the fragment being the maximum of the two values):

```
IF condition == TRUE
THEN    [10 units]
ELSE    [6 units]
FI
```

Depending upon the control flow through the fragment, gain time will become evident. For example, if the **ELSE** clause is executed, 4 units of gain time will be realised - the difference between the WCET's of the two clauses. We note that both Haban's and Dix's approach could detect this gain time after the **FI** statement, not before. These approaches also detect gains due to hardware speed-ups.

The *gain point* is introduced to enable detection of gain time in the manner described above. The gain point is a software trigger, named to reflect that it is inserted at a point in the application code where gain time can be detected. The gain point can be considered to be a call into the kernel interface (we return to implementation issues in section 6.4). The value of the gain point reflects the amount of gain time that can be detected due to the control flow of the code. Consider the following code fragment:

```
IF condition1 == TRUE
THEN    [6 units]
ELSE    [10 units]
FI
IF condition2 == TRUE
THEN    [6 units]
ELSE    [5 units]
FI
IF condition3 == TRUE
THEN    [5 units]
ELSE    [7 units]
FI
```

This corresponds to the code fragments used to illustrate Haban's and Dix's approaches earlier. Gain points are inserted into the code:

```
IF condition1 == TRUE          /* Block 1  */
THEN    GAIN_POINT(4)
        [6 units]
ELSE    [10 units]
FI
IF condition2 == TRUE          /* Block 2  */
THEN    [6 units]
ELSE    GAIN_POINT(1)
        [5 units]
FI
IF condition3 == TRUE          /* Block 3  */
THEN    GAIN_POINT(2)
        [5 units]
ELSE    [7 units]
FI
```

At run-time, the **THEN** clause is executed in `Block 1`; the **ELSE** clause in `Block 2`; the **THEN** clause in `Block 3`. The calculation of gain time is illustrated in Figure 6.2. When comparing the detection of gain time by gain points in the figure, with Haban's triggers and milestones in Figure 6.1, two observations may be made:

(i) the total gain time detected is equal;

(ii) the time at which the gain time is detected is earlier using gain points.

Indeed, using pure static code analysis only (without clairvoyance or use of semantic analysis), gain points enable gain time to be identified as early as possible).

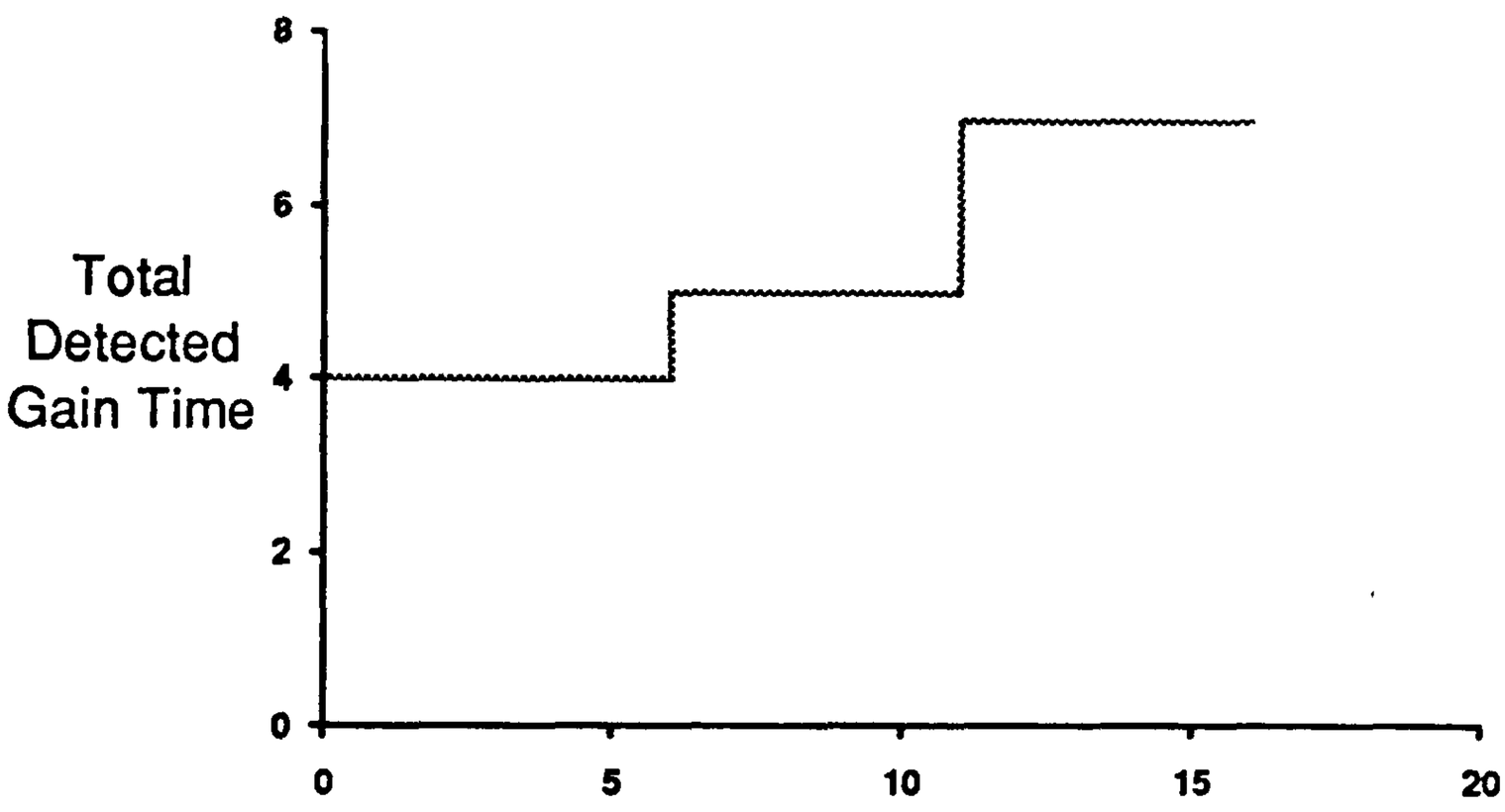


Figure 6.2: Gain Time Detected Using Gain Points

As well as gain time, all other forms of spare capacity can be detected using the gain point mechanism. To achieve this, four separate forms of gain point are required:

(i) *static* for static code;

(ii) *dynamic* for loop constructs;

(iii) *efficiency* for detecting hardware speed-ups;

(iv) *resource usage* for identifying spare resources.

These are discussed in the following sections, together with the detection of slack time.

6.2.1 Static Gain Points

Static gain points detect gain time due to branches in the control flow (as shown in the previous section). In general, whenever a branch occurs in process code, a gain point is placed as the first statement in each branch. The value of the gain point is the difference between the WCET of that branch and the maximum WCET of all branches, to the next point in the code where all the branches converge. For efficiency, if a gain point's value is 0, it can be removed (this will occur for one of the clauses of an `IF` statement).

6.2.2 Dynamic Gain Points

Gain points can be used to note the amount of time gained by not using the maximum number of iterations in loops. This can occur in two ways:

(i) *conditional loops*: the loop body is continually executed whilst a conditional expression evaluates to true;

(ii) *set loops*: the number of executions of the loop body is determined by the value of a variable set before loop entry.

The first, for example, corresponds to `WHILE` statements; the second to `FOR` constructs. In both cases, the maximum number of iterations is specified by a constant declared with the loop statement, to constrain the code to have bounded execution time.

In conditional loops, the gain point is placed on exit of the loop, with the value calculated dynamically. For example, if the maximum and actual number of iterations are given by `max` and `loopvar` respectively, with the WCET of the loop body being C, then the gain point value is given by:

```
(max - loopvar) * C
```

For example, consider the following statement:

```
WHILE condition == TRUE MAX 20

DO  [10 units]

ENDWHILE
```

```
GAIN_POINT ((20 - loopvar) * 10)
```

The loop counter `loopvar` is assumed to be compiler generated to ensure that the variable name is not known to the application code so that it cannot be altered within the loop body. It is incremented immediately before the end of

the loop body, with the compiler extending the application declared WHILE condition to test to see if `loopvar > max`.

In set loops, the maximum number of iterations is declared as a static constant (as in conditional loops). However, the number of iterations is also limited by the value of a variable set before the loop is executed. Consider the following statement:

```
FOR i = 1 TO j STEP 1 MAX 20
DO  [10 units]
ENDFOR
```

The number of iterations of the loop is equal to the minimum of `max` (where `max = 20` in this example) and `j`. Hence, a gain point maybe placed immediately preceding the loop statement with value equal to

```
min(j, 20) * C
```

where `C` is the WCET of the loop body. This assumes that the value of `j` is not varied within the loop. Although this would only matter if `j` were increased, the value of `max` is set to the initial value of `j` for the duration of the loop.

Additionally, a gain point may be set after the loop statement to detect gain time due to less than `min(j, max)` loop iterations. This caters for the possibility of the code within the loop exiting the loop before `j` iterations have occurred. Thus, the annotated code is:

```
GAIN_POINT (min(j, 20) * 10)
FOR i = 1 TO j STEP 1 MAX 20
DO  [10 units]
ENDFOR
GAIN_POINT ((min(j, 20) - loopvar) * 10)
```

6.2.3 Efficiency Gain Points

Static and dynamic gain points do not enable gains due to hardware speed-ups to be detected. Consider a basic block whose worst-case execution time has been found by WCET analysis. Since a basic block contains no control-flow statements, theoretically, the worst-case and actual-case execution times of the block should be equal. However, due to inaccuracies in WCET analysis (e.g. hardware instructions which have an actual execution time less than expected,

for example rotate etc.) and hardware speed-ups, the actual-case is less than the worst-case by g time units. This gain may be detected by the placement of an efficiency gain point after the basic block. The value of the gain point is equal to the worst-case execution of the block minus its actual execution time (assumed to be determined by the kernel in a similar manner to Haban's and Dix's approaches).

Since hardware speed-ups can also occur for execution of any code (for example WHILE and IF statements and there inherent conditions), efficiency gain points can be placed after such statements as well.

6.2.4 Resource Usage Gain Points

Used and unrequired resources can also be detected using gain-points. The former are those resources that have been used by a process during its execution; the latter consists of those resources a process requires at the start of its execution but, due to the control-flow path taken through the code at run-time, does not use.

Initially, we examine unrequired resources. Consider the following statement:

```
IF condition == TRUE
THEN    GET (resource1)
        [6 units]
        RELEASE (resource1)
ELSE    GET (resource2)
        [5 units]
        RELEASE (resource2)
FI
```

The statement above requires exactly one of two resources, either the resource controlled by `resource1` or that controlled by `resource2`. That is, at the start of the statement, the resources required during execution are `resource1` and `resource2`. After the branch, one resource is no longer required: at the start of the **THEN** branch a gain point `resource2` is inserted; a gain point `resource1` is placed at the start of the **ELSE** branch. This is illustrated below:

```
IF condition == TRUE
THEN    GAIN_POINT(resource2)
        GET (resource1)
        [6 units]
        RELEASE (resource1)
ELSE    GAIN_POINT(resource1)
        GET (resource2)
        [5 units]
        RELEASE (resource2)
FI
```

The gain point statements inform the scheduler that the resource is no longer required by the particular execution of the process.

Since a process may use a resource at more than one point in its code, the gain point may only be inserted if the process will definitely not require that resource at any future point in its current execution. In general, at the start of each branch, a gain point may be inserted if a resource is not required by the branch, but is required by another of the branches before the end of the process's code. When a resource is accessed inside a loop statement, the gain point must be placed outside the loop.

6.2.5 Resource Blocking

When detecting the feasibility of a set of processes, the maximum blocking time for a process set must be considered. This depends largely upon the run-time resource allocation policy employed. For example, non-blocking policies imply no blocking at run-time; blocking policies limit potential blocking to a number of critical region accesses by lower priority processes. When determining the feasibility of a process, both the WCET and the WCBT must be guaranteed before the deadline of the process. It is apparent that since the worst-case estimation of blocking time may be pessimistic, some blocking time may be unused at run-time. For example, consider the use of one of the family of Priority Ceiling Protocols for resource allocation in the context of static priority scheduling. Blocking time guaranteed for process τ_i may be used for one or more of the following reasons:

(i) the lower priority blocking process does not execute for the WCET of its critical region;

(ii) the lower priority blocking process executes for some of its critical region before the release of τ_i;

(iii) the lower priority blocking process does not have the longest critical region amongst all processes that may block τ_i.

Intuitively, any unused blocking time may be detected as gain time.

Consider the Ceiling Semaphore Protocol. Blocking is constrained to occur before a process begins its execution (i.e. between the release of a process and the time at which it first executes) [Rajkumar89]. Thus, a single gain point could be placed at the start of the process's code. Consider two processes, τ_1 and τ_2, that share a resource (τ_1 has the higher priority):

$$C_1=5 \quad B_1=2 \quad T_1=D_1=7 \quad O_1=0$$
$$C_2=2 \quad B_2=0 \quad T_2=D_2=7 \quad O_2=0$$

The processes are feasible using conventional rate-monotonic analysis [Lehozky89]. Initially, both processes are released at time 0, with τ_1 executing without experiencing any blocking. The above discussion would indicate that a gain point could detect 2 units of unused resource blocking time. However, it is clear that unless this time is used for the normal execution of τ_2, this process will miss its deadline.

The reason that unused blocking time cannot be detected as gain time is that the blocking time guaranteed to τ_1 during offline analysis is actually for the execution of τ_2 at the priority of τ_1. Thus in general, for process τ_i, B_i "maps" at run-time onto the execution of the critical region of a lower priority process, not onto the execution of τ_i.

Gain time will become apparent if the lower priority process does not execute for the worst-case execution time of its critical region. However, this will be detected in the context of the lower priority process.

Unused blocking time does have one useful property. Since blocking time is to permit lower priority processes to execute temporarily at a higher priority level, unused blocking time may be used to execute lower priority processes at a higher priority level. In the example above, τ_2 could execute at τ_1's priority for the duration of any unused blocking time. We note that this relies upon the property of the Ceiling Semaphore Protocol that blocking occurs before a process actually executes, so permitting unused blocking time to be used after

the process has commenced execution. This property of unused blocking time is explored further in Chapter 7.

6.2.6 Detecting Slack Time

Slack time becomes evident when no process with guaranteed execution time is runnable, that is when no crucial process has outstanding computational requirement. Slack time could be detected by use of a dynamic gain point at the completion of a process. The value of the gain point would be the amount of time remaining before the next release of a crucial process. Given the presence of sporadic crucial processes, this estimation of time can only take into account periodic crucial processes: when a sporadic crucial process is released, the amount of slack time remaining immediately becomes zero. However, this means that gain points are used for the detection of both guaranteed and unguaranteed execution time. Also, slack time is related more to the complete system rather than individual processes. Therefore, the detection of slack time is the responsibility of the kernel.

6.2.7 Summary

In this section a method for detecting spare capacity has been outlined. Gain points are inserted into a process's code at the earliest point at which gain time can be detected. In general, gain points identify gain time at an earlier stage than either Haban's or Dix's approach. Static gain points are inserted to detect the gain time generated by taking a branch in the code that requires less processor time than alternative branches. Dynamic gain points are inserted to detect unused computation time due to loop constructs not executing for their worst-case number of iterations. Efficiency gain points are inserted to detect code that executes faster than anticipated due to hardware speed-ups. Finally, resource gain points are inserted to detect resources that are not required by a process.

It is noted that the gain point approach is applicable for both periodic and sporadic processes, processes that block on shared resources, and those processes related by precedence constraints.

6.3 Formal Model Of Spare Capacity

An important property of the gain time model is that all spare capacity is identified accurately. That is, for any path that a process may take through its

code, the sum of the actual execution time and the detected gain time should equal the worst-case execution time. A formal framework for the placement of gain points and their associated values is now developed which enables this property to be shown for the gain point model.

In the following section, the relationship between slack time and gain time is articulated, showing that they account for all processor utilisation not occupied by the actual execution of guaranteed crucial processes. Subsequently, a formal model of the control-flow language described in section 6.1 is given, together with a method for the placement of gain points.

6.3.1 Relationship Between Slack and Gain Time

The theoretical worst-case utilisation of the system is given by:

$$W_u = \sum_{\tau_i \in \Delta} \frac{C_i}{T_i}$$

This represents the utilisation of the system if all processes in Δ require their WCET for each execution, and experience their worst case blocking time (noting that blocking time for one process is actually the execution time of another, hence B_i may be ignored). The inherent slack time due to the system having less than 100% utilisation is given by:

$$S_{su} = 1 - W_u \tag{6.1}$$

This slack time will always exist at run-time.

Equation (6.1) assumes that sporadic tasks execute at their worst case frequency. At run-time, additional slack time is generated by sporadic tasks executing at intervals greater than their minimum inter-arrival time. This is given by:

$$S_{ss} = \sum_{\tau_i \in \Delta} \frac{C_i}{T_i} - \sum_{\tau_i \in \Delta} \frac{C_i}{T_i^m} \tag{6.2}$$

where T_i^m is the mean inter-arrival time of τ_i. We observe that $T_i^m = T_i$ for all periodic processes. Therefore the total slack time at runtime is:

$$S = S_{ss} + S_{su} \tag{6.3}$$

At run-time, gain time is generated by processes executing for less than their WCET. Let the mean execution time and mean blocking time of τ_i be denoted C_i^m. The gain time available at run-time is given by:

$$G = \sum_{\tau_i \in \Delta} \frac{C_i}{T_i^m} - \sum_{\tau_i \in \Delta} \frac{C_i^m}{T_i^m} \tag{6.4}$$

The total spare capacity (slack time and gain time) in the system at runtime is given by:

$$T_{sc} = S + G$$

Expanding in terms of equations (6.1), (6.2), (6.3) and (6.4):

$$T_{sc} = S_{su} + S_{sa} + G$$

$$= 1 - \sum_{\tau_i \in \Delta} \frac{C_i}{T_i} + \sum_{\tau_i \in \Delta} \frac{C_i}{T_i} - \sum_{\tau_i \in \Delta} \frac{C_i}{T_i^m} + \sum_{\tau_i \in \Delta} \frac{C_i}{T_i^m} - \sum_{\tau_i \in \Delta} \frac{C_i^m}{T_i^m} = 1 - \sum_{\tau_i \in \Delta} \frac{C_i^m}{T_i^m}$$

Hence, the model of spare processor capacity accounts for all processor execution at run-time.

The model above quantifies the amount of spare capacity in the system. However in practice, the amount available for re-assignment to processes is less, due to the cost of detecting the spare capacity. Assume that the cost of a gain point is constant c, with the mean number of gain points executed by a process being t. Therefore, the amount of time available for re-use is:

$$1 - \sum_{\tau_i \in \Delta} \frac{C_i^m + tc}{T_i^m}$$

The cost of spare capacity detection is considered further in section 6.4.

6.3.2 Code Representation

To enable detailed discussion regarding the placement and value of gain points, a formal model of the language described in section 6.1 is now given. Let the code for the processes in a system be written in the language. Syntax analysis can provide a control-flow graph (CFG) for each process's code [Aho89]. This is also required for worst-case execution time analysis [Puschner89]. The nodes in the graph are statements in the code, connected by directed arcs showing the possible routes through the code. Consider the following code fragment:

```
IF a > c
THEN    d := a; c := b
ELSE    GET (device)
        CALL device_driver ()
        RELEASE (device)
FI
WHILE d > 0 MAX 5
DO  e := e * 2
```

```
ENDWHILE

FOR c = 0 TO j STEP 1 MAX 10

DO  k := k * 2

ENDFOR
```

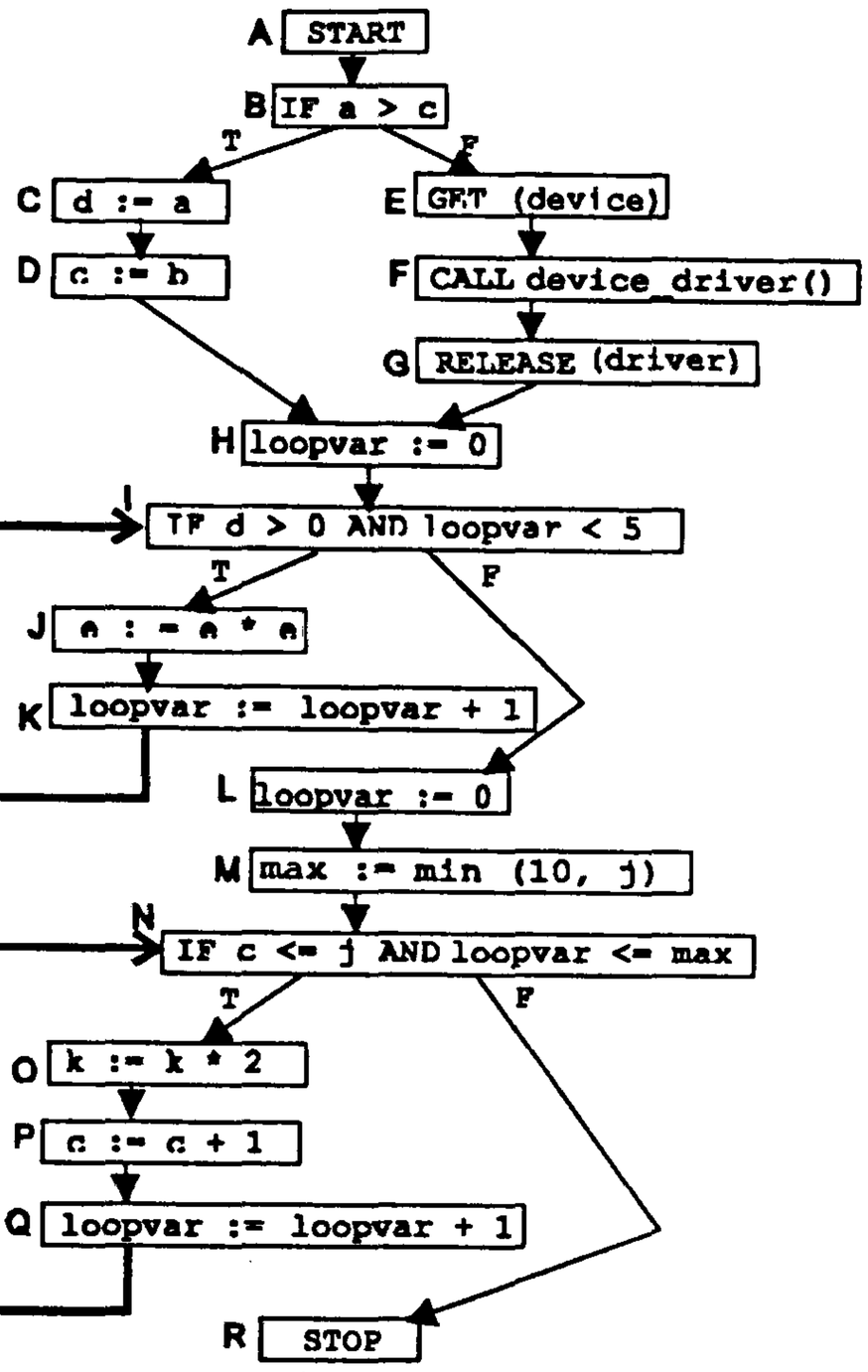


Figure 6.3: Example CFG.

