A Constrained Computational Model for Flexible Scheduling

Charles Gerard McElhone

Submitted for the Degree of Doctor of Philosophy

University of York

Department of Computer Science

November 1996

To my parents.

CONTENTS

ACKNOWLEDGEMENTS

Particular thanks are due to my supervisor Alan Burns for his help and encouragement over the last five and a half years. I would also like to thank my assessor Andy Wellings for some stimulating and fruitful discussions during the course of that time. Thanks are also due to the University of Teesside, and in particular Dr Derek Simpson, for providing financial support and teaching relief.

Finally, a special thanks to my wife, Jane, for much support and patience.

DECLARATION

I declare that the research described in this thesis is original work, which I undertook between April 1991 and March 1996. Certain parts of this thesis have been published previously as technical reports or conference proceedings.

Chapter 4 describes research previously published in "Adapting and Evaluating Algorithms for Dynamic Schedulability Testing" *YCS 225, Department of Computer Science, University of York* (March 1994).

Chapter 5 describes research previously published in "Hybrid Algorithms for Dynamic Schedulability Testing", *YCS 241, Department of Computer Science, University of York* (November 1994).

Chapters 4 and 5 also draw on research which was included in "Hybrid Algorithms for Dynamic Schedulability Testing" *Proceedings of the 7th Euromicro Workshop on Real-Time Systems, Odense* (June 1995).

ABSTRACT

Future real-time systems will require to be adaptive in response to their environments and to system failures, as well as meeting their time constraints for mission and safety-critical functions. Currently, the critical functions of real-time systems are guaranteed before run-time by performing a worst-case analysis of the system's timing and resource requirements. The result is that real-time systems are engineered to have *spare capacity,* under normal operation. A challenge of current research is to make use of this spare capacity, in order to satisfy the requirements for adaptivity in the system. Adaptivity can be implemented by *optional computation*s with *firm* deadlines. Optional computations, can be scheduled, and even guaranteed at run-time, by methods of *flexible scheduling*.

This thesis starts by surveying the complex requirements for adaptivity within real-time systems. There is evidence that the run-time support for a computational model which incorporates all such complex requirements, would incur such large overheads that little spare capacity would remain for the optional computations themselves. The solution devised in previous research is to employ specialised hardware, or additional processors, in order to facilitate the support of a complex computational model. This thesis provides an alternative approach by developing a *constrained* computational model, which is nevertheless general enough to support many of the requirements for adaptivity. The claim is, that the relatively small overheads incurred by the run-time support for a constrained model, will leave adequate capacity for the performance of optional computations.

In order to demonstrate the viability of the run-time support for the constrained computational model, the thesis develops and evaluates (i) efficient algorithms for the on-line acceptance testing of optional computations (ii) allocation methods which enhance the throughput of optional computations within multiprocessor systems, and (iii) cost-effective policies for the admission of optional computations which pass their acceptance tests. The thesis also addresses programming issues by demonstrating that the constrained model can be implemented in a standard programming language i.e. Ada 95.

A major conclusion of this work is that the constrained computational model is viable, so long as acceptance tests, allocation methods and admission policies are chosen, which are appropriate to the spare capacity which exists on the processor(s).

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Real-time applications are characterised by their requirement to respond to their environments within finite and specified time constraints. In *soft* real-time systems, a failure to meet such a time constraint is merely inconvenient. In *hard* real-time systems such a failure may have catastrophic results. Such hard real-time applications include medical monitoring systems, process control systems, control systems for power stations, and flight control systems for aircraft.

Real-time systems are often modelled and implemented as concurrent *tasks*. Each task is a schedulable entity which delivers some of the functionality of the application within its required time constraints. Tasks can be *periodic* in which case they run regularly at intervals e.g. sampling a sensor. Alternatively tasks can be *aperiodic* in which case they run irregularly e.g. in response to some change in the environment in which the real-time system operates. *Critical* tasks are vital to the system, and usually have hard deadlines, in line with the definition of 'hard' given above. *Non-critical* tasks may have soft deadlines. Tasks have a worst-case execution time (WCET) which can be derived from their code and is an estimate of the highest amount of processor time they will use in one execution. When real-time tasks complete their computations within their deadlines, they provide some service to the real-time system which may be quantified as having some *utility* or *value* to the system. The characteristics and behaviour of the tasks in a real-time system can be described in a *computational model*. Whether this model includes scheduling issues is a matter for debate.

Hard real-time systems may often be *safety-critical systems* because certain failures of the system may result in catastrophic consequences such as loss of human life. Hard real-time systems may also have *mission critical* components whose correctness and reliability is crucial to the key services delivered by the system. There may well be requirements for fault-tolerance within such systems, and graceful degradation under failure. Hard real-time systems are often embedded in larger systems e.g. a real-time control system for an aircraft. Therefore the size and weight of hardware may be a constraint, along with the memory available for software.

In order to guarantee the performance of hard real-time systems, *predictability* is important. This ensures that time and resource constraints can be known to be satisfied even under maximum loading of the system. Therefore safety or mission-critical tasks within the real-time system are analysed before run-time to ensure that they can be scheduled to meet their time constraints even under worst-case conditions. This results in such systems being constructed with a processing capacity which meets the worst-case requirement but is excessive for normal operation. One current area of research is into making use of this spare capacity in order to enhance the total utility delivered by all the tasks in the real-time system [8,27,35]. The work of this thesis is a contribution to this research area.

1.2 MOTIVATION

Over the past few years there have been several keynote articles on the future of real-time systems. According to Stankovic [51] the next generation of real-time systems will be more complex and capable of exhibiting intelligence. They will have long lifetimes and be required to exhibit a great deal of adaptability. They will function in distributed and dynamic environments, and economic, human and ecological catastrophes will result if their timing constraints are not met.

More recently Musliner et al. [38] write about future real-time systems combining Artificial Intelligence with the requirements to perform within real-time constraints. Such *Real-time AI* systems would have to:

- work continuously over extended periods of time
- interface with the external environment via sensors and actuators
- handle uncertain or missing data
- concentrate resources on the most critical events
- handle both periodic and aperiodic events in a predictable fashion with guaranteed response times
- degrade gracefully

An example given is that of the proposed Mars Rover for NASA. This must operate at a distance of about 15 light-minutes from earth and therefore cannot be tele-operated. It must operate continuously and autonomously in an incompletely specified environment. It must react in real-time to unpredictable conditions such as navigation route blockages and dangerous terrain such as sand pits. This requires "adaptability and intelligence beyond the capability of current real-time technology" [38].

One of the aims of current research is to make use of the spare capacity of real-time systems mentioned above, in order to incorporate AI techniques for such adaptability and intelligence. Requirements for fault-tolerance and graceful degradation can also be met by making use of spare capacity. A key problem with AI techniques is that they often have very large bounds on their WCETs. This makes them difficult to integrate with conventional real-time tasks whose worst-case performance can be more easily bound, and guaranteed to meet hard deadlines.

The first steps in solving this difficulty is to distinguish between those tasks which are critical and those which are non-critical. Critical tasks are necessary to achieve the minimum standards of safety and reliability in the real-time system. Because a missed deadline in a critical task could be catastrophic, all critical tasks must have their computation times bound and their deadlines guaranteed by schedulability analysis. In contrast, the non-critical tasks need not be guaranteed before run-time. However some non-critical tasks may have *firm* deadlines which means that, although missing such a deadline in not catastrophic, it does result in a significant loss of utility to the system. Tasks which perform AI functions may well fit into this category. These tasks provide adaptivity, and by their very nature may be required to run unpredictably. Therefore the issue arises as to whether such tasks should be guaranteed at run-time, before they start, in order to ensure that they can meet their firm deadlines.

1.3 INCORPORATING ADAPTIVITY

The Spring Project [52] models the tasks within a real-time system to be either critical, essential or non-essential. Essential tasks have firm deadlines, as described above, whereas non-essential tasks have soft deadlines. The project attempts to incorporate these three types of task into a consistent scheduling scheme which satisfies the need to guarantee critical tasks *before* run-time and the need to be flexible in guaranteeing essential tasks *at* run-time. Non-essential tasks have soft deadlines and need not be guaranteed. The project assumes a distributed real-time system with a number of nodes. Each node of the distributed system has a resident set of critical tasks which have been tested for schedulability before run-time. These tasks are assumed to be periodic whereas essential tasks are assumed to arise aperiodically.

Spring provides *dynamic* or *flexible scheduling* for essential tasks in that they can either be guaranteed on-line on the node on which they arise, or if this guarantee is not possible, an attempt can be made to guarantee them on another node of the system. In the Spring Project, the algorithms used to guarantee the schedulability of essential tasks are called *guarantee algorithms*. These algorithms attempt to construct a schedule which preserves the guarantees given to resident critical tasks and previously accepted essential tasks, but also incorporates the new essential task. If an essential task cannot be guaranteed at one node of the system, then a *distributed scheduling algorithm* is used to direct the task to another node where it is likely to be guaranteed.

As has been said, tasks which provide adaptivity, are not only required to run aperiodically, but may have execution times which are difficult, or impossible, to bound. This makes it difficult to guarantee such tasks. Liu [33,34] has developed techniques for modelling various requirements for the computation of such tasks.

According to Liu, tasks can be divided up into those components which are *mandatory* and must be guaranteed, and those components which are *optional.* Mandatory components are mission or safety-critical and must be guaranteed. They correspond to the critical, periodic tasks on a Spring node. However, in some applications, mandatory tasks can be improved upon by further execution, or can be replaced by longer tasks which would provide more utility to the system if they could be scheduled at run-time. If such additional computations have firm deadlines, then it may be possible to guarantee them at run-time, provided that their WCETs can be bounded.

Various schemes have been developed for mandatory and optional computations, in order to meet the different requirements for flexible computation. Four methods are:

- Imprecise Computations
- Sieve Functions
- Multiple Versions
- Approximate Processing

The methods are now described briefly. It is assumed that all of the components of computation can in some way be bound, so that the only issue is whether or not to attempt to guarantee each component (i) before run-time or (ii) at run-time. (Components of computations whose WCETs are unbounded, cannot be guaranteed, and must simply be executed in the hope that they will produce a useful result in whatever processing time is currently available. How to optimise the execution of such components is not considered here.)

Imprecise Computations model the requirements of tasks which provide some result of minimum precision, reliability or confidence level, which may be improved by further computation. It is an assumption of this method that the improvement in the result will increase steadily or *monotonically* after each stage of further computation. The minimal computation may be modelled as a mandatory task which must be guaranteed off-line, and further computation(s) can be modelled as optional task(s) which may be guaranteed at run-time. Often the extra computation takes the form of a sequence of iterations, each iteration refining the results generated by the previous. An integer number of iterations may therefore be included in the optional computation, depending on how many can be guaranteed to meet the (firm) overall deadline of the Imprecise Computation. Applications which require Imprecise Computation can include numerical computation, statistical estimation and prediction, heuristic search activities, and sorting.

Sieve Functions model the requirements for adaptive processing where a computation consists of alternating compulsory and optional components. At the end of a compulsory component, it is possible to improve upon the result by further computation which is assumed to monotonically improve the quality of the input for the next compulsory stage. The compulsory components can be guaranteed off-line to meet the overall deadline for the sieve function. Each optional part may be guaranteed dynamically by the flexible scheduler. The overall deadline for the sieve function may be thought of as being *hard* for the compulsory components but *firm* for the optional components. Should the optional components fail to be guaranteed, then a minimum utility will be provided by the compulsory components which will simply execute in sequence.

Multiple Versions can be applied widely where there are a number of versions of a computation which provide different utilities to the system. In the simplest case of two versions, the *primary version* is the preferred, more computationally expensive version, which may be guaranteed dynamically by the flexible scheduler. If the guarantee is not given, then a *secondary version,* which is cheaper, but has been guaranteed off-line, will run instead. The secondary version will provide the minimum service required.

Multiple versions which can be bounded and guaranteed, may be extended to more than one alternative version, so that the most expensive (most preferred) version which can be guaranteed at run-time, is the one which is chosen.

Approximate Processing assumes that the value of the WCET of a task may be defined by a set of parameters. Assuming that the mapping from parameter values to WCET is available, then it is possible to select values for the parameters so as to provide maximum utility, within a WCET which is known to be currently schedulable. The selection of parameter values may be facilitated by a knowledge of the spare capacity which is currently available on the processor.

1.3.1 Application Example

Future applications for real-time systems may have more complex requirements for optional computations than can be satisfied by the simple methods described above. Take as an example an *autonomous vehicle control system* [40]. Here, there may be a requirement for a complex hierarchy of tasks and subtasks each of which may be mandatory or optional. Tasks and subtasks may each have deadlines. There may be complex precedence relationships between subtasks. At the top of the hierarchy may be intelligent functionality such as route planning, whereas at the bottom there may be critical functions such as collision avoidance. At run-time a choice is made as to which computations should be scheduled in order to provide optimum utility to the system. Such a system may have a requirement for fault-tolerance and graceful degradation which can be achieved by the progressive abandonment of optional computations leaving the mandatory computations to provide a safe level of service. For example, in town traffic where collision avoidance takes a great deal of processing time, it may be necessary for the human operator to instruct the route to be taken.

1.4 FLEXIBLE SCHEDULING OVERHEADS

1.4.1 Scheduling Concepts

As has been described above, *flexible scheduling* allows the on-line guarantee of optional computations whilst safeguarding the guarantees which an off-line *schedulability analysis* has already given to critical tasks. Flexible scheduling may be supported by a variety of *scheduling policies*. It is the scheduling policy which determines, at any given time, which task is dispatched to the processor for execution. Scheduling policies are *pre-emptive* if the task which is currently running on the processor may be interrupted by a more urgent task. The urgency or importance of tasks can be indicated by allocating them *priorities*, and one of the main issues in scheduling is to decide upon what basis such priorities should be allocated. Priorities may be *static* or *dynamic* according to whether they are fixed off-line or can be varied at run-time. For example the *Earliest Deadline* policy allocates priorities dynamically so that the tasks with the nearest deadlines have the highest priority on the processor [32]. In contrast *Deadline Monotonic* scheduling policy [3] has a static allocation of priorities which corresponds to the monotonic ordering of task deadlines which have been specified off-line.

The scheduling policies used in hard real-time systems must allow schedulability analysis of all tasks in the task set. Analysis is performed *statically* for the set of critical tasks which are periodic. However it may be performed *dynamically* for optional tasks which arise aperiodically. *Acceptance tests* for aperiodic tasks ensure that they are schedulable alongside the resident set of critical tasks, and any aperiodic tasks which have already been accepted. Acceptance tests often require to know the *slack* possessed by each of the existing tasks on the processor. A task's *slack* is defined as the task's relative deadline, minus the task's remaining computation time and any delay in the tasks execution which may be caused by the execution of higher priority tasks.

When there are competing optional tasks, then *admission policies* may be used to arbitrate between them. An example of an admission policy is *Best Effort* [35] in which an aperiodic task of *high utility* may abort aperiodic tasks of *lower utility*, in order that it may pass the acceptance test. (*Utility* provides some measure of the service which a task provides for the system, when the task has completed its computation.)

The scheduling services described here are usually provided by a *real-time kernel* or *run-time support* i.e. low-level software which runs in support of the application code.

1.4.2 Cost-effectiveness of Flexible Scheduling

A key issue in current research is whether the overheads incurred by flexible scheduling are small enough for it to be cost-effective. In terms of the above discussion: are the overheads incurred by admissions policy and acceptance testing outweighed by the increase in total utility gained by the system? This is a crucial issue when the overheads occur on the same processor which runs the applications tasks, because the overheads reduce the processor time which is available for applications tasks. Some researchers [39] have avoided this problem by going to the expense of developing specialist hardware which carries the overheads for acceptance testing and scheduling.

Much of the evidence from recent research would indicate that the overheads for flexible scheduling prohibit its use on the same processor that runs the applications tasks. For example, it has been found by Wendorf [60] that the overheads for the original version of Best Effort admission policy can drastically reduce the time available for applications tasks on the processor.

In the Spring Project [56] Stankovic et al. investigate the acceptance testing of general task sets with resources and precedence constraints between task executions. So high are the overheads incurred that they develop heuristics in order to speed up schedulability analysis. They go on to design a hardware co-processor [39] which will remove the flexible scheduling overhead from the applications processor. The Spring Project also develops methods of distributed scheduling [53,55] which can incur large overheads in order to re-allocate rejected tasks to other nodes in a network where they may be guaranteed.

Audsley [2] has shown that an algorithm which can provide an exact acceptance test at run-time has a pseudo-polynomial complexity. Following the work of Audsley, Davis [8] has found that the overheads incurred by the pseudo-polynomial algorithm can so reduce performance, that a simpler but inexact algorithm can provide an equal, if not better, throughput of aperiodic tasks.

This thesis will claim that, contrary to the above evidence, flexible scheduling can be performed cost-effectively, on the same processor which runs the applications tasks. Therefore, a major aim of this thesis will be to demonstrate that the overheads for admission policy and acceptance testing can be reduced to such a level that greater utility may be provided for the real-time system.

1.5 PROGRAMMING LANGUAGE SUPPORT

This thesis will also consider programming language support for flexible scheduling. One approach to the provision of flexible scheduling in the next generation of real-time systems, is to provide suitable constructs in the programming language(s) which will be used to implement such systems. These constructs must have sufficient expressive power to allow the programmer to request the dynamic guarantee of optional computations in various forms e.g. imprecise computations, sieve functions or multiple versions. A major language issue is whether optional computations should take the form of the tasks or processes within a concurrent programming language, or whether they should be formulated as sections of code *within* tasks.

At present there are few programming languages which offer constructs for the flexible scheduling of optional computations with firm deadlines. Furthermore the ones that do exist are experimental languages which have never been used for real-world applications. For example, the languages *Flex* [26] and *Real-time Concurrent C* [19] have been developed by researchers to provide support for flexible scheduling. However, neither language has been sufficiently implemented to be of practical use in the engineering of real-time systems.

The *Flex* language allows the specification of time and resource constraints for Imprecise Computation and Multiple Versions using the concept of *performance polymorphism*. This allows different real-time functions to be chosen at run-time, according to the time and resources available. However, the *Flex* execution environment concentrates on optimising the *average* performance of these real-time functions. Therefore *Flex* does not emphasise the *guarantee* of firm deadlines.

Real-time Concurrent C [19] is an extension of *Concurrent C* [18]. This language provides full facilities for the run-time guarantee of sections of code within processes/tasks, and also provides for the execution of some alternative action if a guarantee is not given. The language also allows time constraints to be placed on the time taken for the guarantee itself. However, there has been little work on the necessary run-time support for this language. As discussed previously, run-time support can incur excessive overheads. The designers of *Real-time Concurrent C* fail to indicate [42] the likely overheads which would be incurred by such a run-time system. Since *Real-time Concurrent C* is allied to the Spring Project it may be that its designers envisage the use of computationally expensive heuristics in order to guarantee time and resource constraints.

In contrast to such experimental languages, the programming language PEARL has been used extensively in Europe for the programming of large real-time systems. The existing PEARL standards [14,15,16] do not provide for flexible scheduling. However Halang and Stoyenko [22] have proposed extensions for the language which would allow

(i) Multiple Version programming under system overload, and (ii) the selection of different sections of code according to the residual execution time of tasks. However, these features do not provide full flexible scheduling, and they are not yet embodied in a PEARL standard.

A new standard has however been agreed for the programming language Ada which has been widely used for the engineering of real-time systems. The new standard, *Ada 95*, has advanced concurrency features in the core of the language, and an annex which provides specific facilities for the programming of real-time systems. This *Real-Time Systems Annex* does not explicitly address flexible scheduling as conceived here. Nevertheless, a variety of new constructs are provided, and it may be possible to use these, and some of the new concurrency features, in order to implement flexible scheduling in Ada.

1.6. THESIS STATEMENT

1.6.1 Assumptions

Contrary to the difficulties outlined above, this thesis attempts to demonstrate that flexible scheduling is practicable. The contention is that flexible scheduling can be efficiently implemented using existing, conventional technology, without resort to specialist hardware or experimental programming languages.

In regard to the hardware required, this thesis assumes that flexible scheduling is implemented on the same processor that runs the application tasks. As stated earlier, previous researchers have used specialised hardware to reduce the effect of the overheads incurred by flexible scheduling. For example, the Spring project uses a specialised co-processor to perform guarantees of aperiodic tasks. Alternatively performance could be speeded up by implementing real-time kernel functions in hardware on the same processor. Although such specialised hardware can always provide greater performance, its use has a number of significant drawbacks. It is not general-purpose, or generally available, and it incurs greater costs in verification. This thesis presents a different approach by arguing that flexible scheduling can be made to perform efficiently on conventional processors.

In regard to programming languages, the thesis does not require a non-standard language which may never be fully implemented. Instead, the claim is that flexible scheduling may be implemented by using the existing constructs of a standard language, with a wide user-base. Therefore an application which requires flexible scheduling, need only include the appropriate library objects. (It may be necessary to assume that the run-time support for the programming language has been extended to provide some support for flexible scheduling.)

The thesis assumes that a *computational model* embraces both concurrency and scheduling issues. Therefore a computational model is defined as a framework which includes a definition of task characteristics, and also describes how the tasks are to be scheduled. Task characteristics include: whether tasks are periodic, aperiodic or sporadic, whether tasks have hard or soft deadlines, and whether tasks are independent, or share resources and intercommunicate. Scheduling includes a definition of which scheduling policy and which admission policy are used.

1.6.2 Thesis

The central thesis can be stated as follows:

"The application requirements for flexible scheduling can be embraced in a constrained computational model for which cost-effective run-time support can be provided. The model can be implemented in a standard programming language so that applications written in this language can increase their utility."

The key aims of the thesis are:

- to derive a constrained computational model which fulfils the set of application requirements for the flexible scheduling of optional computations.

- to provide cost-effective algorithms and implementations for the run-time support of the model. It is assumed that the run-time support executes on the same processor as the applications tasks. Therefore, the run-time support must be cost-effective i.e. its overheads must be sufficiently small, that applications tasks achieve a net increase in their utility.

- to develop methods of allocating optional computations to processors, such that the throughput of optional computations is enhanced. Again the emphasis is on finding computationally inexpensive methods, so that additional hardware is not needed to support the allocation methods.

- to demonstrate that a standard programming language may be used to implement the computational model, so that the model can be used in practice.

1.7 APPROACH

The above aims suggest three strands of enquiry throughout the work:

1. A survey of the requirements for optional computations in the next generation of real-time systems and the derivation of a computational model which is general enough to incorporate different forms of mandatory and optional computations, but sufficiently constrained to be supported by run-time support which incurs low overheads.

2. A review of existing run-time support for flexible scheduling and the development of (i) more efficient algorithms for the guarantee of optional computations, and (ii) efficient methods of allocating optional computations to processors, such that the number of computations being guaranteed are maximised.

   The success of part (i) of this strand is vital to the thesis because, without cost-effective run-time support, the computational model is not viable under the assumption of conventional hardware, running applications tasks and run-time support, on the same processors. Therefore part (i) will form a major portion of the work, and whatever algorithms are developed, will have to be evaluated for their efficiency.

3. An investigation into programming language support for optional computations, and a demonstration that optional computations may be implemented in a current programming language for the engineering of real-time systems.

Strand 2 above can be elaborated further because of its central importance within the thesis. The two-pronged attack on the development of cost-effective support for the computational model will consist of :

(i) an investigation into efficient algorithms for the guarantee/rejection of requests for optional computation. A particular concern is to find the algorithms which provide the optimum trade-off between the overheads they incur, and the exactness of the schedulability test which they provide. (It is assumed that less exact algorithms can be pessimistic in that they sometimes reject optional computations which may in fact be schedulable).

(ii) an investigation into methods of allocation of optional computations within a multi-processor cluster. In (i) optional computations are considered to "arise" and can be schedulability tested in the same way whether they have been released locally or have originated from some remote client. Here the assumption is made that those optional computations arriving from outside a processor cluster are subject to some allocation policy within the local cluster. This can be considered as the dynamic analogue of the problem of providing a static allocation of tasks within a multiprocessor system. The aim here is to maximise the throughput of optional computations within the cluster. It is assumed that full schedulability testing of the optional computations is still performed on the target processors.

1.8 METHODS USED

Modelling and simulation studies are standard techniques for research in scheduling [2,7,8,56]. In this work, simulation studies are used throughout strand 2 as described above. In part (i) of strand 2, simulations are used to compare the relative performances of guarantee algorithms and to measure their overheads. Task set generators are built in order to provide task sets with a variety of characteristics. Different task sets are used to establish performance profiles for the various algorithms under examination.

Further simulation studies are conducted during strand 2 (ii) of the work. Here the objective is to enhance the performance of the computational model within a processor cluster. Both targeted and random allocation of optional computations within the cluster are simulated. Various configurations of processors within the cluster are modelled, and the effects of optional computations which are generated both inside and outside the cluster are investigated.

The most efficient of the guarantee algorithms from strand 2 (i) is used in simulation studies which evaluate the admission policy used in the computational model of strand 1. Performance parameters are varied in order to establish the windows of operation within which the model and its admission policy, can be supported cost-effectively.

Strand 3 aims to demonstrate that optional computations can be implemented in a standard language for the engineering of real-time systems. This requires verification by reference to the requirements gathered at the review stage of strand 1.

1.9 THESIS ORGANISATION

This section outlines each chapter of the thesis.

Chapter 2: Review of Flexible Scheduling

This chapter reviews recent work relating to the three strands of the thesis which are discussed above. Firstly, there is a survey of future application requirements for flexible scheduling within real-time systems. This section goes on to consider existing computational models and programming paradigms for optional computations.

Secondly, existing run-time support for flexible scheduling is reviewed, starting with algorithms which optimise, but do not guarantee, the execution of optional computations. Next, the Spring Project is considered in some detail, with special reference to guarantee algorithms and methods of distributed scheduling. This section concludes by reviewing Audsley's algorithms for *static* schedulability testing which are considered as a promising basis for cost-effective *dynamic* schedulability testing.

Thirdly, the current provision of programming language support for flexible scheduling is reviewed. The languages included are Flex, Real-Time Concurrent C, PEARL and Ada. This leads to the choice of Ada 95 as a standard language in which to demonstrate the implementation of optional computations.

Chapter 3: A Constrained Computational Model

This chapter develops a constrained computational model which fulfils many of the application requirements for optional computations. The chapter starts by making the case for simplicity in the model in order to reduce the overheads of run-time support. After some discussion of the necessary constraints, the model is presented. It provides 3 levels of utility for optional computations: high, medium and low. Each level is regarded as adding value to a baseline utility provided by mandatory computations. Optional computations of higher utility may abort those of lower utility, according to a version of Best Effort admission policy. A key issue which determines the viability of the model is the relative values of the utilities assigned to each level. Finally the model is 'verified' by a discussion of how the various requirements for optional computations may be met within it.

Chapter 4: Viability of On-line Acceptance Testing

This chapter investigates efficient schedulability test algorithms which could contribute to cost-effective run-time support for flexible scheduling. The chapter describes how existing static schedulability tests may be adapted to make use of dynamic scheduling data and thus provide schedulability tests for incoming optional computations. The adapted algorithms trade off complexity with pessimism. Their performance may be improved by combining them in a *hybrid* algorithm. Further performance improvements may be made in all of the algorithms by inserting a timeout, in order to limit their worst-case execution time. The hybrid algorithm is used in an investigation into the parameters which determine the optimum value of this performance-enhancing timeout. Finally, this optimum value is made use of, in an investigation into the effect of changing the proportions of mandatory (periodic) and optional (aperiodic) utilisations.

Chapter 5: Enhanced On-Line Guarantees

This chapter describes attempts to enhance the performance of the hybrid algorithm developed in Chapter 4. The first component of the algorithm has $O(N^2)$ complexity and the second component has pseudo-polynomial complexity. The chapter describes the development of a number of variations on the hybrid algorithm in an attempt to (i) make the $O(N^2)$ component more exact and (ii) make the pseudo-polynomial component faster by giving it a 'head start'. Both of these approaches trade off schedulability testing overheads against the pessimism of the schedulability test. Depending on factors such as the number of resident periodic tasks, both methods are found to be capable of enhancing the number of optional computations guaranteed.

The chapter also investigates the effect of using an optimal, dynamic placement of optional computations within the task list. Hitherto, static placement has been used, according to the value of task deadlines at release time. The chapter concludes by discussing performance profiles for a number of variations of the hybrid algorithm.

Chapter 6: Allocation Methods in Multiprocessor Systems

The aim of the work in this chapter is to enhance the throughput of optional computations when they are allocated among a number of processors within a cluster. Each processor can receive optional computations generated either externally or internally, to the cluster. Performance is measured by the throughput of optional computation over the whole the cluster. The first cluster configuration investigated is that of a *targeting* processor and three *target* processors arranged in a four-processor cluster. The targeting processor receives optional computations from outside the cluster and uses its knowledge of the current slack on each of the targets in order to direct optional computations to the targets most likely to guarantee them. (Target processors still perform schedulability tests on the optional computations which are discarded if the tests fail.) It is assumed that scheduling data are communicated between the targets and the targeting processor.

The main issues investigated are:

- how to minimise the overheads for monitoring slack on the target processors
- whether targeting provides higher throughput than simple 'round robin' allocation
- how the distribution of mandatory processor utilisation on the targets affects throughput

The second cluster configuration which is investigated is that of a loop within which optional computations are allocated 'round robin', but are shuffled to the next processor along if they fail their schedulability test on their previous processor. This shuffling continues until each optional computation has either been accepted, or has been rejected on all of the processors. The technique is named Shuffle Schedulability Testing.

Chapter 7: Admission Policies

This chapter presents simulations studies which compare the performances of Best Effort Admission Policy (as used in the computational model) with the use of FCFS Admission Policy. The objective is to determine ranges of parameters of the simulations, within which, the computational model with Best Effort, provides a superior performance to that of the simple FCFS Admission Policy used hitherto. Performance is measured by the total utility gained by optional computations throughout a simulation. The simulations use one of the versions of the hybrid guarantee algorithm, developed in Chapter 5.

The simulation parameters which are varied include (i) the Total Processor Utilisation (mandatory plus optional computations) (ii) the proportion of Periodic Utilisation (mandatory computation) and (iii) the ratios of the utility values associated with the three classes of optional computation. The results of the simulations indicate that the computational model with Best Effort, provides superior performance for Periodic Utilisations of less than 50%, and optional computation overloads of up to 100%.

Chapter 8: Implementation of the Computational Model

Firstly, the case is argued for implementing optional computations within Ada tasks, rather than at the task level itself. The chapter goes on to review some of the Ada 95 constructs which may be useful in implementing optional computations. The asynchronous select statement is chosen as a construct which can carry the code for an optional computation, within an Ada task.

The chapter next presents the Ada code for a protected object which can be called by requests for the guarantee of optional computations. This protected object handles all concerns regarding the utilities of optional computations, and implements the algorithm for Best Effort admission policy. However, the assumption is made that the protected object is able to call the Ada RTS in order to (i) perform a schedulability test for each optional computation (ii) withdraw a lower utility optional computation from the task list and (iii) efficiently reinstate all withdrawn computations if a request is finally rejected.

The chapter ends with a case by case demonstration, that the asynchronous select statement can be used in different ways, to fulfil many of the requirements for optional computations which were discussed in the review of Chapter 2.

Chapter 9: Conclusions

This chapter discusses major conclusions, and the contribution which has been made by this work. Future work in this area is also discussed.

CHAPTER 2

A REVIEW OF FLEXIBLE SCHEDULING

2.1 INTRODUCTION

Chapter 1 has outlined the three-stranded approach of this thesis. Each strand begins with a review stage:

1. A survey of the requirements for optional computations.

2. A review of the existing run-time support for flexible scheduling.

3. An investigation into programming language support for optional computations.

This chapter is divided into three main parts corresponding to each strand. Sections 2.2 and 2.3 review examples of real-time applications which require flexible scheduling, and then go on to consider the paradigms and models which have been developed so far by researchers such as J. Liu, Garvey and Lesser, Locke, and those involved in the Spring Project.

Sections 2.4 to 2.7 review existing run-time support for flexible scheduling. Section 2.4 considers methods which preserve bandwidth for aperiodic tasks but do not actually guarantee them. Section 2.5 reviews the guarantee algorithms of the *Spring Project*, and Section 2.6 considers the methods used in the Spring Project for distributed scheduling. Section 2.7 reviews the *off-line* schedulability tests developed by Audsley et al. These tests are included because they provide a promising basis for cheaper on-line guarantee algorithms than those used in the Spring Project.

Section 2.8 surveys the current programming language support for optional computations. The languages included are Flex, Real-Time Concurrent C, PEARL, and Ada 95.

2.2 APPLICATION REQUIREMENTS FOR OPTIONAL COMPUTATIONS

2.2.1 Examples of Applications

As stated in Chapter 1, future real-time systems will need to exhibit adaptivity and intelligence in response to the highly dynamic and unpredictable environments in which they operate. They will also be required to provide a flexible and robust response in the event of system overload or failure. Thirdly, they will often be embedded systems, where constraints in size, weight, or cost, dictate that adaptive performance is required from a system of limited capacity. There follow examples of systems which have these requirements.

2.2.2 Autonomous Vehicle Control System

An Autonomous Vehicle Control System must be safe, reliable and adaptive. Research detailed in [24] indicates that in order to automate the 'driving function' there is a need for the following capabilities:

- accurate and timely sensing of other vehicles and obstacles
- vision and scene interpretation
- real-time decision making
- route and path planning
- communication co-operation with other vehicles

Implicit in the above requirements are a range of real-time constraints. Within a time frame of 10-100 seconds, the system must plan and update routes to reach the assigned destination, whilst taking into account traffic conditions, and minimising fuel consumption and journey time. Within a time frame of approximately 1 second, the system must recognise scenes, assess other vehicles movements, and plan a path which ensures that the vehicle can steer a safe course. Within a time frame of less than 1 second, the system needs to sample sensors and detect, and avoid, possible collisions with obstructions or other vehicles.

In addition, the Autonomous Vehicle Control System may be required to operate in a variety of modes:

- fully autonomous mode
- co-pilot mode: the human driver can intervene and take control
- monitoring, display and alert mode: as an aid to the human driver

Furthermore, such a system must perform safely at all times, and must retain reliability while under overload or failure. A method of achieving graceful degradation in such a system is to distribute functionality throughout a number of nodes connected in a common network architecture. Obviously the mission and safety critical tasks must be guaranteed to execute on their host nodes, within their deadlines, and to provide results of a minimum acceptable quality. In addition to this, optional computations may be used to enhance system utility by increasing the frequency, timeliness, precision or confidence level of the results which are produced.

2.2.3 Radar Tracking

Cheong [6] provides another example of an application which requires a mix of mandatory and optional computations. In radar tracking, a sensor returns signals from a tracked target and the system produces estimates of the target's position, velocity and acceleration. When the periodic task providing the estimates is terminated prematurely, it produces coarse estimates of the targets parameters. It is critical to the continuation of the tracking that a precise measure of the targets position, velocity and acceleration is generated at some longer interval. If this does not occur, then errors generated by coarse estimates accumulate beyond a maximum threshold of acceptability.

Cheong points out that the requirements may be satisfied by a mixture of mandatory and optional computations. Before accumulated errors exceed their threshold values, a mandatory computation must execute in order to renew the precision of the estimates. After the mandatory computation has executed, then optional computations may run. Each optional computation will provide a coarse estimate if it is not allowed to complete. However, if an optional computation does complete, and precise measurements are produced, then a future mandatory computation can be postponed.

2.2.4 Summary

The above applications, and others such as robotics [40] and advanced avionics [37], provide evidence for the requirement for a mixture of mandatory and optional computations. There is also evidence that optional computations may require to change their timing requirements at run-time. For example, unpredictable changes in the environment of the system, or faults within the system itself, can cause optional computations to change their execution time. Deadlines may change from one invocation to the next. Optional computations may even vary their frequency of execution, according to the rate of change of an input from the environment.

2.3 EXISTING MODELS FOR OPTIONAL COMPUTATIONS

The following sections review existing techniques for optional computations. These include computational models, programming paradigms, and scheduling strategies.

2.3.1 Imprecise Computation

Imprecise computation is a paradigm for programming optional computations which comes under the heading of techniques of *iterative refinement*. *Sieve functions* also refine intermediate results. Alternative paradigms to these are provided by *multiple versions* and *approximate processing*. All of these techniques are discussed in Section 1.3 above. However imprecise computation is now covered in more detail.

The model of imprecise computation is mainly due to Liu et al [34]. The technique is based upon the assumption that a real-time task monotonically increases the quality of its results as it is given more time to execute within its deadline. The imprecise computation can be divided into a mandatory component, which is executed first, and produces results of the minimum quality which is acceptable to the application. Subsequent iterations of the algorithm can be implemented by optional computations which improve upon the minimum quality. After each iteration a new (and higher quality) intermediate result is recorded. Scheduling within the system will determine how many iterations are performed before the deadline for the imprecise computation is reached.

Error indicators may be used as a measure of the quality of the result, and to establish whether the result which is finally produced, is acceptable. Liu et al use various measures of the errors produced by imprecise computations whose iterations are aborted. For example Liu et al. [7] use the average error produced by computations and Shih [45] uses the number of iterations which are discarded when they are aborted. Much of the work of Liu et al concerns the development of heuristics which minimise the total error for all imprecise computations across the system [7].

Imprecise computation can be used in a variety of applications including numerical computation, statistical estimation and prediction, heuristic searches, and database query processing.

An interesting extension to the model of imprecise computation is the concept of *conditional performance profiles* which is due to Zilberstein [64]. Here the quality of the result of an imprecise computation depends not only on the length of time it has run, but also on the quality of the input data. This implies a trade-off between the computation time allowed and the improvement on the quality of the input data which can be achieved. For example, in a composite task whose components pass on data, one to the other, there may be an optimal way in which the composite task's total budget can be distributed over the components in order to optimise the improvement in the final output data.

2.3.2 Computational Models for Real-time AI Applications

Requirements

According to Yen and Natarajan [62] there are several important differences between the requirements for *real-time AI* and those of conventional real-time systems:

- greater unpredictability in the timing of AI components e.g. the time taken for tree searches can vary widely.

- very pessimistic worst-case performance which it is impractical to build into the system e.g. depth of searches can be very great in the worst case.

- the time granularity of real-time AI techniques are typically larger than conventional real-time systems e.g. the order of seconds rather than milliseconds.

This has led to the development of the *anytime algorithm* which is the counterpart of imprecise computation in the AI community. An anytime algorithm iterates, and monotonically increases the quality of its result as further iterations proceed. The algorithm can be cut short at anytime, and still give a result of a certain quality. Quality can be measured by extra precision, confidence in the result, completeness of the result, etc. With anytime algorithms there is typically a trade-off between the time and resources used in the computation and the quality of the result produced.

Task Hierarchies

Garvey and Lesser [17] describe the requirements for real-time AI computations in terms of a complex task hierarchy and the AI *methods* which group tasks or subtasks together. They use the concept of *satisficing* which involves finding a solution which is acceptable, but not optimal, given the time and resources available. They broadly define two techniques for satisficing: *iterative refinement* and *multiple methods*. These have much in common with the definitions of iterative refinement and multiple versions given above. However multiple methods is a more complex concept than multiple versions because several *methods* may execute concurrently, and may share intermediate results. For example, a computationally expensive method may be aborted, but the intermediate result which it produced may be used by a less expensive method, which has also been executing. A further complexity is that, in the real-time AI context, the techniques of iterative refinement and multiple methods must often be represented in complex task hierarchies.

In the task hierarchies presented by Garvey and Lesser [17], *task groups* are independent solutions with their own deadlines. Within task groups, tasks are *interdependent* and can be subdivided into subtasks which themselves can be subdivided etc. At the lowest level of the hierarchy are *executable methods* which are the smallest schedulable units of work. For each task in the structure, there may be multiple sets of subtasks which may be combined to "solve the task". Each of these sets is known as a *method* for solving the task. Clearly the overheads in supporting flexible scheduling for such a task model could be very high.

Utilities within Hierarchies

Yen and Natarajan [62] also describe the need for a task/subtask hierarchy, and consider the problems of decomposing imprecise computation down to a subtask level. However, these authors also develop a decision theoretic framework for computations. Essentially, rules are applied in order to decide which combinations of tasks/subtasks should be allocated processor time and other resources within the system. Tasks/subtasks are allocated resources according to their *utilities*, and some overall rule about which allocation is likely to gain maximum utility for the system.

Allocation proceeds according to *decisions* which are expressed formally and are compiled into the implementation. This has the double benefit of allowing developers to reason about the application, and also permitting analysis of the performance of the implementation. *Decisions* can be complex and can involve the probabilities of tasks producing results of acceptable quality, using the resources available e.g. the aggregation of the individual probabilities that a set of subtasks may complete, with a certain quality of result, within a certain time. In order to provide a consistent set of task/subtask utilities within a hierarchy, the authors present a system of (de)composing utilities within a hierarchy.

Yen and Natarajan's decision theoretic treatment provides a powerful framework with great flexibility. According to the authors the decisions themselves take up few resources. However this claim is not substantiated in the paper, and seems to need further justification.

2.3.3 Locke's Value Functions and Utilities

Locke [35] argues for the use of *value functions* by the real-time applications programmer. A value function gives the curve obtained by plotting the value to the system of the completion of a process, against a time axis which represents the possible completion times of the process. Locke goes on to describe scheduling algorithms which use the value functions of the process set, to construct a schedule which maximises the total value obtained from all processes. Locke's value functions can be parameterised. Parameters which could be relevant in a complex application might be the system state, the states of the task itself, the input data to the task, or the state of other tasks within the system. Dynamic parameters such as these can be useful in complex real-time systems, for example those which incorporate AI into real-time applications. However, the run-time support for such dynamic value functions could prove very costly.

A different approach to the characterisation of the value of each task is to use *utilities* [62]. In contrast to Locke's value functions, utilities need not be associated with particular completion times, but represent some numeric value which is gained by the system when the task completes *within its deadline*. The utility associated with a task may be fixed and statically allocated, or it may vary dynamically. For example, dynamic changes in utility may be of use in a fault tolerant system where the utility of a replicated module may decrease if a replicant module completes. Conversely, the utility of the replicated module may increase if the replicant fails. A more sophisticated approach is to model a task as a composition of subtasks each of which may have a different utility associated with it. Hence the utility of the task varies according to the point it has reached in its execution.

It is possible to define either the utility or value function of a task in terms of those already defined for other tasks. For example, Locke [35] shows how a (dynamic) definition of the value function of task may be made by adding the value functions of two other tasks, each weighted with coefficients.

2.3.4 The Spring Model

The requirements for real-time systems assumed by the Spring Project have been outlined by Stankovic and Ramamritham [52]. They assume that the real-time system is a distributed set of nodes which exists in a highly dynamic environment. Nodes are multiprocessor clusters which primarily serve a particular location within the distributed system.

The researchers define three types of tasks within a system. *Critical* tasks have their hard deadlines and resource requirements guaranteed before run-time by worst-case analysis. *Essential* tasks have firm deadlines, so that there is a loss of value, but no catastrophic consequences, to the system if their deadlines are not met. There are assumed to be many more essential tasks than critical tasks. Because it is too pessimistic to reserve full resources for all essential tasks before run-time, these tasks are guaranteed at run-time by a *guarantee algorithm*. If the guarantee algorithm rejects the essential task on one node, then an attempt may be made to guarantee on another node of the system.

Non-essential tasks are the third category. They may have soft deadlines or no deadlines at all, and they execute in such a way as to have no impact on the other categories of tasks.

Spring considers many general requirements for tasks. Tasks may be preemptable or non-preemptable, periodic or aperiodic, have a variety of resource constraints, and may have precedence and communication constraints. Spring integrates the scheduling of tasks with these various requirements by using sophisticated guarantee algorithms which attempt to produce a feasible schedule for all the tasks on an applications processor. However, the requirements for adaptivity within the distributed system are met by higher level *decentralised scheduling* in which nodes can *co-operate* in order to guarantee essential tasks.

Some work has been done [52] to extend the Spring project into support for real-time AI applications. Spring workers envisage the following requirements being supported by the Spring kernel:

- the ability to dynamically change the criticalness, timing requirements, resource needs, precedence constraints, and even the structure, of a computation.

- the ability to plan future execution times of functions that may subsequently need to be re-planned.

- the ability to perform trade-off analyses (on-line).

- the ability to respond to an application program with appropriate system information.

In order to do this, large extensions are needed to the data held in, and the algorithms used by, the Spring kernel.

Sections 2.5 and 2.6 below present a detailed review of the existing run-time support provided by the Spring kernel.

2.3.5 Summary

The material reviewed in this section suggests the need for a complex computational model embracing complex interdependencies between tasks such as task/subtask hierarchies, precedence relations and intercommunication dependencies. Dynamic value functions or utilities would also be required, and would need to be (de)composed within the task hierarchy. Such value functions would depend on parameters such as the system state, the states of the task itself, the input data to the task, or the state of other tasks within the system. Simple paradigms such as Imprecise Computations, Sieve Functions, etc. would be subsumed under a more powerful, general model. Distribution of tasks, and resource allocation, would also be supported within the model. The model would also have to incorporate great flexibility, allowing dynamic changes in planned schedules on the basis of known probabilities of task, or system, behaviour. All this would be required, without adversely affecting the *a priori* guarantees given to mandatory computations.

Clearly such a model would be extremely expensive in terms of run-time support.

2.4 EXISTING RUN-TIME SUPPORT FOR FLEXIBLE SCHEDULING

According to many of the models reviewed above, real-time systems consist of a set of periodic, mandatory computations which are resident on a processor, plus aperiodic, optional computations which may arise locally or via a request from a remote node. The conventional approach is to schedulability test the set of mandatory computations before run-time, while flexibly scheduling optional computations at run-time, and even guaranteeing their firm deadlines. The following section reviews methods for flexible scheduling which optimise response times, or throughput, of aperiodic computations with soft deadlines, but fall short of guaranteeing deadlines.

2.4.1 Methods for Optimising Response Time of Soft Tasks

Background and Polling Server

The problem of scheduling soft tasks on a processor which runs its own set of resident periodic tasks with hard deadlines, has been tackled at various levels of sophistication. In background processing, soft tasks are assigned priority levels below those of the hard tasks. This means that soft tasks may have very long response times when the processing demands of hard tasks are high.

Soft task response times may be reduced by the use of a polling server [43]. This is a periodic task with a fixed, high priority whose capacity is set, pre run-time, at a level which allows all hard tasks to meet their deadlines. The polling server is released periodically and during its execution, its capacity is available for aperiodic tasks. The capacity is replenished at the server's next release.

The problem with the polling server is that it does not preserve its capacity. After release, its capacity is spent whether or not there are aperiodic tasks pending. Aperiodic tasks which arrive after the capacity is spent, must wait until the next release of the server until they can execute. Nevertheless the polling server improves upon the response times provided by background processing. However the server's inflexibility, leads to longer response times than for the improved methods which are described below.

2.4.2 Bandwidth Preserving Algorithms

Deferrable Server

The deferrable server [30] also makes use of a high priority periodic server task. However it is able to preserve its capacity when there are no aperiodic tasks pending. It therefore preserves its bandwidth throughout its period. This reduces the average response times of soft tasks to below that of the polling server.

The deferrable server discards any remaining capacity at the end of its period, and then immediately replenishes its capacity for the next period. The fact that the deferrable server preserves unused capacity at a high priority affects the static analysis of the maximum capacity which the server may be allocated. Because the deferrable server can produce back-to-back interference on lower priority hard tasks, its capacity must be smaller than an equivalent polling server. Nevertheless the bandwidth preservation of the deferrable server leads to smaller average response times than the polling server.

Priority Exchange Algorithm

The priority exchange algorithm [30] also uses a high priority periodic server to provide capacity for aperiodic tasks. However the priority of the server is not fixed, but decreases during its period. When no aperiodic tasks are pending the server exchanges its higher priority with the highest priority runnable hard task. The servers capacity is then converted to guaranteed execution time at the lower priority of the hard task. As priority exchange proceeds, capacity may be accumulated at low priority levels. This capacity is not discarded at the end of the servers period, but may be carried over into subsequent periods. The high priority capacity is still replenished at the start of every period.

The priority exchange protocol has as high a capacity as the polling server but also preserves bandwidth like the deferrable server. It does however suffer from the disadvantage that, under overload conditions, soft deadlines are missed in an unpredictable manner.

Sporadic Server

This algorithm attempts to combine the advantages of both the deferrable server and the priority exchange algorithms. Like the deferrable server, it maintains capacity at the original priority, but its capacity is equal to that of priority exchange or polling. A high priority periodic task is used, but instead of being replenished every period, it can be replenished at some earlier time, after higher priority tasks have executed. The capacity of the sporadic server has been shown to be comparable to that of a polling server [44] while early replenishment of capacity can allow a lower response time than the previous methods. Because the server task keeps its high priority, the sporadic server misses deadlines predictably under overload.

Extended Priority Exchange Algorithm

The Extended Priority Exchange Algorithm [49] is an extension to the Priority Exchange algorithm. It has the advantage that it reclaims gain time i.e. time made available when a hard task completes in less than its worst-case execution time (WCET). It replenishes capacity at a particular priority level each time a hard task is released at that level. Furthermore, if the hard task completes in less than its WCET, the gain time is added to the capacity available at that priority level.

2.4.3 Slack Stealing

This algorithm is due to Lehoczky and Thuel [27] and it is optimal in that all spare processing time is made available to soft tasks, as soon as possible, and at the highest priority level. The algorithm depends on the availability of a statically derived schedule of the hard periodic tasks over the complete LCM of their periods. At run-time, counters are used to keep track of slack at each priority level. After the completion of a hard task, the slack at that priority level is incremented according to data on slack time in the static schedule. Slack counters are also decremented when hard or soft tasks run.

Slack stealing has the limitation that it cannot work for sporadic tasks, or tasks which suffer release jitter. It also imposes the overhead of holding a schedule which is the length of the LCM of the periods of the hard tasks. Further work on slack stealing due to Davis [8] is reviewed in Section 6.2.2.

2.4.4 Summary

While the above methods preserve and allocate spare capacity for soft aperiodic computations, they do not guarantee that there is sufficient capacity available in order to meet an aperiodic task with a *firm* deadline. Static schedulability analyses can be applied to the above methods in order to guarantee the deadlines of aperiodic tasks off-line. However, this approach is pessimistic, in that run-time capacity has to be reserved on the processor, regardless of whether the aperiodic task arrives at its maximum rate or not. What is really required are *on-line* schedulability tests for aperiodic tasks with firm deadlines. These would either guarantee that sufficient capacity is available within the specified deadline, or reject the aperiodic task so that some alternative action may be taken. The following section reviews such dynamic guarantee algorithms.

2.5 THE SPRING PROJECT

The Spring Project [52] considers real-time systems which are physically distributed and consist of a network of nodes which are multiprocessors. Each node consist of one or more applications processors, and one or more system processors. The Spring kernel includes guarantee algorithms, and algorithms for co-operative scheduling between nodes. It runs on the systems processor(s) which frees up the applications processor(s) to simply dispatch applications tasks according to a schedule constructed by the system processor(s). The following sections describe the guarantee algorithms and the distributed scheduling algorithms which are used in Spring.

2.5.1 Spring Guarantee Algorithms

Spring guarantee algorithms [53] are aimed at guaranteeing newly arrived aperiodic tasks alongside resident periodic tasks plus any aperiodic tasks which have already been guaranteed. The algorithms take into account many task characteristics including the arrival time of the aperiodic task, its WCET, and its deadline. Other characteristics which can be included are: what resources are required by the task, whether these are required in shared or exclusive mode, whether the task is pre-emptive, and whether there are precedence constraints between tasks. To guarantee all such requirements is, in general, NP-hard [52].

Therefore Spring attempts to guarantee by using heuristics to facilitate a search for a feasible schedule.

The Spring guarantee algorithm starts at the root of a search tree which represents an empty schedule. It then tries to extend the schedule by moving to one of the vertices at the next level of the search tree, and so on, until a full feasible schedule is determined. A heuristic function is applied individually to some or all of the tasks which remain to be scheduled at each level of the search. The task with the smallest value of the heuristic function is chosen to extend the current schedule. As the (partial) schedule is extended, the algorithm determines whether it is *strongly feasible* or not. A partial schedule is strongly feasible if *all* of the schedules obtained by extending the schedule, with any of the remaining tasks, are also feasible. Once a partial schedule is found *not* to be strongly feasible (e.g. when it is extended and the added task misses its deadline) then the search is aborted along that particular branch of the search tree. The algorithm then backtracks and extends the partial schedule by a different task. The search continues until either a full schedule is determined, or the number of evaluations of the heuristic function reach an upper bound, set by the system. This upper bound ensures that the systems processor has sufficient time to perform its other activities.

2.5.2 Complexity of the Algorithms.

A Spring guarantee algorithm is applied to a list of the tasks to be scheduled which is sorted into order of increasing deadline. The insertion of the newly arrived aperiodic task into this list carries O(N) complexity (where N is the size of the task set). Spring researchers claim [53] that the complexity of the subsequent search for a full schedule is also O(N) because the heuristic function need only be applied to a small subset, *k*, of the full task list, N, each time a partial schedule is extended.

2.5.3 Heuristics

The heuristics investigated by the Spring researchers are divided into (i) simple and (ii) integrated heuristics. They take into account not only the timing requirements of tasks, such as their earliest start times, deadlines and WCETs, but also the earliest time that tasks can execute, due to the availability of the resource(s) which they require. Simple heuristics include *minimum deadline first*, *minimum processing time first,* and *minimum earliest start time first*. (*Minimum earliest start time first* is the latest time, chosen between the specified *earliest start time* for the task, and the earliest time at which its required resource(s) are available.)

Simulations carried out by Spring researchers proved more successful when *integrated heuristics* were used, with weightings applied to certain components. The most successful of these heuristics was *minimum deadline + minimum earliest start time*, where *minimum earliest start time* has a weighting applied to it. In general, this heuristic provided the best guide to the task most likely to extend a feasible (partial) schedule.