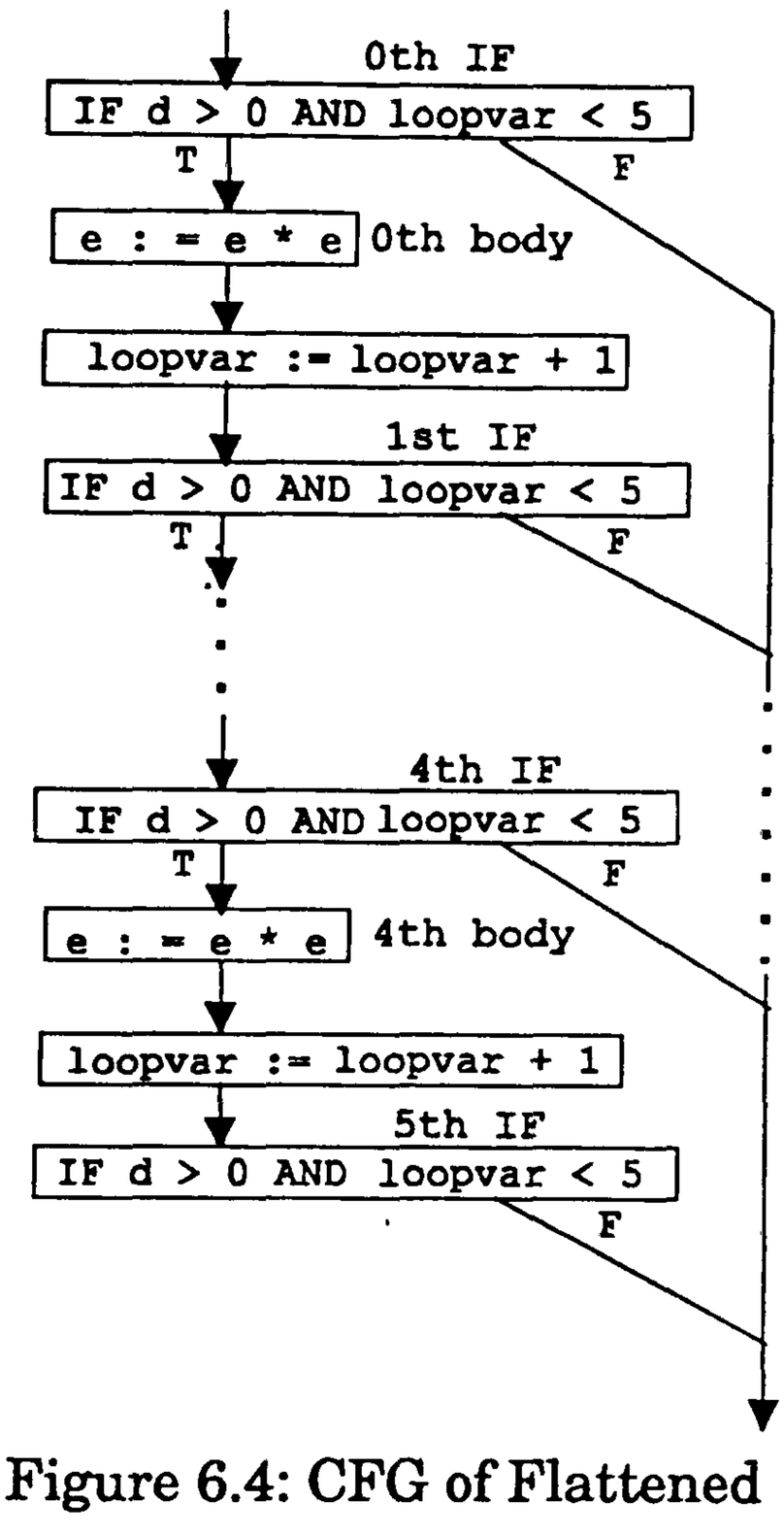


Figure 6.4: CFG of Flattened Loop Construct.

The CFG of this code produced by syntax analysis is shown in Figure 6.3. In the CFG, the condition expressions in **WHILE** and **FOR** statements are expressed as **IF** statements, with the provision of a variable to count the actual run-time loop iterations (`loopvar`) and ensure the maximum number of iterations is not exceeded. After executions of the loop body, the return to the **IF** expression at the head of the loop is achieved by an arc back up the graph. Such an arc is termed *retreating* (e.g. from the node labelled K to the node labelled I). This represents a shorthand form, since all loops could be flattened. For example, the **WHILE** loop in the above could be flattened to the CFG in Figure 6.4 with no retreating arcs. We note that if the maximum number of loop iterations (set by **MAX**) is x the body is executed a maximum number of x times, the **IF** condition a maximum of x+1 times. This observation also applies

to CALL statements. Two additional nodes are introduced into the CFG, namely START and STOP. There are placed at the top and bottom of the CFG respectively, and contain no code.

Formally, CFG = (N_C, E_C), where N_C is the set of unique node labels and E_C the set of directed edges. We denote an edge between nodes A and B by (A, B). This implies that the statement represented by node A is immediately preceding that represented by node B in the CFG. The set E_C is the union of the sets of retreating edges (E_C^-) and the forward edges (E_C^+):

$$E_C = E_C^+ \cup E_C^-$$

where

$$E_C^+ \cap E_C^- = \emptyset$$

In general, the CFG is acyclic (since loops and calls can be flattened). This is a direct consequence of the restrictions placed upon the language.

Consider the following definitions:

Definition 6.4:

For the edge (A, B) we define node B to be a successor of node A. The set of successor nodes of A is given by $\Lambda^+(A)$.

Definition 6.5:

For the edge (A, B) we define node A to be a predecessor of node B. The set of predecessor nodes of B is given by $\Lambda^-(B)$.

We note that *START* is the only node with no predecessor; *STOP* the only node with no successor.

Definition 6.6:

The function $\phi: N_C \rightarrow$ {START, STOP, SIMPLE, CALL, IF, FORIF, WHILEIF, GET, RELEASE} defines, for each node, a type.

Nodes containing an IF statement are assigned type IF, except if derived from a FOR or WHILE construct, where the type of the node is given by FORIF or WHILEIF respectively. Nodes containing non-control flow statements are assigned type SIMPLE. For example, for the CFG in Figure 6.3:

$$\phi(A)=\text{START};\ \phi(B)=\text{IF};\ \phi(F)=\text{CALL}$$

Definition 6.7:

A path from node A to node B, denoted $[A, A_1, \ldots, A_i, A_{i+1}, \ldots, A_n, B]$, exists if and only if edges $(A, A_1), \ldots, (A_i, A_{i+1}), \ldots, (A_n, B)$ exist in E_C^+.

We note that a path contains one or more forward edges. The set of all paths in a CFG is denoted P_C.

Definition 6.8:

The function $\alpha: N_c \rightarrow Z^+$ defines, for each node of type WHILEIF or FORIF, the value of the MAX constant in the statement.

Thus, for node K in Figure 6.3, $\alpha(K)=5$ and $\alpha(N)=10$. For all other nodes in the Figure, the function is undefined. We note that values for the function are easily obtained during syntax analysis of the code.

Definition 6.9:

The functions $\beta^{best}: N_c \rightarrow Z^+$ and $\beta^{worst}: N_c \rightarrow Z^+$ define for each node the best and worst case execution times for that node (when it is executed).

Definition 6.10:

The functions $\gamma^{best}: N_c \rightarrow Z^+$ and $\gamma^{worst}: N_c \rightarrow Z^+$ define for each node the best case and worst case execution times of any path from that node to either the STOP node, or the next retreating edge.

Consider a CFG consisting of only SIMPLE and IF type nodes (together with START and STOP nodes). For any node, γ^{best} and γ^{worst} correspond to the BCET and WCET of all paths from the node to the STOP node.

Consider the inclusion of loop constructs. Now, no path (consisting entirely of forward edges) exists from any node in the loop body, to the STOP node. For nodes in the loop body, γ^{best} and γ^{worst} represent the BCET and WCET respectively, for all paths from the node to the end of the loop body. Therefore, for the first node in the loop body, γ^{best} and γ^{worst} define the BCET and WCET of the loop body. For nodes of type FORIF or WHILEIF, the value of γ^{best} is merely β^{best} since the loop conditional expression is always evaluated at least once. The value of γ^{worst} for node A (of type FORIF or WHILEIF) is given by:

$$((\alpha(A)+1)*\beta(A))+ (\alpha(A)*\gamma^{worst}(B)) + \gamma^{worst}(C)$$

where nodes B and C are both successors of A; B representing the first node in the loop body, C the node after the loop body. We note that the FORIF and WHILEIF node is executed, in the worst-case, one more time than indicated by the value of the α function: this reflects the fact that for x executions of the loop body, the conditional expression is executed $x+1$ times.

The set of resources required in a CFG is denoted R_c.

Definition 6.11:

The function $\varepsilon: P_c \rightarrow R_c$ defines the set of resources that are accessed when traversing a given path.

6.3.3 Gain Point Placement And Value

Within the language model defined in the previous section, a gain point becomes a property of an edge in E_C^+ (no edge in E_C^- has an associated gain point).

Definition 6.12:

GP(A, B, v) defines a gain point which exists on edge (A,B) $\in E_C^+$. The gain point has value v: by traversing that path at run-time, v units of gain time, or gained resource v, has been detected.

The placement and value of the gain points outlined in section 6.2 is now formally defined. The mathematical notation of the Z specification language is adopted [Hayes87].

The set of all sets of nodes in CFGs is introduced [NODE], together with the set of all sets of paths [PATH] and resources [RESOURCE]. The set of nodes (N_c), paths (P_c) and resources (R_c) are defined:

N_c : 𝔽 NODE

P_c : 𝔽 PATH

R_c : 𝔽 RESOURCE

We now turn to the formal definition of gain point value and placement.

Static Gain Points

$$\forall n : \text{NODE} \mid n \in N_C \ \wedge \ \phi(n) = \text{IF} \ \bullet$$
$$\exists s_1, s_2 : \text{NODE} \mid s_1 \in \Lambda^+(n) \ \wedge s_2 \in \Lambda^+(n) \ \bullet \ s_1 \neq s_2 \wedge \gamma^{worst}(s_1) > \gamma^{worst}(s_2)$$
$$\Leftrightarrow \ \text{GP}(n, s_2, \gamma^{worst}(s_1) - \gamma^{worst}(s_2))$$

We note that nodes of type IF have exactly two successors.

Efficiency Gain Points

$$\forall n : \text{NODE} \mid n \in N_C \ \bullet$$
$$\forall s : \text{NODE} \mid s \in \Lambda^+(n) \ \bullet$$
$$true \Leftrightarrow \ \text{GP}(n, s, \gamma^{worst}(n) - \text{ACET}(n))$$

The ACET of a simple block is assumed to be available from the kernel.

Dynamic Gain Points

For nodes of type WHILEIF a single gain point is placed on the edge between the WHILEIF node and its successor that represents the first node after the loop body (i.e. not the successor node that represents the first node in the loop body). This gain point is defined:

$$\forall n : \text{NODE} \mid n \in N_C \ \wedge \ \phi(n) = \text{WHILEIF} \ \bullet$$
$$\exists s_1, s_2 : \text{NODE} \mid s_1 \in \Lambda^+(n) \ \wedge s_2 \in \Lambda^+(n) \ \bullet \ \ s_1 \neq s_2 \wedge$$
$$\exists s : \text{NODE} \mid s \in N_C \ \wedge \phi(n) = \text{STOP} \wedge [s_1, s] \in P_c$$
$$\Leftrightarrow \text{GP}(n, s_2, (\alpha(n) - loopvar) * (\gamma^{worst}(s_1) + \beta^{worst}(n)))$$

Nodes of type FORIF have two gain points associated with them. The first is placed on all edges between predecessor nodes and the FORIF node:

$$\forall n : \text{NODE} \mid n \in N_C \ \wedge \ \phi(n) = \text{FORIF} \ \bullet$$
$$\forall p : \text{NODE} \mid p \in \Lambda^-(n) \ \wedge$$
$$\exists s_1, s_2 : \text{NODE} \mid s_1 \in \Lambda^+(n) \ \wedge s_2 \in \Lambda^+(n) \ \bullet \ \ s_1 \neq s_2 \wedge$$
$$\exists s : \text{NODE} \mid s \in N_C \ \wedge \phi(n) = \text{STOP} \wedge [s_1, s] \in P_c$$
$$\Leftrightarrow \text{GP}(p, n, (\alpha(n) - max) * (\gamma^{worst}(s_2) + \beta^{worst}(n)))$$

The second gain point is placed on the edge between the FORIF node and its successor that represents the first loop after the loop body.

$$\forall n : \text{NODE} \mid n \in N_C \ \wedge \ \phi(n) = \text{FORIF} \ \bullet$$
$$\exists s_1, s_2 : \text{NODE} \mid s_1 \in \Lambda^+(n) \ \wedge s_2 \in \Lambda^+(n) \ \bullet \ \ s_1 \neq s_2 \wedge$$
$$\exists s : \text{NODE} \mid s \in N_C \ \wedge \phi(n) = \text{STOP} \wedge [s_1, s] \in P_c$$
$$\Leftrightarrow \text{GP}(p, n, (max - loopvar) * (\gamma^{worst}(s_2) + \beta^{worst}(n)))$$

Resource Usage Gain Points

The primary observation made in section 6.2.4 is that all resource usage gain points are placed outside loop constructs. Firstly, resource usage gain points may be placed on edges between an IF node and its successors, if resources will no longer be required if that path is taken:

$$\forall n : \text{NODE} \mid n \in N_C \ \wedge \ \phi(n) = \text{IF} \ \bullet$$
$$\exists s_1, s_2 : \text{NODE} \mid s_1 \in \Lambda^+(n) \ \wedge s_2 \in \Lambda^+(n) \ \bullet \ \ s_1 \neq s_2 \wedge [s_1, \text{STOP}] \in P_c \wedge [s_2, \text{STOP}] \in P_c \ \bullet$$
$$\forall r : \text{RESOURCE} \mid r \in \varepsilon([s_1, \text{STOP}]) \ \bullet$$
$$r \notin \varepsilon([s_2, \text{STOP}]) \Leftrightarrow \text{GP}(n, s_2, r)$$
$$\vee \ \ \forall r : \text{RESOURCE} \mid r \in \varepsilon([s_2, \text{STOP}]) \ \bullet$$
$$r \notin \varepsilon([s_1, \text{STOP}]) \Leftrightarrow \text{GP}(n, s_1, r)$$

Secondly, resource usage gain points may be placed on the edge leading from a WHILEIF or FORIF node to the first node after the loop body if resources used in the loop body are not required by the rest of the CFG:

$$\forall n : \text{NODE} \mid n \in N_C \ \wedge \ \phi(n) \in \{\text{FORIF, WHILEIF}\} \bullet$$
$$\exists s_1, s_2 : \text{NODE} \mid s_1 \in \Lambda^+(n) \ \wedge s_2 \in \Lambda^+(n) \bullet \ s_1 \neq s_2 \wedge [s_1, \text{STOP}] \in P_c \bullet$$
$$\forall n_1 : \text{NODE} \mid n_1 \in N_C \ \wedge [s_2, n_1] \in P_c \bullet$$
$$\forall r : \text{RESOURCE} \mid r \in \varepsilon([s_2, n_1]) \bullet$$
$$r \notin \varepsilon([s_2, \text{STOP}]) \Leftrightarrow \text{GP}(n, s_1, r)$$

It is noted that no resource usage gain points are placed in nested loops.

6.3.4 Preservation Of Utilisation

If the detection of gain time is accurate, whichever path is taken from the START to STOP node in a process execution, the sum of actual execution time and detected gain time is equal to C_i, the WCET of a process: utilisation is preserved. This could be achieved by placing an efficiency gain point immediately before the STOP node. However, the insertion of gain points according to the definitions of placement and value given in the previous section enables earlier detection. The remainder of this section shows that such gain points preserve utilisation.

Consider static gain points. If the WCET of the paths leading from a divergent node differ, a static gain point is placed onto the shorter (in terms of execution time) path. Formally, divergent node A has two successor nodes, B and C. Let γ^{worst} (B) > γ^{worst}(C). A gain point is placed onto (A, C) to value γ^{worst} (B) - γ^{worst}(C). Given C_A, the actual execution time up to node A, and G_A, the gained time up to node A, then

$$C_A + G_A + \gamma^{worst}\ (\text{A}) = C_i \tag{6.5}$$

Expanding for node B, the actual computation becomes $C_A + \beta^{worst}(\text{A}) + \gamma^{worst}(\text{B})$, with the gain time detected up to node A remaining at G_A. Thus,

$$C_A + \beta^{worst}(\text{A}) + \gamma^{worst}(\text{B}) + G_A = C_i \tag{6.6}$$

Expanding for node C, the actual computation becomes $C_A + \beta^{worst}(\text{A}) + \gamma^{worst}(\text{C})$, with the gain time detected $G_A + \gamma^{worst}$ (B) - γ^{worst}(C). Thus,

$$C_A + \beta^{worst}(\text{A}) + \gamma^{worst}(\text{C}) + G_A + \gamma^{worst}\ (\text{B}) - \gamma^{worst}(\text{C})$$
$$= C_A + \beta^{worst}(\text{A}) + G_A + \gamma^{worst}\ (\text{B}) = C_i \tag{6.7}$$

The equivalence between equations (6.5), (6.6) and (6.7) for paths from node A is noted.

For dynamic gain points it is observed that the value of any such gain point is equal to the amount of execution saved by not executing the maximum number of iterations. Thus, if one iteration is not executed, the actual execution time after the loop must be at least w less than the worst-case execution time to that point, where w is the worst-case execution time of the loop. Additionally, the gain time up to this point will have increased by w.

Trivially, the property also holds for efficiency gain points since they only detect the difference between the anticipated and actual execution times of a block of code analysed to have constant execution time.

6.3.5 Summary

The informal model of gain point placement and value given in section 6.2 has been formalised. A model of process code and possible control-flow through the code was defined using an acyclic graph. Then, the placement and value of gain points was defined, where gain points are a property of an edge between two nodes in the graph. It is noted that in the formal model, a succession of nodes each with a single successor and a single predecessor (i.e. a linear chain of nodes) were not joined to form a basic block as implied by the informal model of section 6.3. Thus, efficiency speed-ups are detected after every statement rather than every basic block. This is clearly more accurate, as gain time is detected sooner, although the number of gain points, and therefore the associated overheads, are greater. The trade-off between accuracy and overheads is discussed further in section 6.4.

The formal model of gain points is shown to preserve utilisation: all gain time is detected. Although this could be achieved by noting the actual execution time compared to the worst-case on process completion (i.e. no active gain time detection), gain points allow the majority of gain time to be detected before process completion. Dix's and Haban's approaches also identify (most) gain time before process completion, although later than the gain point approach. It is noted that these approaches are equivalent to the insertion of a single efficiency gain point after each basic block.

6.4 Implementation And Overhead Considerations

Ideally, process set feasibility should not be affected by the insertion of gain points: the WCET of a process must be the same before and after gain point insertion. This property holds if the value of a gain point (i.e. the amount of gain time detected) is always at least the cost c (assumed constant) of executing the gain point. Effectively, gain time is used to execute the gain point itself. However, achieving this property in practice is difficult. Indeed, it is noted that Dix's and Haban's approaches do not maintain this property.

One method of ensuring that the property holds is by using clairvoyance: only gain points that will have values of at least c are inserted. This is clearly impossible in practice as the values of dynamic and efficiency gain points cannot be determined offline. Alternatively, dynamic and efficiency gain points could be omitted, with only static gain points having values at least c being inserted. With this approach, the accuracy of gain time detection decreases (although all gain time not detected by gain points will be detected as spare capacity on process completion). Clearly, the first method of ensuring that feasibility is not affected by gain point insertion is not practical. The second method greatly reduces the accuracy of gain point detection.

In Chapter 3 the trade-off between accuracy of detection and overheads incurred was outlined. As the accuracy of gain time detection increases, so do the overheads, with the likelihood of the process set being feasible decreasing. For example a process with a single gain point placed at the end of its CFG (effectively detecting the worst-case minus the actual-case execution time of the process) detects g gain time, at a cost of c. In contrast, the gain time placement outlined earlier in the chapter where (at least) one gain point is placed on each edge in the CFG, detects g gain time, but at a cost of nc where n is the number of gain points. Thus, the WCET of the process is increased by c in the first case and nc in the second.

If the process set remains feasible, despite the additional overhead of nc for each process (where n will vary for different processes), then all the gain points suggested by the previous sections in this chapter could be inserted. However, if the process set is not feasible with all those gain points, but is feasible without them, then the number of gain points needs to be decreased. Also, even if all n gain points can be inserted without the process set becoming infeasible, many gain points will not be worthwhile.

For example, it is observed that the number of efficiency gain points inserted is high compared to the number of static and dynamic gain points, as

efficiency gain points are inserted after each node. However, in general, the gain time detected by efficiency gain points will be small compared to that detected by static and dynamic gain points. This is due to hardware efficiency gains being small relative to gains made due to control flow decisions not to execute nodes (in a CFG). Therefore, if the overall number of gain points is to decrease, it is preferable to reduce the number of efficiency gain points rather than static or dynamic gain points.

Consider an implementation of efficiency gain points which reports the gain time due to hardware speed-ups since the last efficiency gain point rather than merely the last node executed in the CFG. This requires less efficiency gain points to be inserted, each detecting more gain time. The ratio of overheads to detected gain time has decreased, along with detection accuracy, compared to insertion of efficiency gain points after every node.

In the extreme, all efficiency gain points could be omitted, with the accuracy of detection of gain time generated by hardware speed-ups decreasing further. However, this problem can be (partially) overcome by the implementation of gain points given in the following section. Subsequent sections providing an implementation for the insertion of gain points into process code and the detection of slack time respectively.

6.4.1 Gain Point Implementation

The architecture assumed by this section is given in Figure 6.5. This is an abstraction of conventional hard real-time kernel design [Kopetz85, Ramamritham87, Burns92] also assumed by Haban *et al* [Haban90]. Resident in the kernel is the scheduler implementing a gain time policy (discussed further in Chapter 7). Within this context, the implementation of gain points involves insertion of code into a process, that code being responsible for communicating the value of detected gain time to the kernel (and therefore the gain time scheduler). This is achieved by a kernel interface call to record gain time with parameters including the value of the gain time detected, and the type of gain point making the call. The call is termed the Gain Time Kernel Call (GTKC). For static gain points, only the GTKC need be inserted into process code (since the gain point value is constant). For dynamic gain points code to calculate the value of the gain point is required as well as the kernel call.

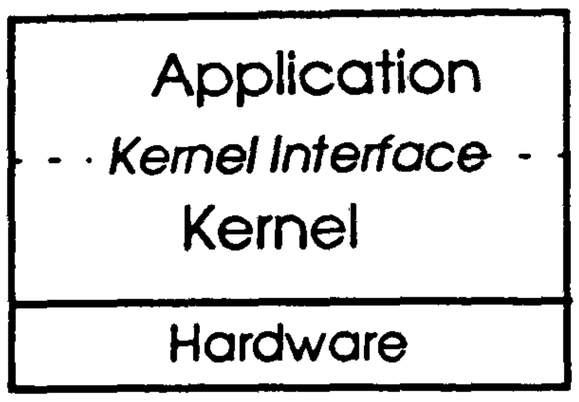


Figure 6.5: Architecture.

Efficiency gain points require the actual and worst-case execution times of the preceding block in order to calculate their value. This would require at least one kernel call. This overhead can be reduced by merely inserting a GTKC of null value, with the kernel obtaining the actual and worst-case execution times of the preceding block and calculating the gain point's value.

In discussions above, it was indicated that the number of efficiency gain points can be reduced by calculating the value of an efficiency gain point with respect to all nodes in the CFG executed since the last efficiency gain point, not just the previous node. Also, the value of the gain point is more efficiently calculated by the GTKC itself. Thus, the value of an efficiency gain point becomes:

$$C_i - \gamma^{worst}(n) - a_i - g_i$$

where n is the node in the CFG after the call. The actual execution time of τ_i up to (and including) the call is a_i, with the gain time reported by τ_i prior to the call given by g_i. We note that $\gamma^{worst}(n)$ is available as a static array instantiated at system start-up.

A key issue when developing hard real-time kernels is that of protection [Burns92]. This manifests itself in the implementation of gain points and the associated GTKC. Consider static gain points, where the value of the gain point is constant. Conventionally, this constant would be embedded in the process code, with little likelihood of the process being able to maliciously alter it (assuming that code is write protected). However, dynamic gain points calculate the amount of gain time before making the kernel call. A malicious (or faulty) process may report more gain time than actually generated causing potential failure of processes to meet deadlines (if that gain time were assigned to another process). To counteract this problem, the GTKC must check to see if the value of reported gain time is possible.

Intuitively, this could be achieved by checking that the following condition holds when a GTKC is made:

$$a_i + g_i + g + \gamma^{worst}(n) \leq C_i$$

The actual execution time up to the call includes all nodes executed prior to the gain point call, whilst the worst-case execution time of remaining code is $\gamma^{worst}(n)$ for the node after the gain point call *(n)*. The amount of reported gain time is *g*. The inequality is due to the possibility of hardware speed-ups not having yet been detected by efficiency gain points.

Consider a rogue gain point of value $g+\delta$, being δ greater than its actual value, with the actual execution time up to the call being at least δ better than expected due to hardware speed-ups. The above condition will hold. Now, a GTKC is made at an efficiency gain point, of value δ' (where $\delta' \geq \delta$). The net effect is that $g+\delta'+\delta$ has been reported, even though only $g+\delta'$ has actually been generated, with the above condition failing.

The problem can be solved by noting efficiency gain time when a GTKC is made for a static or dynamic gain point. Initially, the reported gain time is increased by the efficiency gain time since the last GTKC (either efficiency, static or dynamic). This is used, along with the value in the gain point call, to evaluate condition given above.

```
procedure GTKC ( type : NodeType; value : INTEGER;
                        snode : N_C) is
begin
    -- add on efficiency gain time since last GTKC
    gt[i] = C_i - γ^worst(snode) - actual[i]     ;
    -- now do static / dynamic
    if type = static or type = dynamic then
        if value + gt[i] + actual[i] + γ^worst(snode)
                                  ≤ C_i then
            gt[i] = gt[i] + value    ;
        end if    ;
    end if    ;
end GTKC ;
```

Figure 6.6: Algorithm For GTKC.

The pseudo-code for the GTKC is given in Figure 6.6. The array `gt[i]` is used to accumulate the gained time generated by process τ_i (where *i* is available within the kernel, being the identifier for the currently executing process). In Chapter 7, this method of recording the gain time is expanded. The array `actual[i]` is used to record the actual execution time of a process. The

parameter snode represents the node immediately after the gain point call in the CFG. This is a constant, available at compilation time. The values for γ^{worst} for the nodes in a process's CFG are initialised at system start-up.

The implication of implementation of GTKC is that efficiency gain time is detected during a GTKC for static or dynamic gain time, implying that efficiency gain points need not always be inserted into a process's CFG.

6.4.2 Gain Point Insertion Into Process Code

The WCET of a CFG can be determined by analysing the graph from START to STOP node. For each node, the functions β^{best}, β^{worst}, γ^{best} and γ^{worst} need to be defined. Nodes of type START and STOP are null, and therefore require 0 execution time. Nodes of type IF, LOOP and SIMPLE consist of 1 or more statements executed in strict sequence. Each statement is assumed to take constant time: hence β^{best} and β^{worst} of each statement, and therefore the node, are equal. The value of β^{best} (and therefore β^{worst}) is provided by basic block analysis of conventional WCET analysis, e.g. instruction cycle counting [Puschner89].

Node of type CALL must be treated differently: β^{best} and β^{worst} reflect the best and worst-case execution times of both the CALL node itself and the called procedure. To obtain the WCET of the procedure, the CFG corresponding to the procedure is analysed. Now, β^{best} and β^{worst} for a CALL node correspond to the sum of the best-case execution time of the procedure call and procedure code, and to the sum of the worst-case execution time of the procedure call and procedure code, respectively. Thus, even if the same procedure is called at different points in a process's CFG, the procedure code is only analysed once. An alternative strategy would be to expand the CFG of the procedure within the CFG for each process that calls the procedure, and at each point within the process that the procedure is called. This is inefficient compared to the first approach which requires a single pass through the CFG only.

To obtain γ^{best} and γ^{worst} (and the best and worst-case execution times of a procedure), each possible path from START to STOP nodes in the CFG is analysed. One method of analysis is via a depth-first recursive scan of the CFG (i.e. from STOP to START). This approach reflects the observation that γ^{best} and γ^{worst} for a node depends upon the values of γ^{best} and γ^{worst} being defined for its successor nodes. Whenever a CALL node is encountered, the first node in the CFG for the associated procedure is examined to see if γ^{best} and γ^{worst} have been

defined for that node. If they have not, a recursive scan of the procedure's CFG is initiated.

Having defined β^{best} , β^{worst}, γ^{best} and γ^{worst} for a process's CFG (and any procedures invoked directly or indirectly by the process), gain point insertion may proceed. This could be achieved by several alternative methods, including a bottom-up scans of the CFG (c.f. WCET analysis). Simplistically, two passes are required through the CFG, one for determination of γ^{best} and γ^{worst} and the other for insertion of gain points. I

6.4.3 Evaluation Of Gain Time Detection

In this section the detection of gain time using gain points is compared with the alternative approaches of Dix and Haban. Specifically, three approaches are compared:

Pragmatic Gain Points (PGP) : all static and dynamic gain points are inserted, (with most efficiency gain time detected by these gain points), with a single gain point inserted before the final (STOP) node in a process's CFG.

Dix / Haban Approach (DH): an efficiency gain point is inserted after each basic block in a process's CFG.

Experimental Control (EC): a single efficiency gain point is inserted prior to the final (STOP) node in a process's CFG.

The PGP approach is that outlined in sections 6.3 and 6.4. The EC approach provides a control for the comparison.

During comparison, time at which gain time is detected within a process's execution is noted for a number of processes, along with any overhead costs that have been incurred. It is assumed that any gain time detected has a useful lifetime, or scope, up to the deadline of the process generating that spare capacity (see Chapter 7 for further details). This enables comparison between approaches that detect gain time at different points in a process's execution.

The generation of random process sets to use with the above gain time detection approaches is difficult since actual process code structure needs to be generated, i.e. loops, conditional statements etc. To overcome this problem, the WCET, offsets, deadlines and periods were chosen randomly for 5 processes (see section 4.7). The randomly generated process timing characteristics are given in Table 6.1. It is noted that the worst-case utilisation of the process set is 95.96%. The process set does have a critical instant (the first occurring at

time 4107), and is feasible with priorities assigned in a deadline-monotonic manner (i.e. in descending order in the table).

An arbitrary process structure was then composed for each process, with portions of WCET assigned to basic blocks within that structure. The size of the blocks (in terms of WCET) were differed across the process set. Also, a minimum execution time was assigned to each basic block, with the actual execution time of that block determined by a random number (from a normal distribution). The process code is given in Appendix B.

Process	O	C	D	T
τ_1	47	77	92	406
τ_2	435	173	264	612
τ_3	504	393	971	1201
τ_4	1235	89	1062	1436
τ_5	1740	232	2360	2367

Table 6.1: Process Set For Gain Point Evaluation.

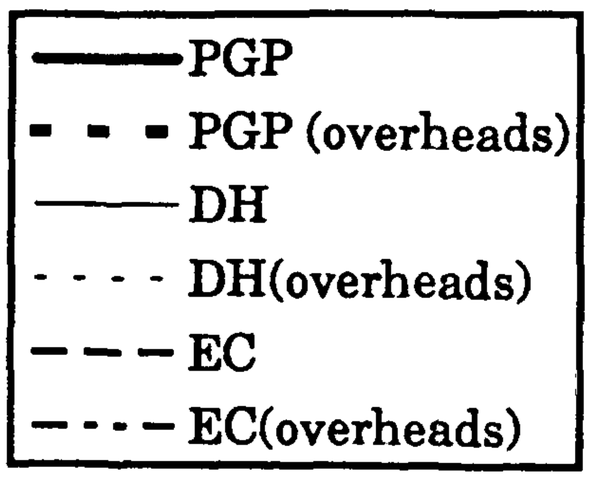
The cost of each gain point is assumed to be constant for an application on a given hardware platform. However, this may vary across applications and platforms. Consider the gain point implementation given in section 6.4 where a kernel interface call is prescribed. The overheads of such a call are, in general, greater than an application level procedure call due to protection [Burns92].

For example in a system implemented on an Intel 486DX processor [Intel89], a kernel call with protection could be implemented using the hardware "call gate" mechanism. The low privilege application process has its privilege increased for the duration of the call. The cost of the call, due to hardware checking of the callers right to make such a call, is 94 cycles on a 33MHz processor, that is about 3μseconds. This is the equivalent to about 4 multiplication instructions. In contrast, an implementation which does not utilise protection, the GTKC is the equivalent of a normal procedure call. On an Intel 486DX processor this requires 23 cycles (about 0.7 μseconds).

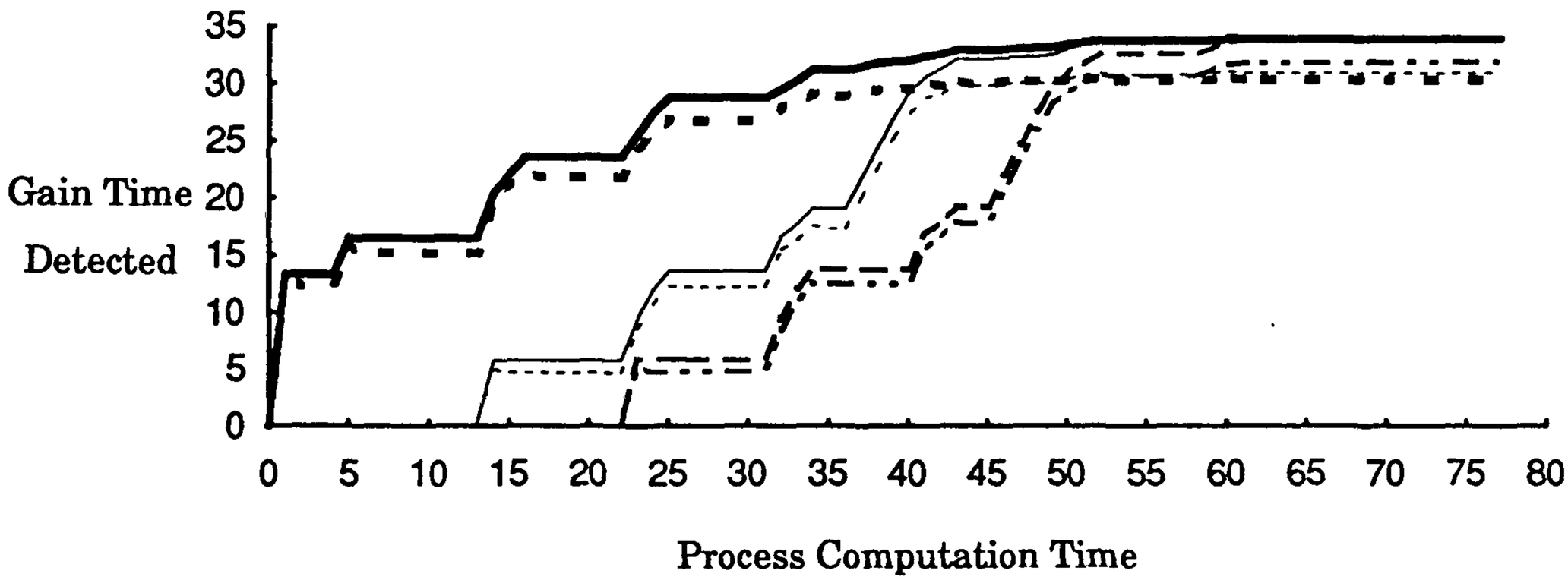
These differing overheads are recognised in the comparison, with the detection approaches considered with respect to overhead costs of 1 and 4. These values are approximately 5% and 20% of the average block size of the processes.

A number of runs of the process set were performed, with differing actual execution times of basic blocks between runs, although the same actual

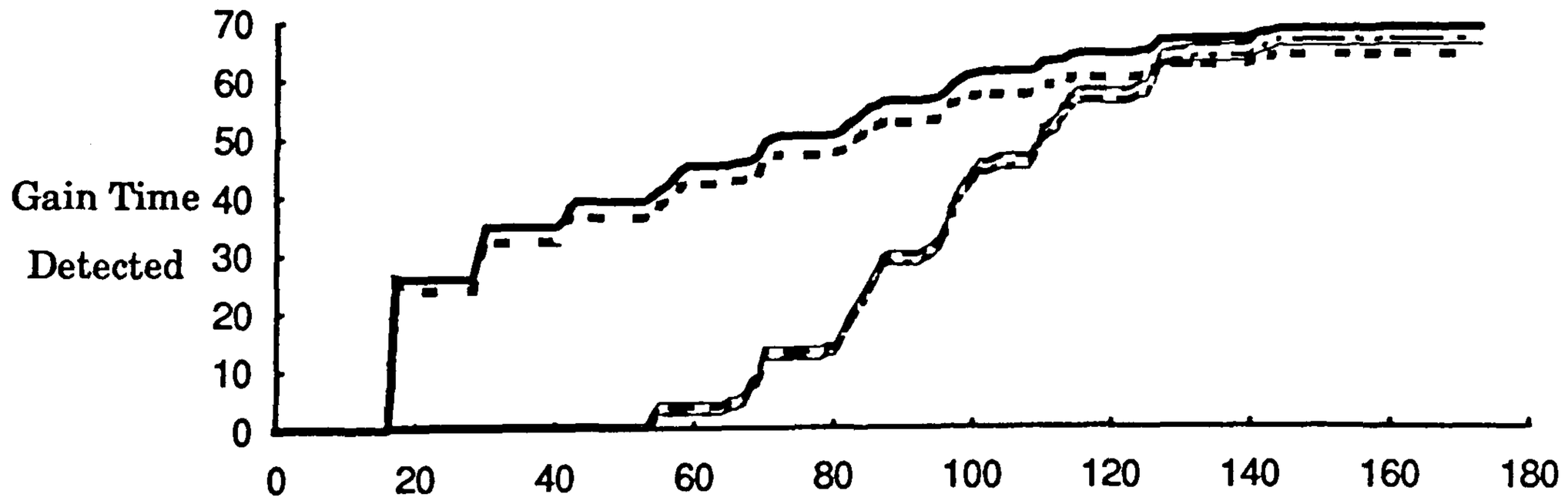
execution times were used for each of PGP, DH and EC for a given execution in a run. In all, there were 1201, 792, 409, 336 and 162 executions of processes τ_1, τ_2, τ_3, τ_4 and τ_5 respectively (the differences being due to the relative periods of the processes).



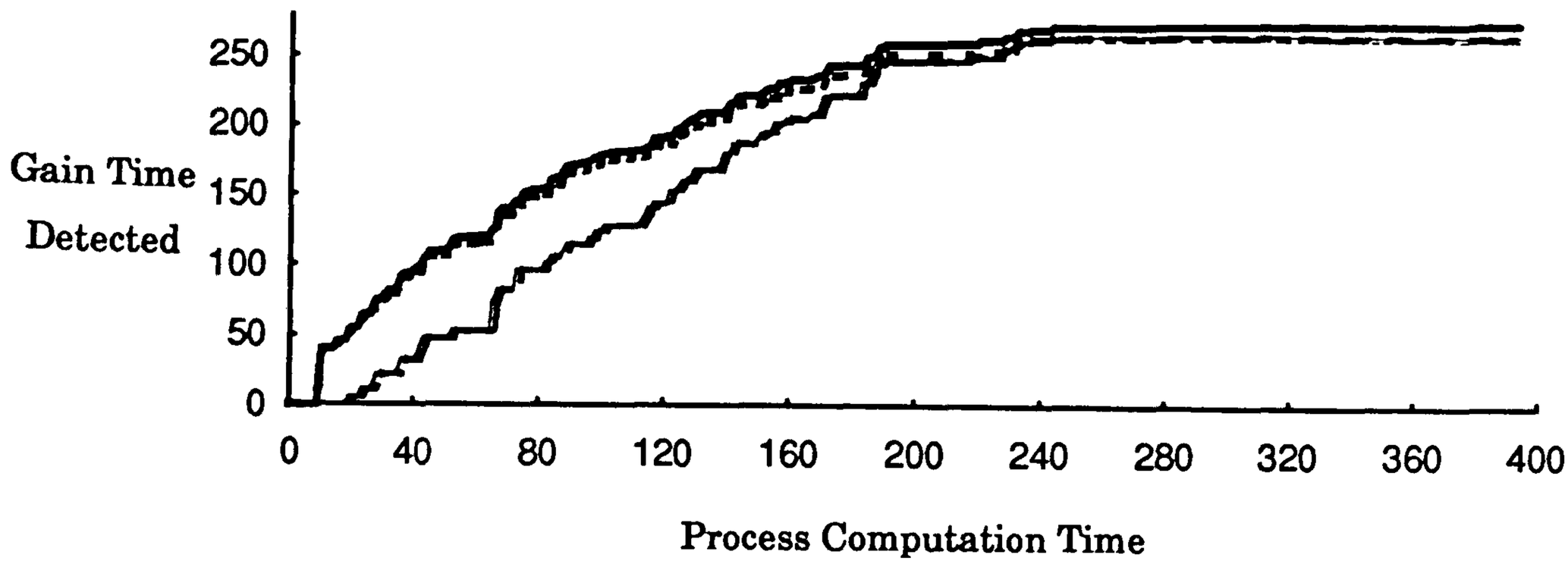
Key For Graphs 6.1 - 6.10



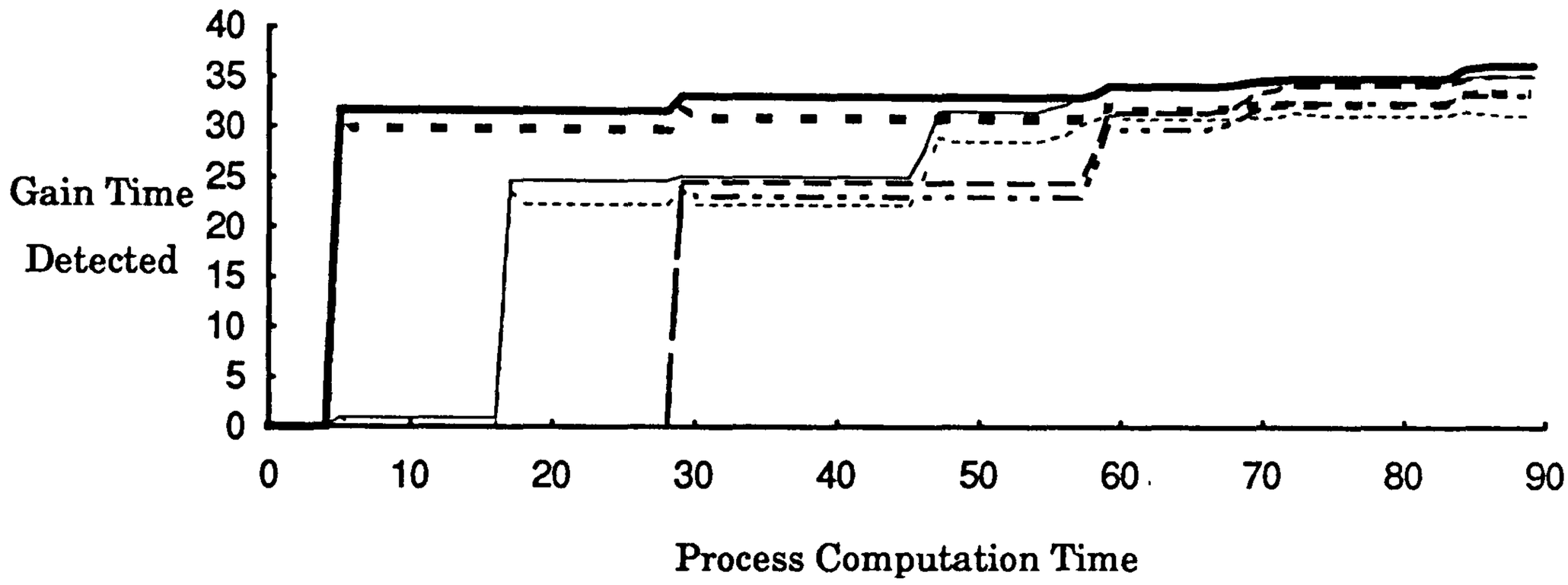
Graph 6.1: Gain Time Detected From Executions Of τ_1 (Overheads = 1).



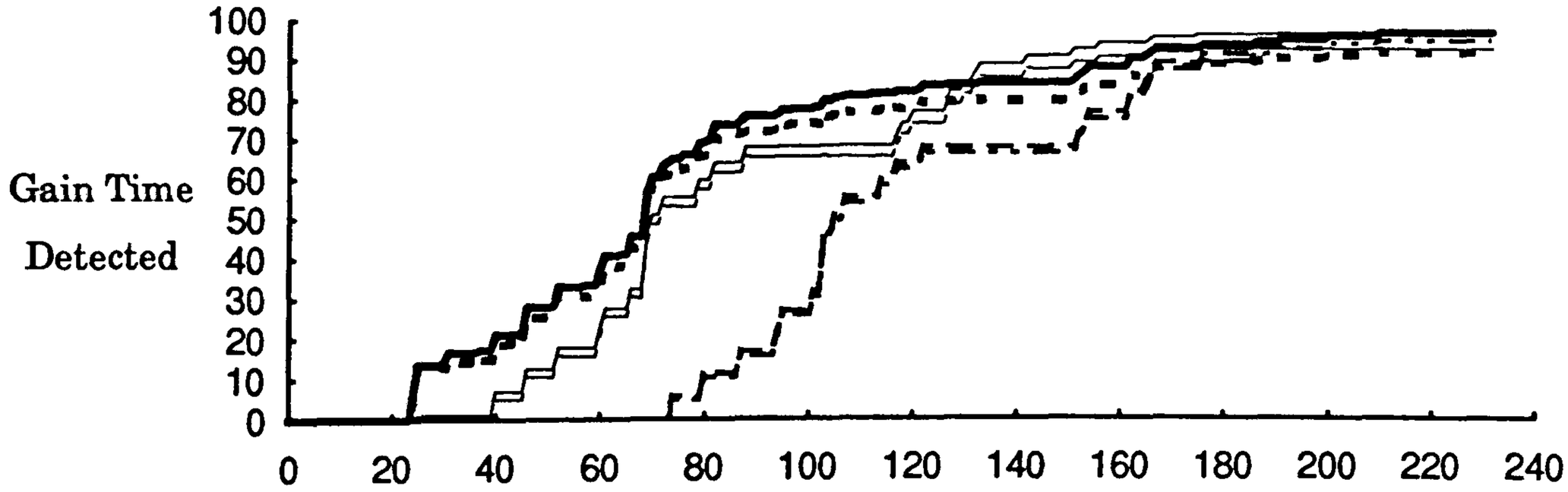
Graph 6.2: Gain Time Detected From Executions Of τ_2 (Overheads = 1).