2.6 DISTRIBUTED SCHEDULING IN SPRING

2.6.1 The Distributed Algorithms

In the Spring Project, when tasks are not guaranteed locally as described above, methods of distributed scheduling are provided for the guarantee of tasks at other nodes in the system. The distributed scheduling algorithms which are investigated by Spring are *focused addressing, bidding* and the *flexible algorithm* [56]. In addition, two simpler algorithms are used as benchmarks: the *noncooperative algorithm* and the *random scheduling algorithm.*

In the *noncooperative algorithm* a task is rejected when it cannot be guaranteed locally, and no attempt is made to request its execution at other nodes. In the *random scheduling algorithm*, the local node which cannot guarantee the task, sends a request for the tasks execution to some other *randomly* selected node. Obviously this cheap method suffers from the disadvantage that there is only a random chance that the task will be schedulable at a randomly selected node.

2.6.2 Focused Addressing, Bidding and the Flexible Algorithm

Focused addressing, bidding and the *flexible algorithm*, each use information about the availability of time and resources on remote nodes, in order to decide where to send requests for the guarantee of tasks which were failed locally. Each node in the system periodically calculates its *node surplus* and sends this data to a subset of the nodes in the system. A *node surplus* is a vector, with one entry per resource on the node. Each entry indicates the total amount of time, within a recent window, during which the resource was not used by local tasks. Each node also holds a list of remote nodes, ranked according to how many requests from them have been guaranteed locally during the recent time window. Each node sends its *node surplus* to a subset of the nodes held in its ranked list. Obviously this targeting of information is intended to provide data only to those remote nodes which have successfully forwarded aperiodic tasks in the recent past. This reduces the exchange of information across the network.

The three algorithms differ in the way they use information from remote nodes, in order to select a remote node for a request for guarantee. The *focused addressing algorithm* determines the remote node with the highest surplus of time and resources required by the aperiodic task which has failed the local guarantee. If this surplus is greater than the *focused addressing surplus* (a tuneable system parameter) then the request is immediately sent to the chosen remote node. If no node exists whose surplus exceeds the *focused addressing surplus*, then the aperiodic task is rejected.

The *bidding algorithm*, is a more expensive algorithm which makes a more sophisticated decision regarding which remote node to choose for a likely guarantee. The local node which has failed to guarantee the aperiodic task, selects k nodes with sufficient surplus in the resources needed to guarantee the aperiodic task. (The value of k is chosen to maximise the chances of finding an appropriate node for the aperiodic task.) A request-for-bid message is sent to each of the k nodes. When a node receives a request-for-bide message, it calculates a bid, which indicates the likelihood that the aperiodic task can be guaranteed by it. If the node's bid is higher than a pre-set minimum level, then the bid is sent to the requesting node. After receiving the bids, the requesting node sends the aperiodic task to the node which has offered the highest bid. If no acceptable bids are forthcoming, then it is assumed that the aperiodic task cannot be guaranteed within the system.

The *flexible algorithm* is a combination of focused addressing and bidding, intended to achieve 'the best of both worlds' at the expense of more processing at nodes and more communications over the network. First, focused addressing is used to select a focused node, to which the aperiodic task is immediately sent. (This is done according to the same proviso that the *focused addressing surplus* must be exceeded by the surplus on the focused node.) The $k -1$ nodes remaining are then sent request-for-bid messages along with the identity of the focused node. The $k - 1$ nodes then calculate their bids and send them to the focused node.

In parallel, the focused node, which has received the aperiodic task, attempts to guarantee it. If the guarantee is successful then all the bids which are received from the $k - 1$ nodes are ignored. If not, then the focused node sends the task to the highest bidder. If there are no acceptable bids then the task is rejected. (A message about whether and where the task has been guaranteed is sent to the original node so that it can update its information on other nodes.)

In the case where no nodes are eligible to be the focused node, then the *flexible algorithm* defaults to *bidding* where bids are returned to the original node.

2.6.3 Summary of Spring

(i) The *guarantee algorithms* developed in the Spring Project enhance system performance as measured by the *guarantee ratio* at each node. (Guarantee ratio is defined as the proportion of the aperiodic tasks arrivals at a node which are guaranteed by that node.) However, the guarantee algorithms were found to incur considerable overheads, with the result that Spring researchers have designed a hardware coprocessor specifically to perform guarantees [39]. The approach of this thesis is to avoid the use of dedicated or specialist hardware and to implement guarantee algorithms on the same processor which runs the applications tasks. Therefore, less computationally intensive methods must be sought in order to provide a schedulability test for aperiodic tasks, and also considers their resource usage. One line of approach is to adapt for on-line use, the *static* schedulability testing algorithms of Audsley et al. [2]. These algorithms assume that a concurrency control protocol such as *priority ceiling protocol* allows upper bounds to be placed on blocking caused by exclusive access to resources. The range of static schedulability tests developed by Audsley et al. are reviewed in Section 2.7.

(ii) The Spring simulation results show that, in general, *distributed scheduling* improves the throughput of aperiodic tasks with firm deadlines [56]. For example, the *flexible algorithm* was found to outperform the *noncooperative algorithm* under all load distributions. The *flexible algorithm* also outperformed both *bidding* and *focused addressing,* under conditions of average communications delay across the network. However, these algorithms incur such large overheads that extra general-purpose system processors are required in order to support them [52]. This thesis will investigate the development of less expensive but equally effective methods, which do not require dedicated hardware, but nevertheless serve to direct aperiodic tasks to the processor most likely to guarantee them.

2.7 ALGORITHMS FOR STATIC SCHEDULABILITY TESTING

In his thesis, Audsley [2] has reviewed the topic of *Static Schedulability Testing*. He goes on to develop an extensive analysis of Deadline Monotonic scheduling which generalises previous work on Rate Monotonic scheduling [32]. Using his analysis of Deadline Monotonic, Audsley presents a set of static schedulability test algorithms with a range of complexities. It is assumed that pre-emptive priority scheduling is used for a set of

N fixed priority tasks, which are listed in order of increasing, static deadline. The tasks are considered to be periodic, such as a set of critical tasks which are resident upon a processor, and must be guaranteed *a priori*. Audsley discusses four sufficient and not necessary algorithms which he refers to as Tests 1 to 4. He also presents a sufficient and necessary schedulability test which shall be referred to as PP on account of its pseudo-polynomial complexity.

2.7.1 Sufficient and Not Necessary Tests

The tasks in the task list are assumed to be ranked in priority order according to the deadline monotonic algorithm. The period *(T)*, deadline (*D*) and WCET (*C*) of each task are known. It is assumed that all tasks are released simultaneously (worst-case critical instant). If B is the worst-case blocking time a task may experience, due to the operation of some concurrency control protocol, and I is the worst-case interference a task may suffer from higher priority tasks, then for any task to be schedulable:

$$D \geq C + B + I \qquad (2.1)$$

Techniques for the determination of C and B are not given by Audsley except to say that C may be estimated during compilation, and B may be upper bounded by, for example, the use of the *priority ceiling protocol*. He presents four algorithms for the determination of I *for the duration of the deadline* of whichever task is being schedulability tested (known as the test task, i). This may include interference which does not occur during the *lapsed* execution time of the test task. Hence these tests are sufficient but not necessary. In general the list of higher priority tasks is scanned to provide the following sum which is the total interference from all higher priority tasks j:

$$\Sigma_j(\lceil D_i \div T_j \rceil C_j) \qquad (2.2)$$

Inequality (2.1) may then be used to test the schedulability of the test task.

Test 1 uses exactly the above procedure. Every task in the list takes a turn as the test task so the complexity of Test 1 is $O(N^2)$. Note that this test is pessimistic (sufficient and not necessary) since, depending where the deadline of the test task occurs, a final interference by a higher priority task j within the deadline may not be the full value of that task's computation time. Tests 2 to 4 use increasingly expensive methods in order to decrease the pessimism of this aspect of Test 1.

Test 2 uses the fact that the maximum interference from the final hit of an interfering task j with the test task i is given by:

$$\min (C_j, D_i - \lfloor D_i \div T_j \rfloor T_j) \qquad (2.3)$$

$D_i - \lfloor D_i \div T_j \rfloor T_j$ is the interval between the release time of the final hit of an interfering task and the test task's deadline. If this interval is less than the value of the WCET (C_j) for the interfering task, then the worst-case final interference of the interfering task can be taken as this interval, rather than the full WCET of the interfering task. Therefore, in some cases, Test 2 is able to make a less pessimistic estimate of the final interference of the higher priority task within the deadline of the test task. The complexity of Test 2 is still $O(N^2)$ although the extra comparison above will impose a further overhead.

Test 3 also uses (2.3) in an attempt to find a lower bound on final interferences. However, when considering interferences within the test task, i, by a *jth* higher priority task, Test 3 uses the fact that, if $D_i - \lfloor D_i \div T_j \rfloor T_j \leq C_j$ then $D_i - \lfloor D_i \div T_j \rfloor T_j$ may be subtracted from D_i in order to reduce the deadline of the test task to an *effective deadline*. The next task (j + *1th*) which is considered for interference in the test task, now has its interferences calculated as occurring within the effective deadline established by the *jth* task. By definition, the interval by which the test task deadline has been reduced cannot contain interferences form the $j + 1$, $j + 2$, etc tasks because the interference from the *jth* task will cause $j + 1$, $j + 2$, etc to execute later. Therefore, by using the effective deadline, concurrent (overlapping) interferences from $j + 1$, $j + 2$, etc are not included as interferences within the test task deadline, and this reduces the pessimism of the schedulability test. In turn the $j + 1$, $j + 2$, etc tasks may provide further reductions to the effective deadline. However, Test 3 is still a sufficient but not necessary test because, in general, concurrent interferences may still be counted when considering the final hits of interfering tasks. For example, interfering tasks may have final hits which are released slightly earlier than the interval of their WCET from the test task deadline. In this case no reductions can be made to the effective deadline, and concurrent interferences will be counted within the test task deadline.

Test 4 applies effective deadline reductions in the same way as Test 3, except that it further reduces the possibility of overlap by reiterating through all interfering tasks in deadline monotonic order until the effective deadline can be reduced no longer. This means that if a *jth* task just misses a reduction in effective deadline at the first iteration, and the j + *1th* task is subsequently able to reduce the effective deadline, then the next iteration through the interfering tasks may allow a reduction in the effective deadline at the *jth* task. However, it is still possible that concurrent interferences may be counted. For example, overlapping interferences may fall just short of the best effective deadline established by repeated iteration. Therefore Test 4 is also a sufficient and not necessary test. Audsley gives its complexity as pseudo-polynomial [2].

2.7.2 A Sufficient and Necessary Test

Unlike the above algorithms, this algorithm is sufficient *and necessary*. It has a pseudo-polynomial complexity and will therefore be referred to as *PP*. It accurately calculates the total interference of all higher priority tasks *during the course of the test task's execution*. In effect, it calculates the exact *response time* of the test task, under the assumption that all higher priority tasks perform their worst-case executions. The algorithm proceeds by repeatedly increasing the test task's window (w_i) in which higher priority tasks interfere. At each iteration the following sum over all higher priority tasks j is calculated:

$$\Sigma_j(\lceil w_i \div T_j \rceil C_j) \qquad (2.4)$$

The initial value of the window is the WCET of the test task. The window size at the next iteration will be the value of sum (2.4) from the last iteration. And so on, until the window size does not increase. Audsley [2] shows that the algorithm will converge if processor utilisation is less than 100%. This convergence yields the total interference required. As before, the algorithm is repeated for all tasks in the list. Because the tasks are tested in deadline monotonic order, the algorithm can be speeded up by using the final I value obtained for the *ith* test task as the initial value of w for the *i+1th* test task, etc. Audsley shows that the algorithm is pseudo-polynomial and points out that any particular test task deadline (assumed to be an integer number of ticks) will provide an upper bound on the number of iterations required.

2.8 LANGUAGE SUPPORT FOR OPTIONAL COMPUTATIONS

Existing programming language support for optional computations and flexible scheduling is confined to experimental languages, and non-standard extensions to existing languages. Experimental languages may not be fully implemented and non-standard extensions to existing languages may be *ad hoc*, and fail to provide full support. According to this thesis, it is not sufficient for a programming language to merely support the *scheduling* of optional computations but there should also be support for the on-line *guarantee* of firm deadlines. In the following section, two experimental languages (Flex and Real-time Concurrent C) and two standardised languages (PEARL and Ada) are reviewed.

2.8.1 Flex

The experimental programming language Flex [25,26] is a derivative of C++, which is designed to support optional computations in the form of imprecise computation and multiple versions. The language uses RTL-type notation to specify task constraints, and uses object-oriented concepts of *polymorphism* and *late binding* in order to program flexible scheduling. Tools for the static and dynamic analysis of Flex programs have also been developed.

In Flex, the constraints on the timing, and resources used by sections of code are defined by a *constraint block*. Temporal constraints include the start and finish times for the code, intervals for periodic tasks, and the earliest and latest times for events. Exceptions are defined in the case of any constraint failing to be met. It is important to realise that Flex does not provide an on-line *guarantee* of constraints, but rather optimises the chances of constraints being met.

Flex supports imprecise computation, and also the maximising of the values gained by imprecise computations within the application. Multiple version programming is also supported in the form of *performance polymorphism*. This is the temporal counterpart of *polymorphism,* as defined by the types of input parameters. Performance polymorphism allows the dynamic choice of one of several versions of a function. According to the values of the parameters passed in the call to the function, the resources which are available, and the execution time which is available for the function, the version which has the highest chance of generating the highest utility, is chosen. An example given by Kenny and Lin [26] is that of a *sort* function where one of several sort algorithms may be chosen according to the system resources available, or the data which needs to be sorted. The authors acknowledge that the overheads for performance polymorphism can be large, and that this requires the versions themselves to have relatively large execution times.

Flex programs can contain *pragmas* which allow the applications code to interact with tools. These analytical tools can (i) measure on-line, and statistically analyse, the performance of different versions (ii) determine the parameters which influence the performance of each version (iii) provide a static analysis of different versions which can be used to optimise future compilation.

The main advantage of the Flex programming system is that it provides tool support for detailed analysis of the performance of optional computations which can aid in optimising the performance of future runs of an application. However, Flex does not provide the support for the *dynamic guarantee* of optional computations which is sought by this thesis.

2.8.2 Real-Time Concurrent C

In contrast to Flex, *Real-time Concurrent C* [19] does provide support for the guarantee of optional computations, and even for time constraints upon the guaranteeing itself. Real-Time Concurrent C is based on *Concurrent C* [18] which supports processes with synchronous and asynchronous communications. Real-Time Concurrent C extends this by allowing processes to (i) execute sections of code with specified periodicity or deadline constraints (ii) seek guarantees that such timing constraints will be met and (iii) perform alternative actions when either the timing constraints cannot be met, or the guarantees are not available.

The designers of Real-Time Concurrent C acknowledge their debt [19] to the Spring Project which originated the model of attempted guarantee, followed by alternative action when the guarantee is denied. They describe a section of code within a process as an *activity*, and state that "an activity can be guaranteed to complete execution within its deadline if a schedule can be created for the activity, and also for other activities that have been previously guaranteed, such that all these activities will meet their timing constraints". If such a schedule cannot be created, then the new activity is not guaranteed.

The following is a review of the Real-Time Concurrent C constructs associated with the specification of time constraints for optional computations, and the guarantee of optional computations.

Activities with Deadlines

Deadliness can be associated with any activity or statement using the *within deadline* statement which has the form:

```
within deadline(d) statement1
[else statement2]
```

The semantics of the construct are, that if control reaches the *within deadline* statement at time t, if *statement1* is not executed before $t + d$, then its execution is terminated, and *statement2*, if supplied, is executed.

Periodic Activities

Periodic activities are reviewed here because, as shown later, they can be guaranteed in Real-Time Concurrent C. Periodic activities are specified using the *every* statement which has the form:

```
every (p) [until expression | until accept statement]
        statement1
[else stsememt2]
```

expression is a boolean condition. *statement1* repeatedly executes at interval *p*. However, at the start each period either (i) *expression* is evaluated or (ii) in the case of an *until accept statement*, any outstanding transaction is accepted. The *every* statement terminates when either (i) *expression* evaluates to true or (ii) a transaction has been accepted. Should an outstanding transaction take the form of an interrupt, then *statement1* can be aborted when the interrupt is raised, and *statement2* executed instead, followed by the termination of the *every* statement.

Guaranteed Activities

Real-Time Concurrent C provides the programmer with the facility to guarantee, before an activity starts, that it will complete before its deadline. The *guarantee* statement takes the form:

```
[within deadline (gd) ] guarantee
        time_constrained_statement
[else statement]
```

gd is a deadline for the guarantee itself, and *time_constrained_statement* is either an *every* or a *within deadline* statement. The run-time system attempts to determine, within *gd* if specified, whether or not *time_constrained_statement* can be guaranteed to complete within its time constraints. If the guarantee is not possible, or if it cannot be reached within *gd*, then the else *statement*, if provided, is executed.Otherwise *time_constrained_statement* is executed. There follow two examples of the use of the *guarantee* statement.

Example 1:

This is an example of the use of a *guarantee* statement which attempts to guarantee a *within deadline* statement:

```
within deadline (gd) guarantee
        within deadline (d) statement1
        [else ;]
[else statement2]
```

statement2 is executed if it is not possible to give the guarantee by *gd*. If the guarantee is given, then *statement1* will be executed within *d*.

Example 2:

This example shows the use of the *guarantee* statement with the *every* statement:

```
within deadline (gd) guarantee
        every (p) [until condition] statement1
        [else ;]
[else statement2]
```

The above attempts to guarantee that *statement1* will execute at every interval, *p*. The guarantee is performed once, before the first iteration of the loop. As the description of *every* semantics above would indicate, the *every* statement can still terminate if *condition* becomes satisfied.

Flexible Time Constraints

It is planned to add to Real-Time Concurrent C constructs, the capability of using multiple time constraints. This is useful, for example, in applications where there is still some value in completing a computation after a first (preferred) deadline. To introduce such flexibility, the designers of Real-Time Concurrent C plan to allow a *slop* to be associated with a deadline. This provides an additional period of time, after the preferred deadline, during which the activity should be allowed to continue because it may still provide value to the system. After the expiry of this extra time, the activity should be terminated, if it has not already completed.

```
within deadline (d) [(slop)] statement1
[else statement2]
```

The construct has the following semantics. Note that the semantics cater for negative values for slops, which are an equivalent way of expressing a preference between two deadlines:

- If *statement1* is not completed by *max(d, d + slop)*, the processing of *statement1* is terminated and *statement2*, if provided, is executed.
- If *slop* is not specified, it is assumed to be zero.
- If a guarantee is requested, the guarantee algorithm will first attempt to guarantee *statement1* with respect to *min(d, d + slop)* and if unsuccessful, will attempt to guarantee with respect to *max(d, d + slop)*. If the latter attempt is also unsuccessful then the *else* clause, if specified, is executed.
- A time-constrained component of *statement1* can also have a slop which can increase the WCET if *statement1*.

Run-Time Support for Real-Time Concurrent C

It is worth noting that there are five separate algorithms which are required to support flexible scheduling in Real-Time Concurrent C. Time on the processor is partitioned into slots, each of which is divided, in a fixed ratio, between time for periodic activities and time for aperiodic activities. Within each slot 'fraction' (periodic and aperiodic), the scheduler operates according to the following priorities:

- time for guaranteed activities is allocated first
- non-guaranteed activities with time constraints take preference in the remaining time
- non-guaranteed activities without time constraints use what time is left.

According to the latest published work [19], the implementation of run-time support for Real-Time Concurrent C is incomplete. No figures are given for the overheads associated with the five algorithms above.

In conclusion, it seems that Real-Time Concurrent C provides many useful constructs for optional computations, but that the language is not yet fully supported, and may require specialised hardware support to make it viable. It has certainly not been established as a standard.

2.8.3 PEARL

PEARL stands for "Process and Experiment Automation Real-time Language". The language was developed under the West German Ministry of Research and Technology as a real-time programming language for process control applications [61]. The language has been widely used by German industry and several versions of it have been embodied in German (DIN), and ISO standards.

Basic PEARL [14] provides a Pascal-like language with data types for *clock* and *duration*. *Full PEARL* [15] provides separate compilation of modules which are split into hardware-independent and hardware-dependent *divisions*. *Multiprocessor PEARL* [16] is a version of the language which allows the programming of distributed applications in which collections of modules may be configured and reconfigured within a network.

Halang and Stoyenko [21,22] propose extensions to *Full PEARL* which would constitute a new standard called *High Integrity PEARL*. Their proposed standard would make PEARL programs fully analysable for schedulability. They have developed a schedulability analyser for High Integrity PEARL, which works in conjunction with their High Integrity PEARL compiler. Their proposed features for High Integrity PEARL include the ability to program the detection of events, parallel processing with precedence relations within task sets, and greater programmer control of resources and tasks, especially under transient overload.

Greater task control is achieved by the availability to the programmer (and to the run-time system) of the deadline, accumulated execution time and (worst-case) residual execution time of each task. An *update* statement is available should a programmer wish to refine the estimate of the worst-case residual execution time of a task. This is possible when it is known at run-time, which particular path has been taken through the task code:

```
update task_identifier.residual := duration_expression;
```

According to Halang and Stoyenko, process control applications seldom have the monotone property required for imprecise computation [22]. They therefore restrict support for optional computations to a form of multiple version programming. In High Integrity PEARL, task declarations can include the attribute *runtime selectable* which means that the programmer is providing alternative task bodies for a task. The compiler calculates the WCETs of the alternatives and stores them in decreasing order of WCET, so that, at run-time, the scheduler can chose the alternative with the greatest WCET to be schedulable. The assumption is that alternatives with greater WCETs are preferable because they produce results of greater quality.

In summary, High Integrity PEARL provides support for multiple versions programming, but not for other paradigms for optional computation. The availability of run-time data on tasks (e.g. residual execution times) allows the programmer more control of scheduling and could, for example, facilitate scheduling according to *best effort*. However, the scheduling code for this would have to be explicitly written by the applications programmer. In short, High Integrity PEARL provides only partial support for optional computations, and at present exists only as a proposed standard [21].

2.8.4 Ada

Ada 83 was developed by the US Department of Defense [23]. It is a large imperative language which includes strong type checking, limited object-orientation in the form of derived types and generics, and features for concurrent programming using tasks. Tasks can communicate by means of the Ada *rendezvous* in which one task makes a synchronous call to an *entry* in another task.

Some of the criticisms of Ada 83 are that it has inadequate facilities for real-time programming. It has limited provision for expressing timing constraints, tasks have static priorities only, entry calls are always queued FIFO, and there are inadequate implementation standards for scheduling.

Ada 95 is a new standard for Ada [1], which addresses many of the shortcomings of Ada 83. Ada 95 has separate *annexes* for several application domains: *Real-Time Systems, Safety and Security, Distributed Systems,* and *Systems Programming.* Ada 95 introduces such features as *protected objects* for shared access to a resource, and the *requeue* of entry calls from one entry to another.

For the programming of real-time systems, Ada 95 provides improved clock facilities, dynamic priorities, and standardised scheduling of tasks which can be integrated with the scheduling scheme used for inter-task communication. Other new features which are useful for real-time applications include improved priority inheritance, and asynchronous transfer of control.

Optional computation is not directly supported by Ada 95. However, dynamic features such as asynchronous transfer of control may be able to be programmed to provide optional computation. This is the subject of Chapter 8, where the relevant Ada 95 language features are reviewed in more detail.

2.8.5 Summary

Current programming languages do not yet provide adequate support for optional computations. *Flex* only optimises the chances of optional computations meeting their deadlines. *Real-time Concurrent C* provides language constucts for optional computations, but it is an experimental language, and is not yet fully implemented. *PEARL* is a widely used language in which proposed extensions may provide limited support for some forms of optional computation. *Ada* has no direct support for optional computations but the new *Ada 95* standard contains flexible constructs which may allow optional computations to be programmed.

2.9 SUMMARY OF REVIEW

This review has surveyed previous work relating to the three strands of enquiry outlined in Chapter 1 i.e. application requirements for optional computations, run-time support for flexible scheduling, and programming language support for optional computations.

Section 2.2 reviewed the requirements for complex real-time applications which require adaptivity. Section 2.3 reviewed existing computational models, and programming paradigms. To merge all of the requirements, and combine computational models, would result in a large set of requirements supported by a complex computational model. This, in turn, would require support from a complex, and computationally expensive, run-time system. The motivation for Chapter 3, which follows, is to distil the requirements and develop a *constrained* computational model.

Sections 2.4 and 2.5 reviewed existing run-time support for flexible scheduling, and found that the *Spring Project* provides a high level of support, but at the expense of complex software and specialised hardware. Section 2.7 reviewed static schedulability tests due to Audsley et al. which Chapters 4 and 5 adapt, in an attempt to develop computationally cheaper guarantee algorithms than those of Spring. Section 2.6 above reviewed Spring support for distributed scheduling. This also requires complex algorithms, and Chapter 6 focuses on attempts to develop simpler methods. The constrained computational model of Chapter 3 uses Best Effort Admission Policy, which is evaluated by the simulation studies reported in Chapter 7.

Section 2.8 found limited programming language support for optional computations except for the language *Real-Time Concurrent C*. Unfortunately this language is experimental and not yet fully supported. In contrast, *Ada 95* is a new standard which does not explicitly support optional computations, but does provide constructs which may be used to do so. Chapter 8 investigates the use of these constructs in order to program optional computations in Ada 95.

CHAPTER 3

A CONSTRAINED COMPUTATIONAL MODEL

3.1 INTRODUCTION

As stated in Chapter 1, this thesis assumes that the Run-Time System required to support the application requirements for optional computations within adaptive real-time systems, will run on the same processor as the applications tasks. This chapter begins by citing evidence that the overheads incurred in the support of the complex application requirements which are reviewed in Chapter 2, would prohibitively reduce the throughput of optional computations. The chapter goes on to present a different approach by simplifying the complex application requirements, and formulating a constrained computational model, which, it is claimed, can be supported cost-effectively on the same processor as applications tasks. The chapter concludes by discussing some programming language constructs which optional computations may require, and which could be supported by the constrained computational model.

3.2 COMPLEX REQUIREMENTS

Complex requirements and models for adaptive real-time systems have been discussed in Sections 2.2 and 2.3. In the following sections, each of these requirements or models is considered in turn, and evidence is cited which shows that the provision of support for them would incur prohibitive overheads.

3.2.1 Value Functions

In his thesis [35] Locke used processes (tasks) with Value Functions. He found that his Best Effort algorithm achieved consistently high total values for the system, but he did not address the overheads which Best Effort scheduling incurs, except to suggest an architecture in which scheduling is performed on a different processor from applications tasks. Later work by Tokuda et al. [59] and Wendorf [60] investigated these overheads. Wendorf showed that with Best Effort scheduling running on the *same* processor as the applications tasks, the algorithm can incur very large overheads. For example, for a potential load of 200%, up to 80% of processor time was spent in Best Effort scheduling. This drastically reduced the time for application tasks to run, and therefore the total value obtained for the system. A further criticism is that Locke's Best Effort scheduling only increases the probability that tasks will meet their timing constraints, and this is insufficient when critical tasks are required to be guaranteed.

Davis et al. [12] present the results of simulations in which Best Effort scheduling has been adapted to guarantee tasks. In this algorithm, a task of a higher value density can oust a task of lower density from the task list. (Here value density is a constant value, associated with an aperiodic task upon its arrival. It is defined as the value to the system of the task, divided by the task's worst-case computational requirement.) Davis et al. found that a simple FCFS policy, which automatically rejects aperiodic tasks whose value-density falls below a threshold, can provide better performance than Best Effort, under conditions of system overload. This is because processing capacity is saved for later aperiodic tasks of greater value-density. This is evidence that superior performance can be obtained by a scheduling method which is simpler than the Best Effort algorithm.

3.2.2 Interdependencies between Tasks

Interdependencies between tasks have been discussed in Section 2.3.2. These can also greatly increase the overheads involved in scheduling. Communications between tasks, complex task hierarchies, and resource and precedence constraints may all greatly complicate the on-line schedulability analysis required to guarantee newly arrived optional computations. In the Spring Project [52,56] Ramamritham et al. investigate the complexities of constructing schedules for task sets which have resource, or precedence constraints. Because the construction of such schedules is, in general, NP-hard, Ramamritham et al. develop heuristics which are used to guide the search for a feasible schedule. As explained in Section 2.6.3, such heuristics were found to incur large overheads.

3.2.3 Schedulability Testing

A general concern with the facilities for optional computation discussed in Section 2.3 is that they may significantly increase the overheads incurred in scheduling and schedulability testing. For example, it has been shown [60] that value functions can incur unacceptable overheads. Audsley [2] has shown that algorithms which provide an exact schedulability test at run-time have a pseudo-polynomial complexity. Davis [8] presents results which show that the overheads incurred by pseudo-polynomial schedulability test algorithms, so reduce the throughput of aperiodic tasks, that an inexact algorithm can provide equal performance. If tasks are a composite of sections with different utilities, then overheads can be further increased. When value functions or composite utilities are able to be redefined at run-time, then overheads can be greater still. Concerns also exist regarding unbounded computations. For optional computations whose computation times are unbounded upon their arrival, guarantee is impossible. Further, an attempt to give them preference could undermine guarantees already given to other tasks.

3.2.4 Summary

The complex requirements for optional computations which have been discussed in Chapter 2 are likely to incur prohibitive overheads when implemented on the same processor as the applications tasks. Therefore the next step is to simplify these requirements, and to develop a constrained computational model which, on the one hand, incurs acceptable run-time overheads, but on the other, supports programming language constructs of adequate expressive power.

3.3 ATTEMPTS TO CONSTRAIN COMPLEX REQUIREMENTS

3.3.1 Constraining Value Functions and Utilities

As discussed above it is necessary to reduce the complexity of value functions, in order to achieve an acceptable overhead for the schedulability testing and scheduling of optional computations. In any case, it is arguable whether value functions as described by Locke are the most useful measure of the value to the system, of a task's execution. Locke's functions map the values obtained to the possible completion times of the task. However the value of completing the task could be represented more simply by a constant value or 'utility' which is set upon arrival of the aperiodic computation.

With a simple utility, the completion time of a task is still constrained, by the guarantee that the computation meets its deadline. The *exact* completion time of the aperiodic task may be immaterial, and the simple utility which has been allocated to the task is all that is required for the scheduler to make decisions regarding the task's execution. A task with an exact completion time which is *critical*, should not be an optional computation, but instead should be implemented as a high priority task which is resident on the processor and has been schedulability tested off-line.

3.3.2 Categories of Tasks

Having decided to use utilities and not value functions, there must be some way of trying to limit the range of utilities required, in order to constrain the complexity of a computational model, and to reduce the overheads for schedulability testing. For the purposes of a constrained model, three categories of task may be defined, with a separate utility level for each category. The first category (essential tasks) must complete, once guaranteed, the second category (atomic actions) may be aborted at any time between guarantee and the start of execution, and the third category (low utility) may be aborted at any time after guarantee.

The three categories may be said to define different *abortabilities* (the extent to which a guaranteed task may be aborted) and the question arises as to how these may be integrated with different utility levels for the tasks. For example, should a newly arrived aperiodic computation, of high utility, which would otherwise prove unschedulable, be able to abort existing tasks which fall into the second and third categories of abortability? If lower utility tasks may be aborted in this way, then how valid were their original guarantees? Presumably the application will need to know that guarantees have been rescinded, so that this may be handled. Obviously high-utility tasks cannot be aborted and the constraints which they impose might still force a newly arrived request to be rejected. In the case of the lowest utility tasks there may be no need to abort them if they are running in background, and have not been guaranteed.

3.3.3 Guarantee-worthiness

Clearly the utility and abortability of a task are interrelated. For example a task of high utility is more worthy of guarantee and should be less easy to abort. Conversely, there may be little justification for incurring overheads in guaranteeing an aperiodic request which has a low utility and may be aborted by higher utility requests. This raises the issue of the *guarantee-worthiness* of a request for optional computation. In other words can some measure be made of the trade-off between how much time is spent guaranteeing an aperiodic task and how much utility the aperiodic task gains for the system.

Obviously, when a high utility task is guaranteed, and ousts a previously guaranteed task of lower utility, then the utility of the ousted task is lost to the system. Not only that, but the time spent in guaranteeing the ousted task has also been lost. The question arises as to where the trade-off lies between time which is 'wasted' in this way and the higher utility which is gained. This question will be discussed further in the Section 3.5.1.

An important observation is that utility and abortability are intimately related in a scheme which guarantees aperiodic computations in a First-Come-First-Served order. Within this kind of scheme there will be no change to the overall value obtained for the system, unless aperiodic requests of different utilities also have different abortabilities. In other words tasks of different utilities will merely be treated in FCFS order unless incoming high-utility aperiodic tasks have the ability to cause previously accepted low-utility aperiodic tasks to be aborted.

3.3.4 The Problems of Composite Utilities/Abortabilities

One possible requirement for a computational model is that aperiodic requests might have different utilities or abortabilities associated with sections of their computations. For example an imprecise computation can be coded as a sequence of iterative sections, with each section having a lower utility and higher abortability. Another use for a change in utility would be to give the final section of a task a higher utility. This might prevent the task from being aborted just prior to completion, with the resulting loss of value to the system.

From the standpoint of guaranteeing optional computations, there are objections to variations of static utilities/abortabilities within the sequence of code of a single task. Firstly, there are difficulties in the semantics of guaranteeing some utility/abortability sequences within a task. For example, a low-utility section at the start of an optional computation, followed by a non-abortable high-utility section. The non-abortable section should be guaranteed to complete but does this mean that the earlier abortable section should also be guaranteed as *non-abortable,* especially if there is a precedence relation between the sections? Another example is the case of an optional computation whose subsequent sections fall in utility and increase in abortability. In this case 'abortable until started' cannot be used for a later section when 'must complete' has been used for an earlier section. These problems could be solved by the use of 'must complete' with all utility levels in a composite task. However, as stated earlier, there is no benefit to the system in using different utilities for aperiodic requests unless utility levels also have different abortabilities.

A second objection is the additional complexity and overheads in the schedulability testing of newly arrived aperiodic requests, when existing tasks have variable utilities/ abortabilities within their computations. This also creates problems for the schedulability test algorithm when some sections of a task are schedulable but others not. For example, the high utility sections of a composite task may be schedulable (because they have caused existing low-utility task(s) to be aborted) but the low utility sections may not be schedulable. The requirements of the application may dictate that all sections (high and low utility) must be guaranteed, or the requirements may be that the composite task is still viable when only the high utility sections are guaranteed to run (e.g. a sieve function).

A general conclusion here is that, for the purposes of guaranteeing, a constrained computational model must combine the characteristics of utility and abortability in a consistent manner. Any such model must have semantics which are clear to the applications programmer. Using different utilities which are allocated to each section of a task at the time of guarantee, can create sequences of utilities and abortabilities which have contradictory meanings. Therefore the recommendation for the constrained computational model is to allocate a single utility to each task for the purposes of guaranteeing. This will provide clearer semantics and also simplify schedulability testing. As seen below, there are no objections to changes in the utility of a task *after* guarantee, during the task's execution.

3.3.5 Dynamic Changes of Utility

The above section discussed the semantic difficulties in guaranteeing a composite task which has utilities associated with each of its components. Therefore, for guarantee purposes, it is preferable that a task has a single utility which is set upon the task's arrival. However, there is no reason why dynamic changes of utility should not occur later, for example by being programmed within the code of a task. Such utility changes could occur at any time during a task's activation, without semantic complications, or further demands on schedulability testing.

An example of an application where a dynamic change of utility is desirable is that of tasks which are replicated for fault-tolerance. Once guaranteed, each replicated task may have its utility either decreased or increased, depending on whether fellow replicants have either completed before it, or failed prematurely. The utility of each replicated task can be changed easily, without any further need for guarantee. Similarly, little extra scheduling overhead is required if the application decides that a replicant should be killed because its execution is no longer of any value.

3.3.6 Constraining Task Interdependence

The complex requirements described in Section 2.3 also include hierarchies of interdependent tasks with complex precedence relations and resource sharing. Clearly these requirements need to be simplified if scheduling and schedulability testing overheads are to be reduced to an acceptable level. One obvious step is to flatten the hierarchy so that only linear precedence relations between tasks are permissible.

In order to simplify the problem of resource allocation, Priority Ceiling Protocol can be applied. This allows an upper bound to be placed on the worst-case blocking for resource access. Priority Ceiling Protocol does not optimise the allocation of resources, but at least means that true WCETs are used for schedulability testing.

3.3.7 Accommodating Unbounded Computations

Unbounded requests for computation cause difficulties within a scheme which guarantees hard deadlines. If a task has an unbounded computation time then it cannot be guaranteed but merely have its execution 'optimised' e.g. by allocating it available slack. The task cannot be incorporated at a high priority level because it may invalidate guarantees which have already been given to bounded tasks at a lower level. In any case, it can be argued that unbounded computations should be given a low priority (e.g. background) because their utility cannot be great. If it had been, the applications programmer would have bound them, and required them to be schedulability tested in order to meet some time constraint.

In the case of a task whose boundedness or unboundedness is known only at run-time, it can be argued that such late knowledge does not justify the use of schedulability testing and that the task should be placed in background. After all, a task which is allowed such dynamic behaviour is unlikely to be critical. However, an interesting way of accommodating such a request (especially if it is supplied with a deadline) is for the run-time system to *artificially* bound the request. In this way the run-time system can guarantee that the unbounded request receives a certain amount of computation while still retaining the guarantees of existing aperiodic tasks.

3.3.8 Supporting Alternative and Compound Computations

A requirement for some applications may be that groups of optional computations should be requested together. Such requests for the guarantee of several optional computations may be characterised as *alternative* or *compound*. Alternative and compound requests have semantics which correspond to *OR*, and to *AND* respectively. *OR* semantics mean that the application wishes to know which of a list of several alternative optional computations can be guaranteed. The application may prefer those requests which occur earlier in the list, in which case it is sufficient for the application to know the first alternative in the list which turns out to be schedulable. In contrast, *AND* semantics require that *all* of the requests in the list be schedulable, otherwise the compound computation will be rejected.

The above requirements for alternative or compound computations may seem to add greatly to schedulability testing overheads. However, it is possible that schedulability testing may be optimised so that the full schedulability test algorithm does not have to be repeated for each computation in the group. For example, if the alternative or compound computations share the same deadline, then the existing tasks beneath the request position in the task list need not be fully re-tested for each alternative. Therefore it is recommended that alternative and compound computations may be cost-effectively supported by a constrained computational model.

3.3.9 Guaranteeing Sequences of Aperiodic Computations

Guaranteeing sequences of aperiodic computations which arrive simultaneously but have a precedence order (e.g. iterations of an imprecise computation) can also add to schedulability testing overheads. This is especially true if attempts are made to guarantee some members of the sequence at remote nodes.

The issue arises as to whether members of the sequence which cannot be guaranteed locally should be schedulability tested at other nodes in the system, to see whether they can be accommodated there. The difficulty here is to retain the precedence of sequence members. A remote node needs to be able to guarantee the sequence member within a window which follows the execution of that members predecessor and comes before the execution of the member's successor. In practice end-to-end timings would be involved.

The issue can be resolved by comparing the deadline for the aperiodic request with the overheads involved in guaranteeing iterations of the computation at remote nodes. If the overall deadline for the sequence is sufficiently great, then each member of the sequence can be schedulability tested after its predecessor has completed. In this way each member of the sequence is treated as a separate aperiodic request. This has the advantage of being the most dynamic way of handling the sequence and it can involve the forwarding of a sequence member to some remote node, when the host node cannot guarantee it.

When the aperiodic sequence has a relatively short deadline, then complex time constraints due to precedence make it unlikely that remotely guaranteeing some members of the sequence is viable, in a loosely coupled set of processors. (However, in a closely coupled processor cluster, perhaps with synchronised schedulability testing, such remote guaranteeing of sequence members may be more feasible.) Accepting that the schedulability testing for the sequence takes place entirely on the local node allows a simple schedulability test which uses the total deadline for the sequence and determines how much of the sequence can be guaranteed. This can be posed in the form of an *alternative* computation where each alternative includes a different number of sequence members.

3.4 THE CONSTRAINED MODEL

3.4.1 Introduction

Table 3.1 summarises the Constrained Computational Model which is proposed. Aperiodic requests for optional computations may have the characteristics shown in the table. Each row of the table can be considered as a different task 'type', named *Mandatory, High Utility, Medium Utility, Low Utility,* and *Background.*

Task type	Utility	Abortability	Bounded or Unbounded	Guarantee	Deadline
Mandatory	Base	No abort but replaceable	Bounded	Off-line	Hard
High Utility	H	No abort	Bounded	On-line	Firm
Medium Utility	M	Abort before start	Bounded	On-line	Firm
Low Utility	L	Abort anytime	Potentially Unbounded (Budget)	On-line: Budget only	Firm
Background	Null	No need to abort	Potentially Unbounded	Not guaranteed	Soft/ None

Table 3.1: The Constrained Computational Model

The implication of the model is that Mandatory tasks have a baseline utility and that the other task types can add value to the baseline according to their utility. Obviously the higher the utility of a task, the more value it adds to the application, upon the completion of the task. In other words H > M > L in Table 3.1. Any optional computation must belong to one of the High, Medium or Low categories and be allocated the utility which the programmer has set for that category. (A more concise way of expressing the relative utilities of these categories is to use two ratios, as is discussed in Section 3.5.3)

It seems, from the arguments above, that clearer semantics are achieved by associating a single utility level with each aperiodic request, rather than have the complication of guaranteeing a computation which is divided into sections, each with a different utility level. Therefore the constrained model allows only a single utility level to be associated with each request at its arrival. If different utility levels are needed, the applications programmer must split the compound task into smaller tasks which have precedence relations between them, according to their relative deadlines. The use of a single utility per task, allocated at arrival, should also simplify schedulability analysis. Of course, the utility of an optional computation may still be changed dynamically, at any time after its guarantee, as is described in Section 3.3.5.

3.4.2 Utility Levels

It was decided that the constrained model should use *five* utility levels. This choice is a compromise between providing adequate facilities for the applications programmer and incurring greater overheads in schedulability testing if more than five utility levels are used. The close interrelation between utility and abortability has been discussed above and it is clear that these are not orthogonal characteristics. Therefore the constrained model provides consistent semantics by integrating abortabilities with utility levels.

3.4.3 Mandatory Tasks

As can be seen in the Table 3.1, the first task type is *Mandatory* and is associated with a baseline utility. All Mandatory tasks are considered to be critical tasks which are resident on the processor. As indicated in the table, they have bounded WCETs and hard deadlines. It is assumed that they have been guaranteed off-line before the system starts. *No abort* in the table indicates that these tasks cannot be aborted. However it is possible to *replace* a Mandatory task by a preferred High Utility task (see next section).

3.4.4 High Utility Tasks

The next utility level, *H*, is associated with *High Utility* tasks. These are optional computations which arrive aperiodically at the processor and have firm deadlines. (Here, a 'firm' deadline indicates that instances of these optional computations can be missed without critical failure, but that there is no value in executing these tasks if they do not meet their deadlines.) Consequently these tasks require to be guaranteed dynamically and they must have bounded WCETs in order that this may be possible. They cannot be aborted once guaranteed (*No abort*).

It is possible that such a High Utility optional task could replace a Mandatory task at run-time (e.g. where a preferred, expensive version of some computation is required). In this case the utility of the High Utility task would be gained upon its completion and would be added value over the baseline value which would have been gained by the Mandatory task which was replaced. Note that in guaranteeing the High Utility task, the system must take in account the processor time which would be freed, were the Mandatory task to be replaced.

3.4.5 Medium Utility Tasks

Medium Utility tasks are also aperiodic, optional computations. As with *High Utility* tasks, they have bounded computation times with firm deadlines and must be guaranteed dynamically. However, they differ from *High Utility* tasks in that they are allocated a lesser utility level, *M*, and can be aborted within the interval between their guarantee and the start of their execution. An example of an application for this type of task would be an *atomic action* which once guaranteed, may be aborted before it starts, but may not be aborted during its execution without great loss of value to the system.

3.4.6 Low Utility Tasks

Low Utility tasks are similar to *Medium Utility* tasks except that they are allocated an even lower utility, *L*, and can be aborted anytime after guarantee. As with *Medium Utility* tasks, they have firm deadlines, but their computational requirements are *potentially unbounded*. This can mean that they are either impossible to bound, or that their computation times have such large variances that they can only be bound very pessimistically. For example, only a minimum or average computation time may be available. There is no value in executing such a task if it does not meet its deadline, and therefore it is better if some way can be found to increase the chances of the deadline being met. The system attempts to guarantee each low utility task a *budget* which may, for example, cover its minimum or average computation time. This only guarantees that the task has a certain probability of finishing in time, but it is preferable to allowing the task be executed in background, without any guarantee.

Potentially unbounded indicates that such tasks need not necessarily have WCETs which are difficult to bound. It may be that they have tightly bounded WCETs, but that their low utilities require them be *abort anytime* tasks.

It should be noted that the system must monitor whether a low utility task has consumed its budget. When a budget is exhausted, the task must be aborted, or another budget guaranteed. The task must be prevented from using more processor time than it has been allocated, because this could undermine the schedulability of other tasks in the system.

3.4.7 Background Tasks

The last task type, *Background,* are optional computations which also have potentially unbounded execution times but have soft deadlines, or no deadlines at all. In effect, they are not real-time tasks. They cannot be guaranteed and therefore must always be executed at the lowest priority. At this priority, they may be ordered in FCFS or earliest deadline order. (In any case, to guarantee such tasks is less appropriate because their soft deadlines indicate that there may be some value in them executing *after* their deadlines have expired.) Because they are executing in background, and are not guaranteed, these tasks need not be considered when schedulability testing tasks of higher utility. *Background* tasks may of course miss a soft deadline, in which case they may eventually be removed from the task list. They carry Null utility because they are not guaranteed and need not be aborted by higher utility tasks. Therefore the issue of which utility level they require, in order to make it cost-effective to guarantee them, does not arise. It is acknowledged that *Background* tasks still contribute to the system.

3.4.8 Dynamic Changes in Utility

The Constrained Computational Model requires the use of a single utility per optional computation, which is set upon its arrival. As argued above, this clarifies the semantics of the model and should also simplify schedulability testing. The utility which the optional computation carries upon arrival determines how 'hard' the system will try to guarantee it. However, as stated previously, there is no reason why the original utility level of the optional computation cannot be changed as required, during its execution. No extra schedulability testing overhead is incurred in order for the application to change the utility level of a guaranteed task.

An example of utility change is when a low utility (L) optional computation may be nearing completion, and may dynamically upgrade its utility to M in order to avoid the possibility of abortion and a loss of value to the system. Such a change neither affects the schedulability of the task concerned or any of the other tasks in the system. Obviously some programmer-defined method of determining a task's progress is required in order to determine when the utility should be upgraded.

Another interesting application of this facility is to dynamically change the utility of an imprecise computation, at each of its iterations. For example, a task which implements imprecise computation could be programmed to decrease its own utility at each iteration of the computation. In effect, the ability to dynamically change utilities can allow composite utilities without incurring any of the extra schedulability testing overheads associated with guaranteeing a composite task.

3.4.9 Precedence

Precedence is not directly supported by the computational model. However, it is assumed that the Run-Time Support for the model prioritises tasks according to deadline monotonic ordering. Therefore, aperiodic requests which arrive simultaneously can have their precedence indicated by their relative deadlines. If the precedence order in the specification conflicts with deadline order, then a design tool may be used to adjust the deadlines given, so that if the requests are guaranteed, they will be executed in precedence order. This assumes that there are no internal delays in the task(s) which occur earlier in precedence order. Such delays could result in the premature execution a task which should have executed later in the ordering.

3.5 VIABILITY OF THE CONSTRAINED COMPUTATION MODEL

3.5.1 Guarantee-worthiness

One major issue which arises from the computational model is whether the aborting of lower-utility tasks actually benefits the total utility gained by the system. *Guarantee-worthiness* has been defined as a measure of how much the utility gained by the system in guaranteeing new aperiodic requests, outweighs the overheads and aborted computations which are incurred by those guarantees.

Whether a particular aperiodic request is *guarantee-worthy* depends on factors such as its own utility and guarantee overhead, and the utilities and guarantee overheads of any existing tasks which it aborts. (Another consideration is whether the aborted tasks are near completion, since the full utility of an aborted task is lost, however small its residual execution time.) The guarantee overheads incurred will depend upon computation times, deadlines and the complexity of whichever guarantee algorithm is in use.

Clearly, if aperiodic requests are not guarantee-worthy, then guaranteeing may provide less throughput of optional computations than merely accepting or rejecting them according to whether they are schedulable *without* the abortion of lower utility tasks. (In effect this is a FCFS system where all tasks are of the same utility.) Even more extreme is to dispense with schedulability testing altogether and merely accept all aperiodic requests, and attempt to optimise their executions. However, this loses the advantage of being able to pursue some useful alternative, when the request for an optional computation has been rejected, and it may result in a great loss of utility to the system.

3.5.2 Simplistic Measures of Guarantee-worthiness

There are two crude measures of how small a task's computational requirement can be, before the overhead incurred in guaranteeing ceases to be justified. The measures provide necessary but insufficient criteria because they do not take into account the loss of utility due to the abortion of lower utility tasks.

The first measure is whether the deadline of the aperiodic request is considerably larger than the overhead of the likely schedulability test overhead plus the computational requirement of the request. If the deadline is of the same order as this sum, then clearly there is no point in attempting a guarantee, and the request should be rejected outright.

Another comparison which may be useful is between the computational requirement of the aperiodic request and the total overhead for schedulability testing and scheduling the task. This may give some lower bound for the computational requirement of an aperiodic request which is worthy of guarantee and scheduling.

The next section looks at how the full issue of guarantee-worthiness may be addressed, including taking into account the effect of loss of utility due to the abortion of previously guaranteed lower utility talks.

3.5.3 Evaluating Best Effort Admission Policy

In order to fully examine the issue of *guarantee-worthiness* in Best Effort admission, some simulation studies are required. These require some means of expressing and controlling the relative values of *High Utility*, *Medium Utility* and *Low Utility* tasks, in order that the effect of aborting lower utility tasks may be investigated. Note that it is only the *relative* values of these utilities which affects the issue of *guarantee-worthiness*, and therefore the total utility gained by the system.

Two ratios are introduced. The first, R_1, is the ratio of the utility of *High Utility* tasks to the utility of *Medium Utility* tasks. The second, R_2, is the ratio of the utility of *Medium Utility* tasks to the utility of *Low Utility* tasks. (*Mandatory* and *Background* tasks can be considered as different cases, with fixed utilities, and therefore no ratios are needed.) Both R_1 and R_2 are system-wide parameters which can be set by the applications programmer. For example, if R_1 and R_2 are each 10, then the utilities of *High Utility*, *Medium Utility* and *Low Utility* tasks will be 100, 10 and 1 respectively. The applications programmer can set R_1 and R_2 and measure the efficiency of an implementation by accumulating the total value of utilities over a system run. In short, R_1 and R_2 allow the trade-off in guarantee-worthiness of the three intermediate task types to be investigated.

Chapter 7 presents the results of simulation studies in which Best Effort Admission Policy is compared to FCFS. These results include the effect of varying R_1 and R_2.

3.6 NECESSARY LANGUAGE CONSTRUCTS

3.6.1 Compound and Alternative Computations

Compound computations, as described in Section 3.3.8, support requests for multiple optional computations. A compound computation can take the form of an 'AND statement', where all of the ANDed computations must be guaranteed. In other circumstances an application may require the guarantee of one of a selection of optional computations. The language construct for such an *alternative* computation could take the form of an 'OR statement'. The requests listed in the OR statement could be in order of preference, and the run-time system would then attempt to guarantee them in that order until the first schedulable request is found.

Compound computations should all be requests for *high utility* computations, because it is inconsistent to require that all of the components be guaranteed, if some of them are only guaranteed at low utility and are therefore abortable. If the components of the compound computation have a common deadline, then it is more efficient to add their computation requirements and make a single request for the summed computation times. This prevents the schedulability test having to run for each component, but in no way affects the accuracy of the schedulability test. If there is a precedence relationship between the components of a compound computation, then this can be enforced by the programmer allocating appropriate deadlines to the components of the computation (see Section 3.4.9).

The applications programmer may chose to associate a different utility/abortability with each alternative within an *alternative* computation. This may be useful for example when an early preferred alternative is computationally dearer, and a later alternative, which is less-preferred, computationally cheaper. (A cheaper computation has a shorter computation time and/or a longer deadline) Table 3.2 shows the possible alternative computations which can be demanded. The alternative request which is made can have either a different computational expense, or a different utility, or both. The 'Useful' column of the table warns the applications programmer against some combinations which are

Computational Expense of the Alternative	**Utility of the Alternative**	**Is this a Useful Alternative ?**	**Comment:**
cheaper	same	y	Example: a less-preferred version
cheaper	higher	y	Example: a requirement for graceful degradation.
cheaper	lower	y	Example: a less-preferred version
dearer	higher	?	Why not request this as a first alternative?
dearer	same	n	Cannot be guaranteed if earlier alternative rejected
dearer	lower	n	Cannot be guaranteed if earlier alternative rejected
same	higher	?	Why not request this as a first alternative?
same	lower	n	Cannot be guaranteed if earlier alternative rejected

Table 3.2: Possible Alternative Requests

nonsensical e.g. cases where later alternatives cannot be guaranteed, after an earlier alternative is rejected.

3.6.2 Fulfilling Application Requirements

Alternative computations may capture some of the requirements of *graceful degradation*, or *multiple versions*. Other requirements for these may be programmed by the applications programmer. For example, when a low-utility task is aborted by a higher-utility request, the application is informed of the rescinded guarantee, and the programmer may design the application to request a cheaper task in place of the aborted task. This may allow graceful degradation under system stress. Similarly, in the case of multiple versions programming, the application may request a cheaper version in place of an aborted preferred version.

Subsequences of imprecise computations can also be schedulability tested by presenting them as *alternative* computations. In other words the OR construct can be used with the first alternative being the maximum sequence of iterations of the imprecise computation, the second (less-preferred) alternative being a shorter subsequence, and so on. Alternatively, if the Imprecise Computation can run to a large number of iterations, and if the iterations are of low utility, then a low utility request for a *budget* could be made. This will require less schedulability testing than using the OR construct, but the iterations of the Imprecise Computations will be abortable. Note that, should the Imprecise Computation use up its budget before its deadline has expired, then there is no reason why it should not make a further request for a budget.