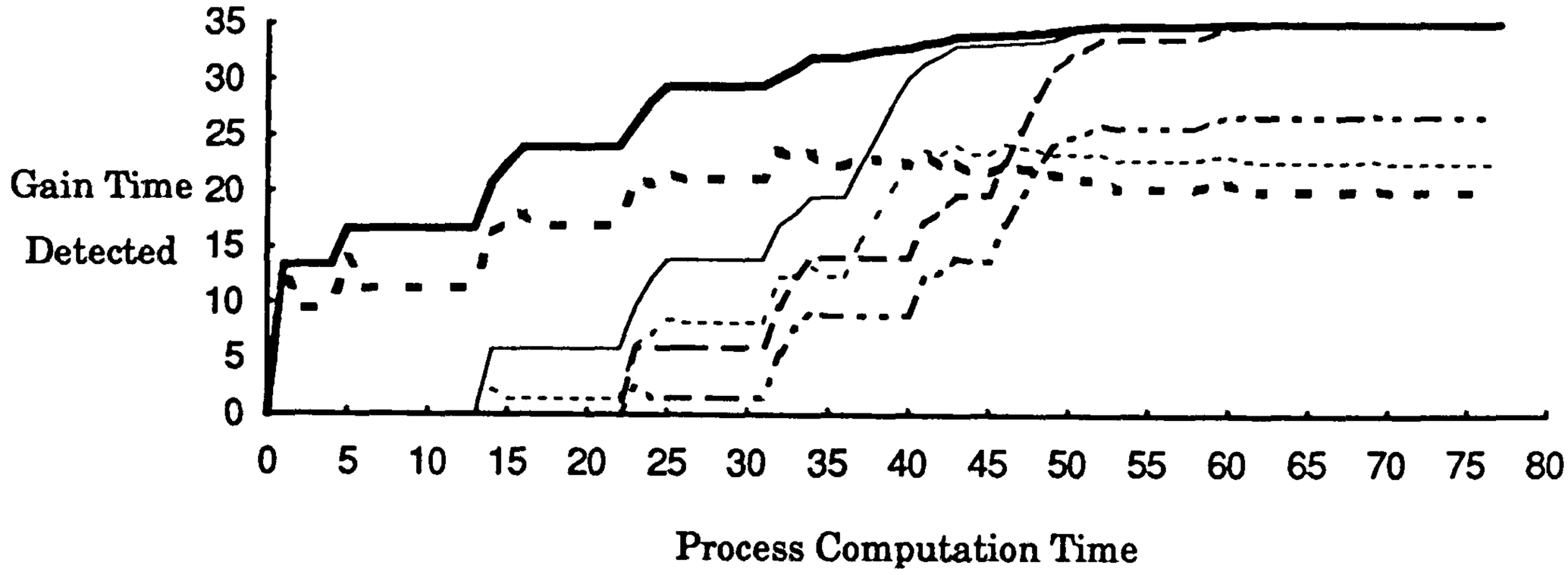
Graph 6.3: Gain Time Detected From Executions Of τ_3 (Overheads = 1).



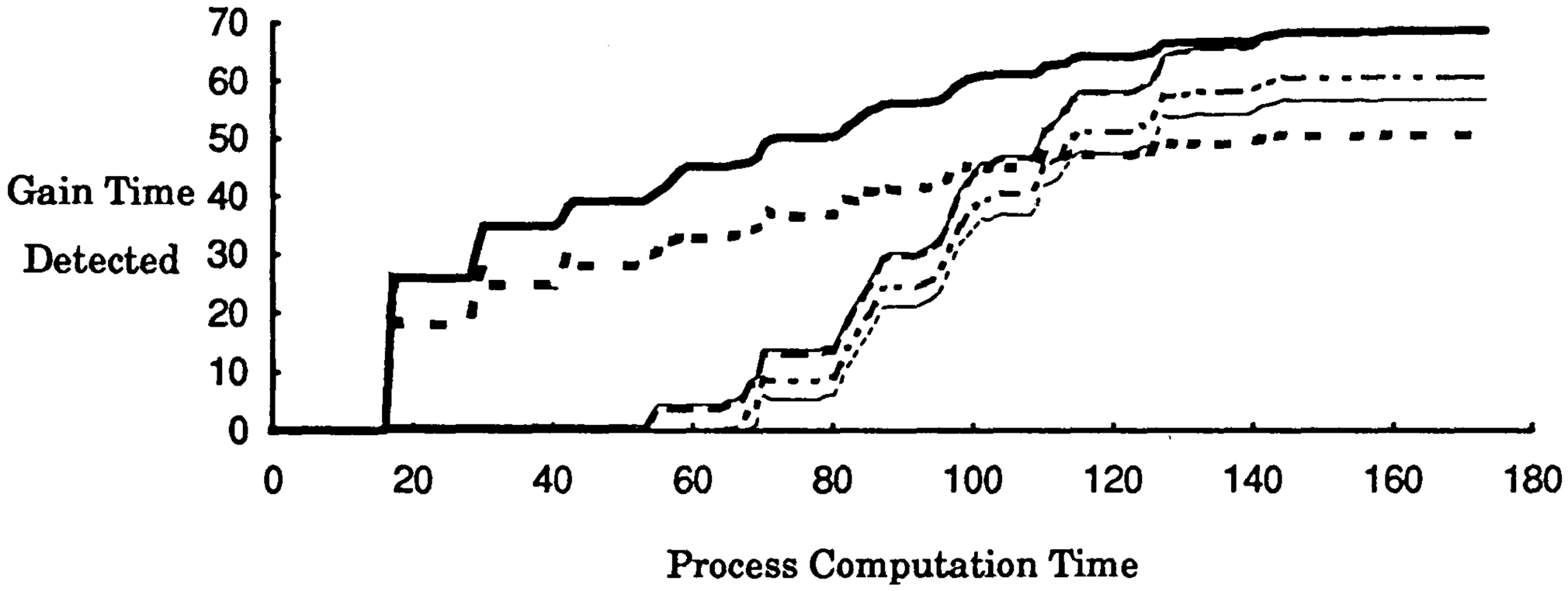
Graph 6.4: Gain Time Detected From Executions Of τ_4 (Overheads = 1).



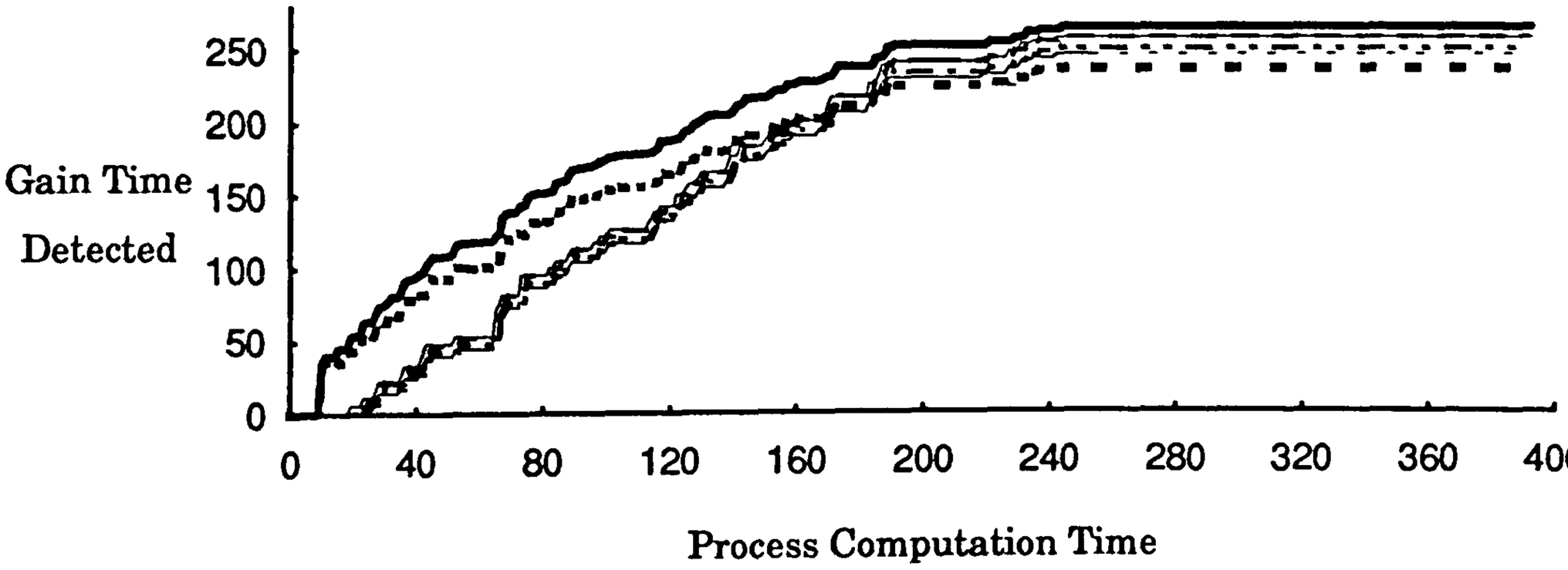
Graph 6.5: Gain Time Detected From Executions Of τ_5 (Overheads = 1).



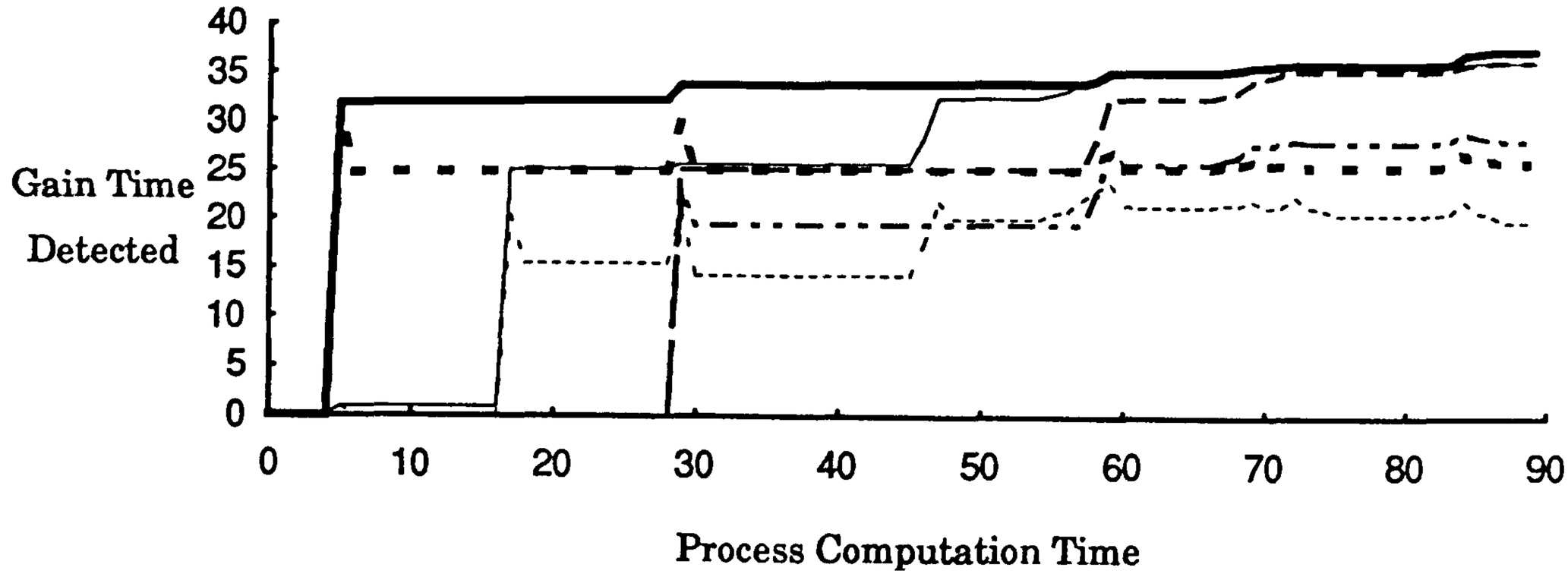
Graph 6.6: Gain Time Detected From Executions Of τ_1 (Overheads = 4).



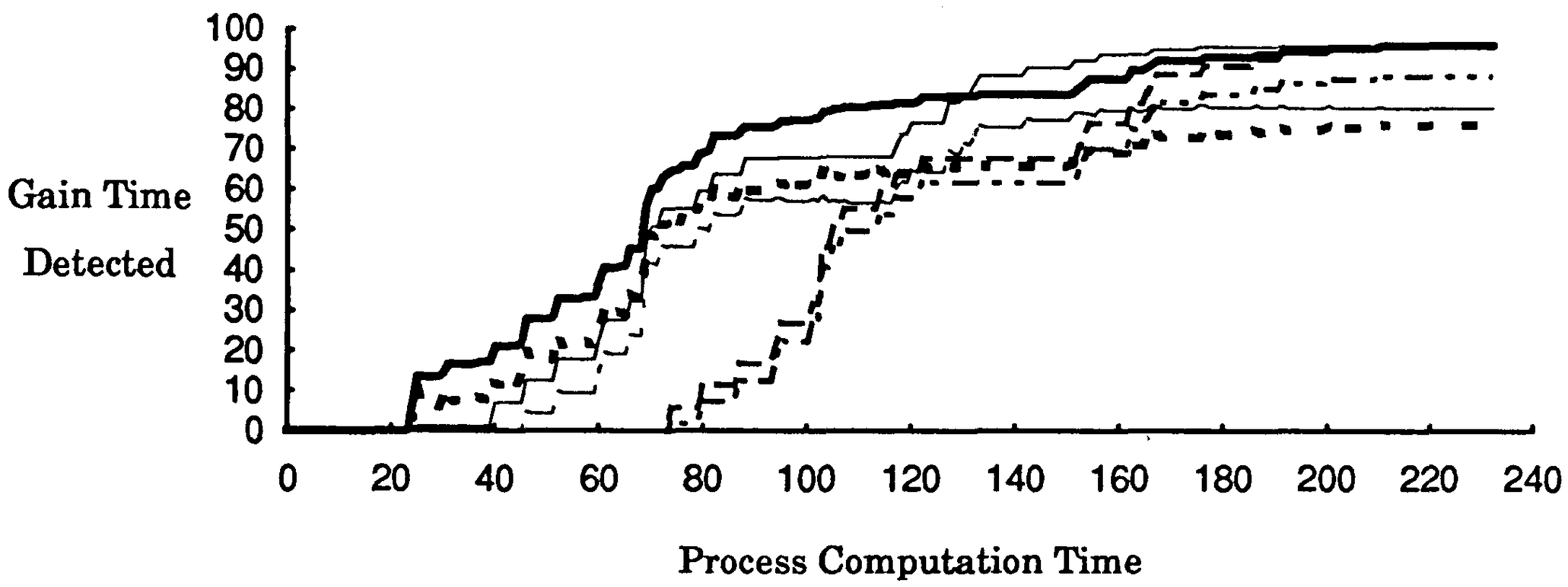
Graph 6.7: Gain Time Detected From Executions Of τ_2 (Overheads = 4).



Graph 6.8: Gain Time Detected From Executions Of τ_3 (Overheads = 4).



Graph 6.9: Gain Time Detected From Executions Of τ_4 (Overheads = 4).



Graph 6.10: Gain Time Detected From Executions Of τ_5 (Overheads = 4).

The graphs plot on the y-axis the average amount of gain time detected during the execution of a process, and the average amount of gain time detected after overheads have been subtracted for each gain point. The x-axis represents the amount of process computation time that has elapsed. This ranges from 0 to C_i for process τ_i. Thus, in graph 6.1, after τ_1 has executed for 20 time units, an average of 24 units of gain time have been detected by the PGP approach, 5 time units have been detected by the DH approach, with no gain time detected by the EC approach. After the subtraction of overheads, 22, 4 and 0 time units are detected respectively.

When overheads are not considered, the graphs show that in general, gain time is detected earliest by gain points, when compared with Dix's and Haban's approach, and the experimental control. Thus, detection using gain

points is, in the majority of cases, the most accurate. For process τ_5, Dix's and Haban's approach appeared the most accurate within interval [130, 200] (see graph 6.5 or 6.10). This was due to the efficiency gain time after the execution of a basic block being detected immediately after the execution of the basic block by the DH approach and not by the PGP approach. It is noted that an efficiency gain points could be inserted after the basic block under PGP (as proposed by the informal and formal models in sections 6.2. and 6.3) although this would add to overheads.

When overheads are considered, the higher overhead of 4 (graphs 6.5-6.10) reduces the amount of gain time that can be re-used when compared with an overhead of 1 (graphs 6.1-6.5). For practical implementations, this overhead must be kept as low as possible, by using the fastest secure mechanisms available for communication to the kernel.

6.5 Extensions

The sources of spare capacity outlined at the beginning of this chapter include the pessimism inherent in the WCET analysis of processes. Previous sections have shown how the pessimism due to control-flow branches in code and inaccuracies in basic-block analysis can be detected earliest by the use of gain points. However, it has been observed that in some cases, processes will never execute for their WCET due to their semantics [Puschner89]. For example, WCET analysis may assume the execution of two basic-blocks that cannot be both executed at run-time (i.e. one block is executed if a condition is true, the other if the same condition is false). *Semantic aids* inserted into process code can reduce such WCET pessimism.

Consider an application process which performs some operations on a 300x200 pixel image (drawn from [Puschner89]):

```
row = 1
while row ≤ 300 loop
    column = 1
    while column ≤ 200 loop
        if pixel[row, column] = 1 then
            -- if pixel set perform operation
            -- 10 time units
```

```
            end if  ;
            column = column + 1 ;
        end loop    ;
        row = row + 1   ;
    end loop    ;
```

Conventional analysis assumes 300x200 iterations, at a cost of 10 time units each. This is clearly pessimistic if the maximum number of pixels that are ever set is $s < 60000$. Puschner *et al* propose the use of *markers*, placed into the loop body indicating the actual maximum number of iterations of the loop [Puschner89]. Only one marker is permitted within the scope of any loop.

Such a marker could be placed inside the above loop code to inform WCET analysis that the pixel operation is called a maximum of 100 times (although the `IF` statement and associated condition will be called 300x200 times). If such markers are utilised, the placement of (dynamic) gain points is unaffected. However, the value of such gain points must consider the value of the marker. Thus the maximum number of iterations of the loop is given by:

$$\min\begin{pmatrix} value\ of\ a\ marker\ within\ the\ scope\ of\ the\ loop, \\ maximum\ iteration\ constant\ declared\ in\ loop\ construct \end{pmatrix}$$

The marker must also be considered when defining γ^{worst} and β^{worst} for nodes representing loop statements.

The gain point approach could also be extended to consider other semantic aids, for example those described by Park [Park93], which enable aspects of high-level knowledge regarding actual control-flow to be captured via a regular expression representation of possible control-flow paths.

6.8 Summary

To guarantee 100% predictability of processes requires the worst-case usage of resources to be considered. Often, the worst-case estimation of resource usage exceeds the actual-case usage. This implies that at run-time, a degree of spare capacity will be evident in the system. This chapter has considered the detection of such spare capacity, in particular gain time, defined as the proportion of spare capacity due to processes not executing for their worst-case execution time.

Gain time could be detected by comparing the actual and worst-case execution times for a block of process code, after the basic block has completed execution. However, much gain time can be detected earlier at control-flow branches in the code. Gain points were introduced to enable such detection. Gain points take the form of software triggers inserted into process code offline at the earliest points in the code that gain time can be detected (without the aid of clairvoyance).

Specifically, static, dynamic and efficiency gain points were described to detect gain time due to static control-flow, loops and hardware speed-ups respectively. A fourth form, namely resource gain points, were introduced to enable spare resources (i.e. non-processor) to be detected. An informal model was presented which described the placement and value of gain points. This was developed into a formal model, based upon an acyclic graph of process code. Within the context of the formal model, gain points were shown to detect all gain time.

An implementation of gain points was described, taking the form of a kernel call placed into process code. The implementation given enables the amount of gain time reported by a process to be checked so that rogue processes cannot report more gain time than has actually been detected. If this were possible, the allocation of gain time to another process could cause a guaranteed deadline to be missed.

The trade-off between the accuracy of detection and overheads incurred was discussed. As accuracy increases, so do the overheads, the latter detracting from the amount of gain time that can be assigned to other processes. One approach for the insertion of gain points dictates that efficiency gain points be inserted after each basic block of code, incurring significant overheads. However, an implementation of gain points was found whereby efficiency gain time can be detected by a gain time kernel call reporting static or dynamic gain time. This reduces the number of gain points required to detect all gain time, although at a cost of reduced accuracy.

The gain point approach was evaluated by comparing it with other approaches for detecting gain time, namely those proposed by Dix *et al* [Dix89] and Haban *et al* [Haban90]. Gain points were shown to detect gain time earlier than the other proposed approaches, although at a higher cost.

In general, the earlier detection of gain time provides a greater potential for the re-use of spare capacity, leading to the increase of overall system utility and flexibility at run-time.

Chapter 7.
Allocation Of Spare Capacity

The efficient and early detection of spare capacity was considered in Chapter 6. Gain time and unrequired resources are signalled to the underlying kernel, via gain points, at the earliest point at which they may be detected. The identification of intervals of slack time was seen to be the responsibility of the kernel itself. Given the ability to detect these forms of spare capacity at run-time, the outstanding issue now becomes the re-use of such spare capacity to extract greater utility from the system. This issue forms the main focus of this chapter.

In the literature, little consideration has been given directly to the allocation of detected spare capacity. In Haban *et al* [Haban90] and Moron *et al* [Moron93] policies are defined which allocate spare capacity to (non-crucial) processes that may otherwise miss their deadlines, so improving the number of deadlines met. Within the context of this thesis, crucial process deadlines are guaranteed offline with the result that any available spare capacity can be used to improve the overall utility of the system. Whilst meeting the (soft) deadlines of non-crucial (unguaranteed) processes may improve the utility of the system, it was argued in Chapter 3 that assignment of spare capacity to crucial processes can also improve overall system utility. Therefore, a computational model is required under which crucial processes may improve their utility by using spare capacity.

Many models have been proposed to enable optional, possibly unbounded, computation to be incorporated into crucial processes. Most are founded upon the notion of processes consisting of mandatory and optional parts, the former being guaranteed offline, the latter unguaranteed. Liu *et al* have identified three general methods by which optional software components can be incorporated into crucial process executions [Chung90, Liu91]:

(i) imprecise computations;
(ii) sieve functions;
(iii) multiple versions.

Imprecise computations produce a result whose accuracy increases monotonically as execution proceeds. A minimum accuracy is achieved using a mandatory (i.e. guaranteed) execution, with subsequent optional (possibly

unguaranteed) executions improving the result. The latter utilise run-time spare capacity. Sieve functions represent code within a crucial process that has not been guaranteed as part of a crucial process. However, if sufficient spare capacity is available, it may be executed at run-time. Again, the motivation is to enable additional accuracy or utility to be achieved. Multiple versions prescribe that several different implementations of a function are available at run-time, with the version with shortest execution time guaranteed as part of a crucial process. At run-time, if sufficient spare capacity is available, one of the more expensive (in terms of execution time) versions may be executed (assuming that greater execution time equates to greater accuracy or utility).

The above three methods have been generalised by Audsley *et al* [Audsley93a], where a process takes the form:

IP, C1, X, C2, OP

IP and OP represent the input and output phases of the process. C1 and C2 are both part of the mandatory computation (with IP and OP) with X forming the optional execution. It is this model that is assumed in this chapter for crucial processes that require spare capacity for additional computation.