Sieve Functions can be defined as a sequence of alternating bounded and unbounded computations. These may be implemented by first using an AND request for the bounded, minimum components of the sieve function. If this request is accepted, then the sieve function is started, and a budget for each unbounded component is obtained by making a low utility request at the point where each of the unbounded components is released in the sequence of computations.

Admittedly, AI applications can require complex task hierarchies and the simple linear form of precedence assumed in this model cannot capture such complex dependencies. In these circumstances, tool support may be used to reduce the task hierarchy to the constrained model used here.

3.7 SUMMARY OF THE CONSTRAINED MODEL

A complex computational model for optional computations would incorporate value functions or utilities which depend upon many parameters. In addition it would model complex interdependencies between tasks which may include a task hierarchy which captures inter-task communications, precedence, and the sharing of resources and subtasks.

There exists considerable evidence that, were such a complex model to be implemented, the run-time overheads incurred would drastically reduce the throughput of optional computations. This leads to the development of a constrained computational model which defines 5 *task types*, ranging from mandatory tasks to background tasks. In between these extremes lie 3 types of optional computations which require to be guaranteed upon arrival. Optional computations in the model, which are of lower utility, can be more easily aborted. All optional computations which arrive at the processor are allocated a single utility according to which task type they belong to. However, no extra overhead is incurred if the task type (and utility) of an optional computation is changed later, during its execution. The constrained model allows only simple linear precedence between tasks. The model can support some useful programming language constructs which would, for example, incorporate requests for multiple optional computations.

The differences in utility levels between the 3 intermediate types of optional computation may be specified by the use of two system-wide ratios. The values of these ratios crucially determines whether it is cost-effective to guarantee each type of optional computation.

3.8 THE WORK WHICH FOLLOWS

The constrained model must now be shown to be viable by the development of algorithms, for run-time support, which are efficient enough to run on the same processor as applications tasks. Chapters 4 to 7 which follow, take a bottom-up approach to this development. Chapters 4 and 5 first establish the viability, and then attempt to enhance, a range of on-line schedulability tests for optional computations. Chapter 6 investigates allocation methods for optional computations in a multiprocessor cluster. Chapter 7 compares admission policies for optional computations, which have passed their schedulability tests. Finally, Chapter 8 implements the computational model in Ada, and provides Ada code which can fulfil many of the application requirements discussed above.

CHAPTER 4

VIABILITY OF ON-LINE ACCEPTANCE TESTING

4.1 INTRODUCTION

Chapter 3 concluded that a constrained computational model for optional computations can satisfy many of the requirements for future real-time systems, but that such a model requires efficient algorithms for run-time support. This chapter develops a range of algorithms for on-line or *dynamic* acceptance testing of optional computations. The acceptance tests are developed by adapting some of Audsley's *static* schedulability tests [2] which are reviewed in Section 2.7.

It is assumed that, as with the computational model of Chapter 3, each processor has a resident set of mandatory computations which have been guaranteed off-line. Mandatory computations may be periodic or aperiodic (e.g. interrupts). For the purposes of off-line schedulability testing the aperiodic mandatory computations must be constrained to be *sporadic tasks* i.e. they have a minimum interarrival time or, in other words, a maximum arrival rate. Worst-case off-line analysis assumes that sporadic mandatory computations continually arrive with a separation of their minimum interarrival time, and that they can therefore be considered as periodic tasks. It follows that, in this analysis, all of the mandatory computations which relate to the processor are considered as periodic tasks.

Optional computations are also considered as sporadic tasks. This allows the overheads for acceptance testing, which occur on the same processor as application tasks, to be upper bound and themselves guaranteed. In the following analysis, optional computations are modelled as sporadic tasks which arrive, in the worst-case, at their maximum arrival rate. Each sporadic task arrives with a specified WCET and deadline.

4.2 THE STATIC ALGORITHMS

The aim of dynamic acceptance testing is to guarantee the relative deadline, D, of each sporadic task which arrives with a known WCET, C, at an arbitrary arrival time which is constrained to be separated from other sporadic arrivals by at least the minimum interarrival time. When a sporadic task arrives at the processor, it is inserted in its correct position, in a deadline monotonically ordered task list, which includes resident periodic tasks plus those sporadic tasks which have previously been accepted, and have not yet completed. The sporadic arrival is then schedulability tested in order to determine whether it can be guaranteed or must be rejected. Note that the acceptance test must also include schedulability tests for all of the tasks which fall below the sporadic task in the task list, in order to ensure that each of these can still meet its deadline.

Two static algorithms, due to Audsley [2], are now chosen as the first candidates for adaptation. These are (i) Test 1, a sufficient but not necessary test, which has $O(N^2)$ complexity and should incur the smallest overheads, and (ii) a sufficient and necessary test, which has pseudo-polynomial complexity, and should incur the greatest overheads. (Both of these static algorithms are described in Section 2.7) From now on Test 1 will be know as **$O(N^2)$**, and the sufficient and necessary test will referred to as **PP**.

$O(N^2)$ requires a determination of the interference (I) for the duration of the deadline of whichever task is being schedulability tested (known as the test task, i). In order to determine I, the list of higher priority tasks is scanned to provide the following sum which is the total interference from all higher priority tasks j:

$$\Sigma_j(\lceil D_i \div T_j \rceil C_j) \tag{4.1}$$

In contrast, PP calculates the exact worst-case response time of the test task, by calculating the interference from higher priority tasks within a time window, w_i. This window is increased at each iteration of the algorithm. Its initial value is the WCET of the test task, and at each iteration the following sum over all higher priority tasks j is calculated:

$$\Sigma_j(\lceil w_i \div T_j \rceil C_j) \tag{4.2}$$

The value of w_i at the next iteration is set to the value of (4.2) from the last iteration, and so on, until w_i converges at a maximum value, which is the response time of the test task.

The adaptation of these two static algorithms for dynamic use, is now described briefly. A more rigorous explanation of the necessary changes may be seen in Appendix A.

4.3 ADAPTING THE STATIC ALGORITHMS

Both $O(N^2)$ and PP can be adapted for dynamic schedulability testing by similar changes to the above equations. Both adapted algorithms avail themselves of the run-time data which is updated by the scheduler for all tasks: R_j, the current residual execution time of each task, j, and NR_j, the next release time of each task j. When a sporadic task arrives, the schedulability tester uses the NR_j for each task which has a higher priority than the test task, in order to calculate the offset, O_j, of the task's next release ($O_j = NR_j$ - *current time*). The changes required in expressions (4.1) and (4.2) above allow for the following dynamic properties:

(i) A higher priority task cannot interfere with the test task until that higher priority task has been released.

(ii) If the next release of the higher priority task is after the expiry of the interference interval under consideration, we must ensure zero, not a negative, interference value is produced.

(iii) Any residual execution of an interfering task must be added to that task's total interference with the test task.

In line with the above, expressions (4.1) and (4.2) above are adapted to:

(i) Reduce by the offset O_j, the interval (D_i and w_i respectively) considered for interference by a higher priority task j.

(ii) Ensure that the result ($D_i - O_j$ and $w_i - O_j$ respectively) is not negative

(iii) Add R_j.

Hence (4.1) becomes :

$$\Sigma_j(\lceil (D_i - O_j) \div T_j \rceil_0 C_j + R_j) \tag{4.3}$$

where $\lceil X \rceil_0$ (i) returns 0 if $X \leq 0$

(ii) returns $\lceil X \rceil$ if $X > 0$

and (4.2) becomes :

$$\Sigma_j(\lceil (w_i - O_j) \div T_j \rceil_0 C_j + R_j) \tag{4.4}$$

Apart from this, the algorithms proceed as in the static case, except that there is no need to schedulability test the tasks which are above the sporadic task in priority ordering. When testing the sporadic task using the above equations, (4.3) will use the sporadic task's deadline for D_i, and (4.4) will initialise w_i to the sporadic task's WCET.

Because the sporadic task is a one-off, each lower task need only be tested against its next deadline. If the lower test task is active (i.e. non-zero residual execution time), then (4.3) will use the remainder of the task's relative deadline for D_i, and (4.4) will initialise w_i to the residual execution time (R_i) of the test task. (Note that in the case of using PP for dynamic testing, the remaining interference intervals do not necessarily increase monotonically down the task list, and therefore a final *ith* window value cannot be used to initialise the window for the $i + 1th$ test task.)

If the lower test task is inactive (i.e. completed its current execution and awaiting its next release) then we must check against the deadline of the task's *next* activation. Strictly, we should calculate interference in an interval starting at the test task's next release. However, to calculate interferences in a future interval of time would incur unacceptable overheads. These can be avoided by a sufficient test (see Appendix A) which supposes that the next release of the test task is at the *current time* i.e. the deadline has effectively been increased by the quantity: *next release time - current time.*

4.4 VARIATIONS ON THE ALGORITHMS

Both static and dynamic $O(N^2)$ and PP algorithms trade off complexity with accuracy. In the dynamic case $O(N^2)$ will be pessimistic but quicker. Therefore, if the schedulability testing is on the same processor as the resident periodic task set, then $O(N^2)$ will allow more time for sporadic task processing, but may reject some sporadic tasks which are schedulable. By contrast, the PP test takes longer, on average, to arrive at an *optimal* result. On the same processor, PP would therefore leave less time for sporadic tasks, but never pessimistically reject a schedulable sporadic task.

A more efficient algorithm may be to combine $O(N^2)$ and PP in a *hybrid* algorithm. All schedulability testing is performed by $O(N^2)$ until a sporadic task is found to be unschedulable. Then PP is used to make a finer judgement on schedulability. Such a hybrid algorithm should be both optimal, and faster on average, than PP.

Another variation would be to reverse the order of schedulability testing by schedulability testing the lowest priority task first and then working up the task list until the sporadic task itself is schedulability tested. For a schedulable sporadic task this would take the same time as the top-down order used previously. However it may be that *unschedulable* sporadic tasks are found out earlier. This will depend on where in the task list the unschedulable tasks are likely to occur. At one extreme (justifying top-down) only the sporadic task may test as unschedulable, while lower tasks pass their tests. At the other extreme (justifying bottom-up), all tasks (including the sporadic task) may be schedulable except the lowest in the list. In other words, if the unschedulable tasks are more likely to be found nearer the bottom of the task list, then bottom-up testing will be faster, on average.

The above discussion indicates that five dynamic schedulability tests should be investigated:

(i) (Pure) $O(N^2)$
(ii) (Pure) PP
(iii) Hybrid ($O(N^2)$/PP)
(iv) Bottom-up PP
(v) Bottom-up Hybrid.

4.5 SIMULATION STUDIES

According to the above discussion, further investigation is required into the statistical behaviour of five adapted algorithms. The most cost-effective way of doing this was to build a simulation of a scheduler and schedulability tester which would input a large variety of periodic task sets and sporadic requests. It was decided that the schedulability testing part of the simulation would run *in real time* in order to measure exactly the overheads incurred by each algorithm. The scheduler itself would run as a simulation. However, in order to establish the scheduling overheads (context switching, etc.) which would be used in the simulation, some initial real-time scheduling was carried out. Measurements were made by timing the scheduling overheads of concurrent programs written in Parallel C, and running on the target hardware, a T800 transputer.

4.5.1 Measuring the Scheduling Overheads

Fixed-priority pre-emptive scheduling was used in the concurrent programs which measured scheduling overheads, as well as in the subsequent simulations. However, at the fine grain level, it was decided that scheduler slots should be implemented by co-operative, rather than interrupt-driven scheduling. This allowed more flexibility in the implementation, and avoided the overheads of descheduling the currently executing task at the start of each scheduler slot.

A small granularity (10ms) was chosen for the scheduler slot size in order to minimise the delay in testing sporadic arrivals, and also to minimise release jitter. (An even smaller slot size might have unreasonably increased the co-operative scheduling overhead.) It was decided to allow only one sporadic arrival to be schedulability tested at the beginning of every tenth slot, in order to permit a smaller upper bound for the overheads of schedulability testing.

The co-operative scheduling overheads which were measured were (i) the minimum co-operative scheduling overhead (0.15ms) and (ii) the extra overhead per task release (0.06ms). Following this, five versions of the simulation were built, one for each of the adapted algorithms listed in Section 4.4.

4.5.2 The Simulations

The average co-operative scheduling overheads which had been measured were now used in the simulations. As explained above, the scheduling itself ran in simulation time, but the algorithms for schedulability testing were measured in real-time so that comparisons in their performance could be made. The following is an outline of the method of scheduling and schedulability testing which occurred in each simulation.

At the start of every *tenth* slot, each simulation checks for the arrival of a sporadic task and, if one is present, tests its schedulability in *real time*. If the sporadic task is schedulable, it is inserted into the task list (dispatch queue) in deadline monotonic order. At *every* slot, the simulation releases any periodic tasks whose reactivations are due, updates next release times and residual execution times, and finally dispatches the topmost runnable task. When a computation completes in mid-slot, each simulation allocates the remainder of the slot to the next topmost runnable task. An indefinite number of sporadic tasks may accumulate in the task list until each completes and is then deleted. The simulations also verify that every guaranteed task actually completes within its deadline.

4.5.3 Task Generators

A task-set generator was constructed to produce large numbers of schedulable sets of periodic tasks. All tasks were independent in order to simplify analysis. The co-operative scheduler was modelled as the highest priority periodic task with a period of 10ms. The schedulability test was modelled separately as the second highest periodic task with a period of 100ms which was equal to the inverse of the maximum sporadic arrival rate. (As indicated earlier, the adoption of a maximum sporadic arrival rate is a pre-requisite for finding an upper bound on schedulability testing.) The generator produced random task sets with task periods, deadlines and WCETs all uniformly distributed. Different numbers of periodic tasks and different periodic processor utilisations were specified.

Sporadic task generators were constructed to produce random sporadic arrival times, deadlines, and WCETs. In some simulation runs, these parameters were uniformly distributed, whilst in other runs the arrival rates were Poisson distributed, with deadlines and computation times normally distributed. The generator allowed the minimum interarrival period for sporadic tasks to be specified.

4.5.4 Measuring Performance

The first aim of the simulation study was to compare the performance of the five adapted algorithms. The best performance index seemed to be the guarantee ratio as used in Spring [56]. This is a ratio obtained over a complete simulation run :

number of sporadics guaranteed by the algorithm $\div$ total number of sporadics sent

Before guarantee ratios can be measured by a simulation, an upper bound for the time taken to run the schedulability test must be estimated. This will be the WCET for the high priority periodic task which models schedulability testing. The task set generator uses this value when performing its own (static) schedulability test of the task sets which it generates. There is obviously a trade-off between the pessimism of this value and the time left for other tasks. Some working value of this bound/computation time must be used to generate the first schedulable task sets, which can *then* be used in a run to yield a better value for the bound. In practice, the approximate maximum values from a feasibility study were used to generate the first task sets. Simulation runs then allowed these values to be refined.

It was found that schedulability testing algorithms based on PP are especially difficult to upper bound. Obviously, a particular maximum value is peculiar to a particular set of test data, and the question arises as to which maximum to use in practice. For example is it overly pessimistic to use the highest value which has *ever* been obtained for a particular algorithm? This problem will be addressed later (see Section 4.7). Meanwhile, the general practice adopted is to use the maximum value for the particular set of test data used.

4.6 COMPARING THE ADAPTED ALGORITHMS

There are a number of parameters which affect the performance of all of the adapted algorithms. The most obvious is the number of periodic tasks in the periodic task set (i.e. N above). Related to this is the ratio:

average periodic task deadline $\div$ average sporadic deadline (henceforth: PD/SD) which determines the average position in the task list in which a sporadic task will be placed. All tasks below the sporadic task must be schedulability tested, so this ratio is an important factor in the actual time taken by the algorithms. Other parameters are the total periodic processor utilisation, and the intrinsic difficulty of scheduling a particular set of periodic tasks. This latter parameter is referred to by Lehoczky [27]. Task sets whose periods are not harmonics have a relatively low *breakdown utilisation* i.e. their uneven occurrence of slack time means that there are intervals of zero slack time during which no sporadic tasks can be scheduled.

Further parameters which may affect performance are sporadic arrival rates, and the average computation time per sporadic task. For example, the average sporadic computation time may affect the number of iterations required to test individual tasks in the PP algorithm. The approach taken in the following investigations is to keep all parameters constant except one, and to measure the performance of the five algorithms whilst varying the single chosen parameter.

4.6.1 Varying the Periodic Task Characteristics

Table 4.1, Graph 4.1 and Table 4.2, all below, show the comparative performance of the five algorithms and two background scheduling methods, when characteristics of the periodic task set only are varied. The performances of the five algorithms are measured by Guarantee Ratio (GR) which has been defined in Section 4.5.4. The background scheduling methods accept all sporadic tasks and execute them at the lowest priority in FIFO order. Their performance can be measured by a Success Ratio (SR) i.e. the proportion of all the sporadic tasks which are found to meet their deadlines. It should be noted that this background scheduling does not schedulability test sporadic tasks and therefore does not guarantee them. In that sense it is not strictly comparable with other five algorithms and serves only as a benchmark.

All simulations results shown in these tables and graph use the same set of 420 sporadic tasks whose arrival rates are Poisson distributed ($\mu = 2.8$, $k = 10$) over the total simulation time of 100,000 ms. The sporadic deadlines and computation times are normally distributed.

Changing the Number of Periodic Tasks

Table 4.1 shows the maximum schedulability test times (in ms) and guarantee ratios obtained from each of the schedulability test algorithms as the number of periodic tasks (N) in the task set is increased. The PD/SD ratio and the number of tasks below the sporadic task position also increases with the number N. The table also shows the success

Number of Periodics	PD/SD Ratio	Ave no of tasks below sporadic	Bottom-up Hybrid		Hybrid		Bottom-up PP		PP		$O(N^2)$		Background (FIFO) 1.	2.
			Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	SR	SR
5	0.340	0.0	3.53	0.742	3.80	0.737	3.41	0.742	3.80	0.734	0.53	0.424	0.170	0.700
10	0.627	3.1	6.30	0.727	6.70	0.719	7.50	0.699	8.00	0.672	1.34	0.606	0.144	0.649
15	0.808	4.4	8.80	0.710	9.20	0.695	10.00 (13.20)	0.636	10.00 (15.50)	0.580	2.76	0.585	0.128	0.624
20	1.017	7.5	10.00 (11.80)	0.686	10.00 (14.00)	0.659	10.00 (24.60)	0.255	10.00 (25.70)	0.173	4.00	0.633	0.099	0.584
30	1.393	13.0	10.00 (20.00)	0.576	10.00 (25.00)	0.529	10.00 (68.00)	0.00	10.00 (69.00)	0.00	9.80	0.547	0.085	0.555

Table 4.1: Comparing performances when changing the number of periodic tasks

ratios (SR) obtained from two versions of background scheduling of the sporadic tasks in FIFO order. (The difference between these versions is explained in the paragraph below.)

Each result in Table 4.1 is based on the output of 10 simulation runs each with 10 different sets of random periodic tasks. Maximum values are the maximum from all 10 runs and guarantee ratios are the average for 10 runs. The periodic processor utilisation is always 85% which includes scheduling overheads for the periodic tasks, but does *not* include any utilisation by the periodic task which models schedulability testing.

As explained earlier, the maximum schedulability test time (upper bound) must first be established before a simulation can produce a meaningful guarantee ratio. For high N the maximum schedulability test time can be greater than the scheduler slot size (10ms). It was decided not to allow the schedulability test to overrun the slot size because this would violate the scheduler's upper bound on release jitter. In any case, more than 10ms constitutes an unacceptably high overhead for worst-case schedulability testing. Therefore, a timeout was placed in the scheduler-tester which thus rejects sporadic requests taking more than 10ms to test. Obviously, this impacts on the guarantee ratios shown in Table 4.1, but it is only significant when the maximum value (shown in brackets) is considerably more than the slot size of 10ms.

Two versions of background scheduling of the sporadic tasks in FIFO order are included at the end of the table. It should be emphasised that neither version guarantees the sporadic tasks in advance. Instead, they accept all the sporadic tasks without the overhead of testing their schedulability. Hence the performance measure is better described as a *success ratio* which is the proportion of the sporadics which turn out to meet their deadlines. The difference between the two versions of background scheduling is that Background 1 is strictly FIFO. It continues to queue, and then execute, all sporadics, *even when their deadlines have expired*. Background 2, however, deletes sporadics when their deadlines are found to have expired.

Table 4.1 shows that bottom-up hybrid consistently performs best. Clearly the guarantee ratios of the PP based algorithms are badly affected by the 10ms timeout when N = 20 and N = 30. This explains why $O(N^2)$ produces the second best guarantee ratio when N = 30.

Variations within the guarantee ratios obtained by $O(N^2)$ may be explained as follows. The deterioration in guarantee ratio between N = 20 and N = 30 is as expected due to the increased number of schedulability tests needed for a longer task list. The small guarantee ratio for N = 5 can be interpreted as the effect of the pessimism of the algorithm. For N = 5, all the periodic computation is above the sporadic task in priority order. Therefore the pessimism of $O(N^2)$, due to the full extra hits of higher priority tasks, is likely to be greater. The increase in the total laxity of the task set as N increases may also account for a greater guarantee ratio when N = 30 than N = 5. The shallow trough in guarantee ratio at N = 15 may be due to statistical variations: it was observed that guarantee ratios for $O(N^2)$ had a particularly wide standard deviation over the 10 task sets (approximately 10% of guarantee ratio).

Table 4.1 also shows that the success ratios for the background methods. The low success ratios for Background 1 show the effect of continuing to queue and execute sporadic tasks even after their deadlines have expired. Both background versions show a deterioration in success ratio as the number of periodic tasks in the task list increase. This can be interpreted as the effect of the sporadic tasks occupying the lowest position in the task list, as the periodic task list increases in length. For example, when N = 5, sporadic tasks with an average deadline of 550 ms are being queued beneath periodic tasks with a maximum deadline of 500 ms. When N = 30, however, the same sporadic tasks queue below periodic tasks with a maximum deadline of 3000 ms. It is clear that such a long periodic task list displaces the sporadic tasks further downward from their static deadline monotonic position in the task list.

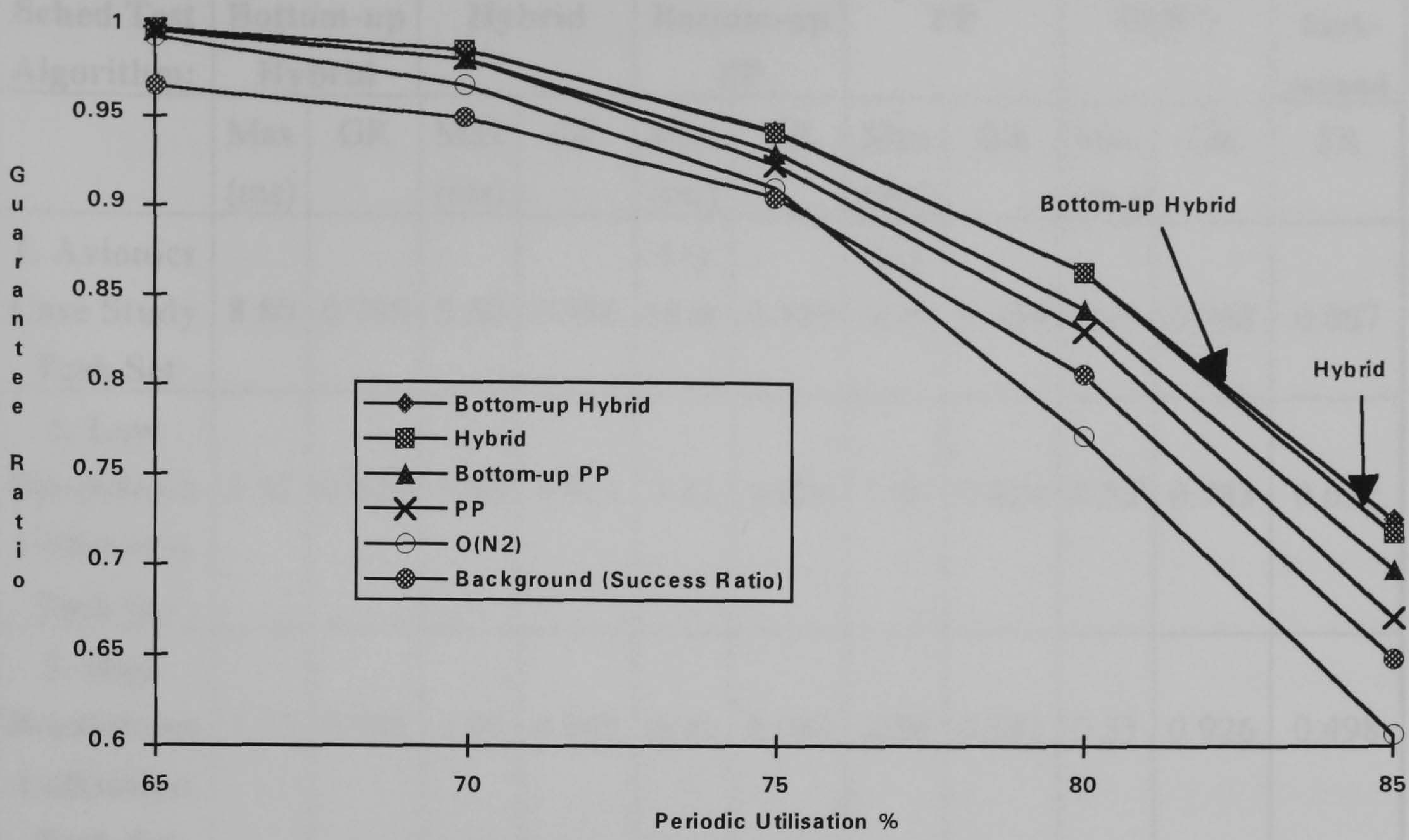
The strictly FIFO *Background 1* method gives such low success ratios, that it was decided to omit it from the rest of the simulation studies. From now on only *Background 2* is included in the results and it is simply referred to as *'Background'*.

Increasing Periodic Task Utilisation

Graph 4.1 shows the comparative performance of the five algorithms and background scheduling when the periodic task utilisation is varied from 65% up to 85%. Lower utilisations were not used because they cause the performance of all of the algorithms to converge at a guarantee ratio of 1.0. In this case N and PD/SD were kept constant (N = 10 and PD/SD as near 0.6 as random task set generation would allow). The set of sporadic tasks were the same as in Table 4.1, and 10 sets of periodic tasks were randomly generated for each processor utilisation in the graph.

As for Table 4.1, the schedulability testing was not included in the total periodic task utilisations. The task sets generated were all statically schedulability tested using a worst-case figure of 10ms for the computation time of the periodic task which models the dynamic schedulability test. As before, a less pessimistic maximum schedulability test time was found, for each set of test data, by repeating each simulation and revising the maximum schedulability test time. Each guarantee ratio and success ratio produced is an average over 10 periodic task sets.

Graph 4.1 also shows that the success ratio of background exceeds the guarantee ratio of $O(N^2)$ at high periodic task utilisations. This can be interpreted as the effect of the pessimism of $O(N^2)$ increasing as the interferences from higher priority periodic tasks grow larger in size (though not in number).



Graph 4.1: Comparing performance with increasing periodic utilisation.

Using a Variety of Periodic Task Sets

Table 4.2 compares the performance of the algorithms across a variety of periodic task sets. Periodic task set (1) is adapted from an avionics case study developed by Locke et al. [36]. It consists of 15 periodic tasks with a wide range of periods from 250ms to 10000ms. Periodic task set (2) has a set of periodic tasks with a low breakdown utilisation (80%), and task set (3) has a high breakdown utilisation (100%). Task sets (2) and (3) evaluate the performance of the algorithms with task sets which are intrinsically difficult to schedule (2), and easy to schedule (3).

All three task sets have a periodic utilisation of 80% and are sent the same sporadic tasks as previously. Task set (1) has 10 tasks below the average sporadic position in the task list while task set (2) has 3 tasks below and (3) has 2 tasks below. It is worth noting that all the periodic tasks for Table 4.2 are rate monotonic (i.e. deadline = period) and have relatively large amounts of slack time associated with them.

Clearly Bottom-up Hybrid consistently outperforms the other algorithms across the variety of periodic task sets. Once again the maximum schedulability test time was not allowed to exceed 10ms which badly affects the guarantee ratios for PP algorithms in the first row of the Table 4.2.

Sched Test Algorithm:	Bottom-up Hybrid		Hybrid		Bottom-up PP		PP		O(N^2)		Back-ground
	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	SR
1. Avionics Case Study Task Set	8.80	0.798	9.50	0.786	(>) 10.00	0.414	(>) 10.00	0.369	2.76	0.762	0.667
2. Low Breakdown Utilisation Task Set	3.53	0.824	3.80	0.824	3.41	0.824	3.80	0.824	0.53	0.783	0.652
3. High Breakdown Utilisation Task Set	3.53	0.945	3.80	0.945	3.41	0.943	3.80	0.941	0.53	0.926	0.498

Table 4.2: Comparing performances over a variety of periodic tasks sets.

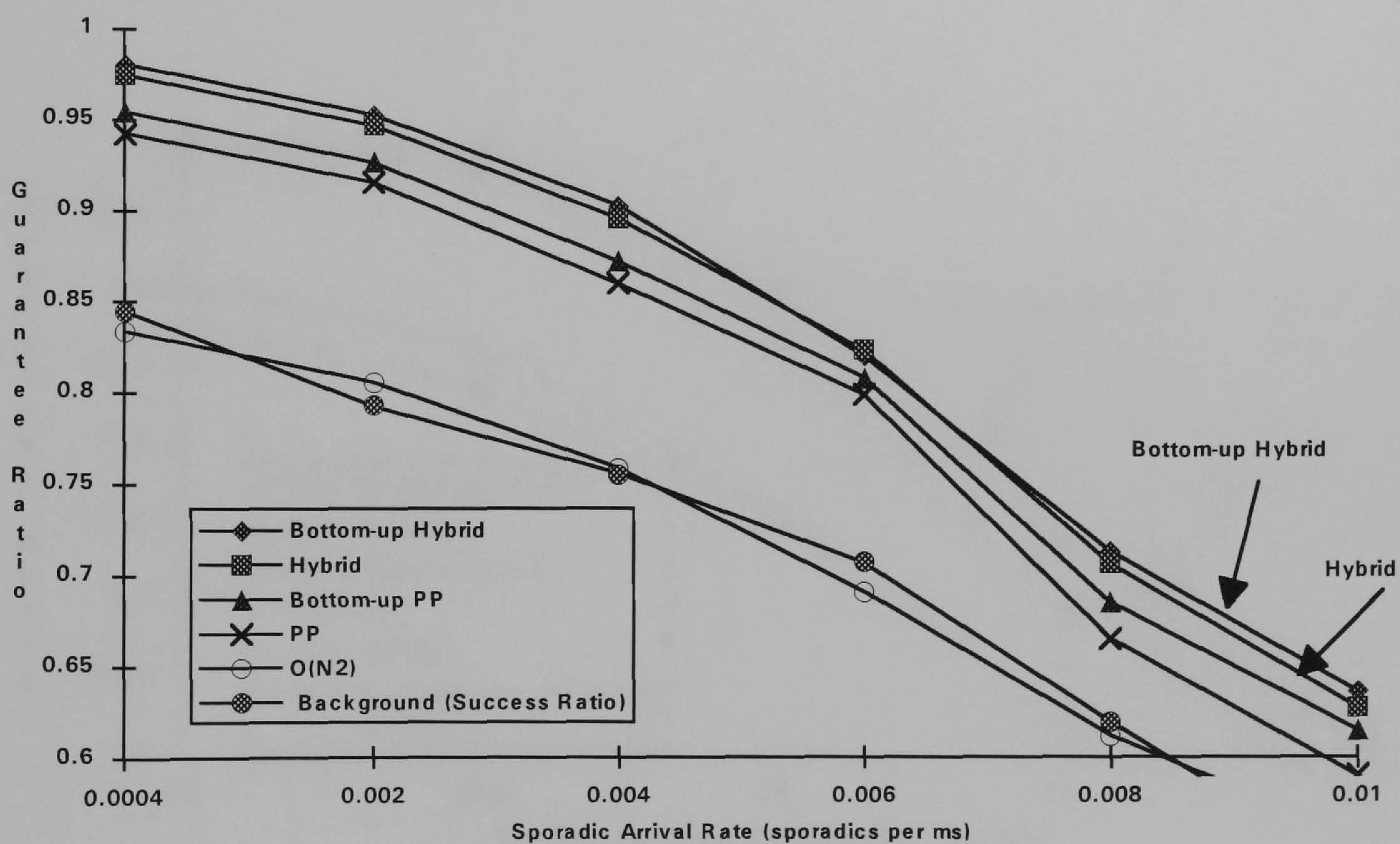
The high guarantee ratios obtained for task sets (2) and (3) are due to the large amount of slack associated with these tasks. In addition it is noticeable that O(N^2) performs better than previously. Again this may be due to greater slack, which means that the pessimism of O(N^2) will count less against it, while O(N^2) retains the benefit of a small upper bound on schedulability testing. The small number of tasks in sets (2) and (3), together with the large amounts of slack, accounts for the closeness of the guarantee ratios across all schedulability test algorithms.

It is interesting to note that Background runs against the trend by performing better with low breakdown utilisation than with high. This can be interpreted as the effect, at low breakdown utilisation, of concentrated intervals of high slack and intervals of zero slack. This benefits background scheduling because this algorithm is penalised less for its indiscriminate processing of sporadic tasks in FIFO order. During high slack, less time is wasted processing sporadic tasks which will eventually fail to meet their deadlines. During zero slack, none of the algorithms can perform sporadic processing in any case. With high breakdown utilisation, and a more even distribution of slack over time, Background wastes more time executing sporadic tasks which eventually fail to meet their deadlines.

4.6.2 Varying Sporadic Task Characteristics

The above results compare the performance of the algorithms while periodic task set characteristics are changed. Now the results of varying *sporadic* task characteristics are presented. Graphs 4.2 and 4.3 show the effect of varying (i) the sporadic arrival rate and (ii) the average sporadic computation requirement. All parameters relating to the periodic task sets remain constant. All results were obtained using the average guarantee ratio from 10 sets of 10 periodic tasks all of which were schedulable and randomly generated to give 85% periodic task utilisation.

Average sporadic arrival rates and average computation times were randomly generated according to a uniform distribution. Realistic arrival rates may be more accurately modelled by a Poisson distribution, however the objective here was to differentiate the performance of the algorithms under different arrival rates. Graph 4.2 uses sporadic tasks with a fixed average computational requirement of 25ms. Graph 4.3 uses a fixed average arrival rate of 0.004 sporadics per ms.

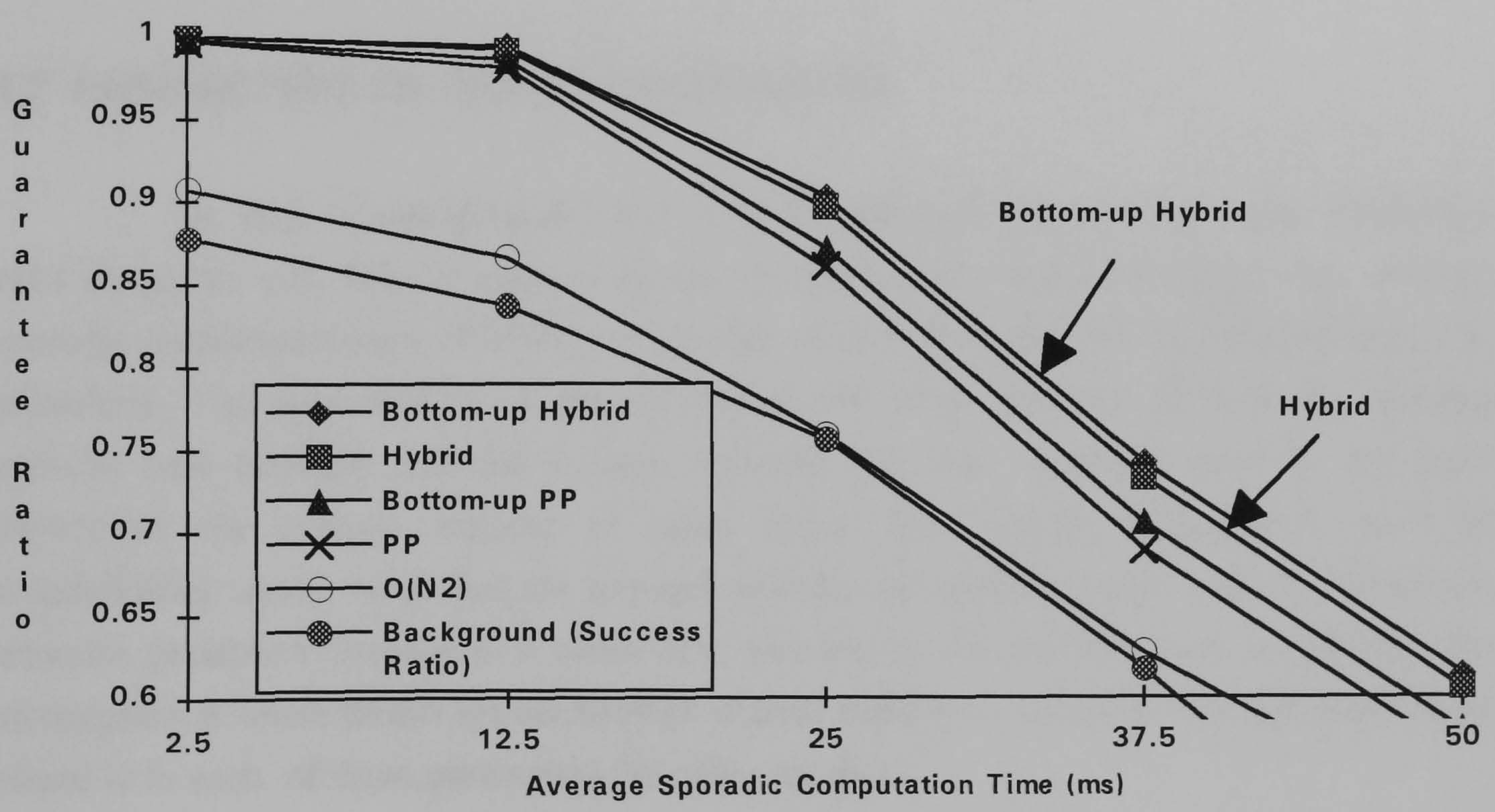


Graph 4.2: Comparing performances over a range of sporadic arrival rates.

As before there is the problem of setting an upper bound on schedulability testing. Here, a different approach is taken. Instead of assuming that the best guarantee ratio occurs when the upper bound has its maximum value, a series of simulation runs were carried out in which the upper bound was reduced in stages and the guarantee ratios measured. The guarantee ratio for each algorithm was seen to peak at a value *less* than the maximum schedulability test value. This was therefore the optimum trade-off between allowing time for schedulability testing and leaving more time for scheduled tasks to run. The peak occurs at what might be called the *optimum upper bound* or *optimum timeout*. The work involved in establishing this bound for each arrival rate, for each schedulability testing algorithm, was prohibitive, so the results in Graph 4.2 use the optimum upper bound established for the *maximum* arrival rate of 0.01 sporadic per ms. This bound was also used for Graph 4.3. The bounds for each algorithm are presented in Table 4.3:

Algorithm:	Bottom-up Hybrid	Hybrid	Bottom-up PP	PP	$O(N^2)$
Bound(ms)	5.5	6.0	7.0	7.5	1.5

Table 4.3: Optimum upper bounds used for the schedulability test algorithms.



Graph 4.3: Comparing performances over a range of average sporadic computation times.

The use of the above bounds for low sporadic loads may be pessimistic, but this should not affect the *comparison* of the schedulability test algorithms. (An investigation into parameters which determine the optimum bound follows in Section 4.7.) Again it is clear that Bottom-up Hybrid consistently outperforms all other algorithms considered.

Furthermore, with some minor exceptions, the results so far are consistent with the following list of algorithms in decreasing order of performance:

Bottom-up Hybrid
Hybrid
Bottom-up PP
PP

Background scheduling is omitted because it does not *guarantee* sporadic deadlines. The question of where $O(N^2)$ comes in the ordering is unclear. Tables 4.1 and 4.2 show that $O(N^2)$ can outperform the pure PP algorithms, and even the hybrid algorithm, when N is sufficiently large. However, this may be due to the PP and hybrid algorithms operating under the handicap of a 10ms timeout. This seems especially likely to be the case for hybrid algorithms because they are based on $O(N^2)$. At the other extreme, $O(N^2)$ gives consistently poorest performance in Graphs 4.2 and 4.3. In summary, none of the results shows $O(N^2)$ outperforming the bottom-up hybrid.

4.7 PARAMETERS OF THE OPTIMUM BOUND

The above investigations show that the optimum value of the upper bound for each algorithm may depend upon a number of parameters: sporadic arrival rate, average sporadic computation time, PD/SD, the number of periodic tasks (N), and the periodic task utilisation. The ratio PD/SD is defined above and takes account of both the average periodic task deadline and the average sporadic deadline. Together with N, this ratio determines the average number of tasks below the sporadic task which must be schedulability tested. Note that the average periodic computation time is not included as a separate parameter because it is taken into account by the periodic task utilisation. The investigations which follow are an attempt to determine how sensitive the optimum upper bound is to each of these parameters. In other words is:

Optimum Upper Bound = f (ave sporadic arrival rate,
ave sporadic computation time,
PD/SD,
N,
periodic utilisation) ?

Investigation of all the schedulability test algorithms would be too time-consuming, so it was decided to select the algorithm with the best overall performance i.e. Bottom-up Hybrid. As before, the approach was to keep all parameters constant, except the one to be varied. The constant values used were:

average sporadic arrival rate (of uniformly distributed times) = 0.004 sporadics/ms
average sporadic computation time (of uniformly distributed values) = 25ms
PD/SD = 0.6
N = 10
periodic utilisation = 85%

A uniform distribution of sporadic task arrival times was chosen in order to provide a constant value and make clearer the effect of varying one other parameter.

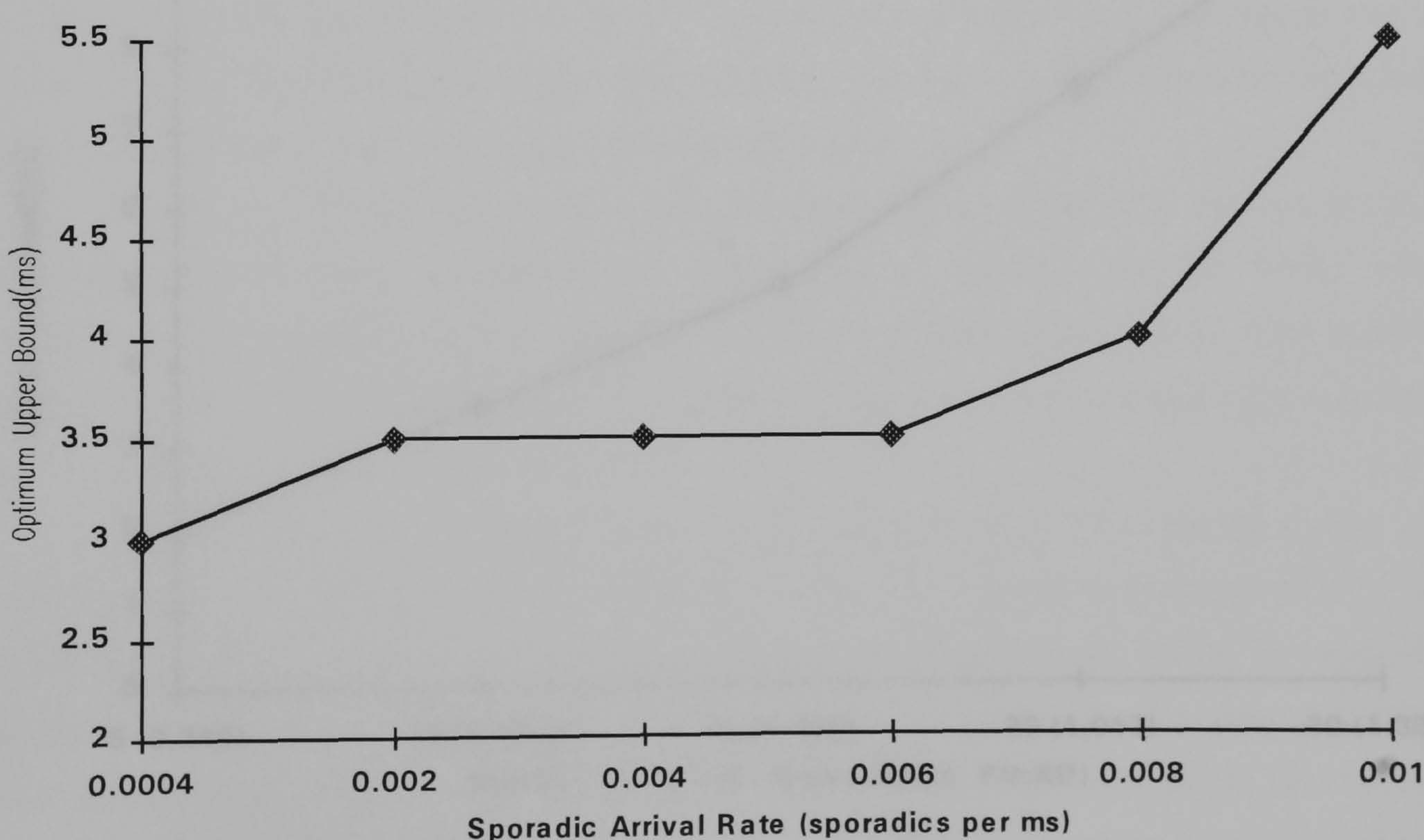
Table 4.4 shows the complete set of guarantee ratios obtained when investigating the effect of average sporadic arrival rate on optimum upper bound. All results are guarantee ratios and the optimum upper bound values are emphasised in bold. All guarantee ratios obtained were averages from 10 sets of 10 periodic tasks. Graph 4.4 is derived from Table 4.4. Noteworthy, is the relatively large increase in bound as the sporadic arrival rate reaches its maximum permissible 0.01 sporadics/ms. This shows that, as sporadic tasks accumulate in a lengthening task list, it rapidly becomes necessary to spend more time schedulability testing, in order to catch those incoming sporadic tasks which are schedulable.

It should be noted in Table 4.4 that the sensitivity of guarantee ratio to the value of the bound is higher at low sporadic arrival rates than at high arrival rates. Table 4.4 also shows that the variation of guarantee ratio with upper bound is a Poisson-like curve. As this curve is compressed by lower bound values, so its shape is emphasised. In other words as the peaks occur at lower bound values, so they become sharper. This has implications for the choice of best optimum bound across a range of sporadic arrival rates.

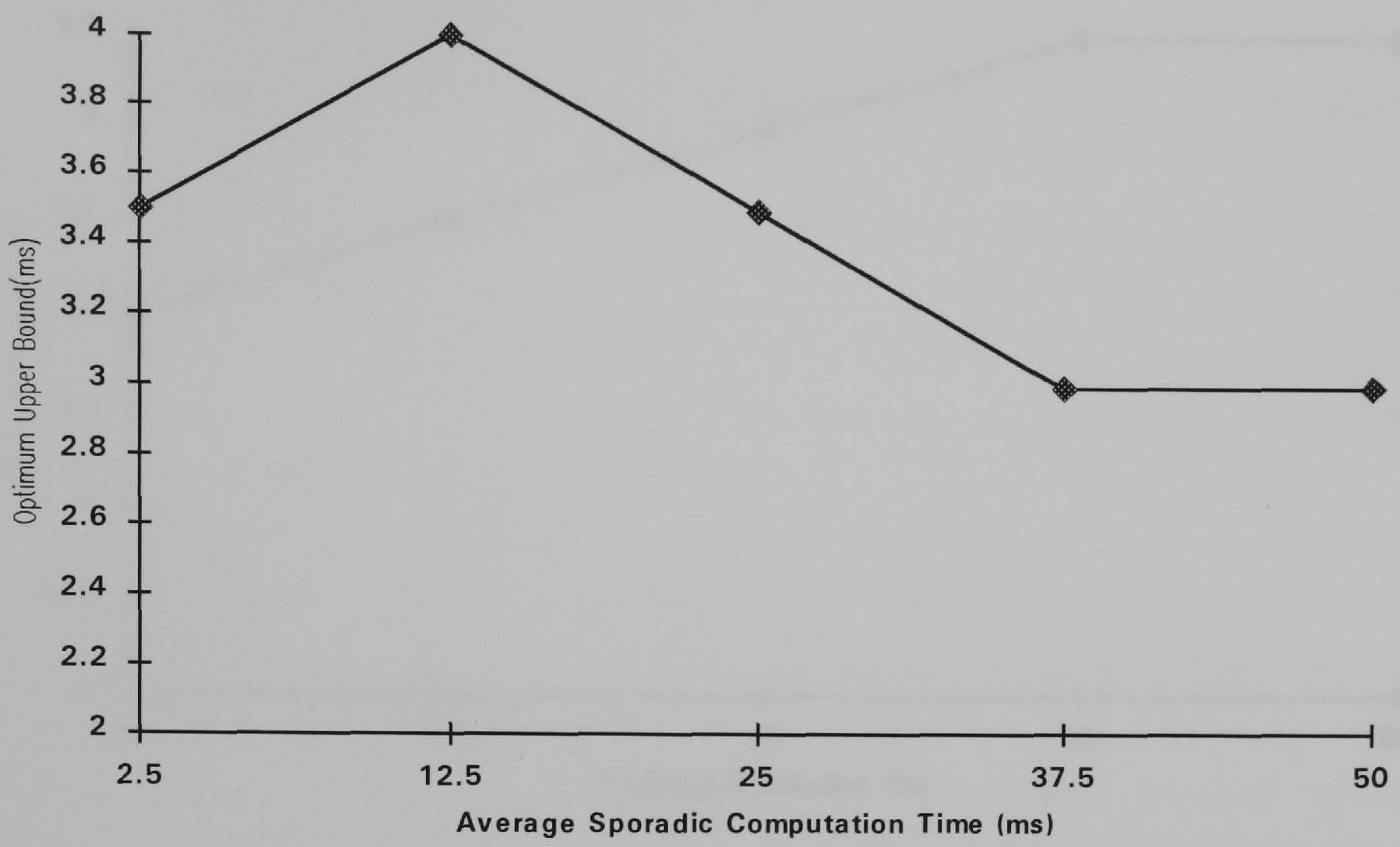
The increase in sensitivity at low peak values was also observed in the data used for Graph 4.5. This graph shows a general decrease in optimum upper bound as the average sporadic computation requirement increases. This can be interpreted as follows: as the average computational requirement of sporadics increases, it becomes less beneficial to spend a long time schedulability testing sporadic tasks which are now more likely to prove unschedulable due to their large computation times. The exceptional result for an average sporadic computation time of 2.5ms can be explained by the guarantee ratio value 'saturating'. The guarantee ratio values for this average sporadic computation time reach a plateau of 1.0 at 3.5ms upper bound and above. In other words all the incoming sporadic

Upper Bound (Timeout in ms)	Sporadic Arrival Rate					
	0.0004	0.002	0.004 (sporadics per ms)	0.006	0.008	0.01
2.0	0.968	0.933				
2.5	0.988	0.962	0.912	0.820	0.704	
3.0	**0.994**	0.968	0.920	0.830	0.714	0.626
3.5	0.993	**0.968**	**0.921**	**0.832**	0.717	0.630
4.0	0.992	0.965	0.919	0.832	**0.718**	0.633
4.5	0.989	0.962	0.915	0.829	0.717	0.634
5.0	0.985	0.957	0.909	0.825	0.716	0.635
5.5	0.981	0.953	0.903	0.820	0.713	**0.636**
6.0						0.635
6.5						0.635
7.0						0.634
7.5						0.632

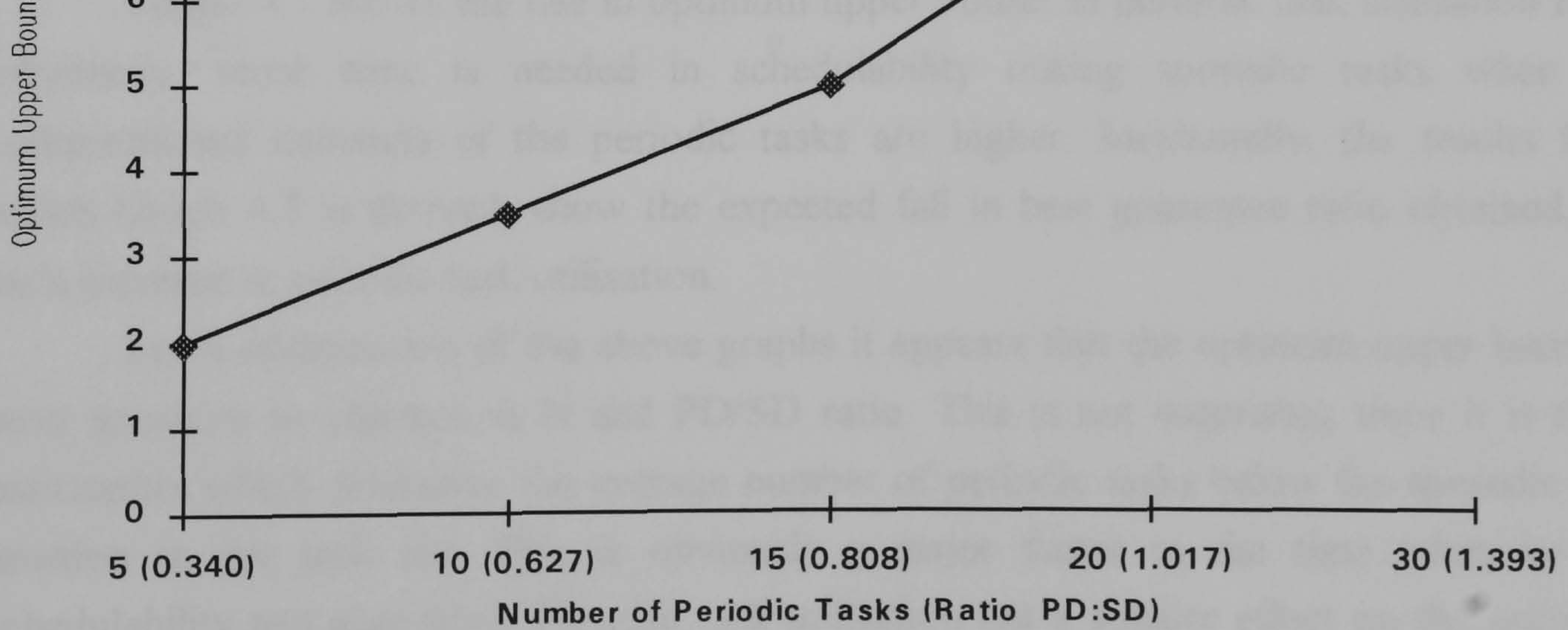
Table 4.4: The effect of sporadic arrival rate on optimum upper bound.



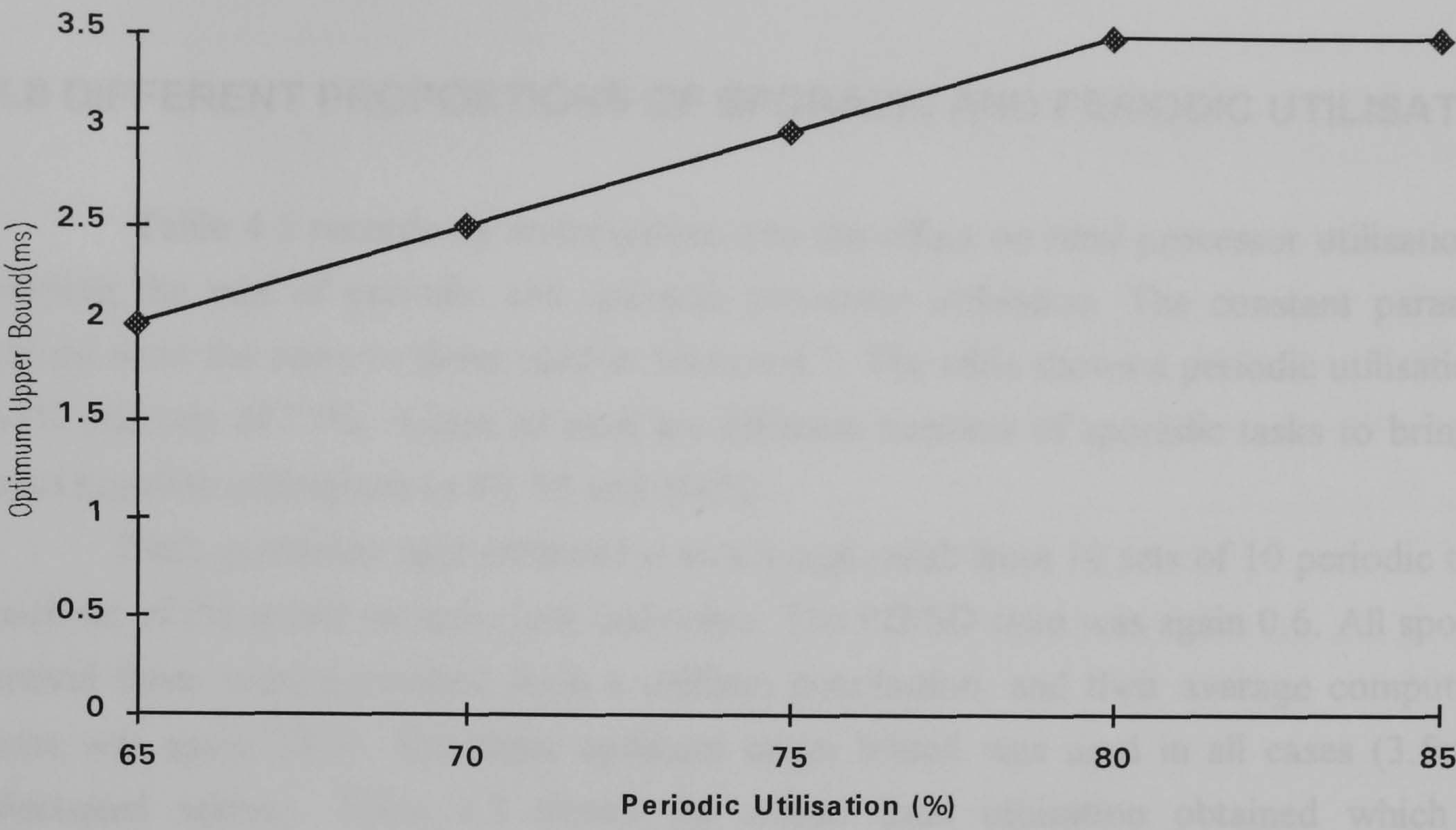
Graph 4.4: Variation in optimum upper bound with sporadic arrival rate.



Graph 4.5: Variation in optimum upper bound with average sporadic computation time.



Graph 4.6: Variation in optimum upper bound with the number of periodic tasks.



Graph 4.7: Variation in optimum upper bound with periodic utilisation.

tasks are being found schedulable even within a tight upper bound of 3.5ms. The algorithm is no longer being stressed, and its upper bound drops.

Graph 4.6 shows the steady increase in the optimum upper bound as N and PD/SD increase. This reflects the need to spend more time on schedulability testing as the number of tasks below the average sporadic task position increases. Unless this is done, schedulable sporadics will be rejected due to a premature timeout.

Graph 4.7 shows the rise in optimum upper bound as periodic task utilisation rises. Obviously, more time is needed in schedulability testing sporadic tasks when the computational demands of the periodic tasks are higher. Incidentally, the results from which Graph 4.7 is derived, show the expected fall in best guarantee ratio obtained, for each increase in periodic task utilisation.

From examination of the above graphs it appears that the optimum upper bound is most sensitive to changes in N and PD/SD ratio. This is not surprising since it is these parameters which determine the average number of periodic tasks below the sporadic task position in the task list. This is obviously a major factor in the time taken by the schedulability test algorithm. Periodic task utilisation has a smaller effect on the optimum bound, and sporadic arrival rates, and computation times, have even less effect. Therefore, in a practical choice of best optimum upper bound, Graph 4.6 is the most important. This suggests an optimum bound of 3.5ms for the final investigation below, which uses PD/SD ratios of around 0.6.

4.8 DIFFERENT PROPORTIONS OF SPORADIC AND PERIODIC UTILISATION

Table 4.5 records an investigation into the effect on total processor utilisation, of varying the mix of periodic and sporadic processor utilisation. The constant parameter values were the same as those used in Section 4.7. The table shows a periodic utilisation of 85% and one of 75%. Added to each are different numbers of sporadic tasks to bring the total possible utilisations to 90, 95 and 100%.

Each guarantee ratio obtained is an average result from 10 sets of 10 periodic tasks, each set of the stated periodic task utilisation. The PD/SD ratio was again 0.6. All sporadic arrival times were generated from a uniform distribution, and their average computation time was again 25ms. The same optimum upper bound was used in all cases (3.5ms as discussed above). Table 4.5 shows the *actual* total utilisation obtained which was calculated from the number of sporadic tasks guaranteed, their average computation times, plus the periodic utilisation. Also to be added, is an estimate of the utilisation used on schedulability testing. This was based on the number of sporadic requests made and measurements of the *average* schedulability test time for Bottom-up Hybrid. This estimate came to 0.48% utilisation per 400 sporadics.

Maximum Possible Total Utilisation %	**85% Periodic Utilisation**			**75% Periodic Utilisation**		
	Number of Sporadics	Guarantee Ratio	Actual Total Utilisation %	Number of Sporadics	Guarantee Ratio	Actual Total Utilisation %
90	200	0.9645	89.823	600	0.9855	89.783
95	400	0.9153	94.153	800	0.9499	93.998
100	600	0.8232	97.348	1000	0.8981	97.453

Table 4.5: Increasing sporadic utilisation by sporadic arrival rate.

Table 4.6 shows the results of a similar investigation in which the sporadic utilisation is increased by increasing the average sporadic computation time, while the number of sporadics is kept constant at 400. (In this case, the utilisation for schedulability testing was the same (about 0.48ms) for all guarantee ratios obtained.)

Maximum Possible Total Utilisation %	85% Periodic Utilisation			75% Periodic Utilisation		
	Average Sporadic Computation (ms)	Guarantee Ratio	Actual Total Utilisation %	Average Sporadic Computation (ms)	Guarantee Ratio	Actual Total Utilisation %
90	12.50	0.9965	89.983	37.50	0.9523	89.285
95	25.00	0.9153	94.153	50.00	0.8655	92.310
100	37.50	0.7610	96.415	62.50	0.7683	94.208
105	50.00	0.6310	97.620	75.00	0.6858	95.574

Table 4.6: Increasing sporadic utilisation by average sporadic computation time (ms).

The conclusions from these limited results are (1) a lower periodic utilisation, and a correspondingly higher sporadic arrival rate, makes no clear difference to the actual total utilisation, and (2) a lower periodic utilisation and a correspondingly higher average sporadic computation time can give a reduction in the actual total utilisation obtained. This second conclusion reflects the difficulty of scheduling sporadic tasks with large computation requirements.

4.9 CONCLUSIONS

This work has investigated dynamic acceptance tests for sporadic tasks arriving at a processor which also runs its own set of resident periodic tasks. Acceptance testing is performed by schedulability tests which run on the target processor itself, and must therefore be upper bound, in order that a worst-case analysis of the processor's load may be made. Knowledge of the minimum interarrival time of the sporadic tasks is a pre-requisite for establishing this upper bound.

The algorithms used for *dynamic* schedulability testing were developed by adapting previously known algorithms for *static* schedulability testing. The adapted algorithms make use of dynamically updated scheduling data. Enhancements were made in order to reduce the run-time overheads incurred by the adapted algorithms. These involved combining two algorithms into a single *hybrid* algorithm, and introducing timeouts into the algorithms in order to enforce tight upper bounds on schedulability testing. Specific conclusions from the simulation results are as follows:

(1) Dynamic schedulability testing of sporadic tasks on the same processor as the periodic task set can incur acceptable overheads of less than 1ms per test.

(2) Bottom-up Hybrid is the most efficient of the dynamic schedulability test algorithms investigated.

(3) The performance of any of the dynamic schedulability test algorithms is sensitive to the choice of upper bound for the worst-case schedulability test.

(4) Constraining the schedulability test algorithm to timeout before the worst-case test time can improve performance. The value of the timeout which gives the best performance is called the *optimum upper bound*.

(5) The optimum upper bound is most sensitive to N (number of periodic tasks) and PD/SD (the ratio of average periodic task deadline to average sporadic task deadline). These parameters determine the average number of tasks below a sporadic task position in static deadline monotonic ordering.

(6) Increasing the sporadic proportion of the total possible processor utilisation, will, if anything, decrease the *actual* total utilisation achieved.

Of all the adapted algorithms, Bottom-up Hybrid consistently performed best over a range of test data which varied all the parameters discussed above. Introducing a timeout improved the performance of all the algorithms, but Bottom-up Hybrid still led the field.

One of the aims of this thesis is to find the most cost-effective run-time support for optional computations. A major part of the overhead incurred by such run-time support will be the acceptance test which is used. Therefore it is important that the overheads of Bottom-up Hybrid, and in particular the high bounds on its WCET, should be reduced. Chapter 5, which follows, attempts to reduce overheads and bounds, by increasing the efficiency of each component of Bottom-up Hybrid.

CHAPTER 5

ENHANCED ON-LINE GUARANTEES

5.1 INTRODUCTION

5.1.1 Approach

Chapter 4 developed and evaluated a set of on-line guarantee algorithms, and found Bottom-up Hybrid to be the algorithm which consistently provided the greatest guarantee ratios. The Bottom-up Hybrid algorithm first attempts a schedulability test by using a pessimistic $O(N^2)$ test, and if this test fails, it then uses the exact pseudo-polynomial (PP) schedulability test. The algorithm schedulability tests the tasks beneath the sporadic task, in "bottom up" order, and finally tests the sporadic task itself. Schedulability testing is abandoned as soon as a task is found to be unschedulable.

This chapter attempts to enhance the performance of the on-line guarantee provided by Bottom-up Hybrid (henceforth be referred to as **BUH**). As in Chapter 4, performance will be measured by guarantee ratio. Three approaches will be explored:

1. Enhancing the performance of the $O(N^2)$ component, by using other sufficient but not necessary schedulability tests.
2. Enhancing the performance of the PP component, by giving the PP algorithm a headstart.
3. Investigating the performance of BUH when *dynamic placement*, rather than static placement, of sporadic tasks within the task list is used.

Each approach is now explained in more detail.

5.1.2 Enhancing the $O(N^2)$ Component

Chapter 4 showed that the performance of BUH degenerated with larger numbers of periodic tasks in the task set. This was due to the large overheads and upper bounds on schedulability testing imposed by the PP component of BUH. Therefore any enhancements in the $O(N^2)$ component which can reduce the need to call upon the PP component, may lead to improved performance. With this in mind, the work of this chapter attempts to make the $O(N^2)$ component less pessimistic by using some of the more exact, sufficient but not necessary schedulability tests, developed by Audsley[2].

Section 2.7 refers to the range of sufficient but not necessary schedulability tests, developed by Audsley as *Tests 1, 2, 3 and 4*. All four tests are able to guarantee tasks, but the tests are numbered in order of increasing schedulability testing overhead, and decreasing pessimism. Test 1 has already been used in Bottom-up Hybrid, and has been referred to as $O(N^2)$. However Tests 2 and 3 also have $O(N^2)$ complexity, and are more exact. Test 4 is the most exact but it has pseudo-polynomial complexity, and therefore, because it is still sufficient and not necessary, it is less promising as a candidate for enhancing the first component of BUH.

The simulations which follow, first compare the performances of the full range of Tests 1 to 4, and then concentrate on the use of Test 3 as an enhancement to the $O(N^2)$ component of BUH.

5.1.3 Enhancing the PP Component

Enhancements to the PP component of BUH could also lower the overheads, and upper bounds, required for schedulability testing. Attempts to enhance this component, centre round the concept of initialising PP with a larger value (or ***headstart***) for the window w_i within which higher priority task interfere. In the original PP algorithm, w_i is set to the WCET of the test task, i, which is being schedulability tested. Section 5.4 below shows that initialising w_i to a value which is greater than the WCET of the test task, can result in PP becoming a sufficient but not necessary test. However, because the overheads and upper bounds for PP can be so large, it may be the case that higher guarantee ratios can be achieved with such an approximate version of PP. This may be especially true under heavy loading. With this reasoning in mind, a range of headstart values were investigated, as are explained in Section 5.4.

The work below goes on to perform further experiments by combining the enhancements to $O(N^2)$ and PP simultaneously in the BUH algorithm. For example, Test 3 is used for the $O(N^2)$ component, while a headstart is also applied to the PP component.

5.1.4 Dynamic Placement of Sporadic Tasks

The third approach to enhancing BUH stems from recent work by Davis [9]. He has proved that the method used so far, for the placement of sporadic requests within the existing task list, is not optimal. Hitherto, sporadic requests have been placed in monotonic order according to the *static deadlines* of existing tasks within the task list. The problem with this method is that, the current *dynamic deadlines* of tasks, which are lower in the task list by static ordering, may be *less* than the deadline of the sporadic task itself. This means that static ordering can cause a sporadic task to be rejected, because the sporadic task has been placed too high in the task list. Instead, the *optimal* position for the sporadic task, is just below the *lowest task* which has a dynamic deadline less than the sporadic deadline. Davis has proven that such placement of the sporadic request is optimal, in the sense that, if the sporadic task is schedulable in any place in the task list, then this placement will also find it to be schedulable.

Dynamic placement is optimal in theory, but this does not mean that it will necessarily achieve greater performance in practice. This will depend on the particular overheads which are incurred, and it may be that in some cases, the overheads of dynamic placement outweigh the fact that it is an optimal positioning. One factor, is that a small additional overhead must be incurred by dynamic placement when it performs a bottom-up search for the optimal position, at the start of a guarantee algorithm.

In order to investigate the effect of dynamic placement on performance, Section 5.7 compares the result of static and dynamic placement for a variety of versions of the hybrid algorithm.

5.2 SIMULATION STUDIES

5.2.1 Introduction

As in Chapter 4, simulation studies were performed, and it was decided that the schedulability testing part of the simulation would run *in real time* in order to measure the overheads incurred by each algorithm. The scheduler itself ran as a simulation according to the scheduling model discussed in Section 4.5.1. As in Chapter 4, the simulations were written in Parallel C, and run on a T800 transputer.

5.2.2 Establishing an Upper Bound for each Schedulability Test

As with the simulations of Chapter 4, it is essential to estimate an upper bound for the execution time of each schedulability test, before performances can be measured. This estimate is the WCET of the high priority periodic task which models schedulability testing. As before, the task set generator uses this value when performing its own (static) schedulability test of the task sets which it generates.

In practice, a WCET of 10ms was used to generate the first task sets. Simulation runs then allowed this value to be refined for each algorithm, and each number of periodic tasks. Algorithms of pseudo-polynomial complexity have particularly large upper bounds. Chapter 4 reported that setting a timeout on the schedulability test generally increases the guarantee ratio obtained. Effectively, this causes the test to be inexact in some cases, but the benefit is to be able to set a smaller upper bound on the schedulability test. For this reason, all simulations were performed with a 10ms timeout in the schedulability test.

5.2.3 Simulation Parameters

All simulation results shown in the following tables and graphs use sporadic tasks whose arrival rates are Poisson distributed ($\mu = 2.8$, $k = 10$) over the total simulation time of 100,000ms. Sporadic deadlines and computation times are normally distributed. The tables show the maximum schedulability test times (in ms), and guarantee ratios obtained, from each of the schedulability test algorithms, as the number of periodic tasks (N) in the task set is increased. The PD/SD ratio and the average number of tasks below the sporadic task position, are also shown in the tables because, as discovered in Chapter 4, these parameters are particularly significant factors in the overheads which schedulability testing incurs.

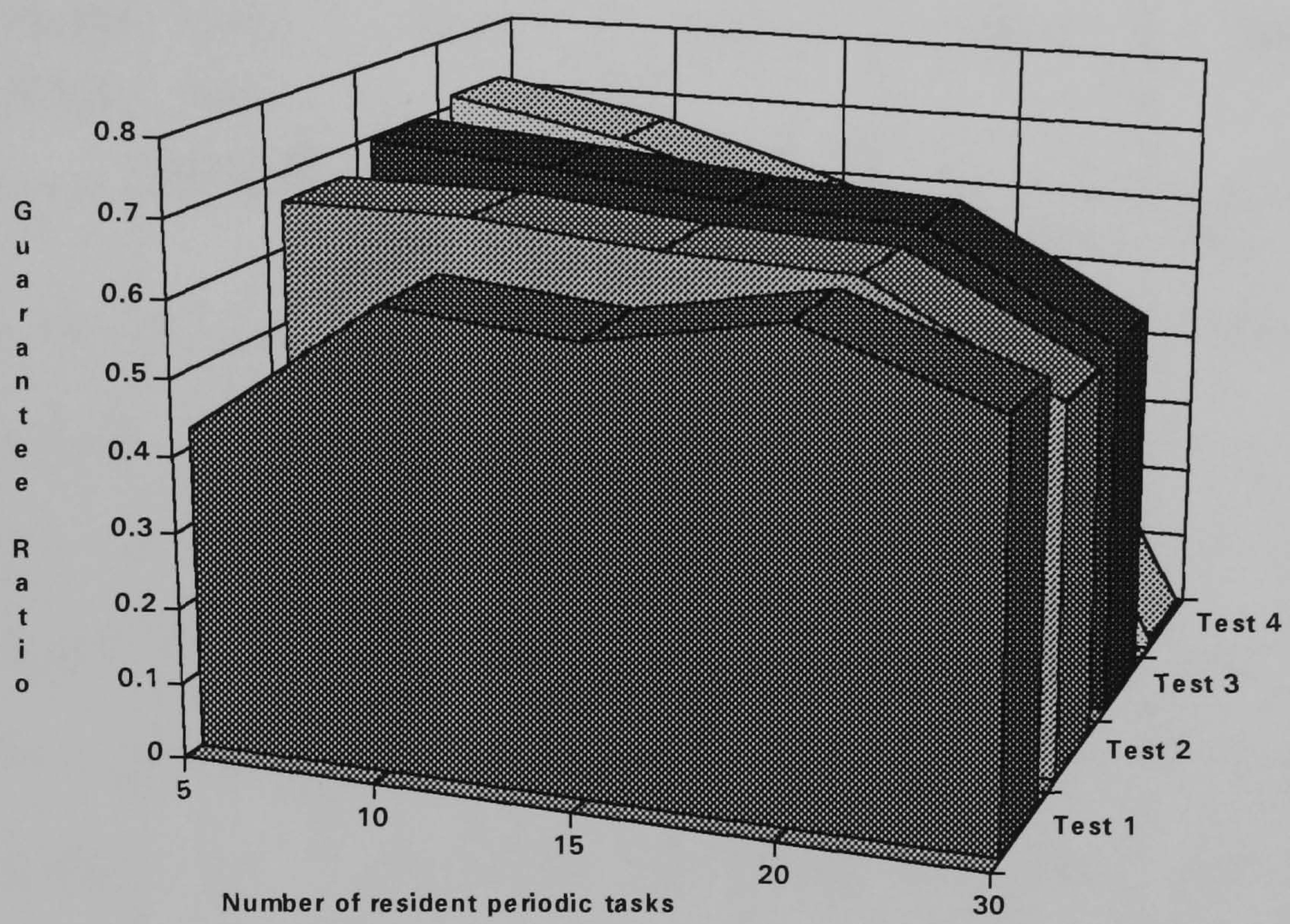
Each guarantee ratio generated, is the overall average of 10 simulation runs, each with 10 different sets of random periodic tasks. Maximum values are the maximum from all 10 runs. Periodic processor utilisations are 85%, which includes scheduling overheads for the periodic tasks, but does *not* include any utilisation by the periodic task which models schedulability testing.

As explained earlier, it was decided not to allow the schedulability tests to overrun 10ms which is the slot size of the scheduler. This curbs the upper bounds required for schedulability testing, and also prevents a violation of the scheduler's upper bound on release jitter (i.e. 10ms). For each simulation the maximum schedulability test time was found. If this maximum was in excess of 10ms it was not used as an upper bound to schedulability testing. However, the maximum is still included in the tables below for information. (It is shown bracketed underneath the "10.00" ms which was actually the maximum schedulability test time permitted). The abbreviation GR in the tables indicates guarantee ratio obtained.

5.3 COMPARING TESTS 1 TO 4

Graph 5.1 and Table 5.1 show the comparative performance of the sufficient but not necessary Tests 1 to 4 which are due to Audsley et al. Static, deadline monotonic placement of the incoming sporadic tasks was used in each case. These algorithms were investigated with a view to improving the performance of the first part of the hybrid algorithm.

It can be seen from Graph 5.1 that, with small periodic task sets (small N), the performance of Tests 1-4 increases with their complexity. This can be interpreted as a result of the small task set which imposes a relatively small schedulability test overhead. In other words, because of the small amount of schedulability testing required, there is sufficient time to make it worthwhile to run the more complex of the Tests. Hence Test 4 achieves the highest guarantee ratio because it is the most exact, and despite the fact that it imposes the greatest overheads.



Graph 5.1: Comparing performances of Tests 1-4 with various periodic task sets

Conversely, with high numbers of tasks in the periodic task set (high N), the schedulability testing overheads for Test 4 increase beyond the 10.00ms timeout, and this rapidly reduces the effectiveness of any extra time spent on testing. The guarantee ratios obtained for Tests 2 and 3 drop off less dramatically. Even for N = 30, Test 2 and 3 rarely impose an overhead which exceeds the 10.00ms timeout. (The highest maximum for Tests 2 and 3 is 11.90ms). Table 5.1 also shows that Test 3 consistently achieves a slightly better performance than Test 2. This can be explained by the use of the effective deadline in Test 3 which makes it less pessimistic. This benefit is obtained with only a very slight increase in overhead.

Test 1 performs worst at low N, because it is the least exact, and therefore most pessimistic, of all these tests. However, at high N, where there are large schedulability testing overheads, Test 1 overheads go up least. (For the test data used, they never exceed

10.00ms.) For this reason Test 1 gives the highest guarantee ratio when N = 30. It is worth commenting that the profile for Test 1 has a different shape than that for Tests 2, 3 and 4: there is a shallow trough in guarantee ratio at N = 15 This can be explained by statistical variations. (It was observed that the guarantee ratios for Test 1 had a particularly wide standard deviation over the 10 task sets used with each simulation run. This standard deviation was approximately 10% of guarantee ratio.)

Number of Periodics (N)	**PD/SD Ratio**	**Ave no of tasks below sporadic**	**Test 1**		**Test 2**		**Test 3**		**Test 4**	
			Max (ms)	**GR**	**Max (ms)**	**GR**	**Max (ms)**	**GR**	**Max (ms)**	**GR**
5	0.340	1.0	0.53	0.424	0.95	0.662	1.02	0.697	1.75	0.716
10	0.627	3.0	1.34	0.606	2.19	0.662	2.15	0.678	4.05	0.671
15	0.808	5.1	2.76	0.585	3.61	0.639	3.54	0.652	6.93	0.602
20	1.017	8.0	4.00	0.633	6.09	0.630	5.95	0.639	10.00 (11.50)	0.506
30	1.393	13.9	9.80	0.547	10.00 (11.90)	0.494	10.00 (11.90)	0.501	10.00 (23.00)	0.014

Table 5.1: Comparing performances of Tests 1-4 with various periodic task sets

Tests 1, 2, 3 and 4 are compared here with a view to improving the $O(N^2)$ component of BUH, which originally used Test 1. It seems that the likeliest improvements might be gained by using Tests 2 or 3 instead. These increase the exactness of the schedulability test whilst still retaining the $O(N^2)$ complexity.

By contrast, Test 4 seems unsuitable for the following reasons. Its complexity is pseudo-polynomial, and therefore its maximum overhead rapidly increases with N. It is therefore unlikely to benefit performance when it is combined in a hybrid algorithm with the exact PP test. This is especially true because a 10ms timeout is being used. (The only possible benefit of using Test 4 in a hybrid algorithm might be at low N where it can give a high guarantee ratio.)

5.4 THE HEADSTART OPTIMISATION

In order to reduce the number of iterations, and thus the overhead, of PP, a possible optimisation to this algorithm is to initialise the interference window to some value which is larger than the computational requirement of the test task, *i*. For example, it could be initialised to the value of the following expression which can be regarded as a 'Headstart' :

$$\Sigma_j(\lfloor (D_i - O_j) \div T_j \rfloor C_j + R_j) + C_i \qquad (5.1)$$

In other words the initial Headstart is the sum of the interferences of each higher priority task, *j* (up to its floor value) plus the computational requirement of the test task itself. Unfortunately, this turns PP into a sufficient but not necessary test. PP becomes pessimistic because the initial window considered may be too large, and therefore may include more interferences from higher priority tasks than is actually the case when a schedule is constructed. The window may then increase in size until the deadline of the test task is exceeded. An example of such a case is shown in Figure 5.1.

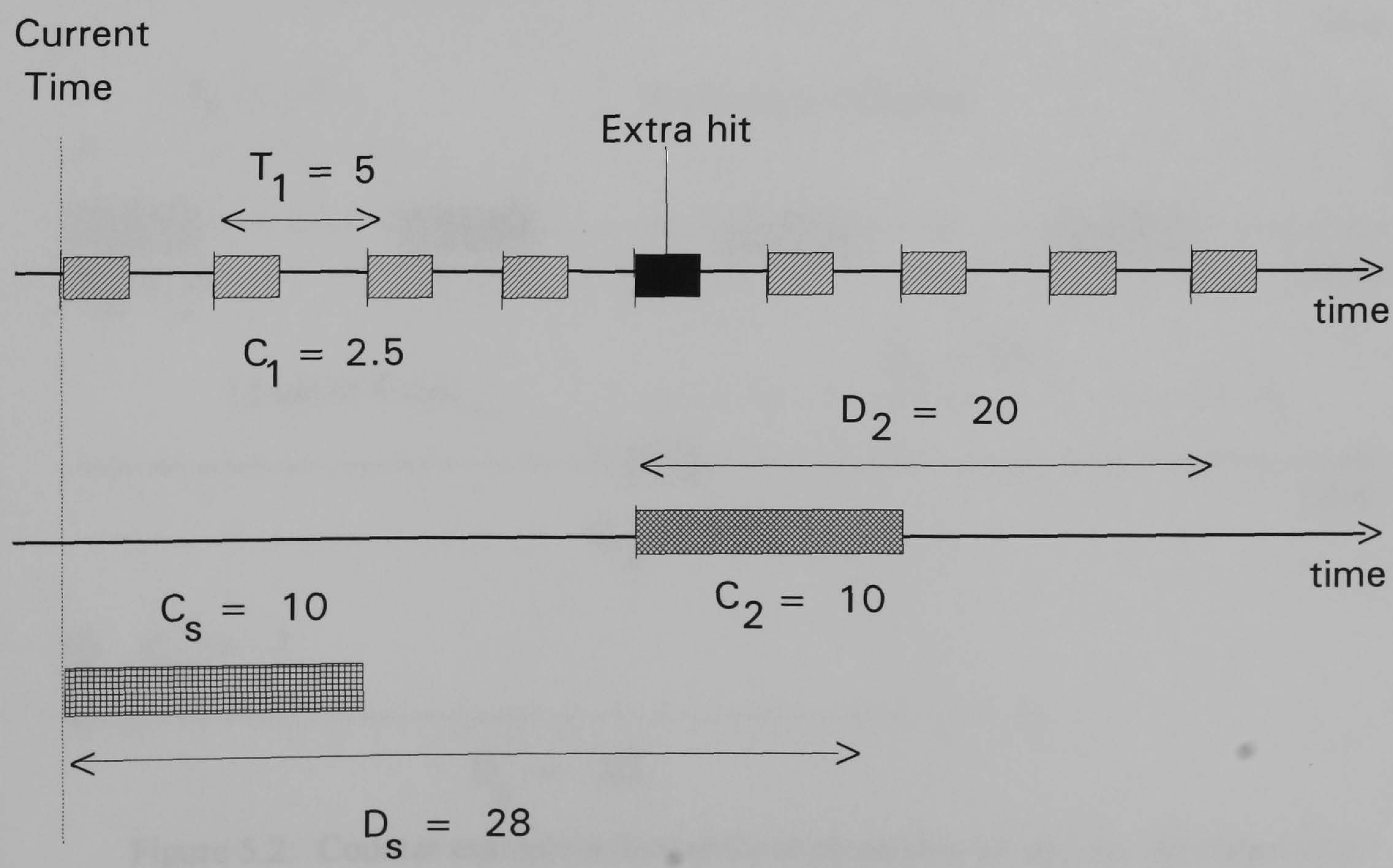


Figure 5.1: Counter example to the original Headstart idea

Figure 5.1 shows two resident periodic tasks and a sporadic task, *s*, which has arrived with a deadline greater than the deadlines of either of the periodic tasks. If *s* is schedulability tested with the Headstart optimisation then expression 5.1 above evaluates to: $(5 * C_1) + (0 * C_2) + 10$. Therefore the interference window will extend further than the release of task 2, and the schedulability test will fail. The problem is that an unnecessary interference (highlighted) of task 1 is included in the initial window. Hence the window becomes pessimistically large. It is worth noting, however, that the test is still *sufficient* because using the floor expression above to calculate interferences from higher priority tasks can never be optimistic.

An attempt to make this optimisation to PP both necessary and sufficient was to first cause the algorithm to calculate the lowest floor of all the interfering higher priority tasks. This is then taken as the initial value of the window of interference for all higher priority tasks. Therefore for a test task i this would be:

$$\min_j (\lfloor (D_i - O_j) \div T_j \rfloor T_j + O_j) \qquad \forall j : (j < i) \wedge (O_j < D_i) \qquad (5.2)$$

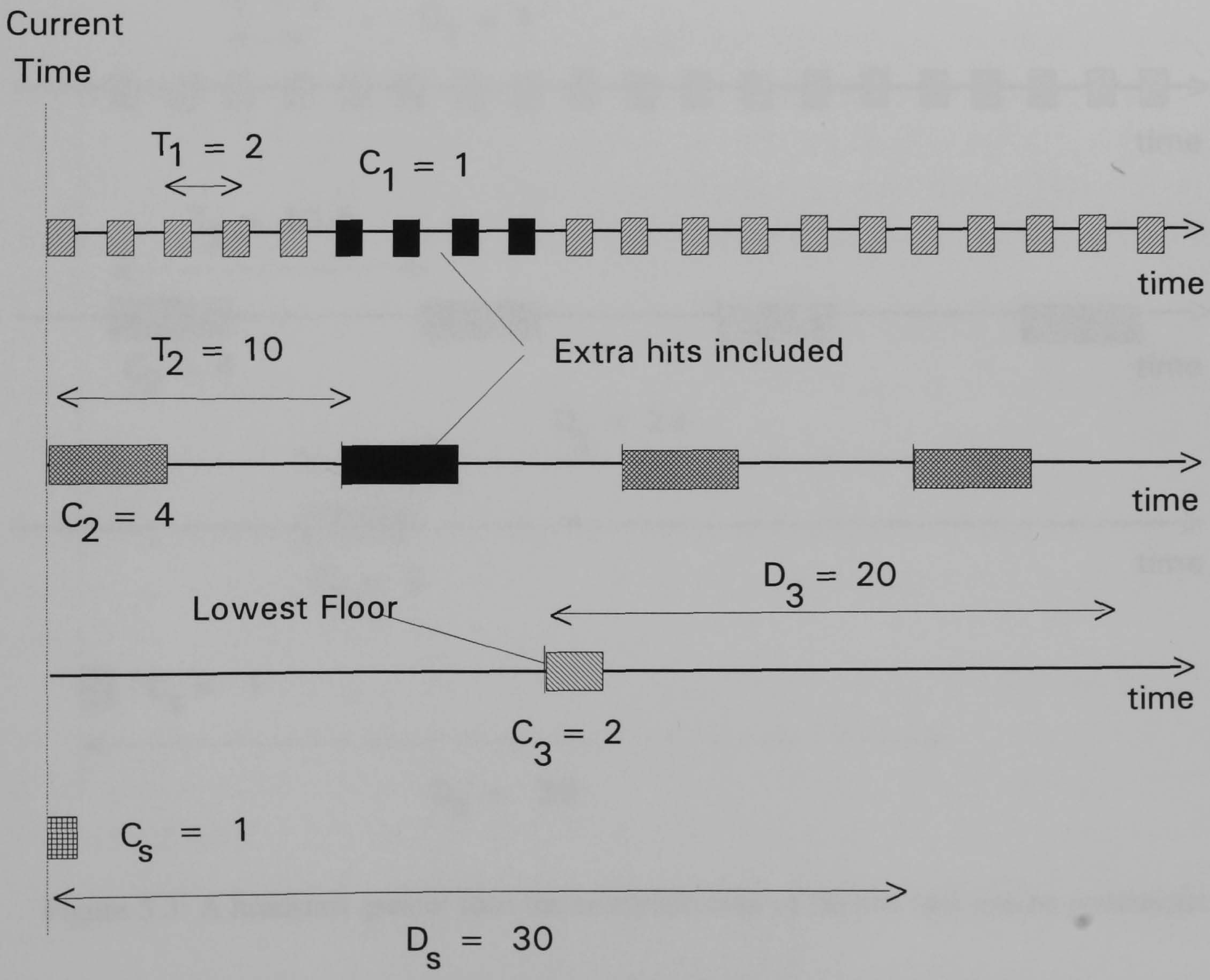


Figure 5.2: Counter example to lowest floor correction to original Headstart idea

The idea is to avoid the inclusion of extra interferences which occur in the latter part of the deadline interval of the test task. Note that it is left to the first iteration of PP to sum the interferences of each higher priority task within the initial window, and then the computational requirement of the test task is added on.

In the case of Figure 5.1, the initial window would be set to 20, the interferences would be calculated within this interval, and the test task would be deemed schedulable. A problem with this approach is that, in general, the lowest floor could be so low as to nullify any optimising effect on PP. Figure 5.2 shows a counter-example which shows that even this method can include pessimistic extra interferences. The early occurrence of sufficient slack for the test task, sporadic task s, is missed at *current time* + *9*. Instead superfluous interferences (highlighted) are included. Hence this amendment to Headstart is also sufficient but not necessary.

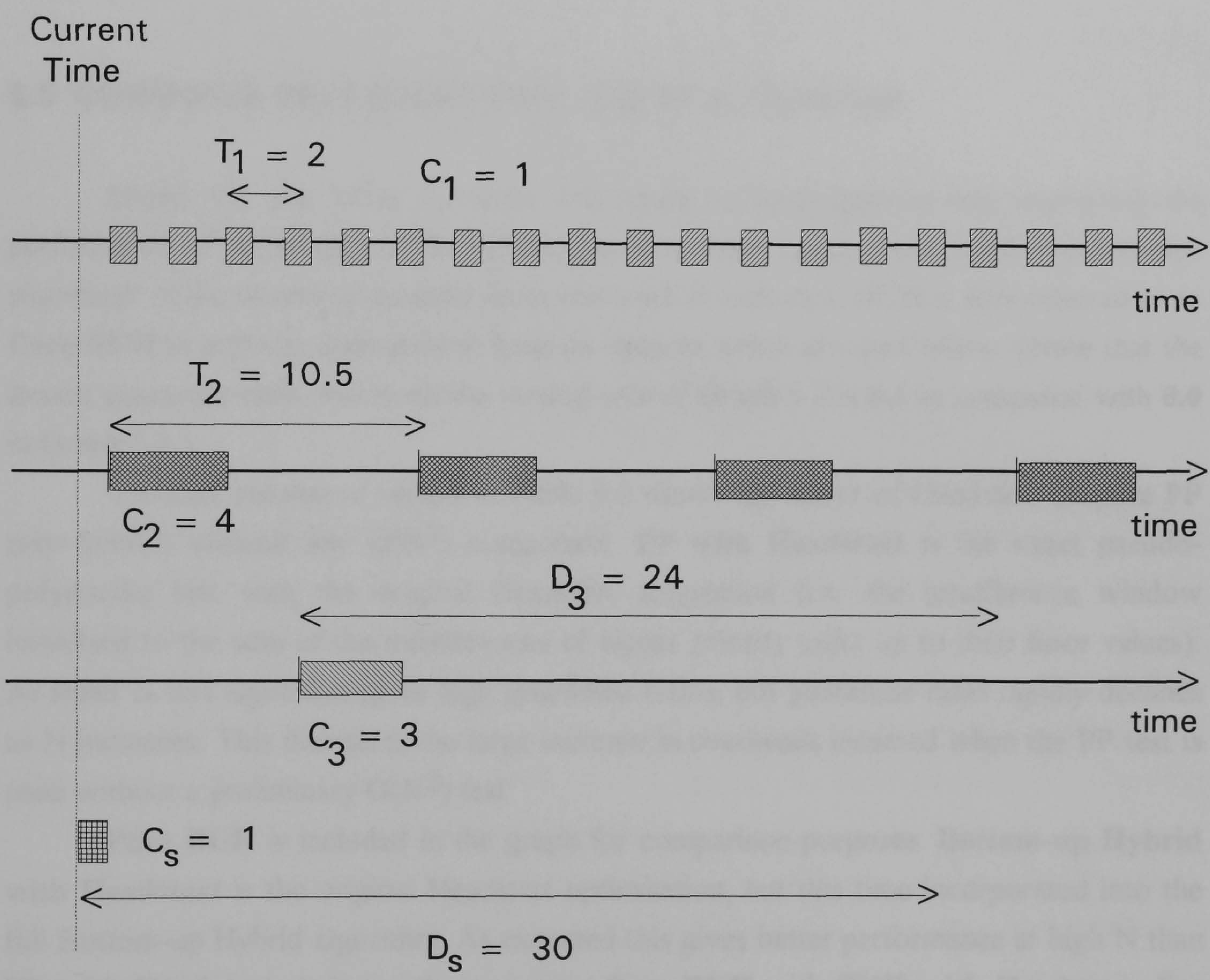


Figure 5.3: A headstart greater than the execution time of the test task can be pessimistic

A more extreme example is shown in Figure 5.3 which shows that, in general, the slack required by the test task may occur at an arbitrarily early point. In this example the necessary slack occurs only at the release of the test task, s. Therefore, to set the initial window to more than the computational requirement of the test task, may lead to the test task pessimistically failing the schedulability test.

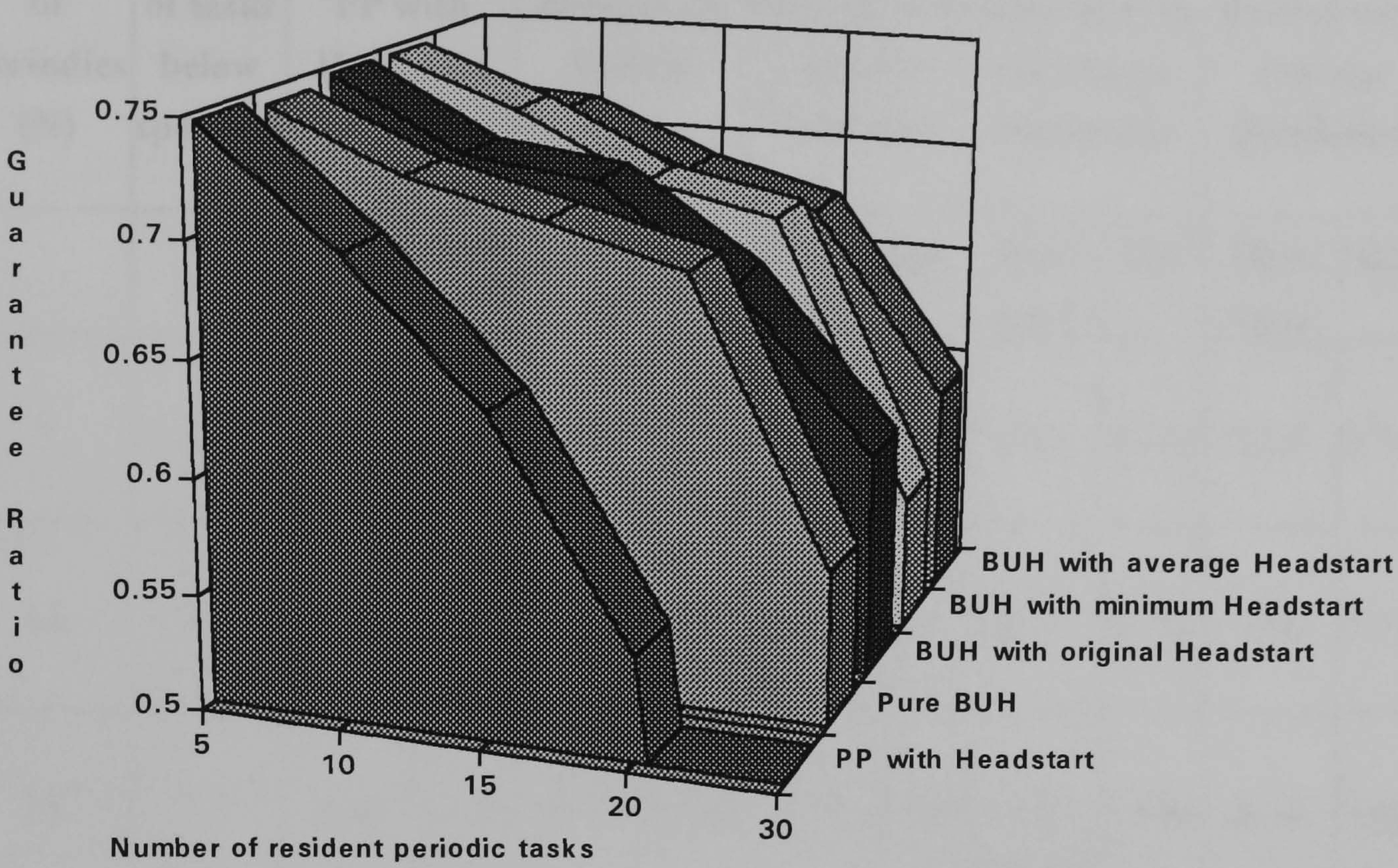
The idea of giving PP a headstart to reduce its large overhead may still have some worth, however. Although it provides a sufficient and not necessary test, a headstart may still increase the guarantee ratio obtained, by cutting down time spent on schedulability testing. Furthermore, a headstart may reduce the maximum bound for the schedulability test and this itself may allow more tasks to be guaranteed. Whether these effects are a benefit in practice requires statistical evidence from simulation studies. For example, it may be that a particular choice of value for the initial window (e.g. half the test task deadline) may provide the best trade-off between the time spent on schedulability testing and the number of tasks which can be guaranteed. Such issues are now investigated.

5.5 COMBINING HEADSTART WITH THE PP ALGORITHM

Graph 5.2 and Table 5.2 show the results of investigations into improving the performance of the PP part of the hybrid algorithm. Once again, static deadline monotonic placement of the incoming sporadic tasks was used in each case. BUH is now referred to as **Pure BUH** in order to distinguish it from its variants which are used below. (Note that the lowest guarantee ratio shown on the vertical axis of Graph 5.2 is **0.5** as compared with **0.0** in Graph 5.1.)

The first column of results in Table 5.2 shows the effect of Headstart on pure PP (non-hybrid) without any $O(N^2)$ component. **PP with Headstart** is the exact pseudo-polynomial test with the original Headstart suggestion (i.e. the interference window initialised to the sum of the interferences of higher priority tasks up to their floor values). At small N this algorithm gives high guarantee ratios, but guarantee ratio rapidly declines as N increases. This illustrates the large increase in overheads incurred when the PP test is used without a preliminary $O(N^2)$ test.

Pure BUH is included in the graph for comparison purposes. **Bottom-up Hybrid with Headstart** is the original Headstart optimisation, but this time incorporated into the full Bottom-up Hybrid algorithm. As expected this gives better performance at high N than **PP with Headstart**. It is worth comparing **Pure BUH** with **BUH with Headstart**. For low N both algorithms give very similar results. However there are differences at N = 20 and N = 30. At N = 20 **Pure BUH** performs better, whereas at N = 30 it performs worse. This can be interpreted as the effect of the pessimism of **BUH with Headstart** showing at N = 20. However, at N = 30, the maximum (**Max**) obtained with **BUH with Headstart** is more significant. (Because the maximum is lower, more sporadic tasks are likely to be guaranteed.)



Graph 5.2: Comparing performances of Headstart Hybrids with various periodic task sets.

The next variation in Graph 5.2 and Table 5.2 is **BUH with minimum Headstart**. This is an attempt to correct the pessimism of the original headstart by taking an initial window equal to the lowest floor of all interfering tasks. As shown in Section 5.4, this "correction" is still pessimistic. However it is worth considering because, statistically, it could yield higher guarantee ratios. As can be seen, its performance is close to that of **Pure BUH**, except for a modest improvement at N = 10, and a deterioration at N = 30. The deterioration at high N can be explained as follows.

For a large number of tasks, it is more likely that the lowest of the floors is near to the current time at which schedulability testing is taking place. In this case **minimum Headstart** algorithm degenerates to **Pure BUH** (i.e. it initialises the window to the computational requirement of the test task). However, it must be remembered that **minimum Headstart** *still* incurs the extra overhead of finding the lowest floor. Hence its disadvantage relative to **Pure BUH** at N = 30. This is confirmed by the maximum schedulability test times obtained for each algorithm at N = 30: **minimum Headstart** is higher (28ms) than **Pure BUH** (20ms), as would be expected.

Number of Periodics (N)	Ave. no. of tasks below sporadic	PP with Headstart		Pure Bottom-up Hybrid		Bottom-up Hybrid with original Headstart		Bottom-up Hybrid with minimum Headstart		Bottom-up Hybrid with average Headstart	
		Max (ms)	**GR**	**Max (ms)**	**GR**	**Max (ms)**	**GR**	**Max (ms)**	**GR**	**Max (ms)**	**GR**
5	1.0	2.50	0.747	3.40	0.743	2.65	0.747	3.11	0.746	2.26	0.718
10	3.0	4.77	0.697	7.42	0.717	4.61	0.715	6.27	0.724	3.90	0.714
15	5.1	8.96	0.638	9.70	0.702	7.75	0.707	10.00 (10.90)	0.695	8.80	0.689
20	8.0	10.00 (14.67)	0.545	10.00 (11.80)	0.686	10.00 (11.00)	0.667	10.00 (15.25)	0.683	10.00 (12.50)	0.674
30	13.9	10.00 (34.90)	0.063	10.00 (20.00)	0.566	10.00 (17.38)	0.598	10.00 (28.00)	0.559	10.00 (24.50)	0.592

Table 5.2: Comparing performances of Headstart Hybrids with various periodic task sets

A comparison of **BUH with original headstart** and **BUH with minimum headstart** is not simple. In general the lower maxima (**Max** values) obtained for original headstart could allow a higher guarantee ratio to be achieved. However, superimposed on this performance trend, is the random effect introduced by headstart. As shown previously, it is an algorithm which can produce pessimistic results. Statistically, it is more probable that original headstart will produce a pessimistic decision. This is due to the larger initial window (i.e. bigger assumption) generated for original headstart. Therefore the relative performances of the two algorithms are subject to statistical fluctuations which impede analysis. An additional effect is introduced by the extra overhead which minimum headstart incurs due to its requirement to find the "minimum floor".

Only at N = 10 does **BUH with minimum headstart** perform better than both **Pure BUH** and **BUH with original Headstart**. This can be explained as follows. It performs better than **Pure BUH** because it still provides some headstart on the initial window size. It performs better than original headstart because it is less pessimistic and this outweighs the fact that it has a higher maximum.

The final variation, referred to in the table as **BUH with average Headstart**, is an attempt to find the "best of both worlds " from original and minimum headstart. The initial interference window is set to *half the value of the test task's deadline*. The hope is that this might strike an optimum balance between the pessimism of original headstart and the larger overhead of minimum headstart. However, Table 5.2 shows that this hope is only marginally realised (only at N = 30).

It is interesting to consider why **BUH with average Headstart** does not perform better. Note that it produces slightly lower guarantee ratios even though its maximum test values are relatively low. Further, note that it incurs no overhead for a search through the interfering tasks (e.g. for the lowest floor). Its puzzling failure can be explained by the fact that *half the test task's deadline* is unlikely to be a *floor value* for any interfering task. This means that the chances of being pessimistic are marginally increased. A floor value excludes the possibility of pessimistically including the whole of a partial hit *for the interfering task whose floor it is.* An arbitrary time (i.e. half the test task's deadline) does not exclude the possibility. This is much the same reason that Test 1 is more pessimistic than Test 2. Such a pessimistic "full extra hit" is more likely to have a greater effect at low N (where the 85% utilisation is distributed between a smaller number of tasks). At high N this effect is less important than the fact that average headstart is actually shortening the time for schedulability testing without imposing any substantial overhead.

5.6 USING BOTH TESTS 1- 4 AND HEADSTART TO IMPROVE HYBRID

Finally investigations were made into the combined use of Tests 1-4 and Headstart in the **Pure BUH** of Chapter 4. As before, static deadline monotonic placement of the incoming sporadic tasks was used in each case.

The first component of **Pure BUH** used Test 1. It was decided to try only Test 3 as an alternative to Test 1 for the following reasons. As shown in Table 5.1, Test 3 produces a higher guarantee ratio, for most values of N, than either Test 1 or Test 2. However Test 3 still benefits from having a complexity of $O(N^2)$. In contrast, Test 4 is pseudo-polynomial and imposes high overheads with rapidly decreasing performance at large N. Therefore

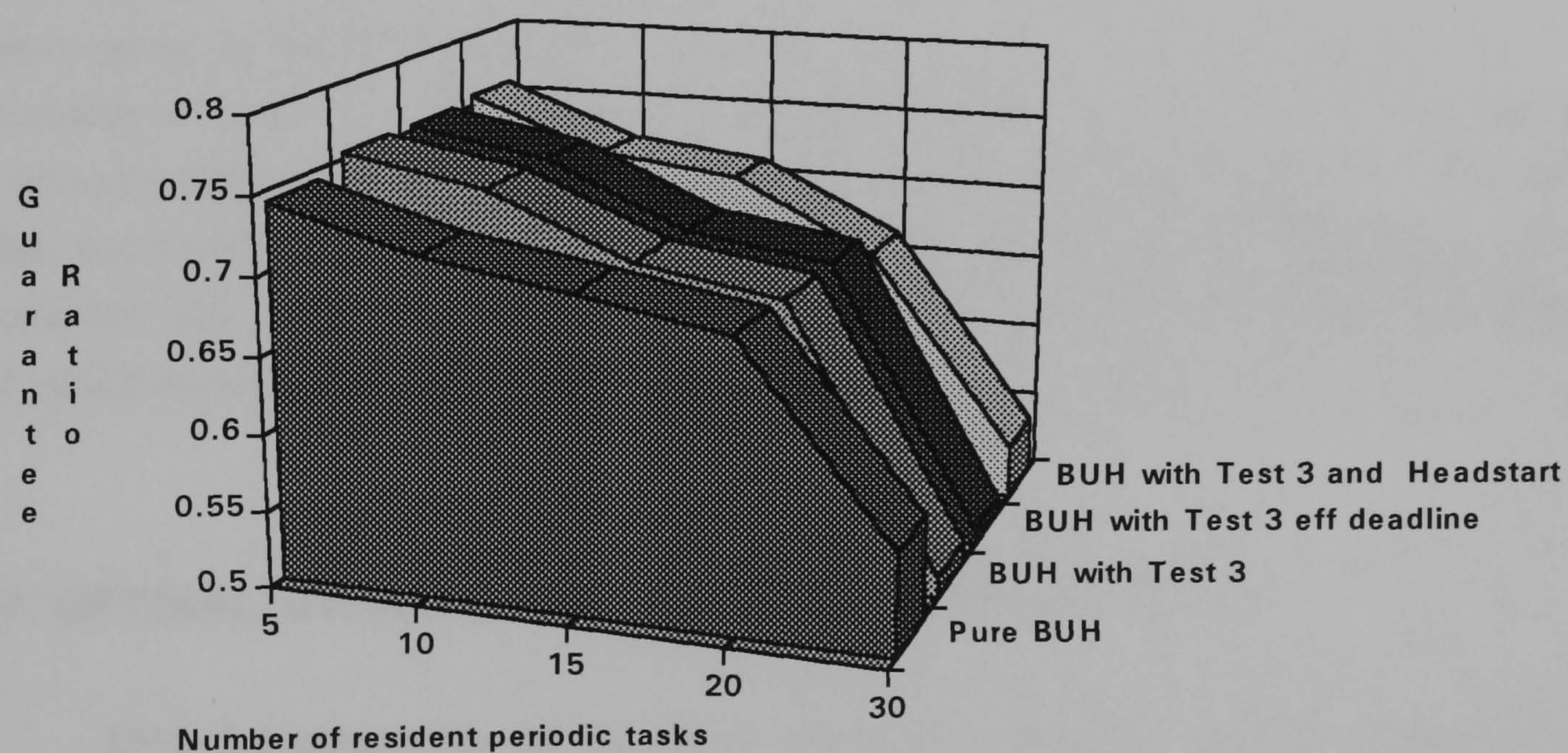
Test 4 has too large overheads to warrant consideration as a replacement for the first component of a hybrid algorithm. It is also worth noting that, despite its complexity, Test 4 is still an inexact test.