To afford X, in the above model, the possibility of executing at run-time, two methods are identified:

(i) after executing C1 (and IP) the process requests spare capacity to execute X, prior to executing C2 (and OP);

(ii) the process is split into three processes: IP and C1 are placed into one process, C2 and OP into another (both these processes are guaranteed), with X forming a third, unguaranteed, process. This process may execute using spare capacity. A linear precedence constraint is placed over the three processes.

In method (i) only gain time can be allocated to execute X since at least one crucial process is runnable (with no slack time by definition). In method (ii), gain time or slack time could be allocated for X, if the first process completes before the release of the third process.

It is noted that in both methods, the crucial process demands spare capacity. This is achieved via a Spare Capacity Kernel Call (SCKC) within the kernel interface (see Figure 6.5 section 6.4.1). The parameters of such a call include the minimum and maximum amount of spare capacity requested, together with the deadline by which that spare capacity is required. If spare capacity is allocated, the exact amount given to a process is between the minimum and maximum values. If a deadline is specified, any allocated spare

capacity is guaranteed to execute before the deadline. Some of the possible forms of optional computation permissible within the constraints of these parameters are:

(i) *bounded optional with a 1/0 constraint (i.e. if started must complete):* minimum and maximum requested amounts of spare capacity are equal to the WCET of the optional computation, with the deadline set to be the time by which the optional must complete;

(ii) *unbounded optional*: minimum amount of spare capacity set to be the least amount of execution time required to produce a useful result, with the maximum set to be at least the mean execution time of the optional component;

(iii) *bounded optional*: minimum amount of spare capacity set to be the mean execution time of the optional computation, the maximum amount set to be the WCET. The deadline is set to be the time by which the optional must complete.

It is noted that in (ii) and (iii) above, the relationship between the minimum and maximum requested spare capacity values, and the maximum, mean and minimum execution times of the process, could be varied. Other possible optional computations include non-crucial processes.

Within this chapter a number of additional assumptions are made. Requests for spare capacity originate from running (or runnable) processes. These could be either application or kernel processes. When requests for spare capacity have been made, via a SCKC, a kernel resident spare capacity allocation policy (SCAP) determines which requesting processes are to receive spare capacity, and the amount of spare capacity they receive. Under normal execution, processes are assumed to have a static priority, with processes dispatched in a priority pre-emptive manner. To enable resource sharing under the auspices of one of the priority inheritance protocols (see Chapter 2), process priorities are permitted to be varied whilst executing part or all of a critical section.

To enable the construction of SCAPs a greater understanding of the exact characteristics of spare capacity needs to be developed. For example, there is little merit in allocating gain time if such an action leads to deadlines that have been guaranteed offline to be missed. To counter this problem a model of spare capacity is derived in this chapter, in terms of its scope and preservation. The former refers to the interval of time that any spare capacity

may be allocated to a process without causing subsequent deadlines to be missed. The latter discusses how, under certain circumstances, the scope of spare capacity may be extended in order to prolong its useful lifetime. Using this model, a set of parameters are defined to which any SCAP must conform so that 100% predictability of crucial process deadlines is not violated. This model is discussed in the following section.

In section 7.2 the conversion of slack time to gain time is discussed, the motivation being to enable more spare capacity to be guaranteed. Section 7.3 presents implementation strategies for spare capacity allocation policies. An evaluation of the main proposals of the chapter is given in section 7.4, with a summary of the chapter given in section 7.5.

7.1 Characteristics Of Spare Capacity

In this section, the characteristics of spare capacity are explored, in terms of a model which defines the properties of gain time, slack time and spare resources. The following section provides some initial observations. Subsequent sections define a formal model and derive a number of properties of that model.

7.1.1 Initial Observations

Gain time occurs when processes, at run-time, do not execute for their WCET. Conventionally, the spare capacity due to gain time becomes apparent when the process completes early. Figure 7.1 shows a single crucial process τ_1 with WCET C_1 and actual execution time A_1 for a release at t. At $t+A_1$ the process completes. Now it is apparent that C_1-A_1 gain time is available (indicated by the dotted box in the figure). This gain time can be assigned to any other process without affecting τ_1 since it has already completed and met its deadline. If the process assigned the gain time executes in the interval [$t+A_1$, $t+C_1$), no other process is affected (assuming that this process executes within the interval at the same priority level as the crucial process and uses no resources), with 100% predictability of crucial process deadlines maintained.

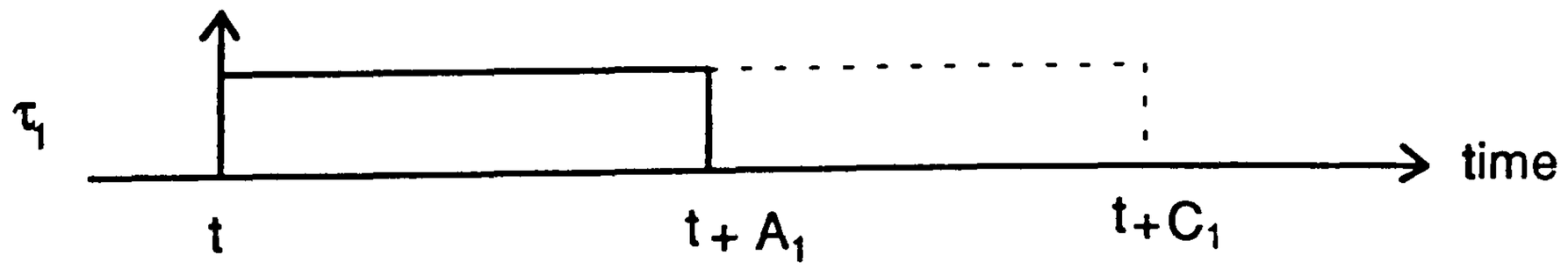


Figure 7.1: Assigning Gain Time Detected At Process Completion.

Using gain points, the gain time may be identified earlier. For example, τ_1 may have gained C_1-A_1 units after a single unit of execution time. This gain time can be utilised whenever detected without affecting the predictability of τ_1. For example, let the processor idle whenever gain time is detected, for an amount of time equal to the length of gain time detected: this cannot affect the deadline of τ_1. This is illustrated in Figure 7.2. Crucial process τ_1 detects t_b-t_a units of gain time at t_a which is consumed by idling the processor for the interval $[t_a, t_b)$. Again, guaranteed deadlines of other processes are not compromised.

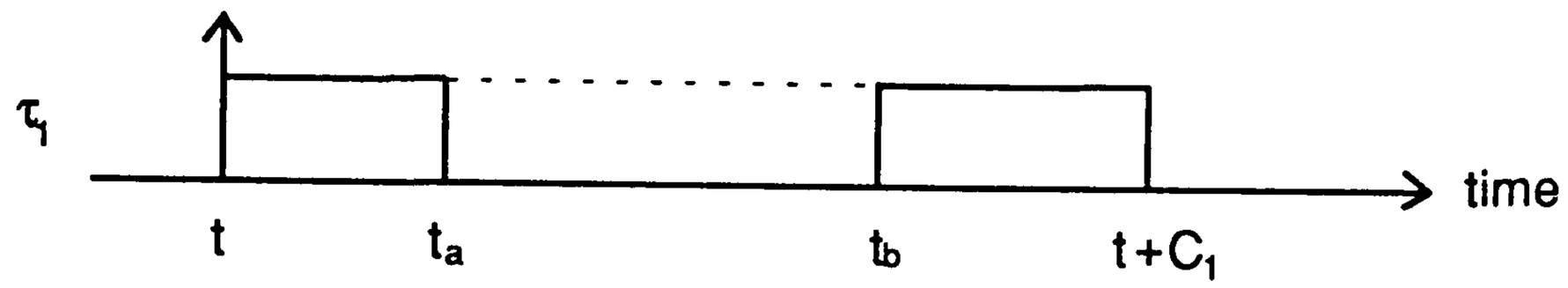


Figure 7.2: Assigning Gain Time Detected During Process Completion.

In both scenarios detailed above, slack time is identified at the completion of τ_1 (see section 6.2.6), up to the release of the next crucial process. When using gain time to idle the processor, at the point at which slack time is identified, all gain time has been utilised. However, when gain time is identified at the completion of a process execution, along with slack time, more spare capacity has been detected than is actually available For example, in Figure 7.1 at time $t+A_1$ there are C_1-A_1 units of gain time detected. If the next release of a crucial process is at $t'>t+C_1$, the $t'-(t+A_1)$ of slack time will be identified. The total spare capacity detected is:

$$t' - (t + A_1) + C_1 - A_1$$

The actual interval of spare capacity is $[t+A_1, t')$. Clearly, the length of this interval is less than the total amount of spare capacity identified, that is:

$$t' - (t + A_1) + C_1 - A_1 > t' - (t + A_1)$$

The implication of this observation is that one or both of gain time and slack time available must be reduced to compensate.

Slack time may be assigned to any process without affecting any crucial process deadlines, since no crucial processes are runnable when slack time is available. It is assumed that slack time is utilised at a priority lower than any crucial process.

Extending these observations, it is noted that gain time detected by τ_1 can be partitioned and assigned to more than one other process at any time after its detection without affecting the generating process. However, since the WCET of τ_1 (i.e. C_1) has been guaranteed by feasibility analysis, and gain time generated by τ_1 can only be used before the deadline of τ_1. Intuitively, this gain time must also be utilised at no more than the priority of τ_1.

The essential principle underpinning these observations is that spare capacity can be assigned to any process, in any quantity, at any time, without affecting the generating process, or any other crucial process, subject to certain rules of scope and assignment. The following sections define a model of spare capacity based upon these initial observations.

7.1.2 Kernel Level Model Of Spare Capacity

When gain time is detected by a gain point a GTKC is made (see Chapter 6) reporting the spare capacity to the kernel. At this level, spare capacity may be represented by a *gain-time tuple* of the form $<\tau_i, j, \delta, t, d>$ where τ_i is the executing process detecting the gain time; j the priority level at which τ_i is currently executing; δ the amount of gain time detected; t the time at which the gain time was identified and d the deadline of the current execution of τ_i (i.e. $d=t'+D_i$ where τ_i is released at t').

At the completion of crucial process τ_i, the kernel itself is responsible for detecting slack time (see section 6.2.6). Such slack time is represented by a *slack-time tuple* of the form $<\delta, t>$ where δ is the amount and t the start time of detected slack time. Typically, δ will be equal to $t'-t$, where t' represents the next release of a higher priority (crucial) periodic process after the completion of τ_i at t. The deadline of this spare capacity is given by $\delta+t$.

When spare resources are detected by a gain point (via a GTKC) a *resource tuple* of the form $<\tau_i, R, d>$ is formed, where R is the resource detected as spare by process τ_i. The deadline d is the next time at which any crucial process may next require the resource. Clearly, the calculation of such a value is problematic. Alternatively, the scope can be set to the next release time of τ_i, or, if such information is available to the kernel at run-time, the minimum execution time of τ_i before it may request R. This approach places emphasis upon the SCAP to consider the requirements of other crucial processes regarding R.

In the worst-case, a gain-time tuple is created for each gain point in a process. This could lead to a large number of gain-time tuples. However, if on the creation of a gain-time tuple another exists with the same process, priority level and deadline fields, the two tuples may be reduced to a single tuple. Hence, if $<\tau_i, j, \delta, t, d>$ exists when $<\tau_i, j, \delta', t', d>$ is created (i.e. $t'>t$) , then the two tuples are merged into a single tuple $<\tau_i, j, \delta+\delta', t', d>$. Since only one activation of a process is runnable at any one time, this limits the number of tuples to a maximum of n.

In the same manner, a tuple may be split into several parts, where the process, priority level, creation time and deadline are the same, with the amount of spare capacity in the original tuple equal to the sum of the spare capacity declared in the new tuples. This applies both to slack time and gain time tuples.

It is noted that if a process detects gain time whilst executing in a critical region at a priority level higher than normal, the created tuple could be merged with tuples representing gain time detected by the process whilst executing at its normal priority level, assuming that all the gain time had been detected at the normal priority level. This maintains the limit of n on the number of gain-time tuples. The priority level of the gain time detected in a critical region can be preserved by noting the behaviour of the run-time resource allocation protocol. Under the Priority Ceiling Protocol [Sha90] (see section 2.3.5), a process could execute at all higher priority levels whilst in a critical section. This limits the maximum number of tuples to i for a process whose normal priority level is i. Thus, $1+2+...+n$ gain time tuples could exist at any time (i.e. $(n^2+n)/2$).

On creation, gain-time, slack-time and resource tuples are placed into GTList, STList or RList respectively. These are lists, although the cost of insertion (i.e. during a GTKC) is minimised by placing new tuples at the head of the list. It then becomes the responsibility of any SCAP to order the respective lists if required (discussed further in section 7.3). At certain times, tuples are discarded from the lists (discussed further in section 7.1.3).

When gain time or slack time is assigned to requesting processes, a *pseudo-process* is created, with release time, computation time, deadline and priority level inherited from the characteristics of the spare capacity assigned to the requesting process. For example, if gain time $<\tau_i, j, \delta, t, d>$ is assigned to process τ_k at time t', the pseudo-process τ_k' is created with release time t', computation time δ, deadline d-t' (i.e. relative to t') and priority level j. If slack

time $<\delta, t>$ is assigned to τ_k at time t', the pseudo-process τ_k' is created with release time t', computation time δ, deadline $t+\delta-t'$ (i.e. relative to t') and priority level lower than all crucial processes. If more than one gain time or slack time tuple is assigned to a process, one pseudo-process is created for each tuple assigned, to reflect the possibility that the assigned tuples may have different priority levels, computation times or deadlines.

7.1.3 Preservation and Scope of Spare Capacity

Since gain time has been guaranteed at a particular priority level by offline feasibility analysis, it must be utilised in preference to the normal execution of a lower priority process. This is summarised by the following theorem:

Theorem 7.1:

If a gain time or slack time tuple exists with a higher priority level than the currently executing process (either crucial or non-crucial), that gain time (or slack time) must be utilised immediately. If more than one such tuple exists, they are utilised in descending order of priority.

Proof:

Consider gain time tuples. During feasibility analysis, processes are assumed to execute as soon as possible. Assume that gain time has been detected by τ_i, that process completing at t. Let process τ_k ($k > i$) execute. Now, if the gain time tuple created by τ_i is utilised at priority level i, the assumption inherent in feasibility analysis that processes execute as soon as possible has been broken: the interference on a lower priority process by τ_i may now be greater than assumed offline (possibly) causing a deadline to be missed. Therefore, if a gain time tuple exists which has a higher priority than all currently runnable processes, that gain time tuple must be utilised. If more than one gain time tuple exists of higher priority level than τ_k then the tuple with the highest priority level must be utilised since this is the priority level at which execution is assumed to occur by offline analysis.

A similar argument applies to slack time tuples assuming they have a priority lower than all crucial processes.

It is noted that a process of priority τ_k could execute in preference to gain time detected at priority level i ($k > i$) if the gain time was then not utilised. Also, if gain time tuples exist with equal priority levels, then an arbitrary choice amongst them can be made with respect to which is utilised if the priority level is greater than the running process.

The implication of Theorem 7.1 is that if no process is runnable whilst gain time or slack time tuples exist, that gain time (or slack time) must be used to idle the processor (highest priority gain time first). It could be argued that since slack time is terminated whenever a crucial process is released (i.e. crucial sporadic process) it is not necessary to utilise slack time to idle the processor. However, a SCAP may decide to execute an optional process using that slack time assuming that a given amount of slack time exists, when in practice it may not.

Theorem 7.1 appears to limit the usefulness of gain time since it needs to be utilised as soon as the detecting process completes. However, an approach for the preservation of gain time is now described.

One method for preserving gain time, the Extended Priority Exchange (EPE) method, has been presented by Sprunt *et al* [Sprunt88] to enable gain time to be used to permit non-crucial sporadic processes to execute. Under EPE, gain time is detected upon process completion (actual execution time is compared to worst-case execution time). If non-crucial processes are available to utilise that gain time then they are executed. If no such process is available, the highest priority runnable crucial process is executed, with the gain time preserved at this lower priority level. Eventually, if the gain time is not utilised, it is used to idle the processor.

However, EPE does not provide any guarantees regarding available gain time for requesting processes. Also, EPE does not incorporate slack time and spare resources. After formal proof of the applicability of the EPE, these restrictions are lifted. Consider the following theorems:

Theorem 7.2:

Let gain time given by $<\tau_i, j, \delta, t, d>$ be assigned to process τ_k (where τ_k may be crucial or non-crucial). If the gain time is not fully utilised by τ_k, it may be preserved up to the deadline of τ_i (i.e. d) at the priority level at which that process detected the gain time (i.e. j).

Proof:

Let τ_k utilise $\delta' < \delta$ units of gain time. This is equivalent to splitting $<\tau_i, j, \delta, t, d>$ into $<\tau_i, j, \delta', t, d>$ and $<\tau_i, j, \delta-\delta', t, d>$ with the former tuple assigned to τ_k. Effectively, the latter tuple has not been assigned and is thus preserved up to d.

Theorem 7.3:

Let gain time given by $<\tau_i, j, \delta, t, d>$ be assigned to crucial process τ_k executing at priority level l. Process τ_k utilises the gain time to execute already guaranteed computation. The original gain time may be preserved in the form $<\tau_k, l, \delta, t', t'+D_k>$ where t' is the release of τ_k.

Proof:

After τ_k has been assigned the gain time, it may use C_k units in $[t', t'+D_k)$ and an additional δ units in $[\max(t', t), \min(d, t'+D_k))$. Assume that τ_k utilises the gain time first. An extra δ units of gain time will become apparent (by use of an efficiency gain point at the completion of τ_k). This has been guaranteed for the interval $[t', t'+D_k)$ offline, implying that gain time tuple $<\tau_k, l, \delta, t', t'+D_k>$ has been created.

Theorem 7.3 proves the applicability of the EPE approach. It is noted that Theorems 7.2 and 7.3 must preserve gain time within the context of Theorem 7.1: the gain time must be utilised if it has been detected at a higher priority than any runnable crucial process.

The EPE approach may be extended to provide guarantees regarding gain time by noting the scope of gain time tuples. Assume that for crucial process τ_i, feasibility analysis has guaranteed C_i units of execution time in any interval $[t, t+D_i)$ where t is a release of τ_i. Let $<\tau_i, j, \delta, t, d>$ be due to a gain point within τ_i detecting δ units of gain time at time t'. Essentially, δ is guaranteed at priority level j until d. Consider the following theorem:

Theorem 7.4:

Gain time defined by tuple $<\tau_i, j, \delta, t, d>$ cannot cause crucial deadlines to be missed if it is used in $[t, d)$.

Proof:

Consider τ_i released at t with δ units of gain time detected at $t_1 > t$, that is tuple $<\tau_1, 1, \delta, t_1, t+D_1>$ is created. Two cases are identified:

(i) δ *units of gain time are assigned to* τ_i *to execute in* $[t_1, t+D_1)$:

Since the actual execution time of τ_1 is at least δ less than C_1, all δ time units have been guaranteed by offline feasibility analysis. The execution of τ_i at priority level 1 for δ units in $[t_1, t+D_1)$ cannot affect other crucial process deadlines.

(ii) δ *units of gain time are assigned to* τ_i *to execute in* $[t+D_1, \infty)$:

Let the interval $[t_1, t_2)$, where $t_2 > t_1 \geq t+D_1$, be required by a crucial process to meet its deadline at t_2. If τ_i executes for between $1..\delta$ time units at priority level 1 within the interval, the aforementioned crucial process will miss its deadline.

In consideration of Theorem 7.4, when the current time equals or exceeds the d field in any gain-time tuple it is removed from GTList, as its spare capacity has expired.

Slack time tuples cannot be preserved by Theorem 7.3 since if a crucial process is runnable which may utilise slack time, no slack time can exist by definition. Slack time tuples have similar scope to gain time tuples (Theorem 7.4): if their deadline expires they are removed from STList. One implication of this is that exactly zero or one tuples will exist in STList at any time. Consider a tuple placed into STList with deadline equal to the next time a crucial process is runnable. The tuple will run out of scope before another slack-time tuple can be created, noting that STList contains no tuples if a crucial process is runnable. Also, if a sporadic crucial process is released before the deadline of a slack-time tuple, that tuple is removed.

If both slack time and gain time exist at any time, both must be decreased concurrently by process execution or idling the processor is idle. For example, consider process τ_1 completing at time t having generated 1 unit of gain time. The deadline of τ_1 is $t+1$, with the next crucial process to be released at $t+1$. Hence, slack time tuple $<t, t+1>$ is created. Let the processor idle in the interval $[t, t+1)$. Clearly, neither the slack or gain time tuples can be preserved past the deadline $t+1$. Hence, both must be utilised (i.e. in parallel) whilst the processor is idle. The implication of this observation is that if the process idles

for 1 time unit, that amount of time must be subtracted from a gain time tuple and from a slack time tuple, whichever exists.

7.1.4 Assignment Of Gain Time And Slack Time

Initially gain time is considered. Any assignment of gain time to a process must ensure that the guaranteed deadlines of crucial processes are not compromised. In section 7.1.2, it was noted that on assignment of gain time to process τ_k, pseudo-process τ_k' is created to utilise the gain time. The computation time and deadline of τ_k' are given by the characteristics of the gain time tuple assigned: if τ_k is assigned gain time tuple $<\tau_i, j, \delta, t, d>$ at time t' then the release time, computation time and deadline of τ_k' are t', δ and d-t (i.e. deadline relative to start time) respectively.

The priority level of τ_k' is now defined. let the crucial processes that are runnable in $[t', d)$ (i.e. are runnable at t' or may become runnable in the interval) be denoted Δ_R where $\Delta_R \subseteq \Delta$. Let $\tau_i > \tau_j$ represent the fact that τ_i has a higher (base) priority level than τ_j. Consider the following theorem:

Theorem 7.5;

If gain time tuple given by $<\tau_i, j, \delta, t, d>$ is assigned to τ_k at time t', then pseudo-process τ_k' must execute at priority level j to ensure that no crucial process deadlines are compromised and that τ_k' executes for δ units in $[t, d)$.

Proof:

At time t' $\Delta_R=\{\tau_i, \tau_a, \tau_b, \tau_c, \tau_d\}$ with the priority ordering over Δ_R defined by $\tau_a>\tau_b>\tau_i>\tau_c>\tau_d$. Three cases are identified:

(i) *τ_k' is assigned a priority level higher than that of τ_b*:

Hence the priority ordering over Δ_R becomes either $\tau_k'>\tau_a>\tau_b>\tau_i>\tau_c>\tau_d$ or $\tau_a>\tau_k'>\tau_b>\tau_i>\tau_c>\tau_d$. In either priority ordering, the worst-case interference assumed on τ_b during offline feasibility analysis has increased by δ. This may cause τ_b to miss its deadline.

(ii) *τ_k' is assigned a priority level lower than that of τ_c*:

Hence the priority ordering over Δ_R becomes either $\tau_a>\tau_b>\tau_i>\tau_c>\tau_k'>\tau_d$ or $\tau_a>\tau_b>\tau_i>\tau_c>\tau_d>\tau_k'$. In either priority ordering, the execution of τ_k' occurs after τ_c (also after τ_d in the

second priority ordering). When the feasibility of τ_i was considered, the executions of τ_c (and τ_d) did not form part of any interference calculations. Therefore, if τ_k' executes at either of the above priority levels, it may not be able to execute for all δ time units in $[t, d)$.

(iii) τ_k' *is assigned to the same priority level as* τ_i:

Hence the priority ordering over Δ_R becomes either $\tau_a>\tau_b>\tau_i>\tau_k'>\tau_c>\tau_d$ or $\tau_a>\tau_b>\tau_k'>\tau_i>\tau_c>\tau_d$. In either priority ordering, τ_k' can execute for δ time units without affecting the deadlines of any other process (including τ_i) since the spare capacity was detected and guaranteed (by offline analysis) at this priority level.