Number of Periodics (N)	PD/SD Ratio	Ave no of tasks below sporadic	Pure BUH		BUH with Test 3		BUH with Test 3 and effective deadline		BUH with Test 3 and Headstart	
			Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR
5	0.340	1.0	3.40	0.743	2.40	0.754	2.40	0.754	2.00	0.757
10	0.627	3.0	7.42	0.717	5.31	0.737	5.31	0.737	4.53	0.722
15	0.808	5.1	9.70	0.702	10.00 (10.80)	0.697	10.00 (11.50)	0.695	7.00	0.711
20	1.017	8.0	10.00 (11.80)	0.686	10.00 (12.13)	0.682	10.00 (12.13)	0.681	10.00 (10.80)	0.664
30	1.393	13.9	10.00 (20.00)	0.566	10.00 (33.00)	0.514	10.00 (33.00)	0.511	10.00 (19.50)	0.536

Table 5.3: Comparing performances of Test 3 and Headstart Hybrids with various periodic tasks.

Table 5.3 shows **Pure BUH** together with three variations involving Test 3. **BUH with Test 3** is a straight replacement of the $O(N^2)$ component of BUH with Test 3. **BUH with Test 3 and effective deadline** goes further in that it takes advantage of the *effective deadline* calculated by Test 3. It uses the effective deadline, and not the original test task deadline, as an upper bound on the response window which is iteratively calculated by the PP component. It is always true that *effective deadline* $\leq$ *test task deadline*. Therefore this method may shorten the computation time of the exact PP test while incurring no extra overhead. The final combination in Table 5.3 uses **BUH with Test 3** plus the original Headstart for the PP component. The idea here is that the $O(N^2)$ component is made more exact by Test 3, and that the pseudo-polynomial component is made faster, though generally pessimistic, by Headstart.

It can be seen from the Table 5.3 and Graph 5.3 that **BUH with Test 3** performs better than **Pure BUH** for low values of N, but the situation is reversed for higher values of N. This can be explained by the extra overhead incurred by Test 3. This overhead increases with N, and becomes less cost-effective as it does so. At low N, the overhead is worthwhile because it reduces the pessimism of the $O(N^2)$ component (consistent with the lower **Max** value for **BUH with Test 3** at N = 5, 10). At larger N, however, the reduction in pessimism is a smaller benefit than the increase in the cost of the overhead. In addition the reduction in pessimism which Test 3 brings is less likely to be significant at large N: the greater slack which each task possesses will in any case reduce the likelihood of an $O(N^2)$ test giving a pessimistic result. These effects have been seen before in Table 5.1 where pure Test 3 performs worse than pure Test 1 at high values of N.



Graph 5.3: Comparing performances of Test 3 and Headstart Hybrids with various periodic tasks

BUH with Test 3 and effective deadline makes use of the effective deadline to optimise the pseudo-polynomial component of the hybrid test. Because it incurs no extra overhead one would expect it to produce, if anything, marginally better guarantee ratios than **BUH with Test 3**. Indeed, Table 5.3 shows that for N = 5 and N = 10 the same guarantee ratio is obtained for both algorithms. However, slightly *worse* guarantee ratios are obtained for N > = 15. Some of this can be explained by random fluctuations (0.001 difference in guarantee ratio represents only 4 sporadic tasks). A fuller explanation, however, is required for N = 30 where a 0.003 decrease in guarantee ratio is observed for the **BUH with Test 3 and effective deadline** algorithm. The following interpretation of this may be surprising but 6.11.3 shows evidence for it.

The **with effective deadline** algorithm actually manages to guarantee, within the 10ms limit, some marginal sporadic tasks which can barely be scheduled, and which the plain **BUH with Test 3** algorithm does not have time to schedule. However, this feat does not work to the long term advantage of the **with effective deadline** algorithm. Statistically, these marginally schedulable sporadic tasks may have higher than average computational requirements or some other "difficult" characteristics. This means that the **with effective deadline** algorithm would have been better to reject each marginal sporadic task because it would soon afterwards have been able to guarantee perhaps two more "easy" sporadic tasks. This is what **BUH with Test 3** does, and therefore it achieves a slightly higher guarantee ratio than **BUH with Test 3 and effective deadline.**

The final combination to be tested was **BUH with Test 3 and Headstart**. The hope here is to reduce the pessimism of the $O(N^2)$ component by the use of Test 3, but also to speed up the PP component by the use of Headstart. This strategy seems to benefit guarantee ratio at N = 5. Once again, however, it appears that the Test 3 overhead ceases to be cost-effective at high N. For example at N = 30, even though Headstart is reducing **Max,** and tending to increase guarantee ratio, the effect of the Test 3 overhead is to make guarantee ratio less than that for **Pure BUH**. (This interpretation is corroborated by Table 5.2 which shows that Headstart alone *increases* guarantee ratio at N = 30).

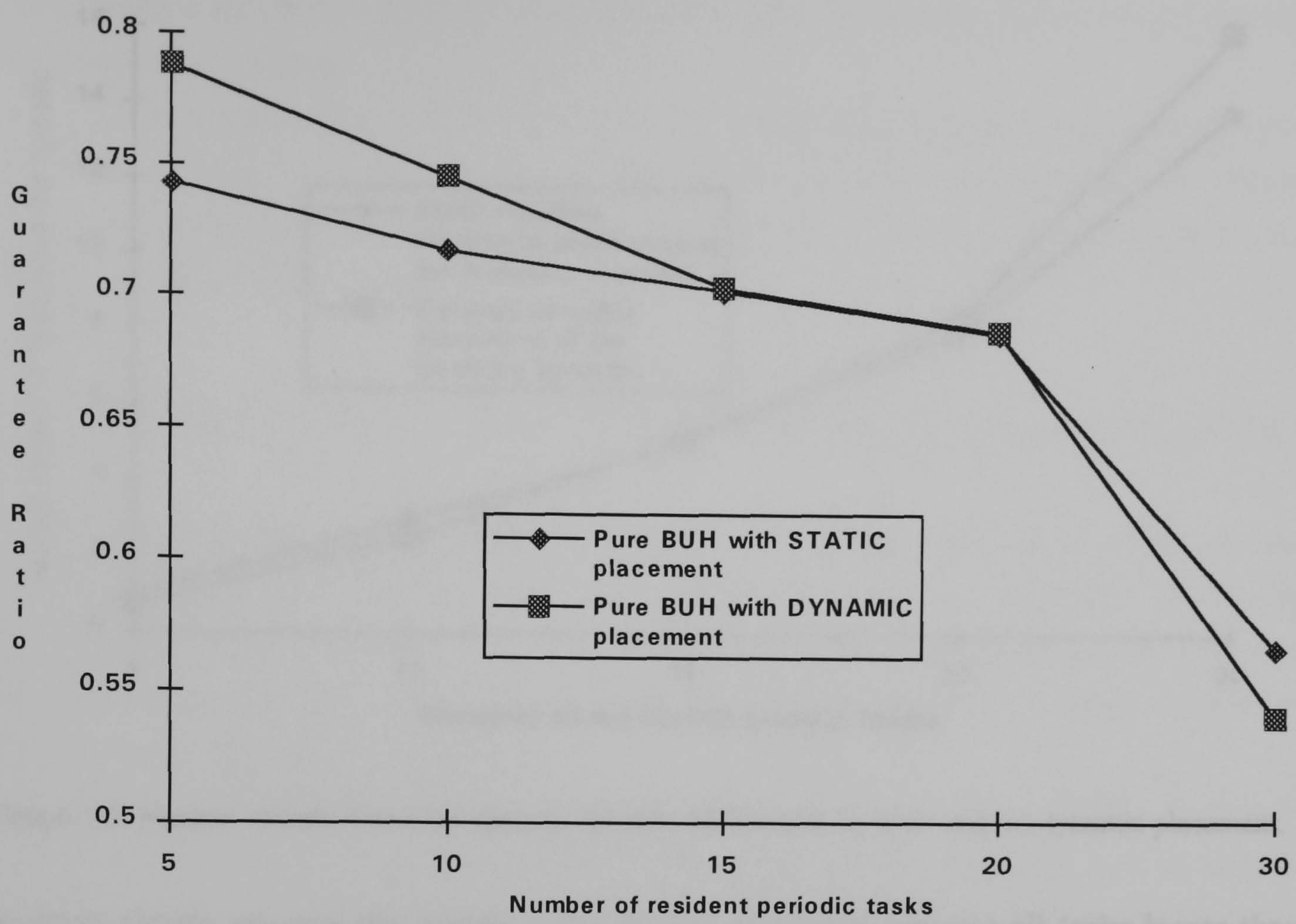
5.7 OPTIMAL, DYNAMIC PLACEMENT OF SPORADIC TASKS

Table 5.4 summarises the algorithms which give the highest guarantee ratios for at least one value of N. Each highest value is shown in bold. None of the other algorithms investigated produced the highest throughput for a particular number of periodic tasks. Therefore the hybrid algorithms shown in Table 5.4 were the ones which were used in an investigation into the effect of placing the incoming sporadic tasks in the task list according to the dynamic deadlines of the existing tasks.

Number of Periodic Tasks	Ave no of tasks below sporadic	Pure BUH		BUH with Headstart		BUH with Test 3		BUH with Test 3 and effective deadline		BUH with Test 3 and Headstart	
		Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR
5	1.0	3.40	0.743	2.65	0.747	2.40	0.754	2.40	0.754	2.00	**0.757**
10	3.0	7.42	0.717	4.61	0.715	5.31	**0.737**	5.31	**0.737**	4.53	0.722
15	5.1	9.70	0.702	7.75	0.707	10.00 (10.80)	0.697	10.00 (11.50)	0.695	7.00	**0.711**
20	8.0	10.00 (11.80)	**0.686**	10.00 (11.00)	0.667	10.00 (12.13)	0.682	10.00 (12.13)	0.681	10.00 (10.80)	0.664
30	13.9	10.00 (20.00)	0.566	10.00 (17.38)	**0.598**	10.00 (33.00)	0.514	10.00 (33.00)	0.511	10.00 (19.50)	0.536

Table 5.4: Hybrids with best performances for particular N for static placement of the sporadic.

As discussed in Section 5.1, recent work by Davis [9] has shown that inserting the sporadic request in monotonic order according to the static deadlines of existing tasks is not optimal. Instead *dynamic placement* must be used. As pointed out already, the question is whether the run-time overheads of dynamic placement are actually justified by an increase in guarantee ratio for sporadic requests. For example, there is an small extra overhead incurred due to the fact that a bottom-up search for the optimal position of the sporadic task must always be made at the start of a schedulability test. (This knowledge is required before a schedulability test can be made.) As has been seen with Test 3 in Section 5.6, such increases in schedulability testing overhead can have a critical effect on guarantee ratio when overall schedulability testing overheads are large.

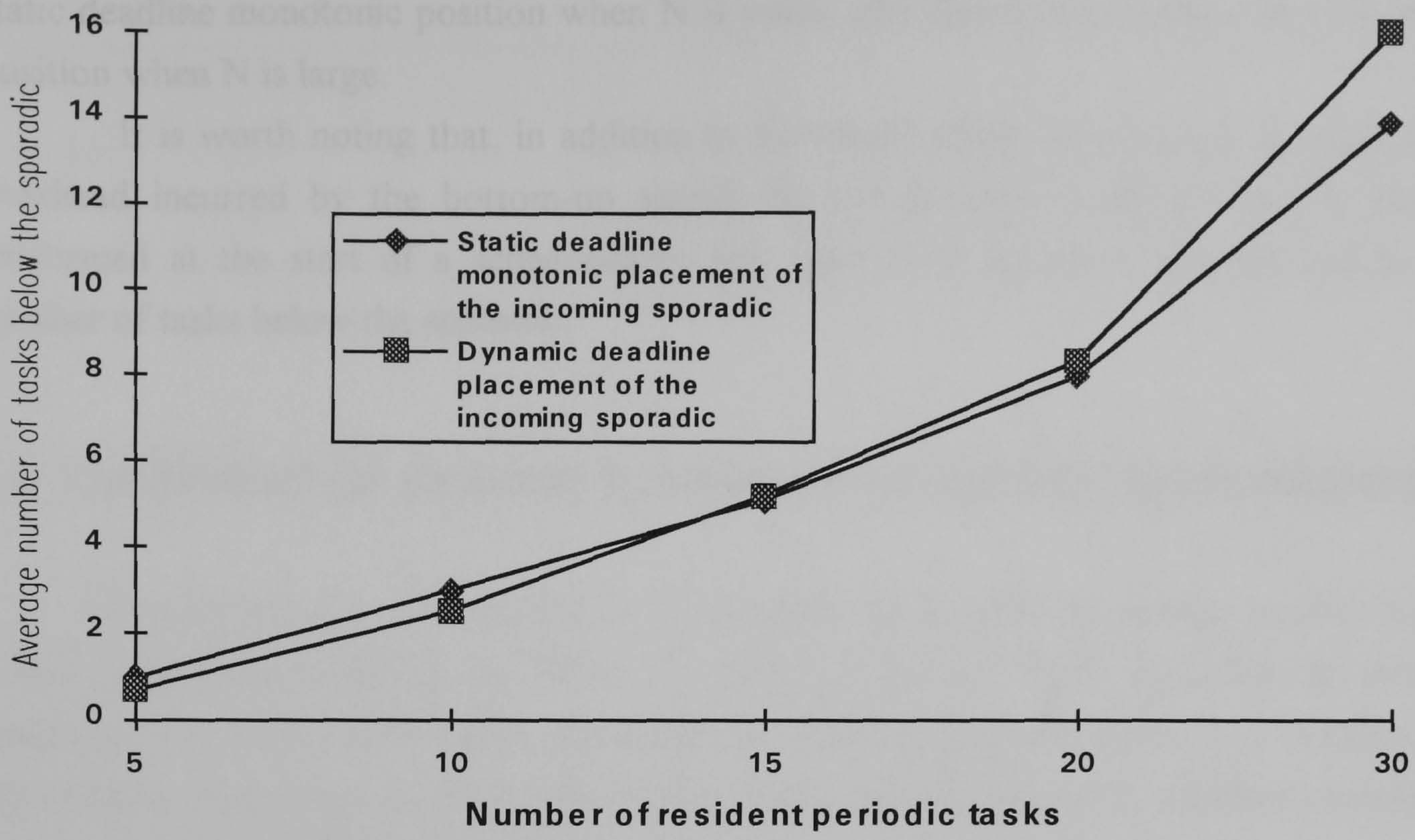


Graph 5.4: Comparing the performances of Pure BUH with (i) static (ii) dynamic placement.

Graph 5.4 shows the results obtained for Pure BUH under the static, and then dynamic, placement of sporadic requests. Clearly dynamic placement improves guarantee ratio at low N, but as N increases the improvement becomes more marginal until, at N = 30, guarantee ratio is actually reduced. In the case of Pure BUH the maximum schedulability testing overheads obtained for each N also confirm that dynamic placement reduces overheads at low N but increases them at high N. At high N this factor will obviously work against the improvement in performance which an optimal placement of sporadic tasks would be expected to give.

Further investigations were carried out into the difference which dynamic placement makes in the behaviour Pure BUH. Graph 5.5 shows the average number of tasks (per sporadic request) below the sporadic position, as the number of resident periodic tasks increases. The averages shown were obtained over simulation runs for both static and dynamic placement as applied to Pure BUH.

Clearly dynamic placement causes fewer tasks below the sporadic at low N but more tasks at high N. Around N = 15 both methods of placement cause approximately equal numbers of tasks below the sporadic. The number of tasks below the sporadic



Graph 5.5: Number of tasks below the sporadic for Pure BUH under (i) static and (ii) dynamic placement.

position closely governs the schedulability testing overheads because all tasks lower than the sporadic task must also be schedulability tested. Hence Graph 5.5 shows why optimal, dynamic placement is clearly beneficial at low N, but not beneficial at high N. At low N, the benefit of optimal placement is combined with smaller schedulability testing overheads, whereas at high N, the benefits of optimal placement are more than offset by the cost in extra schedulability testing.

Possible explanations for the changes in the average number of tasks below the sporadic task are as follows. Note that these explanations assume a constant processor utilisation, while N, the number of periodic tasks, changes.

At low N each task has a relatively high computational requirement and is therefore active (i.e. released but not yet completed) a relatively large proportion of the time. Being active, the dynamic deadline of the task is probably less than its full static value. Hence an incoming sporadic task will be placed relatively low in the task list, beneath the shortened deadlines of active tasks. Contrast this with the likely sporadic position for high N. At high N, each task has a relatively low computational requirement, and is therefore inactive (i.e. completed and awaiting its next release) a relatively large proportion of the time. Being inactive, the dynamic deadline of the task is effectively extended to the deadline following the next release of the task. Hence an incoming sporadic task will be placed relatively high in the task list, above the lengthened deadlines of inactive tasks. This explains why dynamic placement of sporadic tasks places tasks lower than the static deadline monotonic position when N is small, and places tasks higher than the static position when N is large.

It is worth noting that, in addition to the above effect, there is also a small extra overhead incurred by the bottom-up search for the dynamic position. This is always performed at the start of a schedulability test, and its overhead is proportional to the number of tasks below the sporadic.

5.8 THE EFFECT OF DYNAMIC PLACEMENT ON HYBRID PERFORMANCES

The following is a commentary on the effects of dynamic placement on the various hybrid algorithms featured in Tables 5.5 and 5.6 below. Each algorithm is directly compared for static and dynamic placement of incoming sporadic tasks. It is notable that the *relative* performances of the algorithms remain largely the same whether considered under static placement or dynamic.

BUH with Headstart shows the same overall pattern as **Pure BUH**. At low N guarantee ratio is improved, at around N = 15 guarantee ratio is similar, and at high N guarantee ratio drops. The maximum schedulability test values are also affected in the same way as **BUH**. One difference between the algorithms is that the decline in improvement of guarantee ratio seems to start at lower N (N = 15) in the case of **BUH with Headstart**. This may be due to the pessimism of Headstart. As the number of tasks below the sporadic task rises, so does the probability that the Headstart method will prove pessimistic for one of the tasks beneath the sporadic position.

BUH with Test 3 shows similar changes in the pattern of guarantee ratio due to dynamic placement. It is noteworthy that, compared to **Pure BUH**, guarantee ratio drops dramatically at N = 30. This is not surprising when it is observed that the static placement guarantee ratio for **BUH with Test 3** also drops dramatically at N = 30. Section 5.6 explained that this was due to the large increase in Test 3 overhead combined with the fact that, at large N, Test 3 is less likely to make the $O(N^2)$ component of the hybrid test less pessimistic. When sporadic placement is dynamic this effect is further exaggerated by the greater number of tasks which need to be schedulability tested (i.e. below the sporadic position).

It is worth commenting on the fact that Table 5.5 shows the maximum schedulability test value for **BUH with Test 3** actually *increases* at low N. This can be explained by the fact that, under dynamic placement of the sporadic task, the average number of tasks below the sporadic is lower when N is low. Section 5.6 showed that the small maximum values achieved by **BUH with Test 3** are due to the fact that it can make the $O(N^2)$ component of the hybrid schedulability test less pessimistic, and thus prevent a time-consuming invocation of the PP component, and a large maximum test time being

Number of Periodic Tasks (N)	Pure BUH (STATIC)		Pure BUH (DYNAMIC)		BUH with Headstart (STATIC)		BUH with Headstart (DYNAMIC)		BUH with Test 3 (STATIC)		BUH with Test 3 (DYNAMIC)	
	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR
5	3.40	0.743	3.17	0.788	2.65	0.747	2.53	0.791	2.40	0.754	2.88	0.795
10	7.42	0.717	6.82	0.745	4.61	0.715	4.24	0.722	5.13	0.737	6.15	0.751
15	9.70	0.702	10.00 (13.00)	0.703	7.75	0.707	8.00	0.696	10.00 (10.80)	0.697	10.00 (12.50)	0.704
20	10.00 (11.80)	0.686	10.00 (19.00)	0.687	10.00 (11.00)	0.667	10.00 (12.00)	0.659	10.00 (12.13)	0.682	10.00 (18.00)	0.686
30	10.00 (20.00)	0.566	10.00 (35.00)	0.540	10.00 (17.38)	0.598	10.00 (18.00)	0.572	10.00 (33.00)	0.514	10.00 (34.50)	0.439

Table 5.5: Comparing hybrid algorithms for static and dynamic deadline placement of sporadic tasks.

reached. In the case of dynamic placement, there are fewer tasks below the sporadic task. This, in turn, means that there is less chance for Test 3 to make the $O(N^2)$ component less pessimistic, and thus prevent a higher maximum being reached.

BUH with Test 3 and effective deadline shows a slightly different performance profile from **BUH with Test 3**. At low N, guarantee ratio is slightly more improved by dynamic placement whereas, at high N, guarantee ratio is made slightly worse. This is broadly in line with the differences shown between the two algorithms under static placement. Investigations showed that the slight improvement at low N was largely due to an approximately 1% drop in the *average* schedulability test time of the **with effective deadline** algorithm compared to the without effective deadline. This is what might be

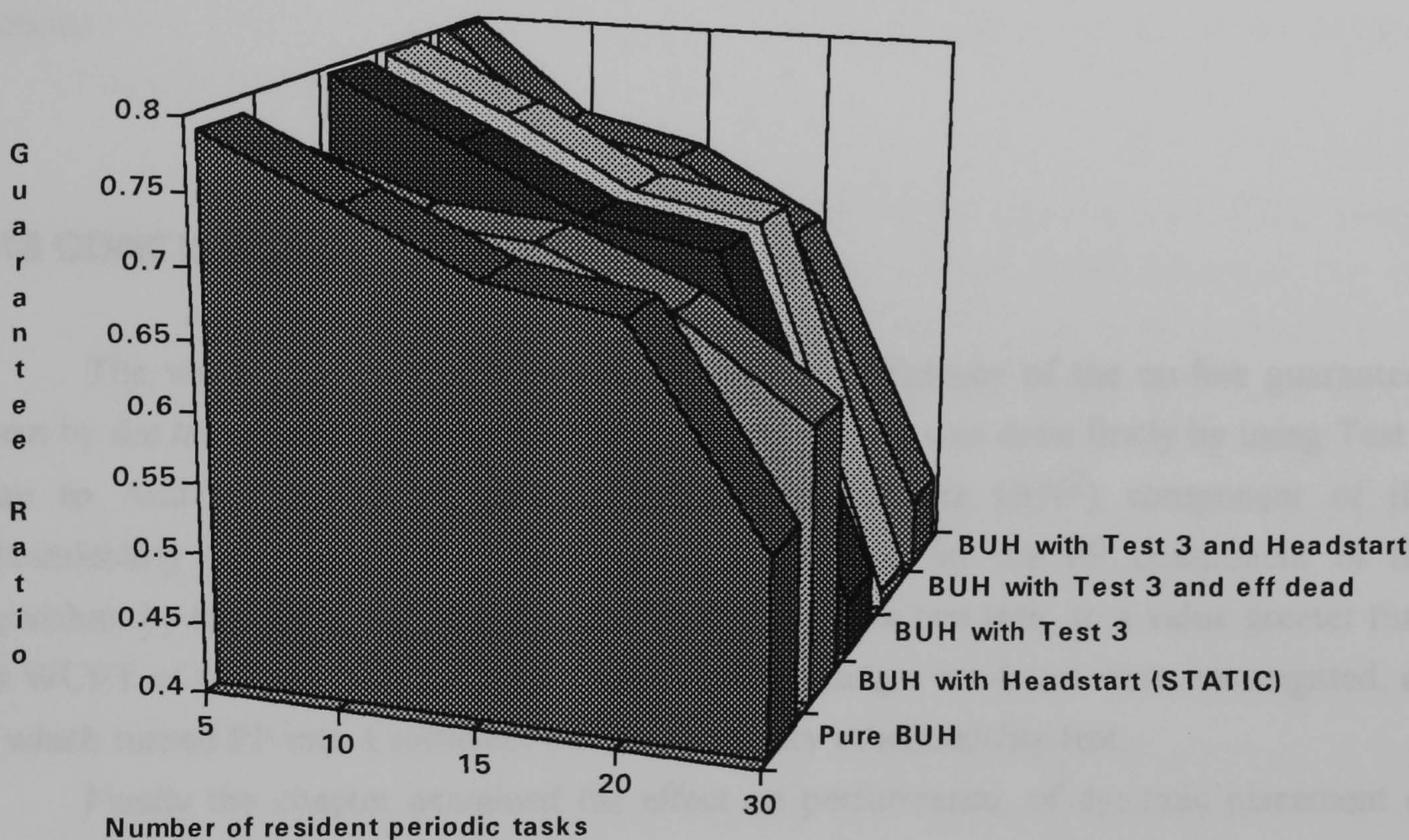
Number of Periodics (N)	BUH with Test 3 and effective deadline (STATIC)		BUH with Test 3 and effective deadline (DYNAMIC)		BUH with Test 3 and Headstart (STATIC)		BUH with Test 3 and Headstart (DYNAMIC)	
	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR	Max (ms)	GR
5	2.40	0.745	2.87	0.796	2.00	0.757	1.89	0.807
10	5.31	0.737	6.05	0.755	4.53	0.722	4.12	0.730
15	10.00 (11.50)	0.695	10.00 (12.50)	0.705	7.00	0.711	7.00	0.710
20	10.00 (12.13)	0.681	10.00 (18.00)	0.687	10.00 (10.80)	0.664	10.00 (11.00)	0.658
30	10.00 (33.00)	0.511	10.00 (34.50)	0.412	10.00 (19.50)	0.536	10.00 (19.50)	0.451

Table 5.6: Comparing hybrids for static and dynamic placement of sporadics.

expected when the effective deadline is used to limit the iterations of the PP component of the hybrid algorithm. In contrast, the drop in guarantee ratio for **with effective deadline** at N = 30 is counter-intuitive. However, it may be explained by the same argument as given in Section 5.6 when comparing these algorithms under static placement. The argument is supported by evidence from Section 6.11.3, and leads to the conclusion that a more efficient algorithm for exact schedulability testing does not always improve the throughput of sporadic tasks, especially when a time limit on schedulability testing is in force.

BUH with Test 3 and Headstart also shows the improvement in guarantee ratio at low N, and deterioration in guarantee ratio at high N, which has been typical of the effect of dynamic placement. The maximum schedulability test times shown for this algorithm are generally low due to the effect of Headstart. This is particularly true at N = 5 where the effect of Headstart, plus that of Test 3 and that of dynamic placement, is to generate a maximum of only 1.89 ms. Guarantee ratio is correspondingly high at 0.807. At N = 30, however, the effect of both Test 3 and dynamic placement is to increase the maximum schedulability test time, and to decrease the guarantee ratio obtained. Nevertheless the effect of Headstart is still apparent in that the guarantee ratio for **BUH with Test 3 and Headstart** is greater than that for **BUH with Test 3** or that for **BUH with Test 3 and effective deadline.**

5.9 SELECTING THE BEST OF THE HYBRID ALGORITHMS



Graph 5.6: Performance profiles of the Hybrids which give maximum GR for some value of N.

The above results show that optimal, dynamic placement of the incoming sporadic request has a favourable, or at least neutral, effect on guarantee ratio except when N = 30. At such high N, the schedulability testing overheads are so large that pessimistic and quicker schedulability test methods will, in general, provide a higher guarantee ratio. Such pessimistic methods can include Headstart, or a schedulability test of purely $O(N^2)$ complexity. Even static placement, though not optimal, can produce greater sporadic throughput at high N.

Graph 5.6 summarises the algorithms which give the highest guarantee ratios for at least one value of N. Dynamic placement is used except at N = 30. Generally **BUH with Test 3 and Headstart** performs best for N <= 15. (Exceptions to this are at N = 10 where **BUH with Test 3** and **BUH with Test 3 and effective deadline** give the highest guarantee ratios.) At high N, the extra overhead of Test 3 is no longer cost-effective, so that at N = 20 **Pure BUH** performs best, and at N = 30 **BUH with Headstart** *and static placement* gives the best performance. Comparison of Graph 5.6 (dynamic placement) with Table 5.4 (static placement) shows that the same algorithms perform best for the same values of N. In other words, the issue of whether static or dynamic placement is used, is independent of the relative performances of the hybrid algorithms. Which schedulability test algorithm is chosen in practice, must be decided by the amount and nature of the schedulability testing which is required. Detailed conclusions are given in the following section.

5.10 CONCLUSIONS

The work of this chapter was to enhance the efficiency of the on-line guarantees given by the hybrid algorithm developed in Chapter 4. This was done firstly by using Test 3 (due to Audsley et al.) to reduce the pessimism of the $O(N^2)$ component of the schedulability test. Secondly, a headstart was provided for the PP component of the algorithm, by initialising the window of interference of the test task, to a value greater than the WCET of the test task. Several methods of providing a headstart were investigated, all of which turned PP into a sufficient but not necessary schedulability test.

Finally the chapter examined the effect on performance, of dynamic placement of sporadic tasks, instead of static placement according to the monotonic ordering of deadlines which are current at task release time. Davis has proven dynamic placement to be optimal in theory, but this chapter has shown that such placement does not always perform better in practice. General conclusions to this chapter are as follows, where *performance* is measured by guarantee ratio:

(1) When used separately as schedulability tests, Audsley's sufficient but not necessary Tests 1 to 4, and his exact pseudo-polynomial test (PP), give relatively poor performances compared to the Pure BUH test.

(2) Headstart optimisations to the PP schedulability test can turn PP into a sufficient but not necessary schedulability test.

(3) Headstarts which provide a sufficient but not necessary schedulability test can nevertheless improve performance beyond that attainable with Pure BUH. (This is particularly true when the overheads incurred by schedulability testing are *high*.) The method used to determine a headstart value will, in general, affect the performance obtained for a particular periodic task set.

(4) When the overheads incurred by schedulability testing are *low*, then Test 3 performs better as the $O(N^2)$ component of the hybrid schedulability test algorithm. Conversely, when the schedulability testing overheads are *high*, then Test 1 performs better.

(5) The performance of the hybrid algorithms *may* be improved by the use of optimal, dynamic placement of sporadic requests instead of static placement by deadline monotonic ordering.

(6) Optimal, dynamic placement can improve performance when schedulability testing overheads are low, but can also decrease performance when schedulability test overheads are high. This can be due to dynamic placement giving the sporadic task a higher average position in the task list than the average position for static placement.

(7) The relative performances of the hybrid algorithms are largely unchanged when all of the algorithms are converted from static placement to dynamic placement of the incoming sporadic tasks.

(8) No single one of the schedulability testing algorithms investigated will perform best for all periodic task sets. In particular, different algorithms may have to be chosen for different sizes of periodic task set.

Chapter 6, which follows, uses the **BUH with Test 3 and Headstart** algorithm as a schedulability test running on each of the processors within a multiprocessor cluster. The chapter investigates methods of allocating sporadic requests across the processors of the cluster, in such a way as to maximise the total number of sporadic tasks which can be guaranteed.

CHAPTER 6

ALLOCATION METHODS FOR MULTIPROCESSOR SYSTEMS

6.1 INTRODUCTION

6.1.1 Approach

Section 2.6 reviewed distributed scheduling in the Spring Project. Spring algorithms allow the re-allocation of 'essential' computations which cannot be guaranteed at their node of origin. However, the re-allocation methods which Spring uses, such as *Focused Addressing* and *Bidding*, incur such large overheads, that one or more dedicated system processors are required within Spring nodes.

The object of the work of this chapter is to adopt a similar multiprocessor architecture to that of a Spring node, but to investigate computationally cheaper methods of allocating optional computations to the processors within the node. It is assumed that optional computations arise in the form of requests for the guarantee of aperiodic tasks which may originate from inside or outside of the node or 'cluster'. Each request must be allocated to a processor within the cluster, for acceptance testing. If the acceptance test fails at the processor, then the aperiodic task is rejected by that processor. As before, it is assumed that each processor within the cluster runs its own set of resident periodic tasks, and performs its own acceptance testing.

This chapter reports on detailed investigations into two multiprocessor configurations, each with its own allocation method. The first is *Targeting*, and the second *Shuffle Schedulability Testing*. It is assumed that communications within each of these configurations is sufficiently fast that its delays are negligible, in comparison to the intervals between acceptance tests (slot width of the schedulability-tester) on each of the processors.

6.1.2 Targeting

The configuration for *Targeting* assumes a processor cluster which consists of a targeting processor and three target processors. All processors are assumed to run a set of resident periodic tasks. However the targeting processor also acts as a channel for aperiodic requests which arrive from outside the cluster. Instead of attempting to schedulability test, and run, aperiodic tasks, the targeting processor performs algorithms which allow it to direct each aperiodic request it receives, to the target processor most likely to guarantee it. The targeting processor may also run other kernel activities, providing that all of its critical, periodic computations are still guaranteed.

The targeting methods used by the targeting processor may range from simple Round Robin allocation of requests, to a relatively sophisticated pre-test which is based on recent slack values for tasks, on each of the target processors. The actual guaranteeing of aperiodic requests is performed by schedulability testing on the target processors themselves. In the case of Targeting, if a schedulability test fails, then the aperiodic request is given a final rejection by the cluster.

The choice of *three* target processors was made to reduce the complexity of the simulation studies, whilst still providing sufficient choice of targets to demonstrate the principles involved. In theory, the findings of this work can be generalised to a larger number of target processors.

6.1.3 Shuffle Schedulability Testing

The second configuration which is considered in this chapter consists of only three applications processors configured in a loop. Each applications processor can independently receive aperiodic requests whether they arise internally, or from the external system environment. Each processor attempts to guarantee the requests it receives, but, should a request fail, it is passed on to the next applications processor in the loop for further schedulability testing. In this way, aperiodic requests are shuffled around the cluster. This method is therefore named *shuffle schedulability testing.*

A consideration which closely affects both the targeting and the shuffle schedulability testing configurations, is the extent to which the processors within the cluster are coupled or synchronised. Clearly targeting is an activity which is global to the cluster and therefore benefits from some synchronisation across the processors within the cluster. Similarly the process of shuffle schedulability testing can be speeded up if each of the processors within the loop is performing its schedulability testing simultaneously. Therefore the general assumption throughout the following work is that there is some means for the processors within the cluster to synchronise. The next section discusses the issues which are to be investigated (i) for targeting and (ii) for shuffle schedulability testing.

6.2 ISSUES TO BE INVESTIGATED

6.2.1 Rationale for Targeting

The first issue to be investigated in targeting is the algorithms which are used to target the aperiodic requests onto the application processors. These algorithms will use a slack-based pre-test and will incur various overheads in mapping requests to target processors. The algorithms to be examined will range from Round Robin allocation with no pre-test, through Targeting by the use of a slack-based pre-test, to 'clairvoyant' Targeting where the slack-based pre-test is replaced by the full schedulability test which is later performed on the target processors. This last algorithm acts as a control experiment which is designed to measure the maximum benefit which targeting can produce.

The second issue in targeting is the frequency and methods with which slack is updated. This determines the accuracy of the slack-based pre-test. Of course the update of slack values will incur some overhead on the target processors and a key issue is the trade-off between this overhead and the benefit to targeting of slack values which are more up-to-date. In order to address this issue it is useful to review recent work by Davis on *slack stealing*.

6.2.2 Davis' Slack Stealing Algorithm

Use with Soft Tasks

In [13] Davis et al. present an algorithm which performs exact on-line calculations of the slack within a fixed priority task list. A task's *slack* is defined as: the task's current deadline minus (the task's remaining WCET plus any interferences by higher priority tasks within the current deadline). When the algorithm determines that slack is currently available at all priority levels within the task list, then tasks with soft deadlines can be executed at the highest priority level. This has the benefit of reducing the mean response time of the soft tasks. This dynamic method of slack stealing is more general than the static equivalent due to Lehovsky and Thuel (reviewed in Section 2.4.3). The method can accommodate hard tasks which exhibit *release jitter*. It also reclaims, as extra slack, any *gain time* which results form hard tasks performing better than their projected worst-case. Unfortunately Davis' algorithm incurs large overheads, so that he proposes a less expensive method which performs exact updates of slack only periodically. Therefore, in between updates, only approximate values of slack are available.

Use with Hard Aperiodic Tasks

In [10] Davis uses his slack stealing algorithm in the acceptance testing of hard aperiodic tasks whose deadlines must be guaranteed on-line. He uses Test 2 due to Audsley (see Section 2.7) as a sufficient but not necessary schedulability test for the hard aperiodic task itself, followed by the use of the approximate slack stealer to determine the schedulability of all lower priority tasks. The advantage of incorporating the slack stealer into the scheme, is that soft tasks can be scheduled within the same framework, in the way described in the previous paragraph.

If soft tasks are not required, or can satisfactorily be executed in background, then the use of slack stealing imposes an unnecessarily high and continuous overhead. As has been seen in Chapters 4 and 5, all that is required for the guarantee of a hard aperiodic task, is a single execution of an algorithm such as BUH, when the task arrives. A further disadvantage of Davis' scheme is that it uses Audsley's Test 2 which is a pessimistic test (see Section 2.7). In contrast to this, BUH uses a less pessimistic test initially (Audsley Test 3) followed by an exact test (PP) if the aperiodic task fails to be guaranteed by Test 3.

6.2.3 The Slack-based Pre-test used in Targeting

In discussion of his slack stealing algorithms, Davis [10,13] has pointed to the prohibitive overheads of performing very frequent updates of the exact slack which is available at each level within a task list. In contrast to this, this thesis provides a method of guaranteeing hard aperiodic tasks by incurring the overhead of a *single* execution of one of the hybrid algorithms discussed in Chapters 4 and 5. The slack-based pre-test which is used in targeting, takes advantage of a by-product of these hybrid algorithms. Although the by-product provides only approximate data on slack, this data is 'free' in the sense that it comes, at virtually no extra overhead, from schedulability tests which already run on the target processors. The following paragraph explains how the slack data is derived.

In effect, the hybrid algorithms commence with an approximate calculation of the slack possessed by the aperiodic task which has arrived, and all the tasks which lie below it in the task list. This is because the first stage of a hybrid algorithm applies an $O(N^2)$ schedulability test to the aperiodic arrival, and to all the tasks which lie below it. The $O(N^2)$ algorithm performs a pessimistic calculation of the total interference from higher priority tasks, within the remaining deadline of the test task (see Appendix A.1). The interference is added to the (residual) WCET of the test task, and the resulting sum is compared to the test task's remaining deadline. The difference between the sum and the remaining deadline is actually a lower bound on the slack which is available for the test task. Therefore, if the $O(N^2)$ component of the hybrid algorithm is amended to record this difference whenever a task is schedulability tested, then approximate slack values are available, at little extra overhead, for the aperiodic tasks and all tasks lying below them.

The proposal is to use the approximate slack values dating from the most recent schedulability tests, in order to guide the allocation of the current set of aperiodic arrivals to the three target processors within the cluster. Allocation will be guided by a schedulability pre-test which compares the WCET of the aperiodic arrival with the (approximate) slack values for all tasks lying below the position of the aperiodic task within the task list. As with slack stealing, a condition of acceptance of an aperiodic task will be that the slack available at all lower priority levels is greater than, or equal to, the WCET of the aperiodic. This is the only condition for schedulability which is considered by the pre-test which allocates aperiodic arrivals to target processors.

The pre-test is also approximate because the slack values on which it is based are one schedulability testing cycle, or more, out-of-date. How outdated a slack value is, will depend on how recently the task was schedulability tested. The issue of whether to introduce more frequent slack updates is investigated by the introduction of *dummy* aperiodic requests which incur extra schedulability tests, and force the update of all slack values.

6.2.4 Issues to be investigated for Targeting

1. **Targeting Algorithms:** in particular the mapping of aperiodic requests to the most suitable target processors.

2. **The Slack-based Pre-test:** how often is slack updated, what overheads are incurred, and how can these overheads be reduced?

3. **Ordering of Aperiodic Requests:** should aperiodic requests be presented to the targeting algorithm in FCFS order or earliest deadline order? This may make a difference in the case of sets of aperiodic tasks for which the pre-test cannot provide a preferred mapping onto target processors.

4. **Bottom-up versus Top-down order of schedulability testing:** in Chapter 4 Bottom-up was been found to be most efficient order of schedulability testing the tasks in the task list. However, the pre-test only estimates the schedulability of those tasks *below* the aperiodic position. Therefore it may be more efficient for the full schedulability test to start by schedulability testing the aperiodic request (i.e. top-down order).

5. **Overheads on the Targeting Processor:** more sophisticated targeting methods may incur larger overheads on the targeting processor. This will reduce the utilisation available on the targeting processor for other systems or applications tasks.

6. **Updating slack values for rejected Aperiodic Requests:** the full schedulability test calculates slack values which include the effect of the aperiodic request. If the request is then rejected, the slack values which have been calculated are more pessimistic. The

issue arises as to whether to update with these pessimistic values, or to revert to the more outdated previous values.

7. **Uniform variation in Periodic Utilisation:** targeting should be investigated for different, periodic utilisations across the target processors. The first stage should be to investigate different, *uniform* periodic utilisations across the targets.

8. **Skewed Periodic Utilisations:** targeting should also be investigated for a skewed distribution of processor utilisations across the target processors.

9. **Internally and Externally generated Aperiodic Tasks:** so far consideration has been restricted to aperiodic tasks which are generated outside the cluster and can therefore be directed to the most suitable target. Consideration should also be given to aperiodic tasks which are generated on the target processors themselves. Such a mixture of internal and external requests could occur in a real application.

6.2.5 Issues for Shuffle Schedulability Testing

1. **Internally and Externally generated Aperiodic Tasks:** a mixture of these should also be considered for shuffle schedulability testing.

2. **Synchronisation:** the synchronisation of schedulability testing for different processors within the loop may affect the performance of shuffle schedulability testing.

3. **Variation of Periodic Utilisation:** the effect on shuffle schedulability testing of different periodic utilisations on the processors within the loop should also be investigated.

The following sections now consider the above issues in turn, presenting the results of the investigations which were performed. As before, it is assumed that aperiodic requests have a minimum inter-arrival time so that an upper bound may be placed on schedulability testing. They are in fact *sporadic* requests. In the case of Targeting, it is assumed, in all cases, that there are *three* target processors, and one targeting processor, in a multiprocessor cluster.

6.3 ALGORITHMS FOR TARGETING

6.3.1 Introduction

A range of targeting algorithms were developed from simple Round Robin to a 'clairvoyant' method which actually performs the full schedulability test in advance of allocating the sporadic request to a processor where the schedulability test is repeated. This latter acts as a benchmark for the maximum performance enhancement which can be achieved by targeting. The purpose in developing a range of targeting techniques, each with increasing overhead, was to investigate the benefits gained from executing the targeting algorithms on the targeting processor. As usual there was a trade-off between the time spent on these overheads, and the increase in the chances of guaranteeing the sporadic tasks. The targeting methods developed were as follows:

(i) Round Robin
(ii) Partial Targeting
(iii) Full Targeting
(iv) Ideal Targeting

As has been explained already, Round Robin merely performs a cyclic allocation of sporadic requests onto the three target processors. It is the simplest algorithm, and involves virtually no overhead. Both Partial Targeting and Full Targeting allocate sporadic requests on the basis of slack values calculated at the previous full schedulability test on each target processor. Hence the slack values used by the pre-test are always somewhat out-of-date. Furthermore, these algorithms use slack values which are updated regardless of whether the sporadic requests are accepted or not. Therefore the slack values can be pessimistic. However, this method has the advantages that slack values are as fresh as possible, and that overheads incurred by the pre-test are slightly smaller. (An investigation is carried out in Section 6.9 as to the effectiveness of updating slack values *only* when the sporadic requests are accepted.)

In order to bound schedulability testing there is a constraint of a maximum of one sporadic request to be schedulability tested, at any processor, in any one schedulability test slot.

6.3.2 Partial Targeting

Partial Targeting is *partial* in the sense that the mapping of sporadic requests to processors which pass the pre-test is performed in an approximate FCFS manner. The

Partial Targeting algorithm is specified by the pseudo-code in Figure 6.1. The algorithm is also described in the following paragraphs.

Firstly, Partial Targeting performs a default pre-allocation of requests to targets using Round Robin (RR). Then each request is taken in turn and is pre-tested against targets until it passes a pre-test, or there are no more remaining targets which have not already been allocated *by the pre-test* in this way. If the request passes its pre-test on the target to which it was originally allocated by RR, then that allocation holds good. If the request passes a pre-test on a target which is different from its RR allocation, then the request is re-allocated to the new target. If the request fails its pre-tests on all of the remaining targets, then it retains its default allocation. Obviously the first request has the chance of being pre-tested against three targets, a second request against only two targets, and a third request is simply allocated to the remaining target.

```
perform default pre-allocation of requests to targets by Round Robin (RR).
for each request in turn
   set the current target to be the first of the remaining targets which have not been
                                                   allocated a request by the pre-test
      while (this request has not passed the pre-test on any target) and (there are
      unconsidered targets which have not already been allocated according to the
      pre-test)
            pre-test this request on the current target
            if this request passes the pre-test on a target different from its RR allocation
                  then swap targets between this request and the request which was
                        allocated by RR to the different target
                     /*no reallocation of targets is required if this request passes the
                        pre-test on the target to which it was pre-allocated by RR*/
            else
                  set current target to next  target not already allocated by the pre-test
            end_if
      end_while
end_for
```

Figure 6.1: Pseudo-code definition of Partial Targeting.

The algorithm uses *swapping* in order to speed up the allocation of targets which have passed a request by the use of the pre-test. When a request passes the pre-test for a target other than its default target, then the default target and the new target are swapped.

This transposition places the new target in an indexed position which is regarded as finally allocated, whereas the default target is now in a position of higher index value which is regarded as unallocated by the pre-test. An alternative method of searching all targets for each request may look simpler on paper. However, such a method would require a final stage of sorting the targets which is avoided by the use of *swapping*.

The starting position for the default Round Robin pre-allocation among the three targets is adjusted at every schedulability test slot in order to spread more evenly the allocation of requests. The position is determined by a random choice between the targets which were *not* allocated a sporadic request at the last schedulability test slot. If three sporadic requests were allocated in the last allocation cycle, then the new starting position will be the same target as before.

Partial Targeting is simple and incurs relatively low overheads on the targeting processor. However, it provides a far from optimal mapping between sporadic requests and target processors. It suffers from the major disadvantage that second and third sporadic requests have a reduced choice of targets. For example, this may be particularly inefficient when the first sporadic request has failed the schedulability pre-test on all targets but has been (arbitrarily) allocated to the first target.

6.3.3 Full Targeting

Full Targeting constructs a matrix in order to achieve a more optimal mapping of sporadic requests to targets. The matrix contains the results of the schedulability pre-test for each sporadic request tested against each target.

	Target 0	Target 1	Target 2
Request 0	1	0	1
Request 1	0	0	1
Request 2	1	1	1

Figure 6.2: Example of a Pre-Test Matrix.

Figure 6.2 shows an example of the pre-test matrix. A '1' indicates that the sporadic request has passed the pre-test for the target, a '0' indicates that it failed the pre-test. Full Targeting is specified by the pseudo-code in Figure 6.3. The algorithm is also described in the following paragraph.

Full Targeting proceeds as follows. Firstly, any outstanding requests which are the only ones which have passed the pre-test for a particular target, are allocated to targets in column order. Secondly, those targets which have passed two requests are allocated outstanding requests in row order, within column order. Finally, all remaining outstanding requests are allocated FCFS to unallocated targets. (In this case, FCFS means row order, within column order.) This last stage can include targets which have passed no requests, targets which have passed three requests, and also requests which failed to be allocated at the previous "attempt to allocate requests for targets which pass 2 requests" stage. All of these allocations are held until last because targeting cannot be applied to them, and hence they may as well be made arbitrarily. As with Partial Targeting, the starting position for request allocation is adjusted according to the previous cycle of allocations at the last schedulability test slot.

```
construct the pre-test matrix
/* allocate the requests which are the only ones to pass the pre-test on particular targets*/
for each target in column index order
        if the target has only 1 request which has passed the pre-test then
                if the request has not already been allocated then
                        allocate it to the target
                end_if
        end_if
end_for
/*attempt to allocate requests for targets which pass 2 requests*/
for each target in column index order
        if the target has 2 requests which have passed the pre-test on it then
                allocate the target the first unallocated request in row index order
        end_if
end_for
/*perform FCFS allocation of all remaining requests, on all remaining targets*/
for each target in column index order
        for each request in row index order
                if (the target is unallocated) and (the request is unallocated) then
                        allocate the request to the target
                end_if
        end_for
end_for
```

Figure 6.3: Pseudo-code definition of Full Targeting.

Obviously Full Targeting is a more expensive algorithm which will provide a more optimal mapping of requests to targets. However the algorithm does not provide 'backtracking' which would be optimal, but even more expensive. Figure 6.4 shows an example of a pre-test matrix in which 'backtracking' would improve allocation. Full Targeting would allocate sporadic request 0 to target 0, request 1 to target 1, but would not have foreseen the problem that request 2 remains to be allocated to target 2 on which it failed its pre-test. In contrast, Full Targeting with Backtracking would be able to deduce that sporadic request 2 should be allocated to target 1, in order that request 1 can be allocated to target 2.

	Target 0	Target 1	Target 2
Request 0	1	0	1
Request 1	0	1	1
Request 2	1	1	0

Figure 6.4: Example of a Pre-Test Matrix which requires backtracking.

6.3.4 Ideal Targeting

Ideal Targeting is an algorithm provided as a 'control experiment'. Here the pre-test matrix contains the results of the *full schedulability tests* of the requests on each target. In other words this algorithm is clairvoyant in that it knows in advance the results of trying to guarantee each request at each of the targets. This provides 'ideal' knowledge of schedulability, which, combined with a near-optimal allocation of requests to targets, provides a benchmark of the maximum improvement which targeting can achieve. In practical terms this algorithm is obviously not cost-effective.

6.4 DUMMY SPORADIC REQUESTS

6.4.1 Rationale for the use of Dummy Sporadics

When performing targeting which is primarily based on somewhat outdated slack values, the issue arises as to whether to introduce methods of updating the slack values more frequently. This may improve upon the targeting, but, because slack calculations are performed on the targets themselves, the extra overhead incurred may actually decrease overall throughput of sporadic tasks. A compromise is to ensure that the slack values of the tasks in the task lists associated with each target, are each updated at every schedulability test slot. This will not occur automatically because, unless requests are flowing in at the maximum rate, there will be slots when targets are not allocated requests. Even when a request is schedulability tested upon a target, not all the tasks in the task list will have their slack values updated. This is because only the tasks below the position of the sporadic request are schedulability tested.

The method of using dummy sporadic requests allows slack values to be updated whether a target has been allocated a sporadic request or not. Effectively, targets which are not allocated a true sporadic request are allocated a dummy request instead. This causes a schedulability test to take place and therefore slack values to be updated. However, because the request is marked as a dummy no sporadic task is inserted into the task list. By setting both the deadline of the dummy request to be the shortest in the task list, and by setting the computational requirement of the dummy task to be a token 1 tick, we can force all of the tasks in the task list to have their slack values updated. This is because all the tasks below the dummy will need to be schedulability tested in order to show that the 1-tick computation is schedulable.

Obviously the use of dummy sporadic requests will add to the schedulability test burden on each target processor and the key question is whether this is outweighed by the improved targeting as a result of more up-to-date slack values. As stated earlier, it is hoped that the use of dummy requests to ensure a schedulability test at each schedulability test slot, will incur lower overheads than those experienced by Davis et al. in [13].

6.4.2 Ordering Sporadic Requests

A lesser issue concerns the ordering of sporadic requests which are presented to Partial, Full or Ideal Targeting. As explained above, all of these algorithms have a degree of FCFS in their allocation of requests to targets. Partial Targeting gives the first sporadic request in the list the maximum choice of targets whereas the second request has restricted choice, etc. Full and Ideal Targeting can provide no preferred mapping of requests to targets which have passed three requests or zero requests, and in these cases requests are allocated (arbitrarily) in request index order.

These FCFS or arbitrary orderings raise the question of whether sporadic requests can be systematically ordered in such a way as to enhance sporadic throughput. One such ordering may be to rank the requests in order of increasing relative deadline from the time at which schedulability testing is performed. This would have the effect of giving preference to shorter deadline (and on average smaller computation) requests. Alternatively, the sporadic requests could simply be kept in the random ordering in which they arrived at the targeting processor.

6.5 SIMULATIONS OF THE TARGETING ALGORITHMS

6.5.1 Introduction

Simulations were performed using three resident periodic task sets on each of the three (simulated) target processors. The slight extra overhead of retaining slack values after the schedulability test on each target was included in the simulations. The overhead of targeting itself was measured but was deemed to take place on the targeting processor and had no effect on the target processors. The simulations include no overheads for the communications of targeting information between the targeting processor and the target processors. Fast hardware and a closely coupled cluster can make such overheads negligible, and in any case they do not affect the principle of targeting.

Targeting Method:	Round Robin	Partial Targeting	Full Targeting	Ideal Targeting
Guarantee Ratio 0	0.883	0.846	0.852	0.849
Guarantee Ratio 1	0.882	0.863	0.878	0.886
Guarantee Ratio 2	0.870	0.870	0.865	0.856
Ave Test Time 0 (ms)	-	2.546	2.598	2.632
Ave Test Time 1 (ms)	-	2.330	2.353	2.354
Ave Test Time 2 (ms)	-	2.372	2.355	2.381
Total Computation Time (ms) :	317,450	306,135	308,790	310,210
Sporadic Utilisation achieved (%)	84.65	81.64	82.34	82.72
Total number of sporadics guaranteed:	13,377	13,095	13,176	13,157
Total Schedulability Test Time for Real Tasks (ms):	32,315	31,812	31,486	32,809
Total Schedulability Test Time for Dummy Tasks (ms):	-	40,683	41,578	40,883

Table 6.1: Bottom-up Hybrid (Test 3, Headstart) with sporadics targeted in random order.

The number of periodic tasks on each target (N) was fixed at 15 because this was in the middle of the range of N values used in previous simulations. The schedulability test algorithm used on each of the target processors was **Bottom-up Hybrid with Test 3 and Headstart** because this performed best for N = 15 (see Section 5.9). Dummy sporadic requests were used as discussed above, in order to ensure the update of slack values at every 100ms schedulability test slot. As in previous simulations the results presented are either accumulated totals or averages over 10 simulations. Each of the 10 simulations used a different periodic task set for each processor, for each simulation. Each of the periodic task sets was randomly generated to be unique, schedulable and give a periodic utilisation of 85%. Sets of randomly generated sporadic requests were used which added a maximum attainable 12.5 % sporadic utilisation to each of the targets.

Table 6.1 shows full results for each of the targeting algorithms when sporadic requests are ordered randomly rather than in order of remaining deadline as discussed above. The Guarantee Ratios over 10 simulations, for each of the three targets (0, 1 and 2) are included in the table in order to demonstrate the difficulty of using Guarantee Ratio as a measure of the performance of targeting. As they stand they do not give a clear indication of the performance of each of the algorithms. The average schedulability test time (*Ave Test Time*) on each of the targets is also included to give an idea of the typical times taken to guarantee (or reject) a request at a target. These averages include the times taken for dummy tasks. Round Robin does not use dummy tasks, so its column contains no entry for *Ave Test Time*.

The *Total number of sporadics guaranteed* is the total of all sporadic tasks guaranteed by all three targets over a run of 10 simulations. It gives a fairly accurate measure of the relative performances of the targeting algorithms. A more accurate measure still, is given by *Total Computation Time*. This is the total sporadic computation time over 10 simulations, achieved on the three targets, and is the best measure of the effectiveness of each targeting method. *Sporadic Utilisation achieved* expresses *Total Computation Time* as a percentage of the total possible sporadic computation.

Total Schedulability Test Time for Real Tasks is the accumulated schedulability test time for the actual sporadic requests sent to all three targets over the 10 simulations. *Total Schedulability Test Time for Dummy Tasks* gives a comparable figure for time spent on the schedulability testing of dummy tasks. Again, because Round Robin does not use dummy tasks, this figure is omitted in the Round Robin column. Not included in the table are the maximum schedulability test times recorded over all of the three targets. These were in the region of 5 ms, and therefore this was the worst-case computation time used for the top-priority periodic task which models schedulability testing within the task lists of all three targets.

6.5.2 Interpretation of the Results

A discussion of the results in Table 6.1 is as follows. Clearly *Total Computation Times* for the targeting methods are disappointing in that they are less than that achieved by Round Robin allocation. However, as expected, the more sophisticated forms of targeting perform better than the cruder forms. (Again note that targeting overheads are not included within the overheads of the target processors).

Total No of Sporadics Guaranteed show a similar ordering of performance between the methods, with the notable exception of Ideal Targeting which guarantees fewer sporadic requests than Full Targeting but still provides a greater *Total Computation Time.* This marginal effect can be explained by the slightly greater ability of Ideal Targeting to guarantee, on average, greater computation times for sporadic tasks than Full Targeting. This conclusion is confirmed by comparing the results for *Total Schedulability Test Time for Real Tasks* with *Total Schedulability Test Time for Dummy Task* for the two methods. Ideal Targeting incurs greater *Total Schedulability Test Time for Real Tasks* but a lesser *Total Schedulability Test Time for Dummy Tasks.* This can be explained by the larger *Total Computation Time* guaranteed by Ideal Targeting giving a small general increase in *Total Schedulability Test Time for Real Tasks* due to the presence of slightly more persistent tasks within the task list. However, dummy schedulability testing requires all the tasks in the task list (beneath a high-priority dummy of negligible computation time) to be schedulability tested. Hence it is more sensitive to the *number* of tasks in the task list. Ideal Targeting may have, on average, shorter task lists which would explain the lower overhead for *Total Schedulability Test Time for Dummy Tasks.*

The larger *Total Schedulability Test Time for Real Tasks* for Partial Targeting as compared to Full Targeting may be due to the larger schedulability test overheads incurred with a more poorly targeted allocation of requests to targets such as occurs in Partial Targeting. For example, schedulability tests which fail can incur large overheads, on average.

6.5.3 Using Earliest Deadline ordering of Sporadics

The next simulations performed repeated the use of **Bottom-up Hybrid with Test 3 and Headstart** as in Table 6.1, but attempted to improve the performance of the targeting methods by presenting the pre-test with requests in order of their earliest remaining deadline. Table 6.2 shows the results. The rows for *Guarantee Ratios* and *Average Test Times* have been omitted because these are not clear indicators of performance.

Table 6.2 shows that giving preference to the requests with the shortest remaining deadline benefits the performances of all of the targeting algorithms, although of course it makes no difference to Round Robin which is in effect a random allocation. The improvement in performance can be explained by shorter deadline tasks being less likely to be guaranteed when they are allocated to remaining processors which have less slack available. Other requests, because of their longer relative deadlines, stand a greater chance of being guaranteed when they are allocated to the remaining targets. Again the relative performances of the targeting methods show similar improvements when greater overheads are incurred in order to increase the accuracy of the targeting. The slightly higher values for *Total Schedulability Test Time for Real Tasks* as compared to Table 6.1 can be explained by the greater number of sporadic requests which are being guaranteed and therefore the 'added computation time' within the task list.

Targeting Method:	**Round Robin**	**Partial Targeting**	**Full Targeting**	**Ideal Targeting**
Total Computation Time (ms):	317,450	307,172	309,250	310,513
Sporadic Utilisation achieved (%)	84.65	81.91	82.47	82.80
Total number of sporadics guaranteed:	13,377	13,111	13,182	13,148
Total Schedulability Test Time for Real Tasks (ms):	32,315	32,020	31,614	32,852
Total Schedulability Test Time for Dummy Tasks (ms):	-	40,671	41,514	41,019

Table 6.2: Bottom-up Hybrid (Test 3, Headstart) with sporadics targeted in earliest deadline order.

6.5.4 Summary

The results in Tables 6.1 and 6.2 show that Targeting performs disappointingly as compared to Round Robin. Closer investigation of the pattern of acceptance and rejection of sporadic requests within the simulations indicated that targeting sometimes results in the acceptance of a difficult-to-schedule sporadic which is rejected under Round Robin. However, shortly afterwards Round Robin may cause the acceptance of two 'easier' sporadics which Targeting rejects. Hence targeting can be counter-productive because it is not clairvoyant. Similar phenomena have been described by Liu et al [58] in the context of maximising the response time of random aperiodic requests with soft deadlines. Their conclusion is that there is no optimal method for maximising response time, unless it is possible to predict the characteristics of future aperiodic requests.

6.6 USE OF TOP-DOWN SCHEDULABILITY TESTING

6.6.1 Introduction

Targeting is currently performed purely on the basis of slack values of tasks which lie below the sporadic request within the task list. In other words the pre-test gives no indication of whether the sporadic task itself will be schedulable when interferences from higher priority tasks in the task list are taken into account. This means that a request which is allocated by the targeting pre-test is relatively likely to allow the existing tasks within the task list to remain schedulable, but less likely to be schedulable itself. If this is the case then the bottom-up order of schedulability testing may no longer be the most efficient method of finding any unschedulable tasks within a task list which will cause the request to be rejected.

The question arises as to whether a Top-down order of schedulability testing would be more efficient than Bottom-up. Top-down starts by testing the sporadic request and then traverses the task list downwards, testing all the existing tasks below the request. If this method is more efficient at rejecting unschedulable sporadic tasks, then it should cut schedulability testing overheads overall and increase the effectiveness of targeting. Therefore the next simulations which were performed used the Top-down Hybrid schedulability test algorithm, with the improvements referred to as Test 3 and Headstart as described in Chapter 5.

6.6.2 Results

Table 6.3 shows that the effect of using Top-down Hybrid is to cause a marginal drop in *Total Computation Time* across all the allocation methods. This is in accordance with the pervious results for Top-down Hybrid (see Section 4.6). Table 6.3 shows the results when sporadic requests are placed in order of arrival (randomly) and it is therefore comparable to Table 6.1. The performances of the targeting algorithms relative to Round Robin are similar to Bottom-up Hybrid.

Targeting Method:	Round Robin	Partial Targeting	Full Targeting	Ideal Targeting
Total Computation Time (ms):	316,364	304,384	307,044	308,131
Sporadic Utilisation achieved (%)	84.36	81.17	81.88	82.17
Total number of sporadics guaranteed:	13,335	13,017	13,106	13,070
Total Schedulability Test Time for Real Tasks (ms):	33,780	33,967	33,437	34,657
Total Schedulability Test Time for Dummy Tasks (ms):	-	43,410	43,960	43,529

Table 6.3: Top-down Hybrid (Test 3, Headstart) with sporadics targeted in random order.

Targeting Method:	Round Robin	Partial Targeting	Full Targeting	Ideal Targeting
Total Computation Time:	316,364	305,368	306,432	308,208
Sporadic Utilisation achieved (%)	84.36	81.43	81.72	82.19
Total number of sporadics guaranteed:	13,335	13,058	13,092	13,063
Total Schedulability Test Time for Real Tasks (ms):	33,780	33,852	33,448	34,744
Total Schedulability Test Time for Dummy Tasks (ms):	-	43,349	43,938	43,558

Table 6.4: Top-down Hybrid (Test 3, Headstart) with sporadics targeted in earliest deadline order.

Table 6.4 shows a similar set of results for sporadic requests ranked in order of earliest remaining deadline. With one exception, *Total Computation Times* for targeting are improved over Table 6.3 which confirms that this ordering of sporadic requests is again beneficial. The exception is the result for Full Targeting which is less in Table 6.4 than Table 6.3. This anomaly indicates how marginal the effect of sporadic request ordering can be, especially for Full Targeting. Comparing similar figures for Full Targeting with Bottom-up Hybrid (Table 6.1 with Table 6.2) it can be seen that the *Total Computation Time* for earliest deadline ordering of requests it only slightly greater than for random ordering. The anomalous result in Table 6.4 may also show that giving preference to shorter deadline tasks shows up more the inefficiency of the top-down schedulability test algorithm. Earlier deadlines correspond to higher positions in the task list, and therefore more top-down schedulability testing before a discovery that a lower, previously-guaranteed task is unschedulable.

6.6.3 Summary

These results tend to dispel the concern that Bottom-up testing might be disadvantageous to targeting. On the contrary, the increased values for *Total Schedulability Test Time for Real Tasks* and *Total Schedulability Test Time for Dummy Tasks* confirm that Top-down is a less efficient order for the schedulability test algorithm.

6.7. TARGETING WITHOUT DUMMY SPORADIC REQUESTS

6.7.1 Introduction

Another issue which may impinge upon the performance of Targeting as compared to Round Robin is the extra overhead imposed on targeting by the use of dummy sporadic requests. The following simulations use Bottom-up Hybrid schedulability test algorithm but without dummy sporadic requests. While this lowers the overhead on the target processors it also means that the slack values used by the pre-test are likely to be more out-of-date.

Tables 6.5 and 6.6 show results which are comparable to Tables 6.1 and 6.2, this time without dummy requests. *Total Schedulability Test Time* refers to the total time spent in scheduling real sporadic requests only, since dummies are no longer used. The Tables also include measurements of the targeting overhead (due to allocation, matrix construction, etc.) on the targeting processor. *Total Targeting Time* on the Targeting Processor is the accumulated overhead due to targeting over the 10 simulations. As before the tables contrast the performance when not ordering sporadic requests and when ordering them.

Targeting Method:	Round Robin	Partial Targeting	Full Targeting	Ideal Targeting
Total Computation Time (ms):	317,450	320,970	322,110	327,001
Sporadic Utilisation achieved (%)	84.65	85.59	85.90	87.20
Total number of sporadics guaranteed:	13,377	13,543	13,575	13,639
Total Schedulability Test Time (ms):	32,315	32,091	31,853	32,953
Total Targeting Time (ms) on the Targeting Processor:	-	2,518	5,545	88,246

Table 6.5: Bottom-up Hybrid (Test 3, Headstart) with sporadics targeted in random deadline order.

Targeting Method:	Round Robin	Partial Targeting	Full Targeting	Ideal Targeting
Total Computation Time (ms):	317,450	321,049	322,940	327,096
Sporadic Utilisation achieved (%)	84.65	85.61	86.12	87.23
Total number of sporadics guaranteed:	13,377	13,543	13,609	13,667
Total Schedulability Test Time (ms):	32,315	32,107	32,052	33,067
Total Targeting Time (ms) on the Targeting Processor:	-	2,886	6,085	88,578

Table 6.6: Bottom-up Hybrid (Test 3, Headstart) with sporadics targeted in earliest deadline order.

6.7.2 Interpretation of the Results

Obviously the removal of the dummy testing overheads on the target processors improves the performance of Targeting so that it now exceeds Round Robin. Full Targeting improves the performance by up to 2% which shows that even the more out-of-date slack values are of some benefit in mapping requests to likely targets. Ideal Targeting provides an improvement in performance of over 3%. It is not surprising that there is now a clearer difference between the performance of Ideal Targeting as compared to Full Targeting. This now reflects the benefit of using the full schedulability test instead of out-of-date slack values as a basis for targeting.

Both Tables 6.5 and 6.6 show a slight increase in *Schedulability Test Time* compared to *Schedulability Test Time for Real Tasks* in Tables 6.1 and 6.2. This is attributable to the increase in the loading of guaranteed sporadic tasks when dummy testing is removed.

6.7.3 Summary

Clearly Tables 6.5 and 6.6 indicate that the overheads incurred by dummy requests are not justified in terms of improvements in the performance of targeting. The results for Ideal Targeting provide some measure of the maximum improvement in performance which targeting can provide.

6.8 OVERHEADS ON THE TARGETING PROCESSOR

A further issue is the overheads which are actually incurred by targeting on the Targeting Processor itself. *Total Targeting Time* in Tables 6.5 and 6.6 shows that these overheads rapidly increase with the sophistication of the targeting methods used.

Table 6.7 contrasts the Total Targeting overheads from Table 6.6 with the increase in *Total Computation Time* over Round Robin. In the case of these simulations, the overheads for Partial Targeting are outweighed by the performance improvement, Full targeting almost 'breaks even' but, as expected, Ideal Targeting incurs a far greater overhead than is justified.

Targeting Method:	**Partial Targeting**	**Full Targeting**	**Ideal Targeting**
Total Gain in Computation Time over Round Robin (ms):	3,599	5,490	9,646
Total Targeting Time (ms) on the Targeting Processor:	2,886	6,085	88,578

Table 6.7: Comparing gain in computation time against targeting overheads incurred in Table 6.6

The results in Table 6.7 support the case for Shuffle Schedulability Testing which is discussed below in Section 6.13.

6.9 UPDATING SLACK FOR ACCEPTED SPORADIC TASKS ONLY

Issue 6 in the introduction concerns less frequent, but sometimes more accurate, updating of the slack values of tasks in the task list of the target processors. The issue is whether (i) slack values in the task lists should be updated every time a request is schedulability tested or (ii) whether slack values should be updated only when a schedulability test has succeeded and a sporadic task is accepted.

Simulations so far have used (i) above. The advantage of (ii) is that, in the case of a request which fails, the sporadic's computation time has not been used pessimistically in the calculation of the new slack values. The disadvantages of (ii) are twofold. One, when a request fails the old slack values are retained, and these will be even more out-of-date. Two, a slightly larger overhead is incurred due to the need to temporarily store slack values and then perform the updating only when a schedulability test succeeds. In contrast (i) above updates the slack values every time schedulability testing is performed, regardless of whether the sporadic request is accepted or rejected.

Version of Full Targeting:	**(i) Update slack every schedulability test**	**(ii) Update slack only if request accepted**
Total Computation Time (ms):	324,098	323,597
Sporadic Utilisation achieved (%)	86.43	86.29
Total number of sporadics guaranteed:	13,628	13,614
Total Schedulability Test Time (ms):	32,337	32,310

Table 6.8: Two versions of Full Targeting both using Bottom-up Hybrid (Test 3, Headstart) with sporadics targeted in earliest deadline order.

Table 6.8 shows the results of two sets of simulation for Full Targeting with (i) and (ii) above. As before, the periodic utilisation is 85% but different sets of randomly generated periodic tasks were used so that the results in column (i) are slightly different from the simulations in Table 6.6. However, the same periodic task sets were used for both (i) and (ii) in Table 6.8, so that these results are directly comparable. As can be seen from Table 6.8, the original method of updating the slack values, at every schedulability test gives a slightly better *Total Computation Time*. No doubt method (ii) suffers from the fact that slack values become even more outdated plus the effect of a slightly greater overhead.

6.10 VARYING PERIODIC UTILISATIONS UNIFORMLY

The next issue to explore is the effect of varying periodic utilisations uniformly across all targets. As mentioned in Chapter 4, sporadic utilisation can be varied either by changing the average computation times of sporadic requests or by changing the arrival rate of requests. Tables 6.9 and 6.10 below show the results of using each of these methods to vary periodic utilisations uniformly across the cluster. Table 6.9 shows periodic utilisations varying from 85% down to 50% while the average computation time for sporadic requests varies from 25ms to 95ms. The arrival rate of sporadic requests is constant at 0.015 requests per ms. The total possible utilisation (periodic plus sporadic) is 97.5% in every case. The performances of Round Robin and Full Targeting are compared by measuring the percentage of the possible sporadic utilisation which is achieved by each. Full Targeting was chosen for comparison because it was the targeting method which provided the best performance with reasonable overheads in the previous simulations.

Table 6.9 shows that Full Targeting clearly outperforms Round Robin at low average sporadic computation time, but the performances are very similar at high average sporadic computation times. This may illustrate the problems of guaranteeing a sporadic task with a large average sporadic computation time. A targeting method may make it more likely for such a computation time to be accepted. However, as remarked above, this may prove disadvantageous in the long run because using up most of the available slack on a single sporadic may forfeit the chance of accepting subsequent sporadics which may be easier to guarantee. Hence for high average sporadic computation times the relative advantages of Full Targeting are lost.

Table 6.10 shows the results of varying periodic utilisation by changing the arrival rate of sporadic requests. Here the average computation time of sporadic request is fixed at 50ms. Arrival rates vary from 0.01 to 0.03 per ms while periodic utilisations vary from 80.8% down to 47.5%. Again the total possible utilisation (periodic plus sporadic) is constant at 97.5%. As before, the table compares the performances of Round Robin and Full Targeting.

Ave Sporadic Comp Time (ms)	25	50	75	95
Resident Periodic Utilisation (%)	85.0	72.5	60.0	50.0
Sporadic Utilisation Possible (%)	12.5	25.0	37.5	47.5
% of Sporadic Utilisation Achieved by Round Robin	84.65	83.31	85.86	85.11
% of Sporadic Utilisation Achieved by Full Targeting	86.13	85.04	85.91	85.04

Table 6.9: Varying sporadic utilisation by changing the average sporadic computation time. Sporadic arrival rate constant at 0.015 per ms.

The results in Table 6.10 show that both Round Robin and Full Targeting increase their *% of Sporadic Utilisation Achieved* as the rate of sporadic arrivals increases. This is because sporadic tasks make up a relatively larger proportion of the total possible utilisation. The performances of both methods are very similar at the lowest arrival rate of 0.01 per ms. This is because there is little difference between Round Robin and Targeting under a low loading of sporadic requests (an average of 1 sporadic arrival at the cluster, at every schedulability test slot). Larger differences between Round Robin and Full Targeting are observed at intermediate arrival rates such as 0.02 per ms. However, at the highest arrival rate of 0.03 per ms the cluster is saturated with a sporadic request for every target at every schedulability test slot. This means that, again, there is less difference between the performances of Round Robin and Full Targeting.

Ave Arrival Rate (per ms)	0.01	0.015	0.02	0.025	0.03
Resident Periodic Utilisation (%)	80.8	72.5	64.2	55.8	47.5
Sporadic Utilisation Possible (%)	16.7	25.0	33.3	41.7	50.0
% of Sporadic Utilisation Achieved by Round Robin	77.06	83.31	85.65	87.73	89.49
% of Sporadic Utilisation Achieved by Full Targeting	77.02	85.04	87.62	89.99	91.32

Table 6.10: Varying sporadic utilisation by changing the sporadic arrival rate. Average sporadic computation time constant at 50 ms.

6.11 SKEWED DISTRIBUTIONS OF PERIODIC UTILISATIONS

6.11.1 Introduction

The above results show that targeting can generate only marginal improvements across a range of periodic utilisations which are distributed uniformly across the target processors. The next issue to investigate is whether targeting may produce clearer benefits when applied to a skewed (uneven) distribution of periodic utilisations across the target processors. There are obvious advantages in targeting a request at a processor with a low periodic utilisation instead of other targets with higher periodic utilisations.

Two sets of simulations were performed in order to explore skewed periodic utilisations. The first (shown in Table 6.11) has a constant, average sporadic utilisation of 45ms and a constant sporadic arrival rate of 0.02 per ms. The second (shown in Table 6.12) has a constant, average sporadic utilisation of 60ms and a constant sporadic arrival rate of 0.015 per ms. Each table shows the % of the total possible sporadic utilisation achieved across the whole cluster by the various targeting methods under consideration. For all results the total possible periodic utilisation, summed across the three target processors in the cluster, is 202.5%. The total possible sporadic utilisation added to this is 90% for all results. Hence the overall, total possible utilisation per target processor is: (202.5 + 90)/3 in other words 97.5% per processor. This is in line with total possible utilisations for all previous results in this chapter.

6.11.2 The Skewed Distributions

The various distributions of the 202.5% periodic utilisation across the three targets are as follows. **No-skew** has a uniform periodic distribution of 67.5% on each of the three targets and it acts as a 'control experiment'. **Uniform Skew** (in Table 6.11) has a uniform gradient of periodic distribution across the targets i.e. 85% : 67.5% : 50%. **Heavy skew** (Table 6.12) has a 85% : 85% : 32.5% periodic distribution. This concentrates most, but not all, of the spare capacity for sporadic tasks in the third processor. However the constraint of a maximum of one request per target per schedulability test slot of 100ms still applies. This constraint is relaxed for **Single Target** (Tables 6.11 and 6.12) which has a 90% : 90% : 22.5% periodic distribution.

In **Single Target** all of the spare capacity for sporadic tasks is concentrated in the third target. This releases the first two targets from schedulability testing altogether which allows the periodic task set aside for schedulability testing to be included as part of the general periodic utilisation. This task has a WCET of 5ms and a period of 100ms, so that the effect is to add 5% to the periodic utilisation for these processors, which brings their total utilisation (periodic only) up to 90%. In contrast, the third target processor may now have to schedulability test up to three requests per 100ms slot. Hence its top-priority periodic task, which represents schedulability testing, must have a WCET of 15ms and a period of 100ms. **Single Target** obviously has is own unique 'targeting method' which is simply to allocate all sporadic requests to the third target processor.

An additional targeting method is **Skewed Round Robin** which is the skewed analog of Round Robin. Here Round Robin is modified so that the number of allocations made are in inverse proportion to the ratio of periodic utilisations on the targets. In other words Round Robin is simply upgraded to adapt rotational allocation so that targets with lower utilisations are given correspondingly more frequent 'turns'. The ratios of the frequency of allocation 'turns' for the set of target processors is equal to the ratios of their spare capacities. Here 'spare capacity' is defined as: (97.5 - Periodic Utilisation)% for each target.

	Round Robin	Partial targeting	Full Targeting	Ideal Targeting	Skewed Round Robin	Single Target
No Skew: % Sporadic Utilisation achieved	85.99	86.89	87.92	88.33	–	–
Uniform Skew: % Sporadic Utilisation achieved	73.26	82.42	83.66	81.32	83.56	–
Single Target: % Sporadic Utilisation achieved	–	–	–	–	–	76.76

Table 6.11: Skewing the distribution of periodic utilisation.
Constant ave sporadic computation time of 45ms and constant sporadic arrival rate of 0.02 per ms.

	Round Robin	Partial targeting	Full Targeting	Ideal Targeting	Skewed Round Robin	Single Target
No Skew: % Sporadic Utilisation achieved	87.82	86.11	87.57	88.88	–	–
Heavy Skew: % Sporadic Utilisation achieved	60.01	78.81	83.51	78.11	86.11	–
SingleTarget: % Sporadic Utilisation achieved	–	–	–	–	–	79.56

Table 6.12: Varying the distribution of periodic utilisation (skew).
Constant ave sporadic computation time of 60ms and constant sporadic arrival rate of 0.015 per ms.

6.11.3 Interpreting the Results

Table 6.11 shows the following results for different degrees of 'skewedness'. **No Skew** shows a steady increase in the sporadic utilisation achieved from Round Robin through to Ideal Targeting (Skewed Round Robin is not applicable for **No Skew**). **Uniform Skew** shows generally poorer performance, especially for Round Robin which is performing simple rotational allocation despite the inequality of periodic utilisation on each of the targets. Targeting methods improve the sporadic utilisation achieved but it is noteworthy that Skewed Round Robin performs almost as well as Full Targeting. Also notable is that Ideal Targeting performs less well than Full targeting. This may be a reoccurrence of the counter-intuitive effect where the greater ability of Ideal Targeting to facilitate the guarantee of 'difficult' sporadic tasks, actually causes later sporadics to be rejected.

In Table 6.11 **Single Target** achieves a poor sporadic utilisation. No doubt this is partly due to the high upper bound for schedulability testing (15ms) which is used in the schedulability testing of a newly arrived request. Also, the high request loading placed on the single target when, for example, three outstanding sporadic requests have to be schedulability tested must have an effect on the sporadic utilisation which is attainable.

Table 6.12 shows similar results to that of Table 6.11. Again **No Skew** shows the best sporadic utilisations achieved with an increase in sporadic utilisation achieved as targeting methods become more sophisticated. However Round Robin performs slightly better than Full Targeting which is echoed by the results for high average computation times and low sporadic arrival rates in Tables 6.9 and 6.10 above. **Heavy Skew** has most of the spare capacity on one of the target processors with relatively little on the others (85% : 85% : 32.5%). It is not surprising that Round Robin performs so badly when using simple rotational allocation among such a biased allocation of periodic utilisation. Steady improvements are made when Partial Targeting and Full Targeting are used. However Ideal Targeting results in a drop in sporadic utilisation achieved. This may be explained by the same observations as for Table 6.11. Under **Heavy Skew** distribution Skewed Round Robin actually performs best of all. Again the relatively high average computation time (60ms) and the low sporadic arrival rate (0.015 per ms) is best served by a Round Robin method. Finally, **Single Target** performs badly as in Table 6.11.

6.11.4 Summary

The results for skewed distributions of periodic utilisations can be summarised as follows. The **No Skew** distribution of periodic utilisation provides the best sporadic utilisations overall. Under a skewed distribution of periodic utilisation Skewed Round

Robin performs as well as the targeting methods, without the disadvantage of their overheads on the targeting processor. Ideal Targeting can be counter-productive when periodic distribution is skewed, and finally the method of targeting all sporadic request onto a **Single Target**, in order to enhance the periodic utilisation on the remaining targets, does not actually increase the overall utilisation across the cluster.

6.12 GENERATING SPORADIC REQUESTS INTERNALLY

6.12.1 Introduction

The final issue to be considered under the heading of Targeting is the inclusion of sporadic requests which arise internally on the target processors themselves. Hitherto the simulations have assumed that requests arrive only from sources external to the cluster and are allocated by a targeting processor. However to fulfil the requirements of the Constrained Computational Model described of Chapter 3, it is necessary to consider the incorporation of those sporadic requests which may also arise internally at each of the target processors.

A major issue which arises here is how the arrival of a stream of internal and external sporadic requests can be constrained and interleaved in such a way that schedulability testing can be bound. It is assumed that internal sporadic requests arise randomly at target processors and that they must be either guaranteed or rejected on their 'home' processor. Furthermore, it is assumed that only one request (internal or external) may arrive at a target processor in any one schedulability test slot. These assumptions enforce the constraint of a maximum of one sporadic request per schedulability test slot and therefore allow schedulability testing to be bounded.

Clearly the targeting of external sporadic requests will be undermined by the (random) arrival of internal sporadic requests at target processors. Therefore in order to retain some value in targeting, it is better to separate external and internal requests. In order to achieve this, and to enable schedulability testing to be bound, it was decided that all methods used should generate, at each 100ms schedulability test slot, either a set of internal or a set of external sporadic requests, but never a mixture of both. As usual a set may consist of up to three sporadic requests. A further issue concerns the use of slack values which are recalculated when an internal sporadic request is schedulability tested. It is argued that these values should be available to the targeting processor by exactly the same mechanism as is used to communicate slack values generated by the schedulability testing of *external* requests.

6.12.2 Adapted Targeting Algorithms

	Random Round Robin	Alternating Round Robin	Full Targeting	Adapted Full Targeting	Adapted Round Robin
% of Sporadic Utilisation achieved	**84.55**	**84.58**	**85.74**	**86.04**	**85.06**

Table 6.13: Randomly allocated internal sporadics and systematically allocated external sporadics.

Table 6.13 shows the range of methods which were developed to deal with a mixture of internally and externally arising sporadic requests. In **Random Round Robin** a random choice is made at each schedulability test slot to decide whether sporadics requests are (i) internal or (ii) external, for that slot. However, this can lead to an effect where sequences of schedulability test slots are composed of 'runs' of internal or external sporadic requests. In contrast, the method used by **Alternating Round Robin** (and also by **Full Targeting** and **Adapted Round Robin**) enforces separate upper bounds on the arrival rates of both internal and external sporadic requests. This may be closer to the requirements of a realistic application. In **Alternating Round Robin**, alternating schedulability test slots are chosen for (i) internal or (ii) external sporadic requests. This has the effect of fixing the maximum arrival rate for either internals or externals to be 3 sporadics per 200ms. Internal sporadics are allocated randomly at each of the target processors with the constraint that each target may take a maximum of one internal sporadic per 200ms slot.

As previously, external sporadic requests may be allocated Round Robin, or Targeted according to some knowledge of the slack available on each of the target processors. In **Random Round Robin** and **Alternating Round Robin**, external sporadic requests are allocated in round robin rotation without taking into account the allocation of internal sporadic requests at the last 100ms slot. In contrast, **Full targeting, Adapted Full Targeting**, and **Adapted Round Robin** make use of some of the knowledge gained at the last 100ms allocation of internal sporadics. **Full Targeting** makes use of the most recently calculated slack values (which includes any values calculated when schedulability testing internal sporadics at a previous 100ms slot). However **Full Targeting** makes no use of the knowledge of *which* target processors were allocated internal sporadics 100ms previously.

In contrast to Full Targeting, **Adapted Full Targeting** avoids the target positions of the last internal sporadic request allocation, when, as described previously, it is necessary to make FCFS allocation of external sporadics. Similarly, **Adapted Round Robin** restarts its rotational allocation of external sporadics at a target where it is known that *no internal sporadic request* was allocated 100ms previously. (This is slightly less than optimal, since it would be even better to ensure that Round Robin restarts at the first previously unallocated target if any 'runs' of two unallocated targets exist from the last internal request allocation cycle.) In the event of all targets being allocated internal sporadics at the last cycle, **Adapted Round Robin** continues where it left off at the last external allocation, 200ms previously.

6.12.3 Results

Table 6.13 shows the percentage sporadic utilisation achieved by simulations using this range of allocation algorithms. For all simulations there is a flat distribution of 67.5 % periodic utilisations across all three processors, an average sporadic computation time of 45ms, and an average sporadic arrival rate of 0.02 sporadics per ms. Hence the results in Table 6.13 are directly comparable to the results in Table 6.11, where all sporadic requests are external.

Clearly the sporadic utilisations achieved in Table 6.13 are down on the **No-Skew** results in Table 6.11. Obviously the random allocation of internal sporadic requests is working against a balanced loading of sporadic computation on each of the target processors. Table 6.13 shows that small improvements in performance can be made by the use of targeting methods which take into account information gained at the last cycle of internal sporadic allocation. **Full Targeting** takes advantage of slack values which are updated by any internal sporadic in the last cycle. **Adapted Full Targeting** enhances performance slightly more by taking into account the allocations of internal sporadics to targets at the last cycle. However, a similar adaptation to Round Robin, (**Adapted Round Robin**) can bring its performance closer to that of the targeting methods.

6.13 SHUFFLE SCHEDULABILITY TESTING

6.13.1 Introduction

Shuffle Schedulability Testing is a different configuration from a targeting cluster. The targeting processor is dispensed with, and the three target processors are configured in a 'loop'. Sporadic requests which arise (either internally or externally) at each processor are first schedulability tested at the 'home' processor and, if they fail their test, are 'shuffled' to the next processor along the loop. At the second processor the test is repeated, and, if it fails, the request is shuffled to the third processor where a final test is performed. In this way each processor in the cluster has an equal status, and each processor behaves symmetrically. As with targeting, it is assumed that communications within the cluster are sufficiently fast that communication delays are negligible, in comparison to the intervals between acceptance testing.

In Shuffle Schedulability Testing, the issue of separating internal and external sporadic requests no longer causes a problem. Neither type of request need be distinguished from each other and each type can be considered as arising at random on any of the three processors in the cluster. (Whether there is actually a targeting processor which is allocating the external requests is not relevant.)

It is assumed that the system constrains the arrival rate of new requests, whether internal or external, at each processor, to be a maximum of one 'home' request per schedulability test slot. However, in order to speed up the 'shuffling', the constraint of one schedulability test per processor per 100ms is relaxed, and processors may now perform up to three tests per slot. This allows a processor to schedulability test up to two sporadic requests which have been 'passed on' from other processors which rejected them. The upper bound for schedulability testing (which is itself used when performing the schedulability test) must therefore be trebled to 15ms.

As previously, the simulations make the idealised assumption of zero communication overheads incurred when a request is passed on from one processor to the next. This assumption must not be confused with the effect of phase differences between the start of schedulability test slots at each of the processors. The effect of such phase differences is now discussed.

6.13.2 The Effect of the Phase Difference

A key performance parameter in Shuffle Schedulability Testing may be the size of any delays before follow-up schedulability tests at processors along the loop can be performed. Loosely coupled processors may have large phase differences between their schedulability test slots and the delays incurred by these differences may degrade the performance of Shuffle Schedulability Testing.

The simulations of Table 6.14 investigate the effect of such phase differences between schedulability test slots. For each simulation in the table, the delays between processor schedulability slots are set to a different, fixed value. **Fully Unsynchronised** carries a set delay of 100ms between processors, Semi-synchronised has a delay of 50ms and Fully Synchronised has zero delay. Regarding the order of schedulability testing at the start of a 100ms slot, it was decided that sporadic requests which have newly arrived at a processor should be given preference over requests shuffled from other processors.

	Fully Unsynchronised	Semi-Synchronised	Fully Synchronised
% of Sporadic Utilisation achieved	**87.34**	**87.73**	**88.36**

Table 6.14: Shuffle Schedulability Testing with varying degrees of synchronisation between the processors.

6.13.3 Interpretation of the Results

The simulations in Table 6.14 carry a flat distribution of 67.5% periodic utilisations across all three processors, an average sporadic computation time of 45ms, and an average sporadic arrival rate of 0.02 sporadics per ms. Therefore these results are directly comparable to the **No Skew** results in Table 6.11. **Fully Synchronised** in Table 6.14 shows around 2% higher Sporadic Utilisation achieved than **Round Robin** in Table 6.11. This is the appropriate comparison to make because Shuffling, like Round Robin, incurs no overheads for targeting. In fact, Table 6.11 shows that Full Targeting, by incurring considerable targeting overheads, achieves only similar sporadic utilisations to those of Shuffle Schedulability Testing in Table 6.14.

Results not shown in the Tables indicate that Shuffling makes a small but consistent improvement over Full Targeting, in the total number of sporadics tasks guaranteed. For example, **Fully Synchronised** in Table 6.14 manages to guarantee a total of 18,501 sporadic tasks compared to a total of 18,310 sporadic tasks for **Full Targeting** in Table 6.11. This indicates a slight tendency for Shuffling to favour sporadics with shorter computation times. This could be because a second or third schedulability test within a schedulability test slot at a processor may stand a slightly greater chance of succeeding for requests with smaller computation requirements.

Looking at the delays introduced into Shuffle Schedulability Testing in Table 6.14, it is not surprising that greater performance can be achieved when follow-up schedulability testing can be performed immediately. However, performance deteriorates only slightly when the maximum possible delay of 100ms is introduced at each processor. This can be explained by the fact that delaying can, in some circumstances, reduce schedulability test overheads. The explanation is as follows.

Shuffling with delays is programmed to reduce the current deadline of any request by the amount of any delay which has elapsed in receiving that request from another processor. Furthermore shuffling with delays is judged to be unlikely to result in an eventual guarantee of a request once the deadline of the request has been reduced below a certain threshold. (This is implemented by no longer passing on requests once their current deadlines have been reduced to 100ms or less.) In contrast, **Fully Synchronised Shuffling**. experiences no delays between processors and therefore current deadlines are never reduced. In this case, requests with short deadlines (e.g. less than 200ms) may well incur an expensive schedulability test at each processor before being rejected anyway.

It is interesting to note that the total schedulability test overheads for Shuffle Schedulability Testing are considerably greater than those for Full Targeting. Typical figures (not shown in Tables 6.14 or 6.11) for *Total Sched Test Time* are 54,164ms for Shuffle Schedulability Testing as compared to 39,670ms for Full Targeting. Obviously this is due to some requests being passed on to the second or third processor.

Summary: These results show that Shuffle Schedulability Testing can provide a comparable performance to targeting methods without incurring their overheads. Therefore the final issue is to investigate the performance profile of Shuffle Schedulability Testing under varying sporadic and periodic utilisations.

6.13.4 Varying Periodic Utilisations

Of the different forms of Shuffle Schedulability Testing which were simulated, **Fully Synchronised** performed best, and therefore it was adopted for the simulations in Table 6.15, where periodic utilisation was varied uniformly across all processors. Variation of sporadic utilisation by changing the sporadic arrival rate was chosen because this produced the greatest variation in the performances obtained for original Round Robin and Full Targeting (see Table 6.10). These original results for Round Robin and Full Targeting are included in Table 6.15 for the convenience of comparing them to results obtained for **Fully Synchronised Shuffle Schedulability Testing**. The *0.01 sporadics per ms* column is missing because it was not possible to generate the necessary Periodic Utilisation (80.8%) for Shuffle Schedulability Testing when the upper bound for schedulability testing had been raised to 15ms.

The results in Table 6.15 show that Fully Synchronised Shuffle Schedulability Testing performs 2% to 3% better than original Round Robin and performs marginally better than Full Targeting. Note that the performance improvement due to Fully Synchronised Shuffle Schedulability Testing increases slightly with increasing sporadic arrival rate, and decreasing periodic utilisation. Again, more detailed results (not shown) indicate that Fully Synchronised Shuffle Schedulability Testing slightly favours sporadics with smaller computation times.

Ave Arrival Rate (per ms)	0.015	0.02	0.025	0.03
Resident Periodic Utilisation (%)	72.5	64.2	55.8	47.5
Sporadic Utilisation Possible (%)	25.0	33.3	41.7	50.0
% of Sporadic Utilisation Achieved by Round Robin	83.31	85.65	87.73	89.49
% of Sporadic Utilisation Achieved by Full Targeting	85.04	87.62	89.99	91.32
% of Sporadic Utilisation Achieved by Fully Synchro Shuffle Sched Testing	85.16	87.76	90.36	92.66

Table 6.15: Varying sporadic utilisation by changing the sporadic arrival rate.
Average sporadic computation time constant at 50 ms.

6.13.5 Summary

Shuffle Schedulability Testing is more efficient than targeting methods because it achieves similar performance while incurring no overheads on a targeting processor. It does this despite the greater burden of schedulability testing, and the adverse effect of the higher upper bound on schedulability testing.

6.14 SUMMARY OF WORK DONE

This chapter has reported on the results of extensive investigations into the operational use of schedulability testing within a processor cluster. In general, sporadic requests can arise internally within the cluster or externally from some other part of the system. The cluster may be configured for 'targeting' in which a fourth processor directs requests to three target processors. Alternatively, the three targets may be configured as a 'loop' with each processor receiving its sporadic requests separately.

In targeting, each target processor attempts to guarantee a sporadic request which it receives from the targeting processor. If the attempt fails then the request is rejected. In 'shuffle schedulability testing' the originating processor first attempts to guarantee, but a failed request is passed round the loop for further schedulability testing.

A variety of targeting algorithms were developed which would allow the targeting processor to allocate each sporadic request to the target it judges most likely to guarantee it. These ranged from simple Round Robin allocation of requests, to allocation based on a slack-based pre-test. An examination was made of the trade-off between the value to targeting of more up-to-date slack values on the target processors, and the extra overhead incurred on the targets in order to achieve this.

The behaviour of Targeting was investigated when the distribution of periodic utilisation within the cluster was varied. Firstly, a range of uniform periodic distributions were investigated. Secondly, skewed distributions of periodic utilisations across each of the target processors were simulated. The work on Targeting concluded by investigating the effect of the generation of internal sporadic requests at the target processors within the cluster. It was found that the targeting algorithms could be adapted to allow for the recent allocation of such internal sporadic requests.

Finally, simulations of 'Shuffle Schedulability Testing' were performed using similar test data as for targeting. It was found that the performance of Shuffle Schedulability Testing deteriorated only slightly when failed requests were delayed in their shuffle from one processor to another. The performance of Shuffle Schedulability Testing was also investigated for a range of uniform, periodic utilisations.

6.15 CONCLUSIONS

In the following conclusions, 'performance' is measured by the sporadic utilisation achieved.

6.15.1 General Conclusions for Targeting :

1. Targeting performs marginally better than Round Robin Allocation for a uniform distribution of periodic utilisation upon the processors within the cluster.

2. The performance of targeting can be improved by constructing a more optimal mapping of sporadic requests to target processors. However this can incur large overheads on the targeting processor, which are not justified in terms of the gains in performance across the target processors.

3. The use of extra schedulability testing on the target processors in order to permit a more frequent updating of slack, is not cost-effective and can decrease the performance of the targeting methods investigated to below the level achieved by Round Robin.

4. Allowing targeting algorithms to give preference to sporadic requests with the earliest relative deadlines can enhance the performance of targeting.

Conclusions for Targeting in a Cluster with various Periodic Utilisations:

5. A uniform distribution of periodic utilisation on the target processors achieves the highest overall performance.

6. If there is a skewed distribution of periodic utilisations over the target processors, then targeting provides a better performance than simple Round Robin. However, adapting Round Robin to provide a skewed allocation of *sporadic requests* can achieve comparable performance to that of targeting.

7. Distributing periodic utilisation such that a single target carries all of the spare capacity of the system leads to poorer overall performance across the cluster.

Conclusion for a mixture of Internally and Externally Generated Sporadic Requests:

8. The introduction of random, internally generated sporadic requests into a cluster generally decreases performance. However, some performance can be restored to the targeting methods, and to Round Robin, by adapting them to take into account the recent allocation of internal requests.

6.15.2 Conclusions for Shuffle Schedulability Testing:

9. Shuffle Schedulability Testing incurs no overheads on a targeting processor and can provide gains in sporadic utilisation which are greater than those provided by targeting methods.

10. Shuffle Schedulability Testing is not adversely affected by the random generation of internal sporadic requests and allows both internal and external requests to be integrated for the purposes of schedulability testing.

11. The performance of Shuffle Schedulability Testing deteriorates only marginally when schedulability testing on the processors within the cluster is not synchronised.

12. Shuffle Schedulability Testing cannot be applied at high periodic utilisations, because of the relatively high upper bounds which it requires for schedulability testing.

6.15.3 Overall Conclusion

Targeting provides only marginal benefits over Round Robin methods of allocating sporadic requests within a cluster of processors.

Shuffle Schedulability Testing is a preferable configuration to Targeting because:

- it incurs no Targeting overheads and dispenses with the need for a Targeting Processor.
- it provides greater, or equal performance to that of Targeting.
- it integrates the schedulability testing of sporadics requests which arise both internally within the cluster, and externally from the surrounding system.
- its performance is not degraded by the random occurrence of internal sporadic requests upon its processors.

CHAPTER 7

ADMISSION POLICIES

7.1 INTRODUCTION

7.1.1 Objective

Admission policies arbitrate between optional computations which have passed their schedulability tests, and are competing for admission to the schedule on a processor. In previous chapters FCFS Admission Policy has been assumed, for the admission of requests for the execution of sporadic tasks, which have passed their schedulability tests. However the constrained computational model presented in Section 3.4, requires that *Best Effort* Admission Policy be used in order to support the semantics required by different utility levels of optional computation.

The purpose of this chapter is to compare the performances of Best Effort and FCFS Admission Policies, under a wide range of simulation parameters, such as Periodic Utilisation and Sporadic Arrival Rate. This should establish ranges of values of these parameters within which the constrained model, using Best Effort Admission Policy, provides a higher total utility for the system. Under other parameter values, FCFS may provide higher performance.

```
schedulability test the request with the full task list
if the schedulability test succeeds then
        accept the request
else
        remove all abortable, lower utility tasks from the task list
        schedulability test the request within the reduced task list
        if the schedulability test succeeds then accept the request
                for each lower utility task which has been removed, taken in order of (i) the
                highest utility category remaining (ii) lowest residual computation time
                within that category :
                        schedulability test the removed task
                        if the task passes the test then re-instate it in the task list
                end for
        else
                reject the request
                reinstate all removed tasks
        end if
end if
```

Figure 7.1: Pseudo-code definition of Best Effort Admission Policy.

The algorithm for Best Effort Admission Policy which is used in this work, is specified by the pseudo-code in Figure 7.1. The algorithm was adapted from Best Effort as presented by Locke [35] and modified by Davis et al. [12]. Locke's original Best Effort algorithm admits tasks according to their value densities (value or utility upon completion, divided by computational requirement) removing tasks from the run queue if the *probability of overload* is greater than some threshold. In Locke's scheme, tasks are scheduled according to earliest deadline first policy.

In [12] Davis presents a Best Effort Admission Policy which also admits tasks according to value density, but ensures that all admitted or re-admitted tasks are *guaranteed* to meet their deadlines. Davis' algorithm is similar to the one used in this work (given in Figure 7.1). The difference between them is that Davis' algorithm uses many utility levels, while the algorithm in Figure 7.1 restricts utility to three levels as per the computational model of Chapter 3. Both the algorithm above, and Davis' version, apply Best Effort Admission to task lists which are ordered, and scheduled, according to fixed priority.

7.1.2 The Simulation Studies

The main objective of the simulations performed in this chapter, was to compare the performance of Best Effort Admission Policy (BE) with that of FCFS. In the simulations of Sections 7.2 through to 7.4, the processor utilisation of the resident periodic tasks was fixed at 25%, and the possible utilisation due to optional computations was increased in order to build up a profile of the total utilities gained by BE and FCFS. Requests for optional computations were set to arrive with a minimum interarrival time (defined by the 'sporadic arrival rate' in the tables below).

The admission policy itself, whether BE or FCFS, was modelled as the highest priority periodic task in the task list. The period of this task was assumed to be the minimum interarrival time of sporadic requests. The simulations had to be run repeatedly in order to find the required upper bound for the WCET of the task modelling the admission policy. The tables below show the actual admission policy overheads incurred in the simulations, and also the upper bounds for admission policy WCET. The upper bounds were first measured, and then used in the simulations for the schedulability testing of newly arrived requests.

It was decided to simulate 10 resident periodic tasks upon the processor because this number seems to reflect the needs of a realistic application, and to be large enough to show the effect of a sizeable number of mandatory tasks. The schedulability test algorithm which was used was *Bottom-up Hybrid with Test 3 and Headstart,* which has been shown in Chapter 5 to be the algorithm with the best overall performance for task lists which consist of 15 or less resident periodic tasks.

As in previous chapters, all simulation results show totals or averages over 10 simulation runs, with randomly generated tasks sets. Background tasks were not included in the simulations because, as explained in Section 3.4, they have no effect on the optional computations in the constrained computational model.

7.2 COMPARING BEST EFFORT AND FCFS ADMISSIONS POLICIES

7.2.1 Simulating 2 Levels of Utility Only

In order to prototype the simulations, and to establish some trends in the simulation results, the first simulations performed used only two utility levels. These corresponded to optional computations of High and Low Utility.

Tables 7.1 and 7.2 below show the results of the first set of simulations. For both tables, Periodic Utilisation is fixed at 25% and the **Possible Sporadic Utilisation** is increased by raising the **Sporadic Arrival Rate**. The Possible Sporadic Utilisation was increased up to 600% in order to allow BE a greater choice of optional computations, and to examine the trends in the effect of the overheads for admission policy. Table 7.1 compares the total utility gained by BE and FCFS when the average sporadic computation time was 75ms. Table 7.2 shows similar results for an average sporadic computation time of 37.5ms.

Because only two utility levels were used, only one ratio of relative utilities (R) was required. For Tables 7.1 and 7.2, R (the ratio of the utility of High Utility to Low Utility tasks) is 2.

The **Maximum Total Sporadic Utility Obtainable** does not include overheads for admission policy, but assumes that the 75% remaining utilisation is made up of sporadic utilisation with sporadic tasks of the highest possible utilities. An example of how **Maximum Total Sporadic Utility Obtainable** was calculated is given in the following paragraph.

In Table 7.1, the remaining utilisation after subtracting 25% periodic utilisation is 75%. This corresponds to 75,000ms of simulation time. Ideally this should allow 75,000 / 75 = 1000 sporadic tasks to be scheduled. Take as an example the column of Table 7.1 which provides 150% Possible Sporadic Utilisation. This is equivalent to 2000 sporadic tasks of average sporadic computation time 75ms. It is assumed that on average half the sporadic tasks are of higher utility, so that the theoretical maximum Sporadic Utility Obtainable here, is when the 75% remaining utilisation is made up of 1000 optional computations at the higher utility of 2. Over 10 simulations, this gives a **Maximum Total Sporadic Utility Obtainable** of 10 * 1000 * 2 = 20,000. The point to note here is that the 'maximum utility obtainable' is idealised and could not be scheduled in practice. Furthermore it does not take into account the overheads for admission policy.

7.2.2 Interpreting the Results

It can be seen from Tables 7.1 and 7.2 that the results for both BE and FCFS show that, in general, the **Total Sporadic Utility Obtained** increases with the **Possible Sporadic Utilisation**. The **Maximum Total Sporadic Utility Obtainable** also increases, and then levels out. However admission policy overheads and upper bounds continue to steadily increase with **Possible Sporadic Utilisation** and **Sporadic Arrival Rate**. This is why the **% of Maximum Utility Obtained** decreases despite the increase in **Total Sporadic Utility Obtained**.

Possible Sporadic Utilisation (%)		**75**	**150**	**300**	**600**
Sporadic Arrival Rate (per ms)		**0.01**	**0.02**	**0.04**	**0.08**
Total Sporadic Utility Obtained	BEST EFFORT	14,295	16,056	17,264	204
	FCFS	14,237	14,825	15,392	15,290
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	15,000	20,000	20,000	20,000
% of Maximum Utility Obtained	BEST EFFORT	95.30	80.28	86.32	1.02
	FCFS	94.91	74.13	76.96	76.45
Admission Policy Overheads (% of Total Utilisation)	BEST EFFORT	0.96	3.29	6.64	8.29
	FCFS	0.93	2.27	4.46	8.59
Upper Bound for Admission Policy (% of Total Utilisation)	BEST EFFORT	6.0	24.0	40.0	76.0
	FCFS	3.0	10.0	24.0	48.0

Table 7.1: Constant Periodic Utilisation of 25% with an Average Sporadic Computation Time of 75ms.

The tables show that generally BE accrues more sporadic utility than FCFS. However, at the lowest **Possible Sporadic Utilisation**, the gains in sporadic utility between BE and FCFS are comparable. This is because most optional computations are accepted at this relatively low processor loading. Therefore there are few cases where BE can gain over FCFS by scheduling a high utility computation and aborting low utility computations. The tables show that, as **Possible Sporadic Utilisation** increases, the gain in performance of BE over FCFS increases. However, at the highest **Possible Sporadic Utilisation**, the performance of BE falls off rapidly. This is due to the large **Sporadic Arrival Rate** which requires such a large upper bound for BE that few optional computations can actually be accepted.

Possible Sporadic Utilisation (%)		**75**	**150**	**300**
Sporadic Arrival Rate (per ms)		**0.02**	**0.04**	**0.08**
Total Sporadic Utility Obtained	BEST EFFORT	28,583	31,319	332
	FCFS	28,401	30,646	32,936
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	30,000	40,000	40,000
% of Maximum Utility Obtained	BEST EFFORT	95.27	78.30	0.83
	FCFS	94.67	76.62	82.34
Admission Policy Overheads (% of Total Utilisation)	BEST EFFORT	2.32	7.77	8.55
	FCFS	2.22	6.15	11.50
Upper Bound for Admission Policy (% of Total Utilisation)	BEST EFFORT	16.0	60.0	80.0
	FCFS	8.0	32.0	60.0

Table 7.2: Constant Periodic Utilisation of 25% with an Average Sporadic Computation Time 37.5ms.

7.2.3 Overheads for Admission Policy

It is clear from the tables that admission policy overheads for BE increase more rapidly than for FCFS. This is not surprising because BE is the more complex algorithm. In particular the overheads due to BE will increase at higher sporadic arrival rates because there are more chances of an high utility optional computation being schedulable, only when lower utility computations are aborted.

The upper bound for admission policy is expressed as a *percentage of Total Utilisation* by taking into account (i) the maximum time found by repeated simulation runs and (ii) the sporadic arrival rate. The percentage of Total Utilisation increases more steeply, as sporadic arrival rate increases. This to be expected, because the upper bound is the worst case time for admission policy over 10 simulation runs. Therefore the rate of increase of the upper bound is determined by the complexity of the policy.

In the case of BE, the complexity is O(N), where N is the number of optional tasks which have been previously accepted and are still current in the task list. In turn, the schedulability test algorithm which BE calls for each pending task, has a complexity based on $(M + N)$, where M is the number of mandatory, periodic tasks in the task list.