The proof holds whether a high priority level process detects gain time which is then assigned to a lower priority level process or *vice versa.*

It is noted that if $<\tau_i, i, \delta, t, d>$ is assigned to τ_k then the execution of pseudo-process τ_k' could occur at a lower priority level than i without compromising crucial process deadlines, although τ_k' may not receive all δ units of computation time in $[t, d)$.

In the above theorem, there is no implied relationship between the deadline (d) of the process detecting the gain time (τ_i) and the deadline (d') of the process (τ_k) receiving that gain time. It is observed in the theorem that the pseudo-process τ_k' inherits the deadline of τ_i, i.e. d.

It is noted that if the priority level of τ_k' is higher than τ_k then the pseudo-process will complete execution before τ_k, else it will complete after τ_k. This has implications upon the use to which the gain time assigned to a requesting process can be put. Consider using the gain time for executing bounded additional code within a crucial process (i.e. sieve function). This code must either complete before the crucial process continues or its result will be ignored. Clearly, if the priority level of τ_k' is lower than τ_k the pseudo-process will not execute before τ_k has completed. Here, the additional computation could occur during the normal (guaranteed) execution of τ_k, with the allocated spare capacity used for the final part of the normal execution of τ_k.

The assignment of slack time to processes does not cause any problems. Since slack time is not guaranteed, any process utilising it has an effective

priority level below that of all crucial processes. Therefore, any crucial (sporadic) process becoming runnable whilst slack time is being utilised by a process will pre-empt the latter process, with no crucial deadlines compromised.

7.1.5 Scope And Assignment Of Spare Resources

Whilst the scope of gain time and slack time are the deadline of the detecting process and the next release of a periodic crucial process respectively, the scope of a spare resource is the time at which the detecting crucial process may next use that resource. Simplistically, this is assumed to be the next release of the detecting process (see section 7.1.2). It is noted that once the current time is at least the deadline field in any resource tuple, that tuple can be removed since the scope within which the spare resource can be assigned to a requesting process has passed.

In general, any assignment of a spare resource to a process must ensure that the latter process releases the resource at or before the scope of the resource expires. Where more than one crucial process access a given resource, all the processes must have declared that resource spare before it may be assigned to another (possibly non-crucial) process. It is noted that all the required resources of a crucial process have been detected as spare by the completion of that process (by resource gain points).

Any SCAP must ensure that assignment of a spare resource does not compromise assumptions made offline during feasibility analysis regarding resource availability at run-time. If offline analysis assumes that all resources required by a process are available at the release of that process, then if any of the resources are declared as spare, their scope cannot extend past the next release of the process. However, it is noted that the semantics of some run-time resource allocation policies may enable the scope on spare resources to be extended. If the PCP is used, then so long as the worst-case blocking times of crucial processes do not increase, the scope of spare resources may be extended to enable the process that has been allocated the spare resource to complete its critical region. For example, if the length of this critical region is not greater than any critical region that is responsible for the worst-case blocking times of higher priority processes, the scope of the spare resource may be extended to allow the completion of the critical region. It is assumed that the critical region is bounded. If it is not, the scope cannot be extended in this manner.

When a SCAP allocates a spare resource to a process to execute an unbounded critical section, provision must be made within the kernel to automatically release the resource at the end of the scope.

7.1.6 Summary

A model of detected spare capacity has been developed which has enabled a number of parameters to be established regarding the allocation of spare capacity to requesting processes. The priority level at which gain time and slack time can be utilised has been established, along with the interval within which such spare capacity can be used.

Spare resources were also considered. It was noted that the assignment of such resources to processes must be controlled according to the assumptions inherent in the run-time resource allocation policy and the associated offline feasibility analysis for that policy.

7.2 Conversion Of Slack Time To Gain Time

The difference between gain time and slack time is that the former is guaranteed, the latter is not. Whilst either form of spare capacity can be assigned to requesting processes, it is advantageous in some circumstances to be able to provide online guarantees regarding allocated spare capacity. For example, a crucial process requests spare capacity to execute optional computation which has a known minimum execution time, c, before a useful result is achieved. Whilst unguaranteed spare capacity could be assigned, it is preferable to assign at least c guaranteed spare capacity. In the following sections, methods for converting unguaranteed slack time into guaranteed gain time are examined.

7.2.1 Simple Conversion

Slack time occurs when no crucial processes are runnable, although outstanding requests for spare capacity may still exist. Intuitively, in a purely periodic system (with respect to crucial processes) slack time may be guaranteed from the completion of a crucial process (assuming no other crucial processes are runnable) up to the next release of a crucial process at a priority level lower than all crucial processes. Thus if τ_i completes at t with the next release of a crucial process at t', then $[t, t')$ is guaranteed to be available as spare capacity. However, at time t some gain time tuples may remain in scope,

guaranteeing execution time up to t^g, where $t^g > t$ (i.e. the sum of the amounts of gain time in available gain time tuples is given by $t^g - t$). Therefore, slack time in $[t, t')$ may only be guaranteed within the interval $[t^g, t')$ if $t \leq t^g \leq t'$ (assuming that if no gain time tuples are in scope at t then $t^g = t$). This implies that gain time tuple $<-, s, t'-t^g, t'-t^g, t'>$ can be created where s represents a priority level lower than all crucial processes.

When crucial sporadic processes exist in a system, converting slack time to gain time becomes more problematic. This is due to the possibility of the release of a sporadic crucial process in the interval $[t, t')$. Let the earliest release of a sporadic process at or after t be t''. Clearly if $t'' \geq t'$ then slack time can be converted to gain time in the manner outlined above for purely periodic systems. If $t^g < t'' < t'$, only that slack time in $[t^g, t'')$ maybe guaranteed. Thus, the slack time in $[t, t')$ is converted into gain time tuple $<-, s, t''- t^g, t^g, t''>$ with slack time tuple $<t'-t'', t''>$ also created. If $t^g \geq t''$, no slack time can be converted, with slack time tuple $<t'-t^g, t^g>$ created.

Whilst some slack time is converted into gain time by the above approach, it is noted that the intervals during which the gain time can be utilised are between the executions of crucial processes. Often requests for guaranteed spare capacity will be made by crucial processes requiring extra computation time before their deadlines (see section 7.1). By definition slack time and therefore slack time converted to gain time in the above manner will not be available in these intervals. The conversion methods given in the following sections attempt to circumvent this problem.

7.2.2 Conversion By Prediction

The method given in the previous section enables some slack time to be converted to gain time in the intervals when no crucial process is runnable. The weakness with this method becomes apparent when processes request spare capacity for extra computation before their own deadlines: no slack time exists for extra computation before their own deadlines and so no slack time maybe converted to gain time.

One approach to counteract this problem is to shuffle the computation of crucial processes so that they run later once a request for guaranteed spare capacity has been made. This has been proposed by Chetto *et al* in the context of earliest deadline scheduling and the servicing of non-crucial sporadic process requests [Chetto89]. Crucial processes are executed as early as possible until a non-periodic process requests computation. Then, crucial processes are

run as late as possible. This is achieved by plotting a schedule for all processes, noting the latest possible start time of each process whilst maintaining 100% guarantees of crucial process deadlines. Similarly, Lehoczky *et al* map out the execution of periodic crucial processes over a schedule of length equal to the least common multiple of all process periods [Lehoczky92]. The lengths of any intervals of slack time within the schedule are noted. Slack intervals are used to enable the computation of crucial processes to be postponed at run-time. The approach is costly: for each release of a process (in the least common multiple of all process periods) a value for the length of the slack interval needs to be held by the kernel at run-time. An online version of this approach is given by Davis *et al* [Davis93].

The above approaches are, in a sense, predictive: some knowledge of slack time that will become available is derived offline. Neither Chetto's nor Lehockzy's approaches cater for crucial sporadic processes, or processes that may block on access to share resources. Also, these approaches are relatively expensive, in time and space, having to construct schedules over potentially large intervals and hold large tables of values at run-time. Davis's approach is expensive in terms of the time required to calculate available spare capacity at run-time.

7.2.3 Conversion By Preservation

Another approach for conversion of slack time to gain time is to incorporate additional gain time into processes. That is, slack time is converted into gain time offline. This approach was formulated by Sprunt *et al* [Sprunt88] in terms of the Extended Priority Exchange (EPE) algorithm. If a process set is declared feasible, additional computation time is added to the highest priority process whilst ensuring the process set remains feasible. Then, additional computation time is added to the second highest priority process. Again, feasibility across the process set is maintained. This continues for all processes. At run-time, the extra computation is detected (along with any gain time due to the normal computation of the processes) at process completion time. Since the motivation behind the EPE algorithm is to decrease the response times of aperiodic (non-crucial) processes, detected spare capacity is utilised for the execution of such processes.

This approach is directly applicable for use with gain points. Let the amount of additional computation allocated to a crucial process τ_i (periodic or

sporadic) be A_i. The method of allocation has ensured that all crucial processes remain feasible, that is C_i+A_i (and B_i if appropriate) has been guaranteed before D_i (for all releases of τ_i in its feasibility interval). It is noted that the EPE approach detects the availability of A_i at the release of the process but detects gain time at the completion of the process. Thus, the use of a static gain point at the start of the process permits the detection of gain time A_i at the commencement of the execution of τ_i. Formally, a static gain point of value A_i is placed after the start node in a process's CFG (see Chapter 6).

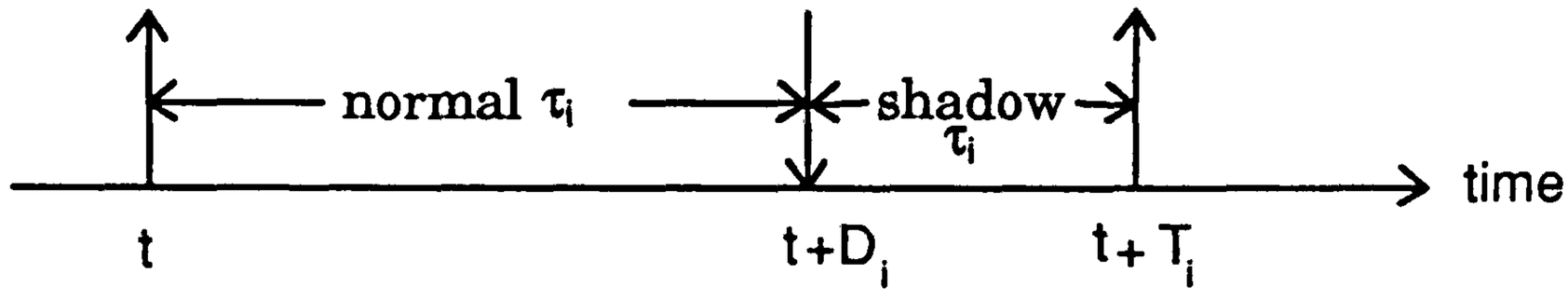


Figure 7.3: Conversion Of Slack Time To Gain Time Using A Shadow Process.

The approach can be extended by observing that for any process, the slack time inherent in the interval between the deadline of one execution of a process and the next release of the process is not converted. A *shadow process* is introduced to enable slack time in this interval to be converted to gain time. Consider Figure 7.3. The normal execution of τ_i at t detects A_i units of gain time at t. A shadow process τ_i^S is introduced with offset $O_i^S=O_i+D_i$, deadline $D_i^S=T_i- D_i$ and period $T_i^S=T_i$. The computation time of τ_i^S is 0, i.e. $C_i^S=0$. If the priority ordering of the process set is ..>τ_i>τ_j>.., the priority level of τ_i^S is set to be between τ_i and τ_j, i.e. ..>τ_i>τ_i^S>τ_j>...

Initially, τ_i^S cannot affect the feasibility of crucial processes (since $C_i^S=0$). Let additional computation be assigned to the shadow processes in the same manner as to crucial processes (in order $\tau_1^S,\tau_2^S,...$) whilst maintaining the feasibility of both crucial processes and the shadow processes. Thus, to each shadow process τ_i^S, A_i^S units of additional computation time have guaranteed (with $C_i^S=0$). This can be detected as soon as τ_i^S is executed at run-time, using a static gain-point.

Practically, the shadow processes are only required for offline feasibility analysis, with the kernel becoming responsible for creating a gain-time tuple of appropriate size at run-time at the deadline of a crucial process. For example, tuple <τ_i, i, A_i^S, $t+D_i$, $t+T_i$> is created at the deadline of the release of τ_i at t.

The approach is applicable for processes which block and those with precedence constraints. In the latter case, the size of shadow processes (in terms of allocated computation) is likely to be small, since the interval between the deadline of one crucial process execution and its next release is probably occupied by other processes in the precedence constraint.

Consider the following example.

Example 7.1:

Process	O	C	D	T	A	A^S
τ_1	0	3	12	16	3	0
τ_2	5	3	14	14	0	0
τ_3	12	4	20	40	0	1
τ_4	8	6	35	50	1	1

Table 7.1: Process Set 1.

Consider the process set in Table 7.1. The processes do not share a critical instant. Prior to additional computation allocation, the worst-case utilisation of the process set is 62.18%. The A column in the table indicates the amount of additional computation allocated to each process. The utilisation converted from slack to gain time by assigning additional computation to crucial processes is 20.75%. The A^S column indicates the amount of computation allocated to shadow processes. This enables an additional 4.5% of utilisation to be converted from slack to gain time. Overall 25.25% of utilisation has been converted. It is noted that at run-time additional slack time could be converted by the methods outlined in section 7.2.1.

The approach is expensive, in that many feasibility analyses need to be performed to determine A_i and A_i^S for each process. However, the number of such analyses can be reduced by observing that the amount of additional computation can be found by a binary search over $[0, D_i\text{-}C_i)$ for crucial processes, or over $[0, T_i\text{-}D_i)$ for shadow processes, rather than performing analysis for each member of the respective intervals. Also, if the exact interference feasibility tests of sections 4.3.2 and 5.4.2 are used, a bound can be found upon the maximum additional computation that could be assigned to a process (where the bound is a member of $[0, D_i\text{-}C_i)$ for crucial processes, or $[0, T_i\text{-}D_i)$ for shadow processes).

The allocation of additional computation to crucial processes (or shadow processes) is applicable for sporadic and periodic processes, those processes that block and those within a precedence constraint. It is noted that the additional computation allocated to a sporadic process will only become available when it is released (i.e. sporadically).

7.2.4 Summary

A number of methods for the conversion of slack time to guaranteed gain time have been described. The methods for conversion by prediction are expensive at run-time either in terms of space or time. The simple conversion and conversion by preservation methods were seen to fit in with the gain point model, catering for crucial periodic and sporadic processes, together with process blocking and precedence constraints. Although the preservation method is expensive offline, at run-time does not suffer the space expense of the prediction method.

7.3 Implementation Strategies For Spare Capacity Allocation Policies

Chapter 3 noted two trade-offs evident at run-time concerning the implementation of, and the complexity of, any spare capacity allocation policy (SCAP). Firstly, as the time between detection of spare capacity and its assignment increases, the system flexibility decreases, assuming that in general, early assignment of spare capacity leads to greater improvements in system utility. However, reducing the time between detection and assignment, in general, leads to the SCAP being executed more frequently. This increases the overheads incurred by crucial processes, so decreasing system feasibility.

The second trade-off concerns the actual policy employed to allocate spare capacity. In general, as the complexity of the SCAP increases, so does the ability to allocate spare capacity to the process which will have the greatest benefit to system utility. However, in general, as the policy complexity increases, so does the amount of time that policy takes to execute.

Conventionally, little attempt is made to re-use spare capacity. In terms of the trade-offs identified above, this approach incurs no overheads due to execution of the SCAP. However, no increase in system utility is achieved. The following sections discuss the above trade-offs in terms of some other potential implementation strategies for SCAPs.

7.3.1 Periodic Execution Of SCAP

The assignment of spare capacity to outstanding requests is performed by executing the SCAP as a periodic process. Potentially, the interval between the detection of spare capacity and its assignment can be large. If the SCAP is executed as the highest priority process, in the worst-case, the interval between detection and assignment becomes the period chosen for the SCAP process. If the SCAP is executed at a priority other than the highest, almost twice the period of the SCAP process could elapse between detection and assignment.

The advantage of this approach is that the overheads imposed on crucial processes are entirely controlled by the priority level, computation time and period selected for the SCAP process. For example, if the SCAP is implemented at the highest priority level, the impact of the SCAP on the feasibility of the process set is due to extra interference on crucial processes. It is noted that with the presence of arbitrary offsets, it may be possible to choose the start time and periodicity of the SCAP process so that interference increases are minimised (see section 5.8).

This approach becomes increasingly less appropriate as the interval within which the spare capacity is requested by a crucial process shortens: it becomes less likely that the SCAP will execute within that interval to allocate spare capacity. If the period of the SCAP process is reduced to counter this problem, the overheads on all processes increase, with the feasibility of the system deteriorating.

Another problem with this approach is that spare capacity is not assigned to requests until at least the next execution of the SCAP process. This is not appropriate for requests by crucial processes for spare capacity to execute optional computation immediately (i.e. sieve functions). The approach is more applicable for requests for spare capacity to execute optional computation between successive executions of a crucial process, or between members of a precedence constraint of crucial processes.

7.3.2 SCAP Execution On Spare Capacity Detection And Request

To minimise the interval between the detection and subsequent assignment of spare capacity, the SCAP can be implemented as part of detection, either within a GTKC or on kernel detection (e.g. from a shadow process). If

outstanding requests for spare capacity exist at the time of detection, spare capacity can be assigned. However, if no such requests exist, subsequent requests will not be allocated until the next detection of spare capacity occurs in the system. This problem may be circumvented by executing the SCAP as part of a spare capacity request also. Now, a minimal time is achieved between detection and assignment of spare capacity.

The overheads incurred by this approach are due to the execution time of a (possibly) complex and expensive SCAP on any detection or spare capacity request. Thus, the amount of computation that needs to be guaranteed for crucial processes increases. It is observed that in many cases the SCAP need not actually be executed: if there are no outstanding requests at spare capacity detection, or if there is no available spare capacity when a request is made. Clearly, the execution time saved under these circumstances could be detected as gain time.

Intuitively, spare capacity itself could be used to execute the SCAP. Now, unless sufficient spare capacity exists at a GTKC (or other spare capacity detection point) or at a spare capacity request, the SCAP will not be executed (assuming that outstanding requests exist). This spare capacity must exist at a priority level at least that of the detecting or requesting process, else the SCAP may not be executed immediately.

A significant problem exists with this approach. Consider two processes τ_i and τ_j whose priority levels are related by $\tau_i > \tau_j$. Process τ_j executes and requests spare capacity, with the SCAP able to commence execution at the priority level of τ_j. The action of the SCAP is to consult the outstanding requests and available spare capacity, and then allocate spare capacity to one or more of the requests. Process τ_i now becomes runnable, pre-empting the execution of the SCAP (and therefore τ_j) after it has consulted the list of available spare capacity. Now, τ_i requests (or detects) spare capacity, with a second SCAP becoming runnable. The running SCAP (i.e. invoked by τ_i) may assign gain time which at a later time may be assumed to be still available by the other execution of the SCAP (i.e. invoked by τ_j).

One solution to this problem is to place the execution of the SCAP within non-pre-emptable critical region. Intuitively, this requires that the execution of a SCAP be included in a crucial process's execution time for each detection of, or request for, spare capacity. Also the execution of the SCAP may form the longest critical region in the system, so increasing all crucial process worst-case

blocking time (assuming the use of one of the family of Priority Inheritance Protocols for run-time resource control [Sha90]). Clearly, this detracts from feasibility.

Alternatively, the approach that prescribes the execution of the SCAP in spare capacity and that which guarantees the execution of the SCAP within critical regions in crucial process code, may be merged. The only part of the detection that actually needs to be guaranteed (as part of crucial process execution) is that given in Chapter 6, that is the part of the GTKC which checks the amount of reported gain time and creates the gain time or resource tuple. Then, if sufficient gain time is available at an equal or higher priority level, the SCAP can be executed. It is noted that the calculation to determine if sufficient spare capacity is available to execute the SCAP is also guaranteed. Similarly for spare capacity requests, the guaranteed computation consists of recording the request. Then, if sufficient spare capacity exists, the SCAP is executed. In both cases the SCAP is executed in spare capacity.

To alleviate the problem of a SCAP being executed whilst another execution of the SCAP has been pre-empted, the SCAP is placed into a non-pre-emptable critical region. Now, the only penalty to process set feasibility is the possibility that the SCAP forms the longest critical region in the system, so increasing the worst-case blocking time of crucial process. The additional penalty of having to guarantee (possibly many) executions of the SCAP for each crucial process has been removed.

Potentially, this approach minimises the interval between spare capacity detection and assignment, whilst ensuring that the impact of SCAP execution upon process feasibility is also minimised. This is in contrast to the approaches of Moron *et al* [Moron93] and Haban *et al* [Haban90] where the policy to allocate spare capacity executes as part of a context switch, increasing the overheads on all crucial processes and detracting from system feasibility. We note that the cost of the SCAP itself detracts from the ability to execute the SCAP since more spare capacity is required.

7.4 Evaluation

In this section, the approach for the allocation of spare capacity given in this chapter is evaluated. This is achieved by comparing its performance with other proposed approaches in two main ways:

(i) guaranteed spare capacity that can be assigned to requesting processes;

(ii) unguaranteed spare capacity assigned to requesting processes.

In both (i) and (ii), the approach given in this chapter (referred to as the SCAP approach in this section) is compared with the Extended Priority Exchange (EPE) approach [Sprunt88]. The EPE approach is assumed to be able to guarantee requesting processes gain time within the constraints defined by Theorems 7.4 and 7.5. Also, additional computation time is detected at the start of a process with gain time detected at the completion of the process. The SCAP approach assumes that gain time is detected via gain points (see Chapter 6). Also additional computation is detected at the release of a process, with each crucial process assigned a shadow processes detecting gain time at the deadline of the associated crucial process.

The allocation policies assumed in both approaches assign spare capacity in a first-come-first-served manner on spare capacity detection or request. This enables overheads to be assumed approximately equivalent between the two approaches, implying that they can be ignored during this comparison.

Process	O	C	D	T	A	A^S
τ_1	5	7	10	40	3	3
τ_2	44	12	27	60	2	0
τ_3	50	27	100	120	3	0
τ_4	124	9	110	180	0	0
τ_5	180	17	230	230	1	0

Table 7.2: Process Set 2.

Initially, crucial process timing characteristics were chosen (given in Table 7.2). These characteristics are similar to those chosen (randomly) in section 6.4.3 (given in Table 6.1) although decreased by an order of magnitude. The basic utilisation of the process set is 72.39%. An additional 21.27% of utilisation is converted to gain time by assigning additional computation time to the crucial processes (column A) and their shadow processes (column A^S).