7.2.4 Average Sporadic Computation Times

Comparison of Tables 7.1 and 7.2 shows that a lower average computation time for sporadic tasks (e.g. 37.5ms) means that a higher rate of sporadic arrivals (and therefore of schedulability testing) is required in order to achieve the same **Possible Sporadic Utilisations**. Therefore, for 37.5ms average computation time, schedulability test overheads and bounds become prohibitive at lower Possible Sporadic Utilisations (i.e. 300% Possible Sporadic Utilisation as compared to 600% Possible Sporadic Utilisation for 75ms average computation time).

This effect of larger admission policy overheads can also explain why the differences between the % of Maximum Utility Obtained for BE and FCFS are smaller at lower **Possible Sporadic Utilisations** for 37.5ms average computation time than for 75ms. (For example compare the differences in the performances of BE and FCFS at 150% **Possible Sporadic Utilisations** in Tables 7.1 and 7.2.) In general one can conclude that, because large **Possible Sporadic Utilisations** are necessary for large utility gains, BE performs relatively badly for smaller average computation times.

7.3 SIMULATING HIGH AND MEDIUM UTILITY OPTIONAL COMPUTATIONS

The previous two-level simulations used optional computations of High and Low Utility, so that a High Utility request might be scheduled by aborting a Low Utility computation which had previously been accepted. The two-level simulation was next modified to compare BE and FCFS admission policies when High and Medium Utility optional computations were used. The difference here is that a High Utility request can only be scheduled by aborting a previously accepted Medium Utility computation *before it has started*.

Table 7.3 shows results for BE and FCFS which are directly comparable to Table 7.1. Periodic Utilisation is 25%, and Average Sporadic Computation time is 75ms. Note that the results for FCFS are identical to the 150% **Possible Sporadic Utilisation** column of Table 7.1.

Comparing the results for BE in Table 7.3 with the relevant result in Table 7.1, it can be see that BE does not gain such a large additional sporadic utility over FCFS in Table 7.3 as in Table 7.1. This makes sense because the simulations recreate identical conditions except that the lower utility computations in Table 7.3 are actually Medium Utility optional computations in the constrained model and are only abortable before they start their computations. Therefore there are, on average, fewer abortable tasks within the task list and correspondingly less chance that BE can schedule a High Utility request by aborting Lower Utility task(s) within the list.

Possible Sporadic Utilisation (%)		**150**
Sporadic Arrival Rate (per ms)		**0.02**
Total Sporadic Utility Obtained	BEST EFFORT	15,642
	FCFS	14,825
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	20,000
% of Maximum Utility Obtained	BEST EFFORT	78.21
	FCFS	74.13
Admission Policy Overheads (% of 100% Utilisation)	BEST EFFORT	2.93
	FCFS	2.27
Upper Bound for Admission Policy (% of 100% Utilisation)	BEST EFFORT	20.0
	FCFS	10.0

Table 7.3: High and Medium Utility Computations.

Constant Periodic Utilisation of 25% with an Average Sporadic Computation Time of 75ms.

Data from the simulations which are not shown indicate that BE in Table 7.3 guarantees, by the abortion of lower utility tasks, around half the number of high utility tasks as are guaranteed by abortion in Table 7.1. Another way of looking at this is to consider Medium Utility computations as *Low Utility* between the time of guarantee and start of computation, and *High Utility* for the rest of their execution. Considered in this way, the simulation of Table 7.3 can be viewed as having considerably fewer Low Utility tasks in the task list as compared to the simulation of Table 7.1.

7.4 SIMULATING 3 LEVELS OF UTILITY

7.4.1 Introduction

The next set of simulations include mandatory tasks, as before, but now *three* types of optional computations with High, Medium and Low utilities, as set out in the constrained computational model.

Table 7.4 shows the results of running the new simulations with exactly the same sets of sporadic requests and resident periodic tasks as for the preceding two-level simulations. The only difference was that sporadic requests were randomly allocated three, instead of two, utility levels. In order to define the three levels, two ratios, R_1 and R_2, were introduced into the constrained computational model (see Section 3.5.3). R_1 was defined as the ratio of the utility of a High Utility task completion to that of a Medium Utility task completion. Similarly R_2 was defined as the ratio of the utility of a Medium Utility task completion to that of a Low Utility task completion. For the simulations in Table 7.4, R_1 and R_2 are each set to 2. Hence the utilities gained by optional computations of each task type are 4, 2 and 1.

Possible Sporadic Utilisation (%)		**75**	**150**	**300**	**600**
Sporadic Arrival Rate (per ms)		**0.01**	**0.02**	**0.04**	**0.08**
Total Sporadic Utility Obtained	BEST EFFORT	22,535	26,761	29,988	321
	FCFS	22,363	23,117	23,996	23,806
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	23,333	33,333	40,000	40,000
% of Maximum Utility Obtained	BEST EFFORT	96.58	80.28	74.97	0.80
	FCFS	95.84	69.35	59.99	59.52
Admission Policy Overheads (% of 100% Utilisation)	BEST EFFORT	0.98	3.74	7.45	9.18
	FCFS	0.93	2.27	4.46	8.58
Upper Bound for Admission Policy (% of 100% Utilisation)	BEST EFFORT	9.0	24.0	48.0	76.0
	FCFS	3.0	10.0	24.0	48.0

Table 7.4: Three levels of utility with Constant Periodic Utilisation of 25% and an Average Sporadic Computation Time of 75ms.

As with the two-level simulations, the **Maximum Sporadic Utility Obtainable** does not include the overheads for admission policy, but assumes that 75% remaining utilisation is made up of sporadic utilisation with sporadic tasks of the highest possible utilities.

7.4.2 Interpreting the Results

Obviously both the **Sporadic Utilities Obtained** and the **Maximum Sporadic Utility Obtainable** are greater than those of Table 7.1 because of the wider range of utilities between the three levels. However, the same trends are observed in Table 7.4. These include the breakdown in BE around 600% **Possible Sporadic Utilisation**, and the increasing difference in utility obtained between BE and FCFS as **Possible Sporadic Utilisation** increases up to 600%.

As before, at 75% Possible Sporadic Utilisation, the difference in utility obtained between BE and FCFS, is small due to the limited choice of higher utility sporadic tasks. Simulation data not shown indicates that, at 75% **Possible Sporadic Utilisation**, only about 0.03% of High Utility tasks and 0.01% of Medium Utility tasks are schedulable by the abortion of lower utility tasks from the task list. The total possible processor utilisation in this case is 100%. It can therefore be concluded that for total possible processor utilisations of less than 100%, BE degenerates into FCFS.

Table 7.4 shows that, at 150% **Possible Sporadic Utilisation**, there is approximately 11% difference between **Sporadic Utility Obtained** by FCFS and BE. This is a promising result because total possible processor utilisation is 175% which is a moderate overload and is an area of genuine interest for applications. For processor overloads much in excess of this, the question arises as to whether the system was designed with too small a processing capacity.

Data for the 300% **Possible Sporadic Utilisation** in Table 7.4 illustrate the point that serious under-capacity undermines the application. The guarantee ratio (not shown) which was measured for FCFS was only around 0.25. This indicates that, due to the overload of sporadic requests, there is only about 25% chance of a request being accepted. (Guarantee ratios for BE cannot be compared in this way because they are 'exaggerated', due to some guarantees being later rescinded.)

Note that the FCFS overheads for admission policy are exactly same as with 2 utility levels except for a slightly greater overhead at 600% Possible Sporadic Utilisation. This result is correct and is simply due to the rounding up of an average schedulability test time which was 1 tick greater because of random variations.

7.5 VARYING THE RESIDENT PERIODIC UTILISATION

7.5.1 Introduction

The next characteristic which the simulations evaluated was the comparative performances of BE and FCFS when the resident periodic utilisation on the processor was varied. The extent of the resident periodic utilisation on the processor makes fundamental changes to the sporadic utilisation which is obtainable. Three sets of simulations were performed with 10%, 25% and 50% resident periodic utilisation respectively. **Possible Sporadic Utilisations** were provided in order to make up the **Possible Total Utilisations** for each simulation to be 100% 150% and 200%. This range of total utilisations was chosen because it represents a reasonable overload on the processor and is therefore of genuine interest for applications.

Tables 7.5-7.7 show the results with columns headed by the **Possible Total Utilisations**. The average sporadic computation time was set to 50ms for each of the Tables. (The average sporadic computation time has been reduced from the 75ms used in previous simulations in order to counter the criticism that a higher average sporadic computation might favour the performance of BE in comparison to FCFS.)

Possible Total Utilisation (%)		**100**	**150**	**200**
Possible Sporadic Utilisation (%)		**90**	**140**	**190**
Sporadic Arrival Rate (per ms)		**0.018**	**0.028**	**0.038**
Total Sporadic Utility Obtained	BEST EFFORT	39,757	50,122	57,860
	FCFS	39,179	45,524	50,492
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	42,000	76,496	116,432
% of Maximum Utility Obtained	BEST EFFORT	94.66	65.52	49.69
	FCFS	93.28	59.51	43.37
Admission Policy Overheads (% of 100% Utilisation)	BEST EFFORT	2.11	5.17	7.15
	FCFS	1.92	3.88	5.58
Upper Bound for Admission Policy (% of 100% Utilisation)	BEST EFFORT	18.0	42.0	57.0
	FCFS	7.2	14.0	22.8

Table 7.5: Constant Periodic Utilisation of 10% with an Average Sporadic Computation Time 50 ms.

Again R_1 and R_2 are each 2 and therefore the utilities gained upon the completion of each task type are 4, 2 and 1 respectively. It is notable that the **Maximum Total**

Sporadic Utility Obtainable varies between Tables 7.5-7-7. This is because the resident Periodic Utilisations are different, and the **Maximum Total Sporadic Utility Obtainable** is calculated from the remaining processor utilisation after Periodic Utilisation has been subtracted.

Possible Total Utilisation (%)		100	150	200
Possible Sporadic Utilisation (%)		**75**	**125**	**175**
Sporadic Arrival Rate (per ms)		**0.015**	**0.025**	**0.035**
Total Sporadic Utility Obtained	BEST EFFORT	33,262	42,969	50,333
	FCFS	32,680	38,537	44,045
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	35,000	58,333	93,333
% of Maximum Utility Obtained	BEST EFFORT	95.03	73.66	53.93
	FCFS	93.37	66.06	47.19
Admission Policy Overheads (% of 100% Utilisation)	BEST EFFORT	1.69	4.38	6.65
	FCFS	1.55	3.29	4.89
Upper Bound for Admission Policy (% of 100% Utilisation)	BEST EFFORT	12.0	42.5	52.5
	FCFS	6.0	15.0	21.0

Table 7.6: Constant Periodic Utilisation of 25% with an Average Sporadic Computation Time 50 ms.

Possible Total Utilisation (%)		100	150	200
Possible Sporadic Utilisation (%)		**50**	**100**	**150**
Sporadic Arrival Rate (per ms)		**0.01**	**0.02**	**0.03**
Total Sporadic Utility Obtained	BEST EFFORT	21,815	30,611	31,631
	FCFS	21,560	27,197	32,671
Maximum Total Sporadic Utility Obtainable	BEST EFFORT & FCFS	23,333	33,333	40,000
% of Maximum Utility Obtained	BEST EFFORT	93.49	91.83	79.08
	FCFS	92.40	81.59	81.68
Admission Policy Overheads (% of 100% Utilisation)	BEST EFFORT	1.09	3.50	5.30
	FCFS	1.00	2.48	3.96
Upper Bound for Admission Policy (% of 100% Utilisation)	BEST EFFORT	10.0	30.0	45.0
	FCFS	4.0	10.0	21.0

Table 7.7: Constant Periodic Utilisation of 50% with an Average Sporadic Computation Time 50 ms.

7.5.2 Interpretation of the Results

Tables 7.5, 7.6 and 7.7 cannot be compared like-for-like because different Periodic Utilisations alter the sporadic utility which can be gained when the remaining capacity on the processor is used by sporadics of different utilities. However, each of the tables is given the same **Possible Total Utilisations** in order that some comparisons can be made.

As with previous simulations, it is generally true that BE obtains a higher **% of Maximum Sporadic Utility** than does FCFS. As before this difference in performance generally increases as **Possible Sporadic Utilisation** takes the **Possible Total Utilisation** beyond 100%. As **Possible Sporadic Utilisation** increases, the **% of Maximum Sporadic Utility Obtained** decreases. As before, this is due to the rapidly rising admission policy overheads and bounds, which decrease the **Sporadic Utility Obtained**. Accentuating this effect is the fact that admission policy overheads are not allowed for, in the **Maximum Total Sporadic Utility Obtainable**, so that this maximum is an over-estimate. What is more, this over-estimate becomes larger as **Possible Sporadic Utilisation** increases.

Tables 7.5 and 7.7 show the extremes in the effect of different **Sporadic Arrival Rates**. Table 7.5 has the lowest Periodic Utilisation, and therefore requires higher Sporadic Arrival Rates in order to achieve the **Possible Total Utilisations** which are common to all three tables. Table 7.7 has the highest Periodic Utilisation and therefore requires lower Sporadic Arrival Rates in order to achieve the same **Possible Total Utilisations**. Therefore the overheads for admission policy are greater in Table 7.5 which as a result shows a more rapid decline in **% of Maximum Utility Obtained.**

Table 7.7 confirms that, when the upper bounds for admission policy become high (e.g. greater than 50% of Total Utilisation) then the performance improvement of BE can tail off. For example, at **200% Possible Total Utilisation**, Table 7.7 shows the **% of Maximum Utility Obtained** is *less* for BE than for FCFS. This occurs even though the **Sporadic Arrival Rate** (and therefore the admission policy overhead) is less than in Tables 7.5 and 7.6. The conclusion must be that, due to the comparatively high Periodic Utilisation of 50%, the **Sporadic Utility Obtained** is very sensitive to an increase in admission policy overheads and bounds. In fact a Sporadic Utility of only 31,631 is obtained at 150% **Possible Sporadic Utilisation**. Tables 7.5 and 7.6 show that, at lower Periodic Utilisations, similar admission policy overheads and upper bounds can be tolerated without reducing the performance of BE below that of FCFS. The dip in the performance of BE in Table 7.7 is probably the beginning of the breakdown observed in Table 7.1 at a **Possible Sporadic Utilisation** of 600%.

The results in Tables 7.5-7.7 show that there is a trade off between the **% of Maximum Sporadic Utility Obtained** and the *absolute* **Sporadic Utility Obtained**. Table 7.5 shows that at low resident periodic utilisation, a low **% of Max Sporadic**

Utility, but high absolute **Sporadic Utility** can be obtained due to relatively high sporadic arrival rates. Conversely, at high resident periodic utilisation in Table 7.7, a higher **% of Maximum Sporadic Utility,** but lower absolute **Sporadic Utilisations** are obtained at lower sporadic arrival rates. Table 7.6 has an intermediate periodic utilisation, and therefore its results may be regarded as a trade off between these two extremes.

7.6 CHANGING THE RELATIVE UTILITIES

7.6.1 Setting the Parameters of the Simulation

The final stage in the comparison of the performances of BE and FCFS Admission Policies was to vary the ratios of the utilities gained by High, Medium and Low optional computations. In other words the ratios R_1 and R_2 were varied in order to examine the comparative effects on BE and FCFS.

A single set of simulation parameters (Periodic Utilisation, Average Sporadic Computation Time, etc.) were adopted for both BE and FCFS so that BE and FCFS could be compared when only the ratios R_1 and R_2 were changed. Parameters were set at average values in an attempt to obtain typical results: Periodic Utilisation was fixed at 25%, Possible Sporadic Utilisation was 125% and Average Sporadic Computation Time was 50ms.

Table 7.8 shows that simulations were performed for equal values of R_1 and R_2 which increase from 2 to 100. In addition, unequal values were used: $R_1 = 100$ and $R_2 = 10$, and vice versa. The **Total Sporadic Utility Obtained** is given, as is the **Maximum Total Sporadic Utility Obtainable**. The Maximum Total Sporadic Utility Obtainable increases rapidly with R_1 and R_2, mainly because the utility gained by High Utility computations is the *product* of R_1 and R_2. For convenience, the final column of Table 7.8 shows the Total Sporadic Utility obtained by BE divided by the Total Sporadic Utility obtained by FCFS.

7.6.2 Interpreting the Results

Table 7.8 shows the expected result that, for both BE and FCFS, the **Total Sporadic Utility Obtained** increases with R_1 and R_2. However, for all values of R_1 and R_2 which were used, BE obtained a higher **Total Sporadic Utility** than FCFS.

A less obvious finding from Table 7.8 is that the **% of Max Sporadic Utility Obtained** stays fairly constant as R_1 and R_2 increase. The exception is at low values (e.g. $R_1 = R_2 = 2$) where a higher **% of Max Sporadic Utility** is obtained.

Clearly, when Periodic Utilisation, Sporadic Utilisation and Average Sporadic Computation Time are all constant, then each BE or FCFS simulation will produce an identical number of High Utility, Medium Utility, and Low Utility task completions. Therefore the only variation in **% of Max Sporadic Utility Obtained** is brought about by changing the relative utilities given to each of the different task completions. Hence an expression for the total utility obtained from a single simulation is:

$$(N_1 * R_1 * R_2) + (N_2 * R_2) + (N_3 * 1) \qquad (7.1)$$

where N_1 is the number of High Utility task completions, N_2 the number of Medium Utility task completions, and N_3 is the number of Low Utility completions. The maximum sporadic utility obtainable in a simulation is calculated according to:

$$(K_1 * R_1 * R_2) + (K_2 * R_2) + (K_3 * 1) \qquad (7.2)$$

where K_1, K_2, and K_3 are the optimum number of High Utility, Medium Utility and Low Utility task completions respectively, as calculated according to the example given in Section 7.2.1. (In the case of the Periodic Utilisation and Possible Sporadic Utilisation in Table 7.8, K_3 is zero.)

R_1	R_2	Total Sporadic Utility Obtained		Maximum Total Sporadic Utility Obtainable	% of Max Sporadic Utility Obtained		Ratio of BE Utility Obtained : FCFS Utility Obtained
		BE	FCFS		BE	FCFS	
2	2	42,969	38,537	58,333	73.66	66.06	1.115
10	10	757,205	600,885	1,125,000	67.31	53.41	1.260
100	100	70,416,475	54,415,503	105,000,000	67.06	51.82	1.294
100	10	7,046,424	5,445,585	10,500,000	67.12	51.86	1.294
10	100	7,534,674	5,958,405	11,250,000	66.97	52.96	1.265

Table 7.8: Varying R_1 and R_2 . (A constant Periodic Utilisation of 25%, Possible Sporadic Utilisation of 125% and Ave Sporadic Computation Time 50 ms.)

It can be seen that for a particular simulation, if the value of expression (7.1) is divided by the value of (7.2), the result tends to a constant value as R_1 becomes very large compared to R_2. Also, (7.1) ÷ (7.2) will be approximately constant when both R_2 and R_1 are large. However, when both R_1 and R_2 decrease towards 1, there is less benefit in the greater number of High and Medium Utility task completions obtained by (7.2), and therefore the utility obtained by (7.1) approaches the maximum of (7.2). This explains why, at $R_1 = R_2 = 2$, both BE and FCFS have higher **% of Max Sporadic Utility Obtained** than at higher values of R_1 and R_2.

Table 7.8 shows that, at $R_1 = R_2 = 2$, FCFS has approximately 13% higher gain in **% of Max Sporadic Utility Obtained** compared to the **% of Max Sporadic Utility** gained at $R_1 = R_2 = 10$ (i.e. 66.06 - 53.41). In contrast BE has an approximately 6% gain. This can be explained by the fact that FCFS, being a random selection of sporadic tasks, contains a higher proportion of Low Utility tasks and so the effect of the (N_3 * 1) factor is greater.

The final results in Table 7.8 compare the performances of BE and FCFS when R_1 and R_2 are unequal. BE still outperforms FCFS, but some shifts in performance can be observed. It can be seen that the effect of $R_1 > R_2$, is to increase the performance gap between BE and FCFS, whereas the effect of $R_1 < R_2$, is to decrease the gap. This is explained by FCFS having proportionally more Medium Utility task completions than BE. Therefore a higher R_2 provides a relative benefit for FCFS as compared to BE, whereas a lower R_2 relatively disadvantages FCFS.

7.7 SUMMARY OF THE SIMULATION WORK DONE

The above work reports on a comparison of BE and FCFS admission policies in order to evaluate this aspect of the constrained computational model. The first simulations used only two levels of utility (High and Low) in order to compare the admission policies. In these simulations, Possible Sporadic Utilisation was increased from 75% to 600% processor utilisation. BE outperformed FCFS except at very high Possible Sporadic Utilisations where the upper bounds required to guarantee BE were so large that its performance broke down. Two-level simulations were also used to investigate the performances of BE and FCFS when High and Medium utility computations were used. As expected, the result was that the performance gain by BE over FCFS was reduced.

The next set of simulations used the constrained model with three levels of utility, and similar results were obtained. However, when these simulations were performed with different resident periodic utilisations on the processor, it was found that the breakdown in BE began at Lower Total Utilisation, when the resident periodic utilisation on the processor was high. A high resident periodic utilisation also obtained a lower *absolute* Sporadic Utility than lower periodic utilisations. However, high resident periodic utilisation incurred lower admission policy overheads, and therefore obtained a relatively high *percentage* of the Possible Sporadic Utility.

The final simulations which were performed allowed the ratios of the utilities of **High : Medium** (R_1), and **Medium : Low** (R_2) optional computations to be varied. As expected, the Total Sporadic Utility Obtained increased with R_1 and R_2, for both admission policies. However, BE always obtained greater Total Sporadic Utility than FCFS. For both BE and FCFS (for a fixed set of simulation parameters) it was found that, as the values of R_1 and R_2 became high, the % of Maximum Sporadic Utility obtained became constant. However, as the values of R_1 and R_2 decrease towards unity, the % of Maximum Sporadic Utility obtained became higher.

7.8 CONCLUSIONS

In the following conclusions 'performance' is measured by the % of Maximum Sporadic Utility Obtained.

1. Best Effort admission policy can achieve a higher performance than FCFS admission policy, when the resident Periodic Utilisation on the processor is low, and the processor is overloaded with Sporadic Arrivals.

2. Under the conditions in 1 above, the performance gain in BE over FCFS increases with Possible Sporadic Utilisation and Sporadic Arrival Rate.

3. The overheads for BE are greater than those for FCFS. The consequence is that, as Sporadic Arrival Rate increases, the upper bounds on the WCET for BE increase more rapidly than those for FCFS. This causes BE to break down at lower Possible Sporadic Utilisations than FCFS.

4. For BE and FCFS, at a given Possible Total Processor Utilisation : as the resident Periodic Utilisation increases, the *absolute* Sporadic Utility which is obtained, decreases. However, the *% of the Maximum Sporadic Utility* which is obtained increases.

5. Larger Periodic Utilisations cause BE to break down at lower Possible Total Processor Utilisations.

6. Lower Average Computation Times cause BE to break down at lower Possible Sporadic Utilisations. This is because, for a given Possible Sporadic Utilisation, lower Average Computation Times require a higher Sporadic Arrival Rate, and therefore higher upper bounds for admission policy.

7. As the ratios between the utilities of High, Medium, and Low Utility computations increase, the % of the maximum possible Sporadic Utility which is obtained becomes constant. (It is assumed that Periodic Utilisation and Possible Sporadic Utilisation are fixed).

8. When the ratios between the utilities assigned to High, Medium and Low utility computations are small, the % of the maximum possible Sporadic Utility obtained increases.

9. When the ratio between the utilities assigned to Medium and Low utility computations, increases, relative to the ratio between the utilities assigned to High and Medium utility computations, the gain in performance of BE over FCFS is reduced.

7.9 VIABILITY OF THE CONSTRAINED MODEL

7.9.1 Windows of Operation

The simulations carried out indicate several windows in the values of performance parameters, within which the Constrained Computational Model using Best Effort Admission Policy, can provide considerably improved performance over FCFS. For example, between 100-200% Total Processor Utilisation the constrained model with BE admission, can gain 10% performance over the use of FCFS. However, either side of this range, the benefits of the model decline. At Total Processor Utilisations which are less than 100%, BE admission degenerates into FCFS. At the other extreme, when the processor is overloaded beyond a Total Processor Utilisation of 200%, the performance of FCFS eventually overtakes that of BE.

Resident Periodic Utilisation also enforces a window on the use of the model. Only between 10-50% Periodic Utilisation does BE gain in performance over FCFS. Below 10% Periodic Utilisation, the upper bounds on BE, required in order to generate a sufficient sporadic overload, become too great to allow BE to outperform FCFS. At greater than 50% Periodic Utilisation, the performance of BE drops below that of FCFS, even when the upper bounds for admission policy would otherwise be tolerable.

The average sporadic computation time also affects the viability of the model. If computation times are too small, then high sporadic arrival rates are required in order to generate reasonable processor overload. The result is that high upper bounds for admission policy cause an early breakdown in the performance of BE.

Increasing the utility ratios of the High, Medium and Low utility computations can augment the performance of BE compared to FCFS, but make little difference to the performance of BE compared to its ideal maximum performance. However, the applications programmer may still wish to set these ratios to reflect the relative importance of each category of optional computation.

7.9.2 Recommendations for the Model

Within the above windows of operation, the Constrained Computation Model with Best Effort Admission, can provide improved performance for optional computations. Furthermore, the parameter ranges which have been established (10-50% Periodic Utilisation, and 100-200% Total Processor Utilisation) are useful for a variety of applications.

Even in the cases where the model with BE provides only comparable performance to that of FCFS, there are still the benefits in being able to (i) distinguish between the utilities of optional computations, and (ii) increase the likelihood of a higher utility computation being performed in preference to a low utility computation. The use of FCFS alone, effectively removes utility as a meaningful concept within the application.

CHAPTER 8

IMPLEMENTATION OF THE COMPUTATIONAL MODEL

8.1 INTRODUCTION

The Constrained Computational Model described in Chapter 3 is not directly supported by any of the real-time programming languages which have been reviewed in Chapter 2. The aim of this Chapter is to show how the Computational Model may nevertheless be implemented in a language which is used for the engineering of real-time systems. The language chosen is *Ada 95* [1]. As described in Section 2.8.4, Ada is a large programming language with many features for the implementation of real-time systems (especially in the *Real-Time Systems Annex*).

Ada allows concurrent programming using *tasks*. Tasks are scheduled according to *static* or *dynamic* priorities. When considering how to implement optional computations in Ada 95, one obvious approach is to implement each optional computation as an Ada task which could be guaranteed by the Ada RTS, and could, if necessary, be aborted before it completes. However, this would give the real-time programmer little control over the optional computations, and would require considerable extensions to the existing Ada RTS.

An alternative is to implement optional computations by using Ada 95 constructs *inside* Ada tasks. This would (i) allow the programmer more ability to tailor the optional computations, (ii) provide a more efficient, lighter-weight implementation, and (iii) require less change to the Ada RTS. With this approach in mind, several Ada 95 constructs which may be of use, are now reviewed.

8.2 ADA 95 CONSTRUCTS

8.2.1 Protected Objects

Protected objects allow mutually exclusive access to data via *protected entries* and *procedures*, which give exclusive read/write access to the encapsulated data. Protected objects may also include *protected functions* which provide concurrent read-only access to the data. Entries to a protected object may be guarded by a *barrier*. Entry calls are enqueued if the barrier evaluates to *false*. In this way, entries in a protected object may be used to implement condition synchronisation. In common with Ada packages and tasks, protected objects have a specification and a body as is shown by the following example of the Ada syntax:

```
protected Specification_Example is
  entry ...;
  procedure ...;
  function ...;
private
  -- hidden subprograms, data etc.
end Specification_Example;

protected body Body_Example  is
  -- full bodies of entries, procedures, functions, etc.
end Body_Example;
```

The Real-Time Systems Annex to Ada 95 defines a locking protocol (*Immediate Ceiling Priority Protocol*) which applies a ceiling priority to each protected object in order to ensure mutual exclusion by tasks which are concurrently accessing any object. The locking protocol also limits the effects of priority inversion.

8.2.2 Requeue

The *requeue* statement can be executed by a server task or protected object which has accepted an entry call. The effect is to queue the call on another entry which may be internal or external to the original task or object which has been called. This *target* entry is named in the requeue statement:

```
requeue target_entry_name [with abort];
```

The original entry is *not* returned to after the target entry call has completed. The target entry must have a parameter profile which is conformant to the original entry statement. Because of this, it is forbidden to give parameters within the requeue statement itself, in case the programmer should erroneously supply a non-conformant parameter profile.

The optional *with abort* clause allows the original client task, which made the call to the server task or protected object to timeout on, or abort, the requeued call, during the period that the call is queued on the target entry. If *with abort* is not present, then a timeout or abort from the original client has no effect on the requeued entry call. Of course, once the requeued call has been accepted at the target, the rendezvous is allowed to complete, regardless of timeouts or aborts from the original client.

8.2.3 The Asynchronous Select

The asynchronous select statement has the following form:

```
select
  -- triggering_alternative:
  -- a triggering statement
  -- [optional code]
then abort
  -- abortable_code
end select;
```

The execution of the asynchronous select begins with the issuing of the triggering alternative which may be (i) an entry call or (ii) a delay statement. If (i) the entry call is queued, or (ii) a delay is issued, then the abortable code is executed. If the abortable code completes before the completion of the triggering alternative then if the trigger is (i) an entry call, an attempt is made to cancel it or (ii) a delay, then the delay is cancelled. After cancellation the asynchronous select is completed. If the entry call cannot be cancelled (e.g. because a rendezvous is in progress) then the call is allowed to complete, followed by any statements which have been included after the triggering statement.

If the triggering statement completes before the abortable code, then the abortable code is aborted, and any optional statements following the triggering statement are executed.

8.2.4 Using Ada 95 constructs for Optional Computations

Having decided in Section 8.1 that optional computations are to be implemented *within* Ada tasks, rather than at the task level, the asynchronous select statement seems a particularly suitable construct for optional computations. The triggering statement can be a call, for the guarantee of an optional computation, to a protected object which implements flexible scheduling. The abortable code within the asynchronous select can be the actual code of the optional computation.

Best Effort Admission Policy can be applied within the protected object which is called, and if the optional computation is guaranteed, then the call can be requeued on another entry within the object. This entry can have a barrier to allow the requeued call never to complete, and therefore the abortable code to run to completion. If required, the code of the optional computation *can* be aborted by lowering the barrier on the requeued entry call. This causes the requeued entry call to complete and the triggering statement of the asynchronous select to finish, thus aborting the code of the optional computation.

The following sections describe an Ada protected object which can handle requests for the admission of optional computations, and in doing so, interfaces with the Ada RTS. After this, the final part of the chapter considers a number of different application requirements for optional computations, together with examples of how each may be implemented in Ada 95.

8.3 AN ADA 95 IMPLEMENTATION FOR OPTIONAL COMPUTATIONS

8.3.1 Overview

Chapter 3 outlined the computational model which is now to be implemented in Ada 95. The model specifies that optional computations are first schedulability tested, and if possible admitted, at the utility level with which they arrive. Furthermore, optional computations can have their utility changed dynamically, during their execution. Such utility changes can be instigated by the tasks containing the optional computations themselves, or by other tasks within the application.

The Ada 95 code which is given below, implements the requirements of the computational model. Optional computations are implemented using asynchronous select statements, as in the following example:

```
-- an optional computation within an applications task

select
  Flex_Sched.Request(C, D, Utility, I);
then abort
  -- Assume ALL optional computations call Flex_Sched.Make_Started before
  -- starting the code for the optional computation.
  -- Optional computation code can call Flex_Sched.Make_High, Make_Medium,
  -- or  Make_Low Utility as required.
  -- Assume ALL optional computations call Flex_Sched.Make_Completed when
  -- finished.
end select;
```

In the above code, the applications task requests the admission and guarantee of an optional computation by calling a *Request* entry in a protected object *Flex_Sched.* This object implements a flexible scheduler in Ada, and interfaces with the Ada RTS. Appropriate parameters are passed to *Flex_Sched* when a call to *Request* is made. These are the WCET (*C*) of the optional computation, its deadline (*D*), its *Utility*, and its index value, I. The index, I, is used as a means of identifying each optional computation with the component of the data structure used to record its status (see Section 8.3.2).

Several assumptions are made about extra facilities provided by the Ada RTS. It is assumed that the Ada RTS can answer a call from *Flex_Sched*, to schedulability test C, within a given D. Should the test succeed, then the Ada RTS is assumed to return to *Flex_Sched*, the priority at which the optional computation should run. If the test fails then the RTS returns a negative priority. The Ada RTS need have no concern with utilities, which are entirely handled by the *Flex_Sched* object. It is also assumed that the RTS can answer a call to withdraw an optional computation from the task list, and in returning, can pass back the accumulated execution time of the computation.

If the request is guaranteed by *Flex_Sched*, then the original call to the *Request* entry is requeued so that it does not return. In this case the code of the optional computation which follows **then abort** is executed. (Note that this code must make procedure calls to *Flex_Sched* in order to register its starting and completion). If the call to *Request* fails to guarantee the optional computation, or the optional computation is aborted during its execution, then the original call to *Request* returns, and the optional computation code following **then abort** is aborted.

8.3.2 Specification of the Flexible Scheduler Object

The Ada code below shows the declarations required for, and the specification of, the protected object *Flex_Sched*. The data for each optional computation is held in an array (of type *Opt_Comps*) of records which is indexed by integer values which identify each optional computation. The protected object *Flex_Sched* is assumed to have the highest priority associated with it. It exports a *Request* entry and a number of procedures to applications tasks. The procedures allow applications tasks to register the start or completion of optional computations, and also permits changes in the utility of optional computations as required by the computational model.

The private part of the *Flex_Sched* specification defines two entries which implement entry families to support the holding of optional computations in (i) the abortable or (ii) the non-abortable state. The procedures *Make_Abortable* and *Make Non_Abortable* are used by *Flex_Sched* to move optional computations from one state to another. Procedure *Best Effort* implements the admission policy which attempts to guarantee a request by, if necessary, aborting current optional computations of lower utility. *Best Effort* calls the auxiliary procedures *Withdraw* and *Reguarantee* which interface with the RTS. Also declared are arrays of flags which are needed to (i) control the changes of state (Abortable <=> Non_Abortable) of optional computations and (ii) control the abortion of optional computations which are in the abortable state.

```
type Utility is (High,Medium,Low);

Max : constant := ....;

type Opt_Comp_Index is new Integer range 1..Max;

subtype Computation_Time is Ada.Real_Time.Time_Span;

subtype Deadline is Ada.Real_Time.Time;

Priority_Reject_Value: constant := ...;

type Flags is array(Opt_Comp_Index) of Boolean;

type Optional_Computation is
record
    Util: Utility;
    Comp: Computation_Time
    Dead: Deadline;
    Accum: Computation_Time
    Name: Task_Id;
    Old_Prior: Priority;
    Started: Boolean;
    Abortable: Boolean;
    Released: Boolean;
end record;

type Opt_Comps is array (1..Max) of Optional_Computation;

-- Flexible Scheduler Specification

protected Flex_Sched is
  entry Request (C : Computation_Time; D : Deadline; U: Utility;
                                                       I : Opt_Comp_Index);
  procedure Make_High (I : Opt_Comp_Index);
  procedure Make_Medium (I : Opt_Comp_Index);
  procedure Make_Low (I : Opt_Comp_Index);
  procedure Make_Started (I : Opt_Comp_Index);
  procedure Make_Completed (I : Opt_Comp_Index);
  procedure Make_Aborted (I : Opt_Comp_Index);
private
  entry Abortable (Opt_Comp_Index)
        (C : Computation_Time; D : Deadline; U: Utility; I : Opt_Comp_Index);
  entry Non_Abortable (Opt_Comp_Index)
        (C : Computation_Time; D : Deadline; U: Utility; I : Opt_Comp_Index);
  procedure Make_Abortable (I : Opt_Comp_Index);
  procedure Make_Non_Abortable (I : Opt_Comp_Index);
  procedure Best_Effort (C : Computation_Time; D : Deadline; U: Utility;
                                               I : Opt_Comp_Index; P : out Priority);
  procedure Withdraw (U: Utility);
  procedure Reguarantee(U: Utility);
  Opt_Comp : Opt_Comps;
  Trans_Abort: Flags := (others => False);
  Trans_Non_Abort: Flags := (others => False);
  Abort_Flags: Flags := (others => False);
end Flex_Sched;
```

8.3.3 Implementing the Public Interface of the Flexible Scheduler

As indicated above a call to *Request* which is guaranteed, is requeued within whichever entry family (see *Non_Abortable* and *Abortable* below) is appropriate for the utility of the computation. Each entry family is guarded by an array of flags which controls the transition of optional computations from one entry queue to another. The procedures *Make_High*, *Make_Medium*, and *Make_Low*, each call *Make_Abortable* or *Make_Non_Abortable* in order to make a required transition when the utility of an optional computation is changed. *Make_Abortable* and *Make_Non_Abortable* manipulate arrays of flags (*Trans_Abort* and *Trans_Non_Abort*) in order to perform transitions. In addition, the entry *Abortable* is guarded by an array of flags (*Abort_Flags*) which allow calls to *Request* to return, and therefore abort optional computations within the applications tasks.

By calling *Make_Started,* a medium utility optional computation ensures that it becomes non-abortable when it starts execution. For the sake of consistency, all optional computations should likewise register that they have started, even though there may be no change in their abortabilities. Calls to *Make_Completed* allow all applications tasks to be set to their previous priority, once their optional computations have finished.

```
protected body Flex_Sched is

procedure Make_High (I : Opt_Comp_Index) is
begin
Opt_Comp(I).Util := High;
if  Opt_Comp(I).Abortable then
  Make_Non_Abortable (I);
end if;
end Make_High;

procedure Make_Medium (I : Opt_Comp_Index) is
begin
Opt_Comp(I).Util := Medium;
if  Started then
  if Opt_Comp(I).Abortable then
    Make_Non_Abortable (I);
  end if;
else -- not Started
  if  not Opt_Comp(I).Abortable then
    Make_Abortable(I);
  end if;
end if;
end Make_Medium;

procedure Make_Low (I : Opt_Comp_Index) is
begin
  Opt_Comp(I).Util := Low;
  if  not Opt_Comp(I).abortable then
     Make_Abortable(I);
  end if;
end Make_Low;
```

```
procedure Make_Non_Abortable (I : Opt_Comp_Index) is
begin
  -- reset flag in Non_Abortable entry in case optional computation
  -- was previously made non_abortable.
  Trans_Abort(I) := False;
  -- set flag in Abortable entry so as to enable requeue on Non_Abortable
  Trans_Non_Abort(I) := True;
end Make_Non_Abortable;

procedure Make_Abortable(I : Opt_Comp_Index) is
begin
  -- reset flag in Abortable entry, in case computation previously made abortable
  Trans_Non_Abort(I) := False;
  -- set flag in Non_Abortable  entry so as to enable requeue on Abortable
  Trans_Abort(I) := True;
end Make_Non_Abortable;

entry Abortable (for I in Opt_Comp_Index) when (Trans_Non_Abort(I)
                                                          or Abort_Flags(I))  is
begin
  if  Trans_Non_Abort(I) = True then
    Opt_Comp(I).Abortable := False;
    requeue Non_Abortable(I) with abort;
  else  -- must be aborted
    null;
  end if;
end Abortable;

entry Non_Abortable (for I in Opt_Comp_Index) when Trans_Abort(I) is
begin
  Opt_Comp(I).Abortable :=  True;
  requeue Abortable(I) with abort;
end Non_Abortable;

procedure Make_Started(I : Opt_Comp_Index) is
begin
  Opt_Comp(I).Started := True;
  if Opt_Comp(I).Util = Medium then
    Make_Non_Abortable(I);
  end if;
end Make_Started;

procedure Make_Completed (I : Opt_Comp_Index) is
begin
  Set_Priority (Opt_Comp(I).Old_Prior, Opt_Comp(I).Name);
  Opt_Comp(I).Released := False;
  -- perform housekeeping on array member Opt_Comp(I)
end Make_Completed;

procedure Make_Aborted (I : Opt_Comp_Index) is
begin
  Set_Priority (Opt_Comp(I).Old_Prior, Opt_Comp(I).Name);
  Opt_Comp(I).Released := False;
  Abort_Flags(I) := True;
  -- perform housekeeping on array member Opt_Comp(I)
end Make_Aborted;

end Flex_Sched;
```

8.3.4 Handling Requests for Optional Computations

The code for the *Request* entry below shows that it calls *Best_Effort* and is passed back a priority value which is used to determine whether the request has been accepted or not. If the request is accepted, then the appropriate record within the array of optional computations is updated with the necessary data on the optional computation, the applications task is set its new priority, and the call is requeued on whichever entry (*Abortable* or *Non_Abortable*) is appropriate. If the request is denied by *Best Effort* then the call to *Request* returns, and the asynchronous select within the applications task is triggered thus aborting the code for the optional computation.

```
entry Request (C : Computation_Time; D : Deadline; U: Utility;
                                                          I : Opt_Comp_Index) is
Prior : Priority;
begin
  --call admission policy
  Best_Effort(C, D, Utility, I, Prior);
  if  Priority_Indicates_Accepted then
    -- initialise the flags for this optional computation's index value
    Trans_Non_Abort(I) := False;
    Trans_Abort(I) := False;
    Abort_Flags(I) := False;
    -- assign record in array of optional computations
    Opt_Comp(I).Util := Utility;
    Opt_Comp(I).Comp := C;
    Opt_Comp(I).Dead := D;
    Opt_Comp(I).Accum := Zero_Computation_Time;
    Opt_Comp(I).Name := Request'Caller;
    Opt_Comp(I).Started := False;
    Opt_Comp(I).Released := True;
    Opt_Comp(I).Old_Prior := Get_Priority(Opt_Comp(I).Name);
    Set_Priority (Prior, Opt_Comp(I).Name);
    case Utility is
      when High =>
        Opt_Comp(I).Util := High;
        Opt_Comp(I).Abortable := False;
        requeue Non_Abortable(I) with abort;
      when Medium  =>
        Opt_Comp(I).Util := Medium;
        Opt_Comp(I).Abortable := True;
        requeue Abortable(I) with abort;
      when Low  =>
        Opt_Comp(I).Util := Low;
        Opt_Comp(I).Abortable := True;
        requeue Abortable(I) with abort;
    end case;
  end if;
    -- request rejected : return to trigger asynchronous select
    -- in the applications task.
end Request;
```

8.3.5 Best Effort Admission Policy

Procedure *Best_Effort* is shown below. It takes the parameters passed to it by *Request* and applies the Best Effort algorithm, with the help of two auxiliary procedures *Withdraw* and *Reguarantee*. (These auxiliary procedures are described in detail in the next section.) *Withdraw* rescinds all optional computations which have lower utility than that of the request. If the request is subsequently accepted, then *Reguarantee* attempts to guarantee each withdrawn request in order of decreasing value density.

Procedure *Best_Effort* interfaces with the RTS by the use of two procedure calls : *GU* and *REIN*. *GU* calls the RTS to perform a single schedulability test of a WCET, *C*, within an absolute deadline, *D*. The index value, *I*, of the optional computation is also passed to the RTS, and the RTS passes back a priority, *P*, which indicates whether the guarantee was given and the optional computation has been accepted within the task list. A positive priority value indicates that the optional computation has been accepted, and that its applications task should be set at that priority. A predefined negative priority value indicates that the optional computation has been rejected.

The RTS procedure *REIN* efficiently reinstates all the withdrawn optional computations when a request has been rejected, even after lower utility computations have been removed. *REIN* provides an optimisation which is more efficient than calling *GU* again, for each withdrawn computation.

```
procedure Best_Effort (C : Computation_Time; D : Deadline; U: Utility;
          I: Opt_Comp_Index; P : out Priority) is
begin
  GU(C,D,I,P);
  if Priorty_Indicates_Rejected then
    if  U /= Low then
      -- remove all lower utility, abortable optional computations,
      -- and repeat attempt to guarantee.
      Withdraw(U);
      GU(C,D,I,P);
      if Priority_Indicates_Accepted then
        -- attempt to re-guarantee all withdrawn optional computations, in
        -- smallest residual computation time, within utility order.
        Reguarantee(U)
      else -- optional computation still refused
        REIN; -- optimised reinstatement of all withdrawn optional computations
      end if;
    end if;
  end if;
end Best Effort;
```

8.3.6 Auxiliary Procedures for Best Effort

The procedures *Withdraw* and *Reguarantee* are given below. (Repetitive code has been replaced by comments.) *Withdraw* calls the RTS procedure *UNGU(I, Acc)* which removes the optional computation of index *I* from the task list and returns, in *Acc*, the accumulated execution time of the optional computation. This allows *Reguarantee* to call the RTS procedure *GU*, passing it the *residual* computation time of the optional computation.

```
procedure Withdraw (U: Utility) is
J : Opt_Comp_Index;
Acc : Computation_Time;
begin
  case U is
    when High  =>
      -- withdraw unstarted medium, and all low utility, optional computations
      -- which are current.
      for J in 1 .. Max loop
        if (Opt_Comp(J).Util = Medium) and (not Opt_Comp(J).Started) and
                                                          (Opt_Comp(J).Released) then
           UNGU(J, Acc);
          Opt_Comp(J).Accum := Acc;
        elseif Opt_Comp(J).Util = Low and (Opt_Comp(J).Released) then
          UNGU(J,Acc);
          Opt_Comp(J).Accum := Acc;
        end if;
      end loop;
    when Medium  =>
      -- withdraw all low utility optional computations
      -- similar code to above
  end case;
end Withdraw;

procedure Reguarantee(U: Utility) is
J, K : Opt_Comp_Index;
Minimum : Computation_Time;
Prio : Priority;
begin
  case U is
    when High  =>
      -- attempt to re-guarantee all Medium utility optional computations
      while Still_Medium_Utility_Opt_Comps_Unconsidered loop
        -- find the Medium utility optional computation with the
        -- lowest residual computation time : first initialise Minimum
        for J in 1 .. Max  loop
          if (Opt_Comp(J).Util = Medium) and (not Opt_Comp(J).Started)
                                                   and  (Opt_Comp(J).Released) then
            if  (Opt_Comp(J).C - Opt_Comp(J).Accum) < Minimum then
               K := J;
               Minimum := Opt_Comp(J).C - Opt_Comp(J).Accum;
            end if;
          end if;
        end loop;
```

```
        -- mark as considered the optional computation with the minimum
        -- residual computation time :
        -- re-attempt to guarantee this minimum optional computation
        GU(Minimum, Opt_Comp(K).D, K, Prio);
        if  Priority_Indicates_Accepted then
          Set_Priority (Prio, Opt_Comp(K).Name);
        else
          -- abort optional computation in the Ada task
          Abort_Flags(K):= True;
          Opt_Comp(K).Released := False;
        end if;
      end loop;
      -- attempt to re-guarantee all Low utility optional computations
      while Still_Low_Utility_Opt_Comps_Unconsidered loop
        -- process the low utility computations as above
      end loop;
    when Medium  =>
    -- attempt to re-guarantee all Low utility optional computations
    while Still_Low_Utility_Opt_Comps_Unconsidered loop
       -- process the low utility computations as above
    end loop;
  end case;
end Reguarantee;
```

8.4 ALTERNATIVE OPTIONAL COMPUTATIONS

The first requirement to be considered is for optional computations which take the form of alternative computations arriving simultaneously. The first alternative requested is the most preferred one, while subsequent alternatives are less preferred and may be computationally cheaper. Therefore the less-preferred alternatives may be guaranteed when preferred versions are not. It is assumed that the preferred version is of high utility (i.e. non-abortable) so that there is no possibility of it being aborted and the less-preferred version being executed as well.

The following fragment of Ada shows alternative versions (v1, v2 ,...) of an optional computation which arrive simultaneously and can be implemented by the following asynchronous select statements:

```
select
  -- request the first preferred version, v1
  -- if this call returns then request an
  -- alternative version using another
  -- asynchronous select.
  select
    -- request alternative version, v2
    -- etc.
  then abort
    -- code for v2
  end select;
then abort
  -- code for v1
end select;
```

8.5 REPLACEMENT OF MINIMUM COMPUTATIONS

In this case the requirement is that there is a mandatory minimum computation which has been guaranteed before run-time and that this should, if possible, be replaced by a preferred version at run-time. Here v1 is the minimum computation while v2 is the version which is preferred at run-time and must be guaranteed dynamically. v2 must be a high utility computation (non-abortable) because it replaces v1 which is mandatory and was guaranteed statically. Therefore, once the request for v2 has been accepted, the call to request will never return and there is no possibility of v1 being performed as well as v2.

```
select
  -- request the version, v2 which is preferable at run-time
  -- code for v1 is here to be executed if request for v2 is denied
then abort
  -- code for v2
end select;
```

8.6 ABORTED OPTIONAL COMPUTATIONS

In this case we are considering Medium or Low Utility computations which if aborted will require some minimum computation to be executed instead. This can occur when there is a requirement for 'graceful degradation', or in the case of multiple versions where a preferred but unbounded computation may have to be aborted when time is left only for the execution of some less-preferred minimum computation. The following is the general form of Ada 95 code which can implement this:

```
select
  -- request the preferred optional computation.
  -- code for minimum computation or recovery is here if preferred
  -- optional computation is aborted or rejected.
then abort
  -- code for preferred computation
end select;
```

In a case of graceful degradation it may be that the abortion of the preferred computation is to be followed by a request for some alternative computation. Then the Ada code below can be used.

In the case of multiple versions it may be that the preferred version is a Low utility computation which is allocated a budget. The implementation must therefore ensure that the preferred version does not overrun its budget and thereby make lower priority computations unschedulable. The Ada 95 code in Section 8.7 can implement this.

```
select
  -- request the preferred optional computation.
  -- if this is aborted (or rejected) then request an
  -- alternative computation using another
  -- asynchronous select:
  select
    -- request alternative computation
    -- etc.
  then abort
    -- code for alternative computation
  end select;
then abort
  -- code for preferred computation
end select;
```

8.7 LOW UTILITY COMPUTATIONS

The computational model of Chapter 3 considers optional computations of Low Utility to be those computations whose WCET cannot be determined or whose WCET is very pessimistic. These computations may nevertheless be guaranteed a budget which, for example, may cover their minimum or average computation time. Because they may exceed their budgets, such computations must have their accumulated execution times monitored. The assumption is that the Ada RTS can be extended in order to detect that the accumulated execution time of an optional computation has reached the allocated budget. The Ada RTS then triggers the abortion of the computation. The following code shows an example of Multiple Versions where an unbounded preferred version has been allocated a budget. If the preferred version is rejected, aborted due to Best Effort Admission Policy, or aborted due to the expiry of its budget, then a minimum version is executed:

```
select
  -- request the budget for preferred version
  -- code for minimum version here is executed if
  -- preferred version is rejected or aborted due to Best Effort.
then abort
  select
    -- a trigger from the RTS when the accumulated execution time has
    -- reached the budget allocated
    -- code for minimum computation version here is executed if
    -- preferred version uses its budget.
  then abort
    -- preferred version code which takes indeterminate length of time
  end select;
end select;
```

The question arises as to whether low utility computations such as the above could be granted an additional budget if they consume their original budget before they complete. The problem here, is that the expiry of the original budget triggers an asynchronous event which would interrupt the flow of control of the optional computation, and therefore make it difficult to resume.

8.8 IMPRECISE COMPUTATIONS

Imprecise Computations consist of a minimum computation which may be mandatory followed by iterations which improve upon the minimum precision (see Section 2.3.1). If the deadline of the Imprecise Computation is relatively small, then it may be most efficient to pose the request in the form of a request for alternative computations in which the alternatives are decreasing numbers of iterations of the imprecise computation. Obviously a higher number of iterations is preferable to a lower number and would be requested earlier in the sequence of alternative computations. See Section 8.4 for an implementation of alternative computations in Ada 95.

In the case of an Imprecise Computation whose deadline is relatively long, it is more flexible to request each new iteration after the previous iteration has completed. This then takes advantage of the dynamic occurrence of slack. In general, iterations of an Imprecise Computation may have bounded or unbounded computation times. Implementations for each case are now presented.

8.8.1 Bounded Computation Times

The server below provides the necessary implementation for the case of iterations of Imprecise Computation with bounded computation times. Each iteration can be guaranteed at High Utility (i.e. non-abortable). With this method, the client can actually specify the deadline by which the imprecise computation must complete. This provides the absolute deadline in the above code, which is 'delayed until' in the outer asynchronous select. Within the inner loop, the Imprecise Server repeatedly requests iterations of the Imprecise Computation with bounded computation times. The server exits the inner loop when a sufficiently precise result has been calculated.

This server can be contrasted with a more asynchronous Imprecise Server given by Burns and Wellings [5]. Their server incorporates a wait upon a persistent signal from a client task which indicates that the client wishes to read the result. The advantage of the method given here is that iterations of the Imprecise Computation can be requested for guarantee at High Utility against the overall deadline for the Imprecise Computation which has been specified by the client. Therefore, once guaranteed, these iterations will never be aborted as they may be in the case of the Burns and Wellings server [5]. The inner loop can still be aborted in between the executions of guaranteed High Utility computations, or if the **exit when** statement evaluates to true.

```
-- declare protected object Shared_Data which Client task can read and
-- Imprecise Server can write

task body Imprecise_Server is
  -- declarations
begin
  loop
    -- initialise absolute deadline to that specified by client
    -- produce result with minimum required precision
    Shared_Data.Write(Result);
    select
      delay until overall deadline;
    then abort
      loop
        select
          -- request next iteration with bounded WCET
          -- request rejected: set flag for delay to avoid 'busy wait'
        then abort
          -- perform next iteration
          -- and compute refined Result
          Shared_Data.Write(Result);
          exit when Best_Result_Obtained;
        end select;
        if  flag_ set then
          -- delay  against busy wait
          -- reset flag
        end if;
      end loop;
    end select;
  end loop;
end Imprecise_Server;
```

The purpose of the **delay** within the inner loop is to ensure that processor time is not wasted by 'busy waiting', if repeated requests for further iterations are rejected by *Flex_Sched*. (This inner **delay** is obviously subordinate to the **delay until** outside the loop.) The flag ensures that the inner delay is not incurred when requests for iterations of the imprecise computation are accepted. (It could be argued that if such busy waiting is carried out at a low priority, then it will not interfere with tasks of higher utility. However, even if this is true, 'busy waiting' might still waste computation time which could be used for the imprecise computation itself.)