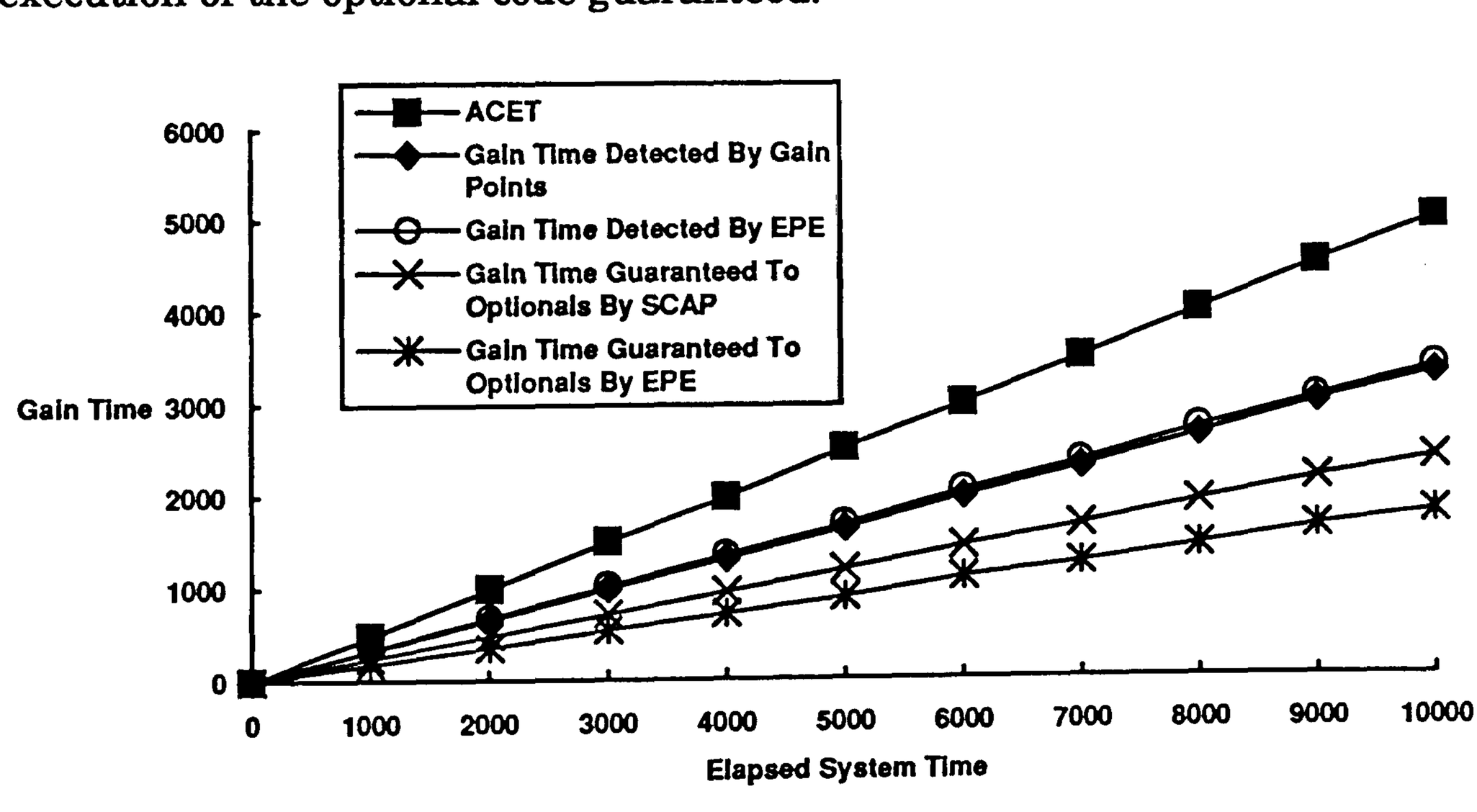
To each process, an arbitrary control-flow structure was allocated, incorporating a mixture of conditional and loop statements (similar to the processes in Appendix B). Within each process's code, a number of calls for guaranteed spare capacity are made to enable execution of optional code before a process's deadline. Also, at the end of each process's execution, spare capacity is requested to execute optional code before the next release of the process.

Given that worst-case utilisation is 72.39%, the total amount of execution requested by a crucial process to execute optional code was on average about 50% of their worst-case computation times, that is about 35% of overall utilisation. Thus, in the worst-case, the load on the processor is over 100% (i.e. 72% + 35%).

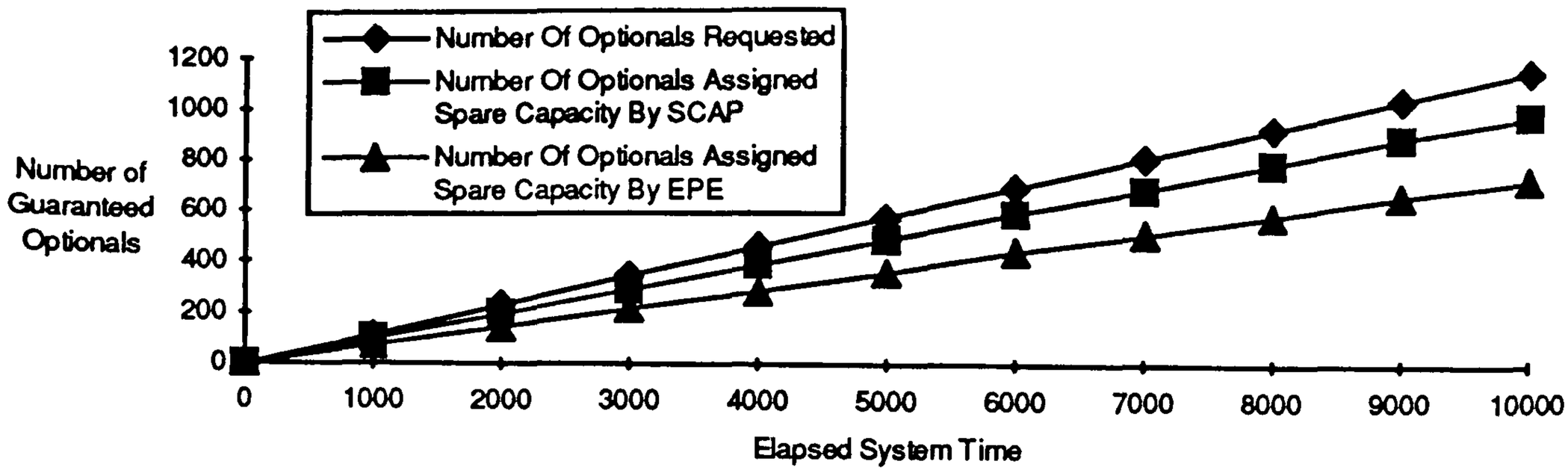
On each release of the process, a random path was taken through each process's control-flow structure. Thus, the points at which gain time could be detected was varied, along with the amounts of gain time generated by a process and the times at which guaranteed spare capacity was requested. For a release of a process at time t, the path taken when considering the SCAP approach was the same as the release of the process at time t when considering EPE. Thus, available gain time and requests for spare capacity were the same for the two approaches all times.

7.4.1 Guaranteed Optional Performance

The process set was executed across the interval [0, 10000) five times (a total of over 7500 process releases). Both requests within process code and the request at the end of process code were for guaranteed spare capacity. If sufficient spare capacity was available before the given deadline, it is assigned, with the execution of the optional code guaranteed.



Graph 7.1: Comparing Amounts of Guaranteed Spare Capacity Assigned.



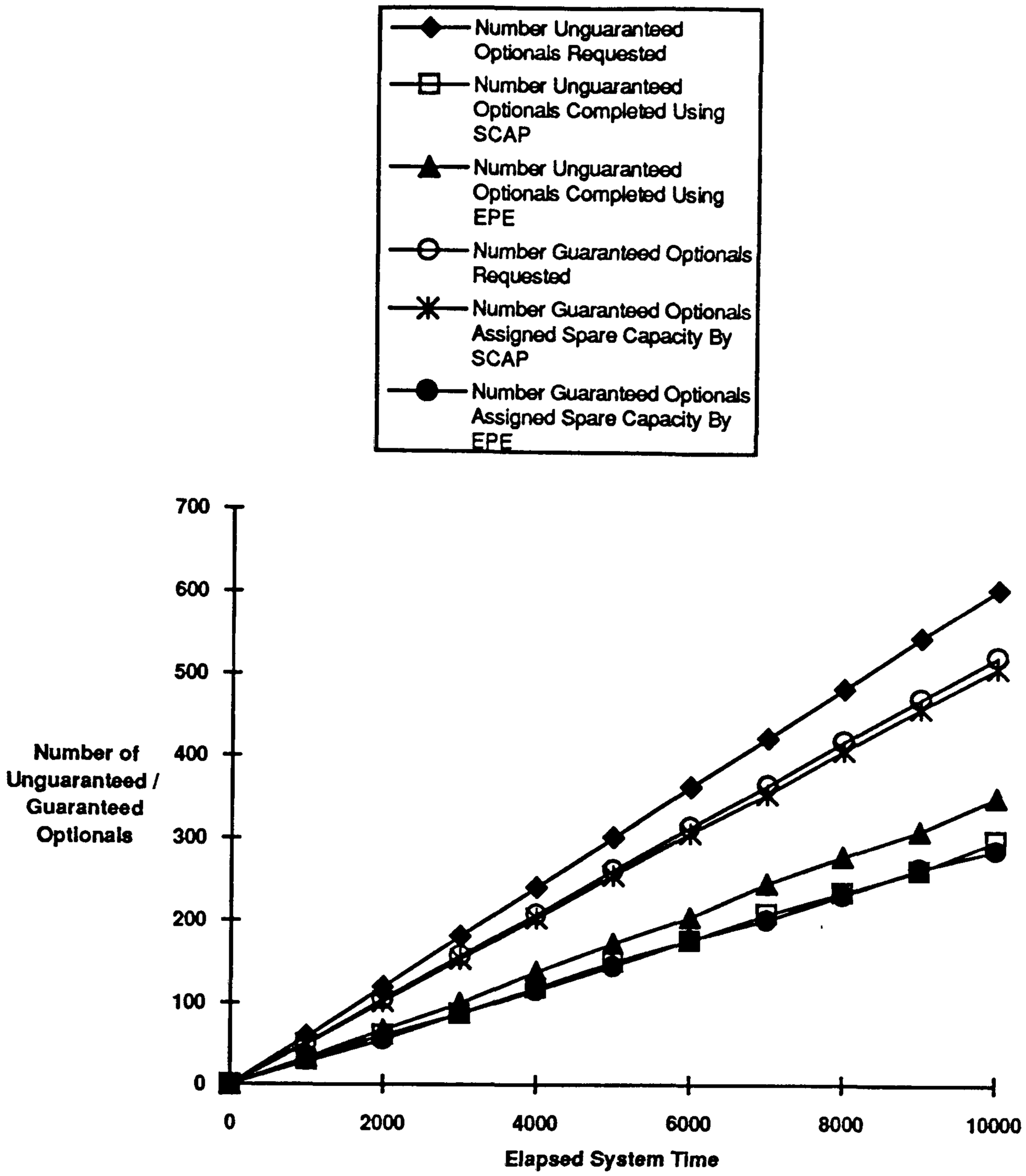
Graph 7.2: Comparing the Number of Optionals Assigned Guaranteed Spare Capacity.

The results obtained are shown in Graphs 7.1 and 7.2. In Graph 7.1, the amount of guaranteed spare capacity assigned by EPE and SCAP to requests is plotted against elapsed time (from system epoch). For reference, the total amount of actual execution time of the crucial processes (i.e. not the optional executions) is also plotted against elapsed time, along with the amount of gain time detected. The latter would indicated that EPE is close in performance to gain points for detecting gain time, although consideration of the scale of the axes indicates the superiority of gain points (a better evaluation is given in section 6.4.3). The main conclusion that is drawn from Graph 7.1 is that the amount of guaranteed spare capacity assigned by the SCAP approach is greater than that assigned by EPE, by on average 30%.

In Graph 7.2, the number of individual requests for spare capacity are plotted against elapsed time. Also, the number of requests guaranteed to complete by the two approaches are plotted against time. The number of requests which are assigned guaranteed spare capacity and complete is higher using the SCAP approach compared with EPE. On average, the former approach assigns spare capacity to 30% more requests than the latter.

7.4.2 Unguaranteed Optional Performance

Again, the process set was executed five times over the interval [0, 10000). Requests within process code were assigned guaranteed spare capacity. Requests for spare capacity at the end of a crucial process's execution, to execute optional code prior to the next release of the crucial process, were not guaranteed. It is noted that in the experiment, when a crucial process starts the execution of an optional unguaranteed computation, if that computation has not completed by the next release of the crucial process, that computation is terminated and assumed incomplete.



Graph 7.3: Comparing the Number of Optionals Assigned Unguaranteed Spare Capacity.

The results obtained are shown in Graph 7.3. For both EPE and the SCAP approach, the number of unguaranteed optional executions started and the number of those that actually completed, are plotted against elapsed execution time. Also, the number of requests for guaranteed spare capacity, and the number of those requests actually assigned spare capacity are plotted against time.

The graph shows that the number of unguaranteed computations completed by EPE is marginally greater than the SCAP approach. This is due to the latter approach being able to utilise more of the spare capacity for

guaranteed computation. This is shown by the disparity between the number of guaranteed requests met by the two approaches.

7.4.3 Increasing System Utility

It has been noted in this chapter that, in general, as the complexity of a SCAP increases, so does its potential for allocating spare capacity in a manner which has greatest benefit to system utility. Disregarding the null policy, the simplest SCAP would appear to be "first-come-first-served". Here, a request for an amount of spare capacity before a deadline receives spare capacity if sufficient has been detected with appropriate scope. Whilst incurring minimal overheads, such a policy is not ideal: spare capacity could be assigned to execute optional code with little benefit to system merely because no other requests are outstanding, even if a request which would benefit system utility greatly were just about to be made.

Another approach that would appear practical for use in a SCAP is that of *time-value* scheduling proposed by Locke [Locke86]. This approach enables the scheduling of processes which have a pre-defined time-value relationship: the notional value for completing a process is known for all possible completion times (i.e. between the release time and deadline of a process). The maximum overall value from the system if the process with the highest *value density* is scheduled at each point in time. The value density of a process is defined as V/C_{rem} where V is the value obtained for completing the process and C_{rem} its remaining execution time.

Clearly, if the relative values or utility of optional components could be obtained, the time-value approach would appear applicable for assigning spare capacity at run-time.

More complex decision algorithms could be used within a SCAP, for example those that consider the possible future requests for spare capacity by other crucial processes. However, the development and discussion of such algorithms is beyond the scope of this thesis.

7.4.4 Summary

This section has evaluated the model for spare capacity in terms of preservation, assignment and conversion (see sections 7.1-7.3) by comparison with the Extended Priority Exchange algorithm. In general, the amount of spare capacity guaranteed to requesting processes is higher using the former

approach: a larger amount of guaranteed spare capacity is assigned to a greater number of requesting processes. When some optional computations do not require guaranteed spare capacity, the approach defined in this chapter was slightly less effective at completing those computations compared with the EPE approach. However, this is directly attributable to the former approach guaranteeing more spare capacity, and utilising that spare capacity for executing optional components requiring guarantees.

Overheads were not specifically considered in the evaluation in this section since the overheads of EPE and that of the approach of this chapter (assigning spare capacity using an inexpensive first-come-first-served policy) are broadly equivalent.

7.5 Summary

In Chapter 6, the efficient and early detection of spare capacity was considered. In this chapter, the issue of allocation of detected spare capacity was considered. Initially, a model of spare capacity was developed as gain time tuples, slack time tuples and resource tuples. Each tuple described the amount of spare capacity detected, the detecting process and the interval of time within which that spare capacity could be utilised.

Several properties of the tuple model were derived, including the preservation and assignment of tuples. The former discussed the circumstances under which the scope of a tuple could be extended. This is important: if at a given time some detected spare capacity has not yet been assigned, extending its useful lifetime increases the chance that the spare capacity may be assigned in the future. Assignment of spare capacity to requesting processes was seen to be constrained by the scope of the tuple, which effectively provides the deadline by which that spare capacity must be utilised, and its priority level (in the case of gain time tuples). Essentially, the model describes a number of criteria that any spare capacity allocation policy must meet so that deadlines of crucial processes are not compromised.

Conversion of unguaranteed slack time to guaranteed gain time was considered. Simplistic conversion enables slack time between executions of crucial processes to be converted to gain time whilst no crucial sporadic process can execute. This approach does not provide extra gain time during the execution of crucial processes. This problem was surmounted by guaranteeing additional computation time to crucial processes offline, available at run-time as gain time on process release.

The implementation of spare capacity allocation policies was also considered. One approach is that of a periodic process executing the allocation policy. This approach does not cope particularly well when requests for spare capacity are made to enable the immediate execution of an optional component. Alternatively, the allocation policy can be executed using spare capacity, whenever spare capacity is detected or requested, providing sufficient spare capacity is available to execute the policy. This approach toward spare capacity allocation was evaluated by comparison with the Extended Priority Exchange approach, which also detects and assigns guaranteed spare capacity. The approach detailed in this chapter was shown to allocate more guaranteed spare capacity to a greater number of requests than achieved by the Extended Priority Exchange algorithm.

The approach toward spare capacity allocation detailed in this chapter enables the online flexibility of hard real-time systems to be improved. Optional software components can be assigned guaranteed computation time, enabling the utility of the system to be increased at run-time. The approach has a minimal effect upon crucial process feasibility, since the computationally expensive allocation policies are executed using spare capacity.

Chapter 8.
Conclusions And Further Work

In general, the requirements of the next generation of hard real-time systems are not catered for by present scheduling approaches and associated feasibility analyses. In particular, support for long lifetime, adaptive and dynamic systems is not provided. To overcome this problem, this thesis has proposed a two-tiered approach to increase the flexibility of hard real-time systems: improvement of offline feasibility and re-use of online spare capacity.

The process model assumed by current scheduling methods is constrained, both in the timing and functional domains: timing and functional characteristics of processes are limited. Such constraints limit the offline flexibility, in that application engineers are not provided with the richness of expression required to enable efficient development of next generation hard real-time systems. Solutions to a number of problems set by such systems have been articulated within this thesis. This has been achieved by extensions to static priority offline feasibility analysis. Primarily, a method was provided for optimal priority assignment of processes with arbitrary release times and deadlines no greater than their periods. Feasibility analyses were provided for such processes (both for processes with and those without common release times). These analyses have been extended to enable the incorporation of sporadic processes, processes that block on shared resources, and those processes which have precedence constraints defined over them. The developments presented in this thesis improve offline flexibility, since analysis of a process model suitable for next generation hard real-time systems is now possible.

A trade-off between the flexibility and complexity of offline analysis was observed. In general, as accuracy of analysis and generality of the process model improves, the complexity of resultant feasibility increases. This was highlighted when sufficient and necessary analyses were compared with sufficient and not necessary analyses for the same process model. The former analyses have in general, non-polynomial complexity whilst the latter have polynomial complexity.

The dynamic and adaptive nature of next generation hard real-time systems is not reflected in current online scheduling. Static priority scheduling assumes a fixed set of processes with bounded execution times.

The assumption is directly attributable to offline feasibility analysis. Such analysis cannot guarantee the deadlines of unbounded (in terms of execution time) software components. Often such components are required to express much of the required dynamic and adaptive behaviour (e.g. AI). Hence, the ability to execute these (optional) components at run-time improves the flexibility of the resultant system. Within this thesis an approach enabling the execution of such components was developed, based upon the observation that the use of offline analysis to guarantee crucial process deadlines implies that, at run-time, resources are under-utilised. The identification of such spare system capacity permits its re-assignment to processes so that additional (optional) computations may be executed.

The gain point mechanism has been developed to identify much of this spare capacity in an efficient manner. By examining the control-flow graph representation of a process's code, gain points are inserted to detect gain time, a guaranteed form of spare capacity, as early as possible.

Having detected spare capacity, the key issue becomes that of effective assignment so that system flexibility and utility can be improved. Initially, a model of spare capacity was developed. This highlighted several important considerations, namely scope and preservation. Guaranteed spare capacity, or gain-time, is detected due to crucial (guaranteed) processes not executing for the worst-case execution time. Such spare capacity has a scope by which time it must be used. However, it was found that under certain conditions, scope could be extended to prolong its lifetime and therefore preserve the usefulness of that scope.

Another important issue is that of the amount of spare capacity available at run-time that can be guaranteed to requesting processes. The presence of this form of spare capacity at run-time enables guarantees to be afforded to optional bounded components or a minimum execution time to be guaranteed to unbounded optional components. These observations motivated the identification of methods which enabled the conversion of unguaranteed spare capacity, to the more useful guaranteed form.

Finally, allocation policies for spare capacity were considered, in particular, their implementation. It was shown that potentially complex and expensive policies, with a large potential overhead on crucial process feasibility, could actually be executed using spare capacity itself.

An online trade-off was observed between overheads incurred during spare capacity detection and allocation, and the benefits to system flexibility. In general, as detection accuracy increases, so do the overheads involved. Also, as the complexity of allocation policy increases, so does the possibility of assigning spare capacity to the requesting process which will

benefit the system most. However, increasing allocation policy complexity implies additional overheads. The gain point approach for detecting spare capacity was seen to occur minimal overheads: a single kernel call. Also, allocation policies can be executed using spare capacity. Therefore, the potential benefits from spare capacity by early efficient detection can be maximised, whilst minimising the effects on crucial process feasibility of detection and allocation.

In essence, the two-tiered approach proposed in this thesis provides a framework within which increased flexibility for next generation hard real-time systems can be achieved. Offline flexibility has been improved by relaxing the common constraints upon the process model, whilst online flexibility is increased by efficient dynamic detection and assignment of spare system resources.

8.1 Further Work

Whilst presenting relatively self-contained research, this thesis provides a framework for further work in the area of providing additional flexibility for hard real-time systems. Such additional work falls into two broad areas, equating to offline and online analysis.

Offline feasibility analysis will always require further development, in terms of accuracy (i.e. reducing pessimism) and coverage: responding to demands from application engineers for ever more flexible process models. In particular, the feasibility analysis of processes using fault-tolerant language structures, or replicated processes is an immediate requirement, together with support for distributed systems (including the additional problem of inter-node communications). Also, feasibility analysis is required for specific hardware and architectural features, for example I/O scheduling (e.g. disks and other multi-media devices), on and off processor caches, processor pipelines etc.

The online mechanisms by which run-time spare capacity can be detected efficiently need to be integrated into hard real-time kernels and programming languages. The criteria described by which spare capacity allocation policies can be built without compromising crucial process deadlines or impairing the feasibility of a process set, provides a framework within which, potentially, a wide range of spare capacity allocation policies can be developed. These could be based on current online scheduling strategies which consider process value, or perhaps using AI techniques. Together with the implementation of the spare capacity detection mechanisms, the implementation of spare capacity allocation policies within

actual systems would lead to a greater understanding of the potential benefits of spare capacity re-use at run-time.

8.2 In Conclusion

This thesis set out to examine the hypothesis that the flexibility of hard real-time systems could be improved in two main ways. Firstly, by choice of scheduling approach. Secondly, by detecting the inherent under-utilisation of system resources at run-time. In both cases, the hypothesis has been shown to be correct. Offline feasibility analysis has been provided for a more flexible process model. Efficient online detection of inherent spare system capacity is given, enabling subsequent assignment of (possibly guaranteed) additional execution time to processes to perform optional computations.

Appendix A.
Generation of Random Process Sets

The generation of random numbers with a uniform distribution was achieved using the linear congruential method [Knuth68]. The generation of random numbers with a normal distribution was achieved using the Box-Muller method [Knuth68]. Assume two independent random variables U_1 and U_2 uniformly distributed upon (0, 1) (U_1 and U_2 provided by the uniform random number generator). The Box-Muller method uses U_1 and U_2 to provide two values X and Y thus:

$$X = \left(-2\left(\ln U_1\right)\right)^{1/2} \cos(2\pi U_2)$$

$$Y = \left(-2\left(\ln U_1\right)\right)^{1/2} \sin(2\pi U_2)$$

By taking pairs of U_1 and U_2 we produce a sequence of values that are normally distributed. The distribution is standard, that is $\mu = 0$ and $\sigma = 1$ where μ and σ are the mean and standard deviation respectively. A value x from a non-standard distribution by:

$$x = \mu' + Z\sigma'$$

where x is a value from the normal distribution with mean μ' and standard deviation σ', and Z is a random value from a standard normal distribution (i.e. X or Y above).