Ideally the problem of 'busy waiting' for a request to be accepted, could be solved by the ability to wait upon an asynchronous signal from the RTS when new slack is available. Again this would require extensions to the Ada 95 RTS.

8.8.2 Unbounded Computation Times

If each of the iterations are unbounded it may only be possible to guarantee a budget which will cover their average or minimum computation times. In this case the server can be coded exactly as above except that nested asynchronous select statements are required within the inner loop (see code below). The outer of these may now be triggered by the rejection *or abortion* of the optional computation. The innermost asynchronous select ensures that the budget which has been allocated for the current iteration is not exceeded.

Note that the same method can be used to avoid wasting processor time on 'busy wait'. In this case the delay to avoid busy wait, placed within the inner loop, will occur in the case of the optional computation being rejected *or aborted.* This may be appropriate, because an aborted request may indicate that the system is heavily loaded with higher utility computation, and therefore the delay is unlikely to waste slack which could have been used by the imprecise computation.

```
loop
  select
    -- request budget for next iteration
    -- request rejected, or low utility computation aborted:
    -- set flag for delay and avoid 'busy wait'
  then abort
    select
      -- a trigger from the RTS when the accumulated execution time has
      -- reached the budget allocated
    then abort
      -- perform next iteration
      -- and compute refined Result
      Shared_Data.Write(Result);
      exit when Best_Result_Obtained;
    end select;
  end select;
    .
end loop;
```

8.9 COMPOUND COMPUTATIONS

The computational model of Chapter 3 describes a *compound* computation as one in which multiple requests are 'anded' together. In other words, two or more optional computations, which arrive simultaneously, must all be guaranteed, or none can be accepted. All the computations should be of High utility because allowing them to be abortable (Medium or Low utility) would be inconsistent with requiring all of them to be guaranteed.

The requirement for compound computations can be implemented in Ada 95 by nested asynchronous selects. In the following code fragment, *C1, C2*, etc. represent the optional computations, both of which must be guaranteed. Because *C1* and *C2* are guaranteed sequentially, there is a possibility that *C1* could be guaranteed, followed by a failure to guarantee *C2*. In this case *C1* must be aborted even though it has been guaranteed as a High utility computation. In order to allow this abortion, *C1* must first be relegated to a Medium utility computation, and then aborted, as is shown in the following code :

```
select
  -- request C1, the first of the computations.
  -- If C1 cannot be guaranteed then the compound request is abandoned here.
then abort
  select
    -- request C2, the second of the computations. If C2 cannot be guaranteed
    -- then the compound request is abandoned here, but must abort C1:
    Make_Medium(C1_Index);
    Make_Aborted(C1_Index);
  then abort
    --  in-line code for C1 and C2.
  end select;
end select;
```

It can be seen from the above code that the computations for *C1* and *C2* are placed in-line. This could, for example, satisfy the requirements for a control system in which a sequence of operations must be carried out, in order, and to consecutive deadlines. The requirement may be that all of the operations must be carried out within their time constraints, or the sequence should not be embarked upon at all. In terms of the above code fragment, *C1* would be required to execute first within a deadline *D1*, followed by the execution of *C2* within a longer deadline, *D2*.

If required, concurrency could be introduced into compound computations within the framework of the asynchronous select statements above. This could be achieved by the use of entry calls which activate code located in other tasks. Another possibility would be to elaborate child tasks, each containing a component of the compound computation, at the point where the in-line code would otherwise be placed. The abortion of such dependent tasks would be an abort-deferred operation, and therefore these tasks could not be immediately aborted if the asynchronous select was triggered. However, this is not a problem in the case of compound computations which are, by definition, non-abortable.

8.10 SIEVE FUNCTIONS

According to Audsley et al. [4] Sieve Functions can be divided into a sequence of bounded and unbounded computations. For example *C1, X1, C2, X2, C3* can represent the computational components of a sieve function. *C1, C2* and *C3* are the bounded computations which are essential to the completion of the function. In contrast, *X1* and *X2* are unbounded components which are desirable but not essential to the completion of the sieve function. It may be possible only to derive an average or minimum computation time for the unbounded components. Therefore the best guarantee that they can be given, is for a budget of time which covers the average or minimum computation time. In other words, according to the computational model of Chapter 3, these components would be classed as low utility computations.

A method for implementing such a sieve function in Ada 95 is firstly to attempt to guarantee the sum of the computation times of the bounded components as a single high utility computations to be carried out within the overall deadline for the sieve function. If the guarantee is given, and the function executes, guarantees for each of the unbounded components can be requested at the points where each of them occur in the sequence of computations. If a request for an unbounded component is rejected, then the component is omitted and the next bounded component is executed.

```
select
  -- request C1 + C2 + C3, the high utility, bounded computations
  -- if rejected then the sieve function is abandoned here.
then abort
  -- perform C1
  select
    -- request a budget for X1, the first of the low utility, unbounded
    -- computations : if X1 budget cannot be guaranteed then abandon X1.
  then abort
    select
      -- a trigger from the RTS when the accumulated execution time has
      -- reached the X1 budget allocated
    then abort
      -- code for X1.
    end select;
  end select;
  -- perform C2
  select
    -- request a budget for X2, the 2nd of the low utility, unbounded,
    -- computations : if X2 budget cannot be guaranteed then abandon X2.
  then abort
    select
      -- a trigger from the RTS when the accumulated execution time has
      -- reached the X2 budget allocated
    then abort
      -- code for X2
    end select;
  end select;
  -- perform C3
end select;
```

It is worth noting that the bounded components have been guaranteed first as high utility, non-abortable computations, and therefore they cannot subsequently be made unschedulable by the guarantee of low-utility components. As usual, provision has be made to prevent the unbounded computations exceeding their budgets.

8.11 PERIODIC TASKS WITH CUMULATIVE ERRORS

8.11.1 With High Utility Optional Computations

```
-- initialise Next_Period
-- set Reject_Count to N -1 to force initial  performance of
-- minimum  + optional computation
loop
  if Reject_Count < (N -1) then
    -- perform minimum computation (guaranteed before run-time)
    select
      -- request optional computation
      -- request rejected therefore increment Reject_Count
    then abort
      -- code for optional computation
      -- set Reject_Count to zero
    end select;
  else
    -- code for minimum + optional computation
    -- set Reject_Count to zero
  end if;
  delay until Next_Period;
  Next_Period := Next_Period + Period;
end loop;
```

In [7] Chung et al. discuss the requirements for periodic tasks with cumulative errors. In applications such as radar tracking, periodic computations may be divided into a minimum and an optional component. The minimum component must run every period but the optional computation component may be terminated early, with the result that an error will be accumulated. The requirement is that the optional component must complete at least every *Nth* period in order that the error is prevented from exceeding a level which cannot be tolerated by the application. (It is assumed that static schedulability analysis ensures that there is at least sufficient computation time for the 'optional' component to execute every *Nth* period, when it is in effect mandatory.)

Such periodic computations could be implemented by the Ada 95 code shown above. In this case optional computations of High Utility are used. They provide a simpler implementation than using Low Utility optional computations (compare 10.2 below). However, High Utility optional computations suffer the disadvantage that are less flexible because they cannot be aborted. *Reject_Count* accumulates the number of consecutive rejections of requests for the guarantee of optional computations. *N - 1* is the number of consecutive periods for which the accumulation of the error can be tolerated.

8.11.2 With Low Utility Optional Computations

The requirements for periodic tasks with cumulative errors are better met by the use of Low Utility optional computations. The disadvantage is that the Ada implementation is more complex due to the fact that Low Utility computations are guaranteed a *budget*, and the implementation must ensure that the budget is not exceeded. As before, this can be done by including a trigger from the RTS which will abort an overrunning computation. In the following Ada, *Fail_Count* plays a similar role to *Reject_Count* above. *Fail_Count* is a counter which represents the current number of consecutive occasions on which the optional computation has been rejected, or aborted. Abortion can occur either because the computation has failed to complete within its budget, or because Best Effort has guaranteed a higher utility computation.

```
-- initialise Next_Period
-- set Fail_Count to N -1 to force initial performance of
-- minimum  + optional computation.
loop
  if  Fail_Count < (N -1) then
    -- perform minimum computation (guaranteed before run-time)
    select
      -- request budget for optional computation
      -- request rejected, or the computation is
      -- aborted due to Best Effort : increment Fail_Count
    then abort
      select
        -- a trigger from the RTS when the accumulated execution time has
        -- reached the budget allocated
        -- budget inadequate : increment Fail_Count
      then abort
         -- code for optional computation
        -- set Fail_Count to zero
      end select;
    end select;
  else
    -- code for minimum + optional computation
    -- set Fail_Count to zero
  end if;
  delay until Next_Period;
  Next_Period := Next_Period + Period;
end loop;
```

In the above code, race conditions can occur around the setting of *Fail_Count* to zero. Such race conditions will not break the constraint that the optional computation must execute at least every *N* periods. Race conditions could be caused by the abortion of the second part of the inner asynchronous select statement, after the code for the optional computation has completed, but just before the inner **'end select'**, or just before the 'set Fail_Count to zero'. In either case 'increment Fail_Count' will execute after the abort has been triggered. If the RTS interrupts just before the inner **end select**, then *Fail_Count* will be incremented to one. If the RTS interrupts just before 'set Fail_Count to zero', then *Fail_Count* will be incremented from its last value. In either case, the worst that can happen is the forcing of an early execution of the full computation after the **else.**

8.12 REPLICATED COMPUTATIONS

Applications may arise where it is necessary for a computation within one task to either change the utility of, or abort a computation in another task. Such a requirement can arise in the case of optional computations which are replicated to enhance fault tolerance. If one of the replicants completes before the other it may be required to lower the utility of its fellow or even abort its fellow altogether. Conversely if one of the replicants fails it may be necessary to raise the utility of its fellow. Two capabilities are required:

(i) the ability to change the utility of another replicated computation.
(ii) the ability to abort another replicated computation.

It is assumed that, to enhance fault tolerance, the replicated code would be guaranteed initially as a High-Utility (i.e. non_abortable) computation. (i) above may be implemented by allowing a replicated computation to make a call to the appropriate procedure in the protected object *Flex_Sched*, in order to change the utility of a fellow replicant. (ii) may be implemented by placing the replicated computation within an outer asynchronous select which can be triggered by a persistent signal from another replicant. The code for each of the replicated computations could take the following form:

```
select
  -- wait upon persistent signal from a fellow replicant
then abort
  select
    -- request the high utility replicated computation
  then abort
    -- replicated code
    -- send signal to fellow replicant to abort
  end select;
end select;
```

Note that the use of a persistent signal means that the signalling replicant not only aborts other replicants, but also aborts itself. Obviously, race conditions may occur in the signalling to abort, of each replicant to its fellow(s). However this should not undermine the requirement for at least one completed execution of the replicated code.

8.13 CONCLUSIONS

Ada 95 does not provide direct support for optional computations. To implement optional computations at the level of Ada tasks would give the real-time programmer little ability to tailor optional computations, and would require considerable extensions to the Ada RTS. However, the *asynchronous select statement* is a construct which can be used to program optional computations *within* Ada tasks. It has sufficient expressive power to implement many different applications application requirements for optional computations.

The Constrained Computational Model of Chapter 3 may be supported by a flexible scheduler implemented in Ada 95. The flexible scheduler has sole responsibility for handling utilities. The scheduler implements Best Effort Admission Policy by making appropriate calls to the Ada RTS, which need only provide functions to (i) schedulability test a single optional computation (ii) withdraw lower utility optional computations from the task list and (iii) efficiently reinstate lower utility computations when a higher utility request has been rejected. Within the flexible scheduler, the Ada 95 *requeue* statement can be used to queue requests for optional computations, which arise from asynchronous select statements within the applications tasks. Requests are requeued on an entry with a barrier which can be manipulated to allow the completion or abortion of optional computation code, within the asynchronous selects. *Requeue* statements are also used to change the state of optional computations form *abortable* to *non-abortable* and vice versa.

Fundamental problems occur unless the Ada RTS can be extended to support accumulated execution times. The Best Effort algorithm, as implemented in the protected object *Flex_Sched*, requires accumulated execution times to be passed back from the RTS in order to attempt the re-admission of withdrawn optional computations, in order of decreasing value density. Lack of availability of accumulated execution times also affects the Ada support for Low Utility Computations which have unbounded WCETs. According to the Constrained Computational Model, these computations should be guaranteed a time budget, which may turn out to be insufficient. In order to avoid these computations exceeding their budgets, their accumulated execution times must be available to be monitored. Computation should be aborted when an accumulated execution time equals the guaranteed budget.

In general, the above work could be extended to provide an Ada implementation which supports the use of optional computations within multiprocessor clusters such as those of Chapter 6. For example the *Request* call for the guarantee of an optional computation could be extended to allow *Shuffle Schedulability Testing*. (If *Request* at processor i fails, then attempt to guarantee at processor $i + 1$, etc.).

CHAPTER 9

CONCLUSIONS

9.1 REVIEW OF THE WORK DONE

Chapter 1 outlined the assumptions of this thesis that standard hardware and programming languages can be used in support of flexible scheduling. The thesis proposition was given as:

"The application requirements for flexible scheduling can be embraced in a constrained computational model for which cost-effective run-time support can be provided. The model can be implemented in a standard programming language so that applications written in this language can increase their utility."

This proposition led to the adoption of three broad objectives or *strands*:

1. To investigate the requirements for optional computations in the next generation of real-time systems, and to derive a computational model which is sufficiently constrained to be supported by a RTS executing on the same processor as the application tasks.

2. To develop more cost-effective support for flexible scheduling than that which currently exists. This is to be done by the development of (i) computationally cheaper guarantee algorithms for optional computations and (ii) methods of allocating optional computations within a multiprocessor cluster, such that throughput of optional computations is maximised.

3. To demonstrate that optional computations may be implemented in a standard programming language.

The preceding 7 chapters have reported on the work done within each strand. Chapters 2 and 3 relate to strand 1. Chapter 2 reviewed, amongst other topics, the complex application requirements for optional computations. Chapter 3 presented a constrained computational model which can support many of these requirements.

Chapters 4, 5 and 6 support strand 2, by investigating the use of guarantee algorithms with FCFS Admission policy. Chapters 4 and 5 developed and evaluated the performance of a suite of on-line schedulability test algorithms. Chapter 6 investigated allocation methods which optimise the throughput of optional computations within a multiprocessor cluster.

Chapter 7 evaluated, by simulation studies, the use of a newly developed guarantee algorithm within the computation model of Chapter 3. The model used Best Effort admission policy instead of FCFS.

Finally, Chapter 8 supported strand 3 by showing that the constrained computational model can be implemented in the standard programming language *Ada 95*.

Each strand of the work of this thesis addresses one aspect of the provision of optional computations within real-time systems. These aspects are inevitably interrelated. Therefore this thesis provides an integrated approach to optional computation which embraces requirements, programming language, run-time support, tasking model and multiprocessor configuration.

9.2 GENERAL CONCLUSIONS FROM THE WORK DONE

The following are a list of the main conclusions from the foregoing chapters:

- A constrained computational model for optional computations can satisfy many of the relevant application requirements, and can be supported cost-effectively.

- Guaranteeing optional computations can provide greater throughput of computations which meet their deadlines, than simply executing them in background.

- Exact schedulability test algorithms are not always the most cost-effective. There is always a trade-off between the overheads incurred by schedulability testing, and the total throughput of optional computations.

- FCFS admission of guaranteed optional computations improves performance generally over background processing. However, Best Effort admission policy can improve performance even further, under certain ranges of operating parameters.

- In a multiprocessor cluster, simple methods of allocating optional computations (such as adapted Round Robin, or Shuffle Schedulability Testing) can provide greater throughput than more complex methods.

- A constrained computational model for optional computations can be implemented in a standard programming language such as Ada 95.

9.3 CONTRIBUTION

This thesis complements and extends previous work in the area of flexible scheduling research. The Spring Project (reviewed in Chapter 2) provides computationally expensive guarantee algorithms and decentralised scheduling. These require the support of a specialised co-processor, and a systems processor respectively. This thesis avoids the need for specialised or dedicated hardware, by developing computationally cheaper methods of (i) guaranteeing optional computations and (ii) directing them to the processor most likely to guarantee them. In devising these methods, the work of Audsley [2] on static schedulability testing is extended into the domain of dynamic schedulability testing.

Elements of the work of this thesis also complement and extend the recent work of Davis. In [10] Davis makes use of schedulability Test 2 (given in Chapter 5) in conjunction with his slack stealing algorithm, in order to provide a sufficient but not necessary acceptance test. Conversely, Chapter 5 adopts and evaluates Davis' method [9] for optimal placement of aperiodic tasks within a task list. Finally, the work of Davis et al. [12] on Best Effort is extended in Chapter 7, where detailed investigations of the overheads incurred by the policy are made, within the context of a complete computational model.

9.4 FUTURE WORK

The work of this thesis indicates a number of avenues for further research:

- Work on setting the implementation techniques developed here, into the larger software engineering context. For example, the mapping of complex requirements for Real-Time AI onto the constrained computational model outlined in Chapter 3.

- Further development of the computational model, and its interface with Run-Time Support. For example, *approximate processing* could be supported within the model. The application could demand, from the RTS, the amount of slack available at a particular priority level, for an optional computation which is to perform approximate processing at that priority level. The application could then set the execution parameters of the optional computation in such a way that its WCET conforms to the slack available.

- The algorithm for Best Effort Admission Policy could be refined. As it is, Best Effort admits higher utility optional computations by, if necessary, aborting lower utility optional computations. No judgement is made as to whether the utility lost by aborted computations is outweighed by the utility gained in the new arrival. Some measure of the utility loss could form a criterion for the admission of the newly arrived optional computation. Such improvements in Best Effort Admission Policy might extend the windows of operation within which Best Effort provides higher utility than FCFS.

- Abortable, low-utility optional computations could be guaranteed more cheaply than non-abortable high-utility computations. The constrained computational model presented in Chapter 3 allows low utility optional computations to be abortable at any time. Even if they are not aborted, their budgets may turn out to be inadequate for their computation to complete. However, such low utility computations still incur the overhead of Best Effort Admission, the most expensive component of which is the schedulability test algorithm itself. There is a case for saying that abortable computations should be given cheaper (pessimistic) guarantees, because their utility may, in any case, be lost.
  This area of the computational model overlaps with the work of Liu et al. [33,34]. These researchers do not guarantee optional computations, but optimise the chances of them meeting their deadlines, thereby gaining greater total utility for the system.

- The Ada RTS could be extended to support flexible scheduling. This would require the implementation of support for the calls to the RTS which are assumed in Section 8.3. These are *GU*, *UNGU*, and *REIN*, which perform the guarantee of an optional computation, the withdrawal of an optional computation from the task list, and the efficient reinstatement of all withdrawn computations, respectively. The implementation of this run-time support would allow the overheads incurred by these calls to be measured, and would provide further evaluation of the computational model. Further extensions to the Ada RTS, for example in support of *approximate processing,* could also be considered.

- The Ada 95 implementation of the computational model (Chapter 8) could be extended to support the use of optional computations within multiprocessor clusters (Chapter 6). For example a call for the guarantee of an optional computation could be extended to allow *Shuffle Schedulability Testing* : if the call for guarantee at processor i fails, then attempt to guarantee at processor $i + 1$, etc.

9.5 FINAL THOUGHTS

Flexible scheduling was introduced to support adaptivity within real-time applications. However, its overheads can be great, and therefore the schemes for flexible scheduling themselves should be adaptable. The foregoing chapters demonstrate :

(i) the large bounds required to guarantee the guarantee algorithms themselves
(ii) the varying overheads of the guarantee algorithms
(iii) the difficulty in bounding some forms of optional computation

In the work of this thesis the above demands are accommodated by static schemes, such as guaranteeing that a fixed, maximum arrival rate of sporadic tasks can be schedulability tested, or, in the case of a multiprocessor, rigid allocation according to Shuffle Schedulability Testing. Such fixed schemes may prove inadequate for intelligent real-time systems which require adaptivity within the flexible scheduling itself. One approach would be to progressively drop the optional computation overheads listed above, as the loading of *critical* tasks on the system increases. For example, under the highest loading of critical computation, schedulability testing may be abandoned altogether, and optional computations rejected out-of-hand. On the other hand, under light loading of critical tasks, exact schedulability tests may be performed, and may be guaranteed to execute at a maximum rate.

Such adaptivity within the flexible scheduling itself, can be partially satisfied within the framework of the constrained computational model presented in Chapter 3. The model allows dynamic changes in the utility and abortability of optional computations. Wider changes within an application may be accommodated by global changes to the parameters of the model such as:

(i) altering the utility ratios R_1 and R_2
(ii) changing the schedulability test algorithm which is provided by the RTS.

More drastic changes within an application may require the constrained model to be amended. For example, an increased periodic utilisation by critical tasks may require admission policy to be changed from Best Effort to FCFS. Adaptivity may also be required in regard to the allocation strategy for optional computations within multiprocessors. For example, if Shuffle Schedulability Testing is employed, it may be beneficial to limit schedulability testing to a single test per optional computation under conditions of high loading.

APPENDIX A

ADAPTATIONS OF STATIC ALGORITHMS

A.1 THE $O(N^2)$ ALGORITHM

In the $O(N^2)$ algorithm, the interference from all higher priority tasks, is calculated *for the duration of the deadline* (D_i) of the test task, i. The number of interferences by a higher priority task is calculated by taking the ceiling of $D_i \div T_j$ where T_j is the period of a higher priority task, j. A dynamic refinement is to first subtract from D_i the offset (O_j) of the next release of the interfering task, j. Also, any residual execution time (R_j) of the interfering task must be added to the total interference of task j. Finally, schedulability is tested by comparing the test task's deadline (D_i) with the sum of interferences over all higher priority tasks plus the current computational requirement of the test task itself. If the test task is currently active, its computational requirement will be its residual execution time R_i, otherwise its WCET (C_i) will have to be considered against D_i, the deadline of the test task's next activation. In the case of the test task being the sporadic task itself (see Figure A.1), then the total interference over all higher tasks, j, is calculated by:

$$I_S = \Sigma_j(\lceil (D_S - O_j) \div T_j \rceil_0 \, C_j + R_j) \qquad \text{(A.1)}$$

where:

D_S is the sporadic deadline

R_j is the current residual execution time of the interfering task

O_j is the offset of the interfering task

T_j is the period of the interfering task

C_j is the WCET of the interfering task

and $\lceil X \rceil_0$ (i) returns 0 if $X \leq 0$

(ii) returns $\lceil X \rceil$ if $X > 0$

Figure A.1 shows that the interference, I_2, of *task 2* in the sporadic task will be pessimistically assumed to include all of the computation time, C_2, of the final hit of *task 2,* despite the expiry of the sporadic deadline before that final hit finishes.

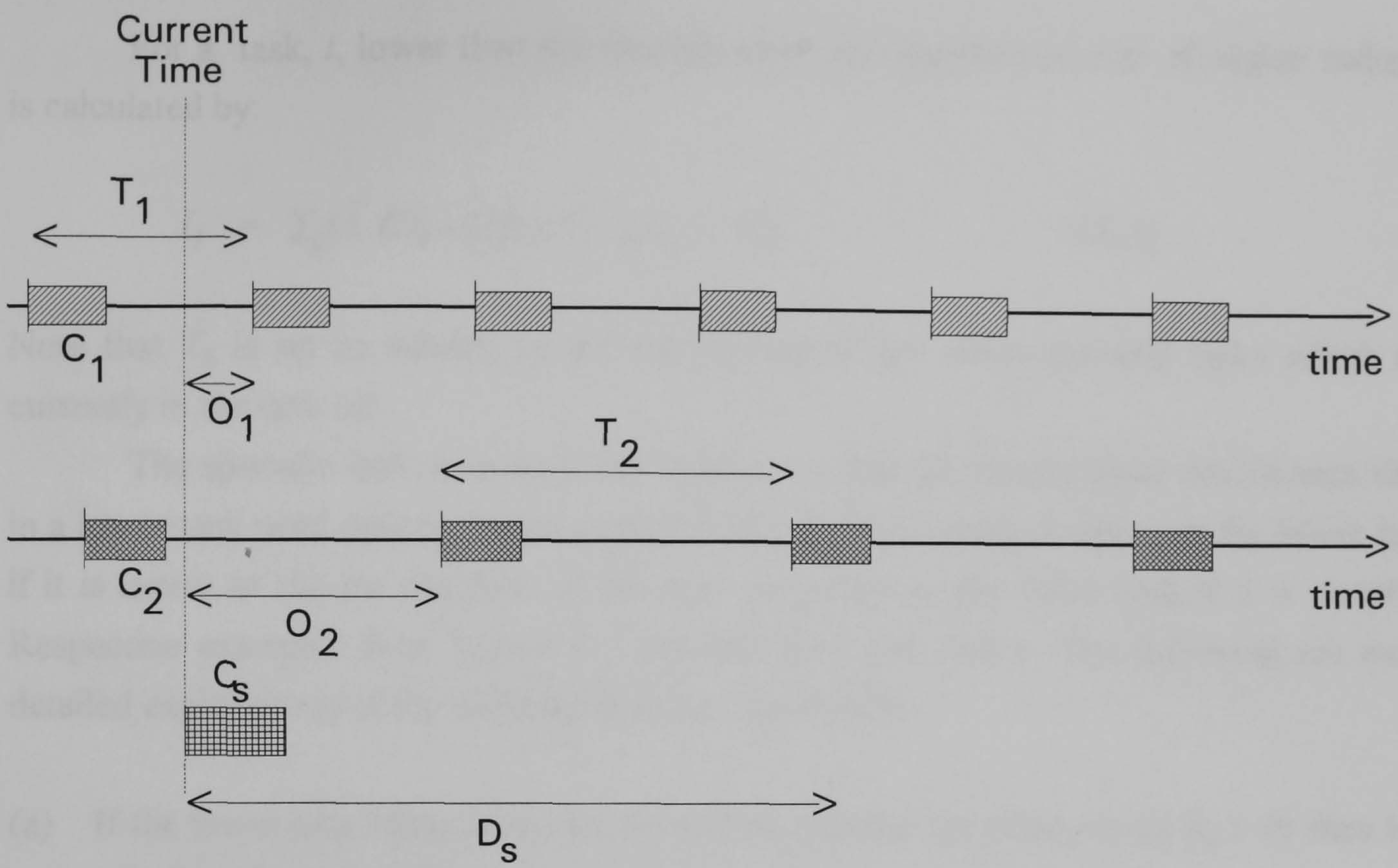


Figure A.1: Computation times (C) for periodic tasks above the sporadic task.

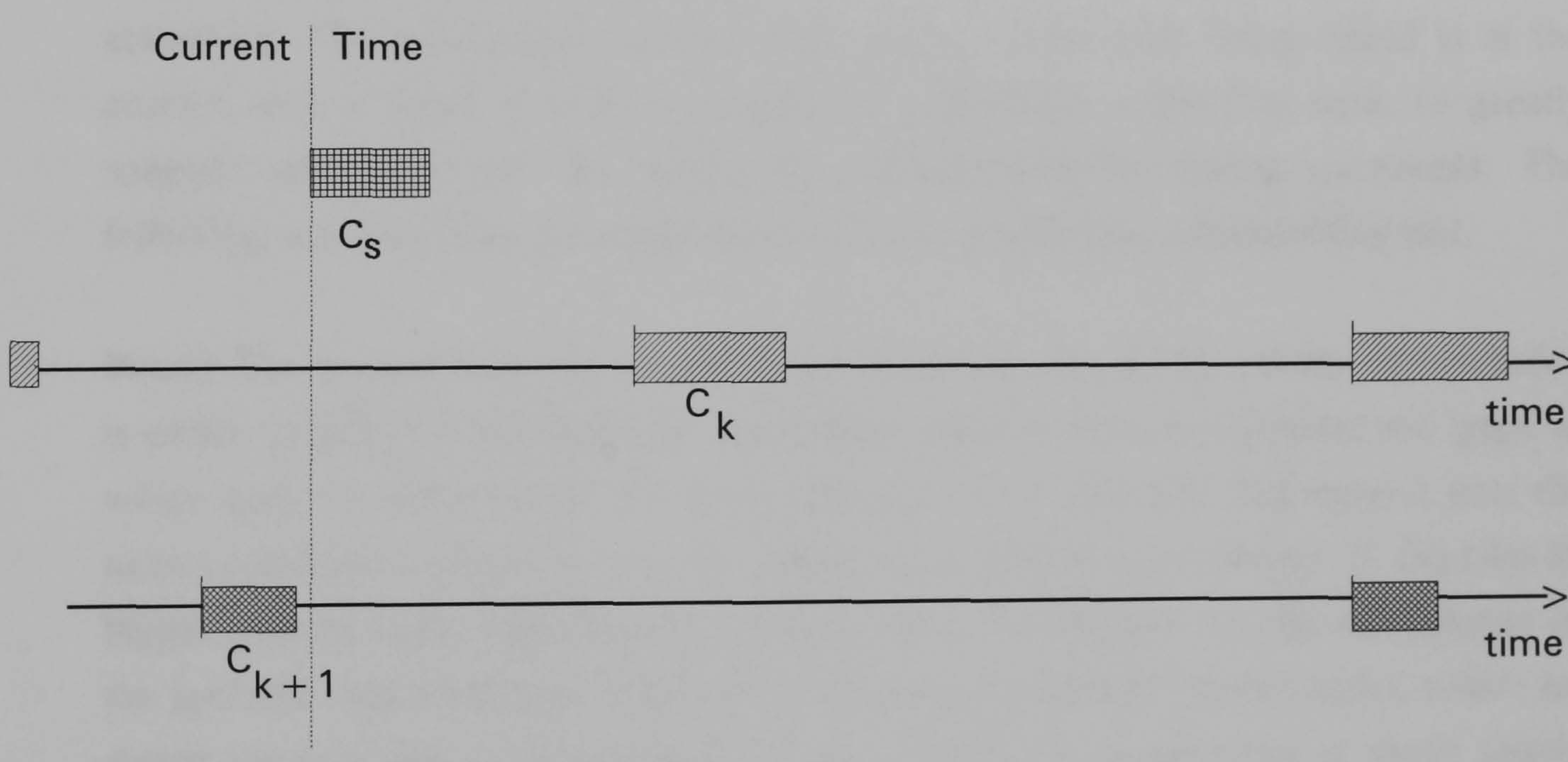


Figure A.2: Computation times (C) for periodic tasks below the sporadic task.

For a task, i, lower than the sporadic task, the interference over all higher tasks, j, is calculated by:

$$I_i = \Sigma_j(\lceil (D_i - O_j) \div T_j \rceil_0 C_j + R_j) \qquad \text{(A.2)}$$

Note that T_S is set to infinity, as are the periods of any other sporadic tasks which are currently in the task list.

The sporadic task, s, is a one-off release, so that the accumulated interference time in a lower task need only be tested against either (a) the *current deadline* of the lower task if it is *active* or (b) the *deadline of the next activation* of the lower task if it is *inactive*. Respective examples from Figure A.2 are *task k+1* and *task k*. The following are more detailed explanations of the different tests for (a) and (b).

(a) If the lower task being schedulability tested is *active* (in other words $R_i > 0$) then test whether $D_i \geq I_i + R_i$.

(b) If the lower task is *inactive*, then the total interference in the lower task's next activation, plus the lower task's WCET, must be tested against its next deadline. A sufficient condition is to suppose that the next activation of the task starts at the *current time*, and to test whether $D_i \geq I_i + C_i$, where D_i is the deadline of the next activation. The supposition that the next release of the task being tested is at the current time is made in order to make use of dynamic scheduling data, to greatly simplify calculation and to thereby reduce schedulability testing overheads. The following is a proof that the supposition provides a sufficient schedulability test.

Proof: The interval between the current time and the actual next release of test task i is either (i) filled by interferences from higher priority tasks or (ii) there are 'gaps' in which task i could execute if it were released. If (i) then this degenerates into the same condition as allowing task i to execute only after its next release. If (ii) then all higher priority tasks, including the sporadic task, are satisfied i.e. the interference of the sporadic task itself, and its knock-on effects on the lower priority tasks, which are above the test task i, have ended. In other words the supposition is never falsely optimistic. Therefore the supposition is a valid basis for a sufficient schedulability test.

A.2 THE PSEUDO-POLYNOMIAL (PP) ALGORITHM

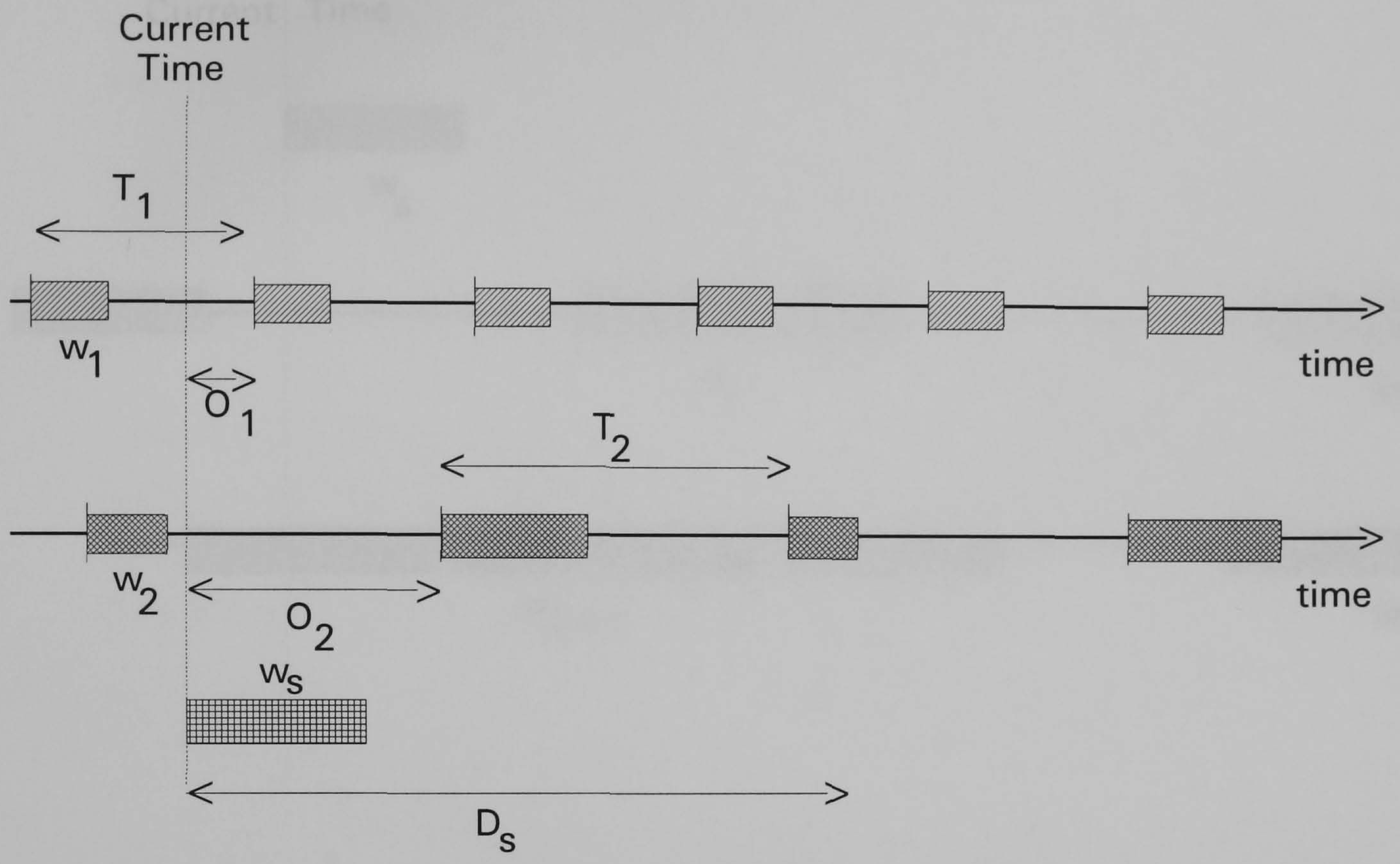


Figure A.3: Response times (w) for periodic tasks above the sporadic task.

The PP algorithm calculates the interference from higher priority tasks *during the elapsed execution time* of the test task. The algorithm therefore generates *response times* (w_i) for each test task, i. Figure A.3 shows the case of the sporadic task itself as the test task. (Incidentally, note that $w_1 = C_1$.) The interference over all tasks, j, above the sporadic task is found by:

$$I_S = \Sigma_j(\lceil (w_S - O_j) \div T_j \rceil_{\mathbf{o}} C_j + R_j) \qquad \text{(A.3)}$$

where:

w_S is the response time of the sporadic

R_j is the current residual execution time of the interfering task

O_j is the offset of the interfering task

T_j is the period of the interfering task

C_j is the WCET of the interfering task

Hence the recursive equation which determines the final value of w_S:

$$w_S^{n+1} = C_S + \Sigma_j(\lceil (w_S^{n} - O_j) \div T_j \rceil_{\mathbf{o}} C_j + R_j) \qquad \text{(A.4)}$$

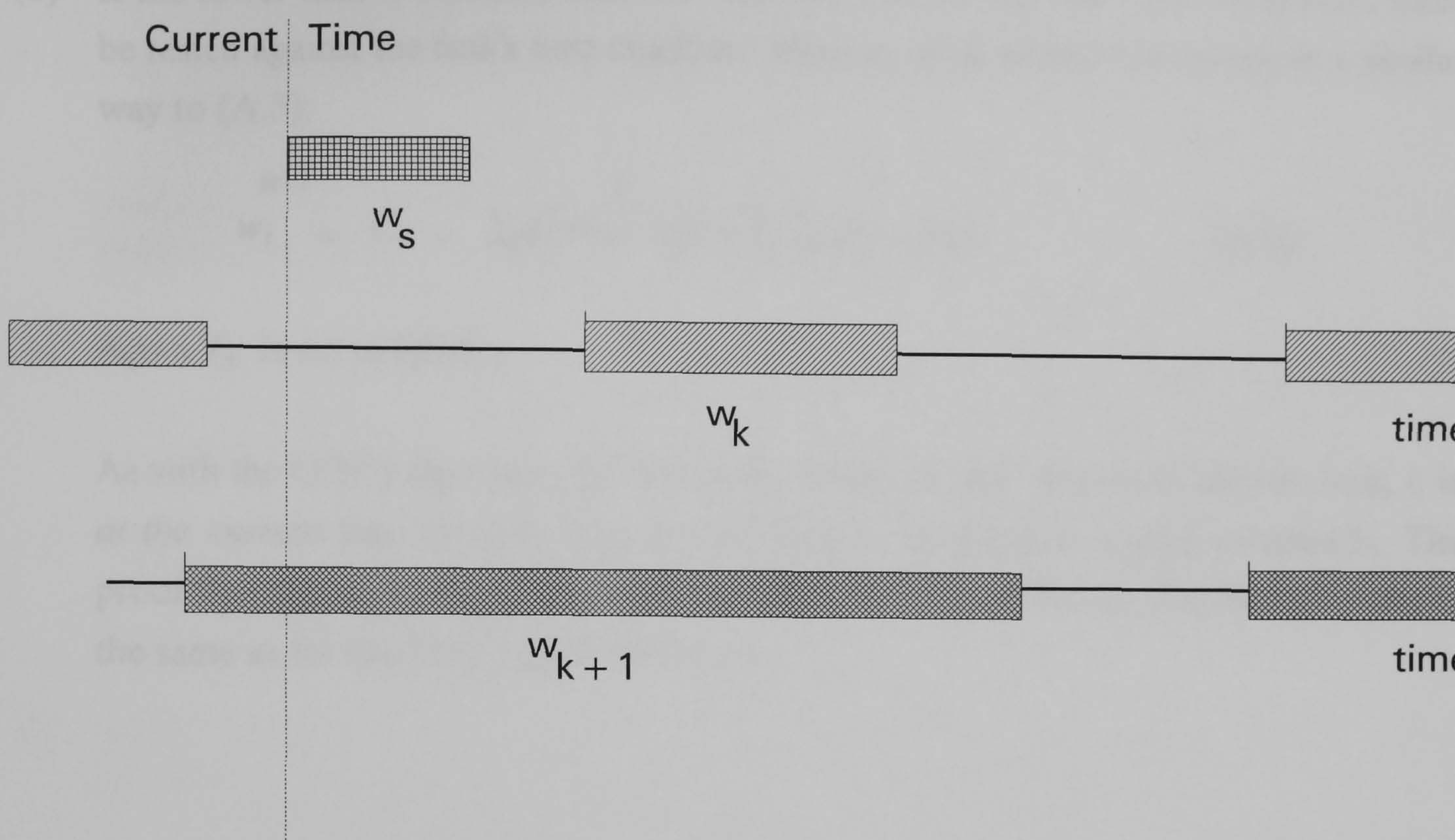


Figure A.4: Response times (w) for periodic tasks below the sporadic task (before the sporadic arrives).

Figure A.4 shows response time for tasks below the sporadic task in priority order. As with the $O(N^2)$ algorithm, the response times of lower tasks need only be tested against either (a) their current deadline if they are *active* or (b) the deadline of their next activation if they are *inactive*. The following are more detailed explanations of the different tests for (a) and (b).

(a) If the lower task being schedulability tested is active (in other words $R_i > 0$) then test whether $D_i \geq w_i$ where w_i is found when the following recursive equation converges:

$$w_i^{n+1} = R_i + \Sigma_j(\lceil (w_i^{n} - O_j) \div T_j \rceil_0 C_j + R_j) \qquad \text{(A.5)}$$

This is the same as (A.4) except that w_i is the response time for the residual computation time of the task, i, being schedulability tested. As before T_s is set to infinity, as are the periods of any other sporadic tasks which are currently in the task list.

(b) If the lower task is inactive, then the response time of the task's next activation must be tested against the task's next deadline. Here w_i is calculated recursively in a similar way to (A.5):

$$w_i^{n+1} = C_i + \Sigma_j(\lceil (w_i^{n} - O_j) \div T_j \rceil_0 C_j + R_j) \qquad \text{(A.6)}$$

Again T_S is set to infinity.

As with the O(N^2) algorithm, the supposition that the next release of the test task, i, is *at the current time* is made in order to reduce schedulability testing overheads. The proof that this supposition provides a valid basis for a sufficient schedulability test is the same as for the O(N^2) algorithm above.

REFERENCES

[1] Ada 95 Reference Manual, Intermetrics, *ANSI / ISO / IEC - 8652 : 1995* (January 1995).

[2] Audsley, N. C., DPhil Thesis, *Computer Science Department, University of York,* (September 1993).

[3] Audsley, N. C., Burns A., Richardson M. F., and Wellings A. J., "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).

[4] Audsley, N. C., Burns A., Richardson M. F., and Wellings A. J., "Incorporating Unbounded Algorithms into Predictable Real-Time Systems", *Computer Systems Science and Engineering, Vol. 8 No 3, pp 80-89* (April 1993).

[5] Burns, A. and Wellings, A, "Concurrency in Ada", *Cambridge University Press,* (1995).

[6] Cheong, I. K., "Scheduling Imprecise Hard Real-Time Jobs with Cumulative Error", *Technical Report UIUCDCS-R-92-1758 (PhD Thesis), Computer Science Department, University of Illinois at Urbana-Champaign* (June 1992).

[7] Chung, J. Y., Liu, J. W. S., Lin, K. J., "Scheduling Periodic Jobs that Allow Imprecise Results", *IEEE Transactions on Computers, Vol. 39, No 9* (September 1990).

[8] Davis R. I., DPhil Thesis, *Computer Science Department, University of York* (July 1995).

[9] Davis R. I., and Burns A., "Optimal Priority Assignment for Aperiodic Tasks with Firm Deadlines, in Fixed Priority Pre-emptive Systems", *Information Processing Letters 53(5),pp 249-254* (March 1995).

[10] Davis R. I., "Guaranteeing X in Y: On-line Acceptance Tests for Hard Aperiodic Tasks Scheduled by the Slack Stealing Algorithm", *YCS 231, Department of Computer Science, University of York,* (June 1994).

[11] Davis R. I., "Scheduling Slack Time in Fixed Priority Pre-emptive Systems", *YCS 216, Department of Computer Science, University of York,* (November 1993).

[12] Davis, R., Punnekkat, S., Audsley, N., Burns, A., "Flexible Scheduling for Adaptive Real-Time Systems", *Proceedings IEEE Real-Time Technology and Applications Symposium, Chicago* (May 1995).

[13] Davis, R., Tindell, K. W., Burns, A., "Scheduling Slack Time in Fixed Priority Pre-emptive Systems", *Proceedings 14th IEEE Real-Time Systems Symposium, Durham, N. Carolina* (December 1993).

[14] DIN, *"DIN 66253 Programmiersprache PEARL, Tiel 1 Basic PEARL", Beuth-Verlag, Berlin-Cologne* (July 1981).

[15] DIN, *"DIN 66253 Programmiersprache PEARL, Tiel 2 Full PEARL", Beuth-Verlag, Berlin-Cologne* (October 1982).

[16] DIN, *"DIN 66253 Programmiersprache PEARL, Tiel 3 Mehrrechner-PEARL", Beuth-Verlag, Berlin-Cologne* (January 1989).

[17] Garvey, A. and Lesser, V., "Representing and Scheduling Satisficing Tasks", *Imprecise and Approximate Computation, Ed. Natarajan, S., Kluwer* (1995).

[18] Gehani, N. H. and Roome, W. D., "Concurrent C", *Summit, NJ : Silicon Press* (1989).

[19] Gehani, N., and Ramamritham, K., "Real-Time Concurrent C: a Language for Programming Dynamic Real-Time Systems", *Journal of Real-time Systems, vol. 3, Kluwer* (1991).

[20] Halang, W. A., "A Proposal for Extensions of PEARL to Facilitate the Formulation of Hard Real-Time Applications", *4. GI/GMR/KfK-Fachtagung Prozessrechner 1984, pp 573-582, Informatik-Fachberichte, Springer-Verlag,* (September 1984).

[21] Halang, W. A. and Stoyenko, A. D., "Extending PEARL for Industrial Real-Time Applications", *IEEE Software* (July 1993).

[22] Halang, W. A. and Stoyenko, A. D., "Constructing Predictable Real-Time Systems," *Kluwer* (September 1991).

[23] Ichbiah, J. D., et al., "Ada Programming Language", *ANSI/MIL-STD 1815A, vol. U.S. Department of Defense* (January 1983).

[24] IEEE Industrial Electronics Society, *Proceedings of the Intelligent Vehicles '92 Symposium* (June 1992).

[25] Kenny K. B., "Structuring Real-Time Systems Using Performance Polymorphism", *Technical Report UIUCDCS-R-90-1641 (PhD Thesis), Computer Science Department, University of Illinois at Urbana-Champaign* (November 1990).

[26] Kenny K. B. and Lin, K. J., "Building Flexible Real-Time Systems Using the Flex Language", *IEEE Computer* (May 1991).

[27] Lehoczky, J. P. and Ramos-Thuel, S, "An Optimal Algorithm for Scheduling Soft-aperiodic Tasks in Fixed-Priority Pre-emptive Systems", *Proceedings 13th IEEE Real-Time Systems Symposium, Tucson, Arizona* (December 1992).

[28] Lehoczky, J. P., and Ramos-Thuel, S., "On-line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems", *Proceedings 14th IEEE Real-Time Systems Symposium, Durham, N. Carolina* (December 1993).

[29] Lehoczky, J. P., "Fixed-Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines", *Proceedings 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, pp 201-209* (December 1990).

[30] Lehoczky, J. P., Sha, L., Strosnider, J. K., "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *Proceedings IEEE Real-Time Systems Symposium, San Jose, California, pp 261-270* (1987).

[31] Leung, J. Y. T. and Whitehead, J., "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", *Performance Evaluation, pp.237-250, Vol. 2, Part 4* (December 1982).

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

[33] Liu, J. W. S., Lin K. J. and Natarajan S., "Scheduling Real-Time, Periodic Jobs Using Imprecise Results", *Proceedings 8th IEEE Real-Time Systems Symposium, Fairmont Hotel, San Hose, California, pp252-260* (December 1987).

[34] Liu, J. W. S., Lin K. J., Shih, W. K., Yu, A. C. S., Chung, J. Y. and Zhao, W., "Algorithms for Scheduling Imprecise Computations", *IEEE Computer, pp58-68* (May 1991).

[35] Locke, C. D., "Best-Effort Decision Making for Real-Time Scheduling", *Technical Report CMU-CS-86-134 (PhD Thesis), Computer Science Department, Carnegie-Mellon University* (May 1986).

[36] Locke, C. D., Vogel D.R., Mesler, T.J., "Building a Predictable Avionics Platform in Ada: A Case Study", *Proceedings 12th IEEE Real-Time Systems Symposium,* (December 1991).

[37] Lovesay, E. J., and Davis, R. I., "Integrating Machine Intelligence into the Cockpit to Aid the Pilot", *Proceedings AGARD Conference on Machine Intelligence for Aerospace Electronic Systems* (1991).

[38] Musliner, D. J., Hendler, J. A., Agrawala, A. K., et al, "The Challenges of Real-Time AI", *IEEE Computer* (January 1995).

[39] Niehaus, D., Ramamritham K., Stankovic, J. A., Wallace, G., Weems, C., Burleson, W., Ko, J., "The Spring Scheduling Co-Processor : Design, Use and Performance", *Proceedings 14th IEEE Real-Time Systems Symposium, Durham, N. Carolina* (December 1993).

[40] Payton, D. W., and Bihari, T. E., "Intelligent Real-Time Control of Robotic Vehicles", *CACM 34(8), pp 48-63* (August 1991).

[41] "Proceedings of the Fourth International Workshop on Real-Time Ada Issues, July 1990", *Ada Letters, vol. X, Number 9, ACM, Pitlochry* (Fall 1990).

[42] Ramamritham, K., "Dynamic Priority Scheduling" *in "Real-Time Systems : Specification Verification and Analysis", pp 66-95, Prentice Hall* (1996).

[43] Sha, L., Lehoczky, J. P., Rajkumar, R., "Solutions for some Practical Problems in Prioritised Preemptive Scheduling", *Proceedings IEEE Real-Time Systems Symposium, pp 181-191* (1986).

[44] Sha, L., Sprunt, B., Lehoczky, J. P., "Aperiodic Task Scheduling for Hard Real-Time Systems", *Real-Time Systems 1(1), pp 27-69* (1989).

[45] Shih, W. K., "Scheduling in Real-Time Systems to Ensure Graceful Degradation: the Imprecise Computation and the Deferred-Deadline Approaches", *Technical Report UIUCDCS-R-92-1765 (PhD Thesis), Computer Science Department, University of Illinois at Urbana-Champaign* (December 1992).

[46] Shih, W. K., Liu, J. W. S., Chung, J. Y., "Algorithms for Scheduling Imprecise Computations with Timing Constraints", *Proceedings 10th IEEE Real-Time Systems Symposium, Santa Monica, California* (December 1989).

[47] Song, X., and Liu, J. W. S., "Performance of Multi-Version Concurrency Control Algorithms in Maintaining Temporal Consistency", *Department of Computer Science, University of Illinois at Urbana-Champaign* (February 1990).

[48] Sprunt, B., "Aperiodic Task Scheduling for Real-Time Systems", *PhD Thesis, Carnegie Mellon University* (1990).

[49] Sprunt, B., Lehoczky, J.P., Sha, L., "Exploiting Unused Periodic Time for Aperiodic Service using the Extended Priority Exchange Algorithm", *Proceedings IEEE Real-Time Systems Symposium, pp 251-258* (December 1988).

[50] Sprunt, B., Sha, L., Lehoczky, J.P., "Aperiodic Task Scheduling for Hard Real-Time Systems", *Journal of Real-Time Systems, 1:27-60* (1989).

[51] Stankovic, J. A., "Misconceptions about Real-Time Computing", *IEEE Computer* (October 1988).

[52] Stankovic, J. A., Ramamritham K., "Overview of the Spring Project", *COINS Technical Report 89-03, University of Massachusetts, Amherst* (January 1989).

[53] Stankovic, J. A., Ramamritham K.,"The Spring Kernel: a New Paradigm for Real-Time Systems", *IEEE Software* (May 1991).

[54] Stankovic, J. A., Ramamritham K.,"The Design of the Spring Kernel", *Proceedings IEEE Real-Time Systems Symposium, San Jose, California, pp146-157* (1987).

[55] Stankovic, J. A., Ramamritham K., Cheng, S., "Evaluation of Flexible Task Scheduling Algorithms for Distributed Hard Real-Time Systems", *IEEE Transactions on Computers, pp 1,130-1,143* (Decmeber 1985).

[56] Stankovic, J. A., Ramamritham K., and Zhao, W. "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", *IEEE Transactions on Computers, pp 1,110-1,123* (August 1989).

[57] Strosnider, J.K., "Highly Responsive Real-Time Token Rings", *PhD Thesis, Carnegie Mellon University* (1988).

[58] Tia T. S., Liu, J. W. S., Mallikarjun, S., "Algorithms and Optimality of Scheduling Aperiodic Requests in Fixed-Priority Pre-emptive Systems", *Department of Computer Science, University of Illinois at Urbana-Champaign* (April 1994).

[59] Tokuda, H., Wendorf, J. W., Wang, H. J., "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems", *Proceedings IEEE Real-Time Systems Symposium, pp271-280* (December 1987).

[60] Wendorf, J. W., "Implementation and Evaluation of a Time-Driven Scheduling Processor", *Proceedings IEEE Real-Time Systems Symposium, pp172-180* (1988).

[61] Werum, W., and Windauer, H., "Introduction to PEARL", *Friedr. Vieweg & Sohn,* (1985).

[62] Yen, J. and Natarajan, S. "A Decision-Theoretic Treatment of Imprecise Computation", *Imprecise and Approximate Computation, Ed. Natarajan, S., Kluwer* (1995).

[63] Zhao, W., Ramamritham, K., Stankovic, J. A. "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", *IEEE Transactions on Software Engineering, Vol. SE-13, No 5* (May 1987).

[64] Zilberstein, S., "Operational Rationality through Compilation of Anytime Algorithms", *PhD Thesis, Department of Computer Science, University of California at Berkeley* (1993).