Using one of the random distributions above (uniform, standard normal or non-standard normal), values for process timing characteristics can be chosen. For example, if a value in the range [a, b] is required from a normal distribution, then $\mu = (x + y)/2$. The standard deviation is now set to $\sigma = (b - a)/4$. This implies that 95.45% of values fall within the interval (since it is 4 standard deviations wide).

When process timing characteristics are unrelated, that is the characteristics of one process do not depend upon the characteristics of another process, their value is chosen from the interval [*minimum, maximum*] (*minimum* and *maximum* are parameters to the random process set generator). When process timing characteristics are related, specifically the period of a process is a function (f) of another process period (T), the value is chosen to be in the interval [$f(T)$, *maximum*]. Functions used include those that ensured that some periods were equal; some periods were multiples of others. This reflects the observation that processes often have periods related due to constraints of hardware or precedence-constraints.

Appendix B.
Processes Used In Gain Time Detection Evaluation

The following psuedo-code describes the processes used during the evalutation of gain-time detection strategies (see section 6.4.3).

Gain time is reported using via the call `gainpoint(priority, v)` where `priority` is the priority level of the process, `v` the amount of gain time detected. Efficiency gain time is detected within the call (effectively by the kernel).

Spare capacity detected by Dix's and Haban's approach is reported via a `dhpoint(w)` call, where the amount detected is calculated by the call by comparing the actual execution time up to this point with the worst-case execution time up to this point, passed as parameter `w`.

The `execute(priority_1, x, y)` call provides the actual execution of the process, that is execute at priority level `priority_1` for between `x` and `y` time units.

```
process process_1 () is
begin
execute (priority_1, 1) ;
if (rand() %3 = 1) then
    gainpoint (priority_1, 26)  ;
    execute (priority_1, 36, 40)    ;
else
    if (rand() %2 = 1)
    then    execute(priority_1, 14, 16);
    else    gainpoint (priority_1, 11)  ;
            execute (priority_1, 4, 5)  ;
    end if ;
    i = rand() % 5  ;
    gainpoint (priority_1, 10*i)    ;
    while (i < 5) loop
        execute(priority_1, 9, 10)  ;
        i++ ;
        if (rand() % 4 = 1 and i < 5)
        then    exit    ;
        end if  ;
    end loop    ;
    gainpoint (priority_1, 10 * (5 - i))    ;
end if  ;
dhpoint (67)    ;
execute (priority_1, 9, 10) ;
dhpoint (77)    ;
gainpoint (priority_1, 0)   ;
end process_1   ;
```

```
process process_2 () is
begin
    execute (priority_2, 17, 18)    ;
    dhpoint (18)    ;
    i = rand() % 3  ;
    gainpoint (priority_2, 31 * i)  ;
    while (i < 5) loop
        execute (priority_2, 12, 14)    ;
        if (rand()%2)
        then    execute (priority_2, 15, 17)    ;
        else    gainpoint (priority_2, 17)  ;
        end if  ;
        i++ ;
    end loop    ;
    dhpoint (173)   ;
    gainpoint (priority_2, (5-i)*31)    ;
end process_2   ;
```

```
process process_3 () is
begin
    execute (priority_3, 10, 11)    ;
    dhpoint (11)    ;
    i = rand() % 3  ;
    gainpoint (priority_3, 76 * i)  ;
    while (i < 5) loop
        execute (priority_3, 1, 1)  ;
        j = rand() % 10 ;
        gainpoint (priority_3, j * 15)  ;
        gt += j * 15    ;
        while (j < 15) loop
            execute (priority_3, 4, 5)  ;
            j++ ;
            if (rand() %2 = 1) then
                gainpoint (priority_3, 5 * (15 - j)  ;
                exit    ;
            end if  ;
        end loop    ;
        i++ ;
        if (rand() %2 = 1 and i < 5) then
            gainpoint (priority_3, 16 * (5 - i))    ;
            exit    ;
        end if  ;
    end loop    ;
    dhpoint (391)   ;
    execute (priority_3, 2, 2)  ;
    dhpoint (393)   ;
    gainpoint (priority_3, 0)   ;
end process_3   ;
```

```
process process_4 () is
begin
    execute (priority_4, 5, 6)  ;
    dhpoint (6) ;
    if  (rand()%4 = 0) then
        gainpoint(priority_4, 27)   ;
        execute (priority_4, 41, 43)    ;
    else if (rand() %4 = 0) then
        execute(priority_4, 66, 70) ;
    else if (rand() %4 == 0) then
        gainpoint (priority_4, 14)  ;
        execute (priority_4, 50, 56)    ;
    else    gainpoint (priority_4, 57)  ;
        execute (priority_4, 12, 13)    ;
    end if  ;
    dhpoint (76)    ;
    execute (priority_4, 12, 13)    ;
    dhpoint (89)    ;
    gainpoint (priority_4, 0)   ;
end process_4   ;

process process_5 () is
begin
    if (rand()%4 = 0) then
        execute (priority_5, 28, 31)    ;
    else    gainpoint(priority_5, 19)   ;
        execute (priority_5, 11, 12)    ;
    end if  ;
    dhpoint (31)    ;
    execute (priority_5, 48, 53)    ;
    dhpoint (84)    ;
    if (rand() % 2 = 1) then
        gainpoint (priority_5, 70)  ;
        execute (priority_5, 32, 34)    ;
    else if (rand() % 2 = 1) then
        gainpoint (priority_5, 56,) ;
        execute (priority_5, 44, 48)    ;
    else    execute (priority_5, 96, 104)   ;
    end if  ;
    dhpoint (188 )  ;
    execute (priority_5, 40, 44)    ;
    dhpoint (232)   ;
    gainpoint (priority_5, 0)   ;
end process_5   ;
```

Bibliography

[Ada83] U.S. Department of Defense, "*Reference Manual for the Ada Programming Language*", ANSI/MIL-STD 1815 A, 1983.

[AEEC91] AEEC, "*Design Guidance for Integrated Mondular Avionics,*" ARINC 651 (Draft 9) (September 1991).

[Aho86] Aho, A. V., R. Sethi, J. D. Ullman, *"Compilers Principles, Techniques and Tools"*, Addison-Wesley, 1986.

[Audsley90] Audsley, N. C., A. Burns, "*Scheduling Real-Time Systems,*" YCS 134, Department of Computer Science, University of York (1990).

[Audsley91a] Audsley, N. C., "*Resource Control For Hard Real-Time Systems: A Review,*" YCS 159, Department of Computer Science, University of York (August 1991).

[Audsley91b] Audsley, N. C., K. Tindell, A. Burns, M. F. Richardson and A. J. Wellings, "*The DrTee Architecture for Distributed Hard Real-Time Systems,*" Proceedings 10th IFAC Workshop on Distributed Control Systems (9-11 September 1991).

[Audsley91c] Audsley, N. C., A. Burns, M. F. Richardson, A. J. Wellings, *"Hard Real-Time Scheduling: The Deadline Monotonic Approach"*, Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta USA, 15-17 May 1991.

[Audsley91d] Audsley, N. C., *"Deadline Monotonic Scheduling"*, YCS 146, Department of Computer Science, University of York 1991.

[Audsley93a] Audsley, N. C., A. Burns, M. F. Richardson and A. J. Wellings, "*Incorporating Unbounded Algorithms Into Predictable Real-Time Systems,*" Computer Systems Science and Engineering, 8(3), pp.80-89, (1993).

[Audsley93b] Audsley, N. C., K. Tindell, A. Burns, *"The End of the Line for Static Cyclic Scheduling"*, pp.36-41, Proceedings of the 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland, June 1993.

[Babaoglu90] Babaoglu, O., K. Marzullo, F. B. Schneider, *"Priority Inversion and its Aviodance in Real-Time Systems"*, TR-90-1088, Department of Computer Science, Cornell University, (1990).

[Babaoglu93] Babaoglu, O., K. Marzullo, F. B. Schneider, *"A Formalisation of Priority Inversion"*, The Journal of Real-Time Systems, 5(3), pp 285-304, (October 1993).

[Baker90] Baker, T. P., *"A Stack-Based Resource Allocation Policy for Realtime Processes,"* Proceedings 11th IEEE Real-Time Systems Symposium (5-7 December 1990) pp.191-200.

[Burns89] Burns, A., A. J. Wellings, *"Real-Time Systems and Their Programming Languages"*, Addison-Wesley, 1989.

[Burns91a] Burns, A., *"Scheduling Hard Real-Time Systems: A Review,"* Software Engineering Journal 6(3) (1991) pp.116-128.

[Burns91b] Burns, A., A. J. Wellings, *"Criticality and Utility in the Next Generation,"* The Journal of Real-Time Systems, vol. 3 (1991), pp.351-354.

[Burns92] Burns, A., A. J. Wellings, *"Safetfy Kernels and the Ada Programming Language"*, pp.56-70, Ada in Transition, Proceedings of Ada UK International Conference, 1992.

[Carlow84] Carlow, G. E., *"Architecture of the Space Shuttle Primary Avionics Software System,"* CACM 27(9) (September 1984) pp.926-936.

[Carre89] Carre, B. A., T. J. Jennings, *"A Subset of Ada for Formal Verification (SPARK)"*, pp.121-126, Ada User, vol. 9, 1989.

[Casavant88] Casavant, T. L., J. G. Kuhl, *"A Taxonomy of Scheduling in General Purpose Distributed Computing Systems,"* IEEE Transactions on Software Engineering 14(2) (February 1988) pp.141-154.

[Chen90] Chen, M. I., K. J. Lin, *"Dynamic Priority Ceilings: A Concurrency Control Protocol For Real-Time Systems,"* Real-Time Systems 2(4) (November 1990) pp.325-346.

[Cheng85] Cheng, S., J. A. Stankovic and K. Ramamritham, *"Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems,"* IEEE Transactions on Computers 34(12) (December 1985) pp.1130-1143.

[Cheng87] Cheng, S., J. A. Stankovic and K. Ramamritham, *"Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey,"*,

pp150-174 in *"Tutorial on Hard Real-Time Systems,"* ed. J. A. Stankovic and K. Ramamritham, pub IEEE Press (July 1987).

[Chetto89] Chetto, H., M. Chetto, *"Some Results of the Earliest Deadline Scheduling Algorithm,"* IEEE Transactions Software Engineering 15(10) (October 1989) pp.1261-1269.

[Chung90] Chung, J. Y., J. W. S. Liu and K.J. Lin, *"Scheduling Periodic Jobs That Allow Imprecise Results,"* IEEE Transactions on Computers 39(9) (September 1990).

[Damm89] Damm, A., J. Reisinger, W. Schwabl and H. Kopetz, *"The Real-Time Operating System MARS,"* ACM Operating Systems Review Special Issue (1989) pp.141-157.

[Davari92] Davari, S., L. Sha, *"Sources of Unbounded Priority Inversions in Real-Time Systems and a Comparative Study of Possible Solutions,"* ACM SIGOPS Review 26(2) (April 1992) pp.110-120.

[Davis93] Davis, R. I., K. Tindell, A. Burns, *"Scheduling Slack Time in Fixed Priority Pre-emptive Systems"*, (to appear) Proceedings Real-Time Systems Symposium, (Dec. 1993).

[Dertouzos89] Dertouzos, M. L., A. K. L. Mok, *"Multiprocessor On-Line Scheduling of Hard Real-Time Tasks,"* IEEE Transactions on Software Engineering 15(12) (December 1989) pp.1497-1506.

[Dix89] Dix, A., R. F. Stone and H. S. M. Zedan, *"Design Issues for Reliable Time-Critical Systems,"* Proceedings of Workshop on Real-Time Systems (September 1989).

[Faulk88] Faulk, S. R., D. L. Parnas, *"On Synchronization in Hard-Real-Time Systems,"* Communications of the ACM 31(3) (March 1988) pp.274-287.

[Fohler89] Fohler, G., C. Koza, *"Heuristic Scheduling for Distributed Real-Time Systems,"* Instiut fur Technische Informatik, Technische Universitat Wien, Austria (April 1989).

[Garey75] Garey, M. R., D. S. Johnson, *"Complexity Results for Multiprocessor Scheduling Under Resource Constraints,"* SIAM Journal of Computing 4 (1975) pp.397-411.

[Garey77] Garey, M. R., D. S. Johnson, *"Two Processor Scheduling with Start Times and Deadlines,"* SIAM Journal of Computing 6 (1977) pp.416-426.

[Garey78] Garey, M. R., D. S. Johnson, *"'Strong" NP-Completeness Results: Motivation, Examples, and Implications,"* Journal of the ACM 25(3) (July 1978) pp.499-508.

[Garey79] Garey, M. R., D. S. Johnson, *"Computers and Intractability"*, Freeman, New York, (1979).

[Gonzalez77] Gonzalez, M. J., *"Deterministic Processor Scheduling,"* ACM Computing Surveys 9(3) (September 1977) pp.173-203.

[Graham69] Graham, R. L., *"Bounds on Multiprocessing Timing Anomalies,"* SIAM Journal of Applied Mathematics 17(2) (March 1969) pp.416-429.

[Haban89] Haban, D., K. G. Shin, *"Application of Real-Time Monitoring to Scheduling Tasks With Random Execution Times,"* Proceedings 10th IEEE Real-Time Systems Symposium (5-7 December 1989) pp.172-181.

[Haban90] Haban, D., K. G. Shin, *"Application of Real-Time Monitoring to Scheduling Tasks With Random Execution Times,"* IEEE Transactions on Software Engineering 16(12) (December 1990).

[Harbour91] Harbour, M. G., M. H. Klein, J. P. Lehozky, *"Fixed Priority Scheduling of Periodic Tasks with Varying Exectuion Priority"*, Proceedings 12th IEEE Real-Time Systems Symposium, San Antonio USA, 1991.

[Hayes87] Hayes, I., *"Specification Case Studies"*, Prentice-Hall International, 1987.

[Hutcheon87] Hutcheon, A., A. J. Wellings, *"Ada for Distributed Systems"*, Computer Standards and Interfaces, 6(1), 1987.

[Intel89] *"Intel i486 Microprocessor"*, Intel Corporation, 1989.

[Jensen91a] Jensen, E. D., *"The Kernel Computational Model of the Alpha Real-Time Distributed Operating System,"* IOS Press (1991).

[Joseph86a] Joseph, M., P. Pandya, *"Finding Response Times in a Real-Time System,"* The Computer Journal (British Computer Society) 29(5) (October 1986) pp.390-395.

[Knuth68] Knuth, D., *"The Art of Computer Programming: Seminumerical Algorithms"*, Addison-Wesley, 1968.

[Kopetz85] Kopetz, H., W. Merker, *"The Architecture of Mars,"* 15th Fault-Tolerant Computing Symposium (June 1985) pp.274-279.

[Kopetz89] Kopetz, H., A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft and R. Zainlinger, *"Distributed Fault-Tolerant Real-Time*

Systems: The Mars Approach," IEEE Micro (February 1989) pp.25-40.

[Lehoczky87] Lehoczky, J. P., L. Sha and J. K. Strosnider, *"Enhanced Aperiodic Responsiveness in Hard Real-Time Environments,"* Proceedings IEEE Real-Time System Symposium (1987) pp.261-270.

[Lehoczky89] Lehoczky, J., L. Sha and Y. Ding, *"The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour,"* Proceedings IEEE Real-Time Systems Symposium (5-7 December 1989) pp.166-171.

[Lehoczky90] Lehoczky, J. P., *"Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines,"* Proceedings 11th IEEE Real-Time Systems Symposium (5-7 December 1990) pp.201-209.

[Lehoczky92] Lehoczky, J. P., S. Ramos-Thuel, *"An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Fixed Priority Preemptive Systems"*, Proceedings IEEE Real-Time Systems Symposium, 1992..

[Leung80] Leung, J. Y. T., M. L. *"A Note on Preemptive Scheduling of Periodic, Real-Time Tasks"*, Information Processing Letters, 11(3), (November 1980).

[Leung82] Leung, J. Y. T., J. Whitehead, *"On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks,"* Performance Evaluation (Netherlands) 2(4) (December 1982) pp.237-250.

[Lin87] Lin, K. J., S. Natarajan and J. W. S. Liu, *"Imprecise Results: Utilizing Partial Computations in Real-Time Systems,"* Proceedings 8th IEEE Real-Time Systems Symposium (1-3 December 1987) pp.210-217.

[Lister84] Lister, A. M., *"Fundamentals of Operating Systems,"* Macmillan Computer Science Series (1984).

[Liu73a] Liu, C. L., J. W. Layland, *"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,"* Journal of the ACM 20(1) (1973) pp.40-61.

[Liu91] Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. S Yu, J. Y. Chung, W. Zhao, *"Algorithms for Scheduling Imprecise Computations"*, IEEE Computer, pp.58-68, May 1991.

[Locke86] Locke, C. D., *"Best-Effort Decision Making for Real-Time Scheduling,"* Computer Science Department, CMU (May 10, 1986).

[Locke92] Locke, C. D., *"Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives,"* Real-Time Systems 4(1) (March 1992) pp.37-53.

[Mok83] Mok, A. K. L., *"Fundamental Design Problems of Distributed Systems For The Hard Real-Time Environment,"* Laboratory of Computer Science, Massachsetts Institute of Technology (1983).

[Moron93] Moron, C. E., H. Zedan, *"Adaptable Scheduler Using Milestones for Hard Real-Time Systems"*, YCS 191, Dept. Computer Science, University of York 1993.

[Nassor91] Nassor, E., G. Bres, *"Hard Real-Time Sporadic Task Scheduling for Fixed Priority Schedulers,"* Proceedings International Workshop on Responsive Systems (3-4 October 1991) pp.44-47.

[Park93] Park, C. Y., *"Predicting Program Execution Times by Analysing Static and Dynamic Program Paths"*, pp.31-62, Real-Time Systems, 5(1), March 1993.

[Pilling90] Pilling, M., A. Burns and K. Raymond, *"Formal Specifications and Proofs of Inheritance Protocols for Real-Time Scheduling,"* Software Engineering Journal (September 1990) pp.263-279.

[Puschner89] Puschner, P., C. Koza, *"Calculating The Maximum Execution Time Of Real-Time Programs,"* The Journal of Real-Time Systems 1(2) (September 1989) pp.159-176.

[Rajkumar87] Rajkumar, R., L. Sha and J. P. Lehoczky, *"On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment,"* Proceedings IEEE Real-Time Systems Symposium (1987) pp.2-11.

[Rajkumar88a] Rajkumar, R., L. Sha, J. P. Lehoczky and K. Ramamithram, *"An Optimal Priority Inheritance Protocol for Real-Time Synchronisation,"* Deptartment of Computer and Information Science, University of Massachusetts (October 17, 1988).

[Rajkumar88b] Rajkumar, R., L. Sha and J. P. Lehoczky, *"Real-Time Synchronisation Protocols for Multiprocessors,"* Proceedings IEEE Real-Time Systems Symposium (December 1988) pp.259-269.

[Rajkumar89] Rajkumar, R., L. Sha and J. P. Lehoczky, *"An Experimental Investigation of Synchronisation Protocols,"* Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software (May 1989) pp.11-17.

[Ramamritham87] Ramamritham, K., J. A. Stankovic, *"The Design of the Spring Kernel"*, Proceedings IEEE Real-Time Systems Symposium (December 1987) pp.146-157.

[Sedgewick83] Sedgewick, R., "*Algorithms*", Addison-Wesley, 1983.

[Sha87a] Sha, L., R. Rajkumar and J. P. Lehoczky, *"Priority Inheritance Protocols: An Approach to Real-Time Synchronisation,"* Computer Science Department, Carnegie-Mellon University (December 1987).

[Sha87b] Sha, L., J. P. Lehoczky and R. Rajkumar, *"Task Scheduling in Distributed Real-Time Systems,"* Proceedings IEEE Industrial Electronics Conference (IECON) (1987) pp.909-915.

[Sha89] Sha, L., B. Sprunt and J. P. Lehoczky, *"Aperiodic Task Scheduling for Hard Real-Time Systems,"* The Journal of Real-Time Systems 1 (1989) pp.27-69.

[Sha90] Sha, L., R. Rajkumar and J. P. Lehoczky, *"Priority Inheritance Protocols: An Approach to Real-Time Synchronisation,"* IEEE Transactions on Computers 39(**9**) (September 1990) pp.1175-1185.

[Shen89] Shen, C., K. Ramamritham and J. A. Stankovic, *"Resource Reclaiming in Real-Time,"* Proceedings 10th IEEE Real-Time Systems Symposium (5-7 December 1989) pp.41-50.

[Simpson90] Simpson, H., *"Four-Slot Fully Asynchronous Communication Mechanism,"* IEE Proceedings Part E 137(1) (Jan 1990) pp.17-30.

[Sprunt88] Sprunt, B., J. Lehoczky and L. Sha, *"Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm,"* Proceedings IEEE Real-Time Systems Symposium (December 1988) pp.251-258.

[Sprunt90] Sprunt, B., "*Aperiodic Task Scheduling for Real-Time Systems*", Ph.D. Thesis, CMU, 1990.

[Stankovic87a] Stankovic, J. A., K. Ramamritham and W. Zhao, *"Preemptive Scheduling Under Time and Resource Constraints,"* IEEE Transactions on Computers 38(8) (August 1987) pp.949-960.

[Stankovic87b] Stankovic, J. A., K. Ramamritham, *"The Design of the Spring Kernel,"* IEEE Proceedings Real-Time Systems Symposium (1987) pp.146-157.

[Stankovic88] Stankovic, J.A., *"Misconceptions About Real-Time Computing: A Serious Problem for Next Generation Systems,"* IEEE Computer 21(10) (October 1988) pp.10-19.

[Stankovic89] Stankovic, J. A., K. Ramamritham and W. Zhao, *"Distributed Scheduling of Tasks with Deadlines and Resource Requirements,"* IEEE Transactions on Computers 38(8) (August 1989) pp.1110-1123.

[Stankovic90] Stankovic, J.A., K. Ramamritham, *"What is Predictability for Real-Time Systems?,"* Real-Time Systems 2(4) (1990) pp.247-254.

[Stankovic91] Stankovic, J. A., K. Ramamritham, *"The Spring Kernel,"* IOS Press (1991).

[Tindell92] Tindell, K., *"An Extendible Approach For Analysing Fixed Priority Hard Real-Time Systems"*, YCS 189, Department of Computer Science, University of York (1992).

[Tokuda89] Tokuda, H., C. W. Mercer, *"ARTS: A Distributed Real-Time Kernel,"* ACM Operating Systems Review Special Issue (1989) pp.29-53.

[Tokuda91] Tokuda, H., C. W. Mercer, *"The ARTS Kernel: Toward Predictable Distributed Real-Time Systems,"* IOS Press (1991).

[Wilf86] Wilf, H. S., *"Algorithms and Complexity,"* Prentice-Hall International (1986).

[Xu90] Xu, J., D. L. Parnas, *"Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations,"* IEEE Transactions on Software Engineering 16(3) (March 1990) pp.360-369.

[Xu91a] Xu, J., D. L. Parnas, *"On Satisfying Timing Constraints in Hard Real-Time Systems,"* Proceedings ACM SIGSOFT '91 Conference on Software for Critical Systems (December 4-6 1991).