Remove this page before binding. (The page number was placed before realizing that this was to be the title page.) Also, don’t forget to move the Contents and Figures pages from the end of this stack to before the Acknowlegments page.

On Exploiting Spare Capacity in Hard Real-Time Systems

Robert Ian Davis

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

July 1995

To Mum and Dad

Acknowledgements

I would like to thank my supervisor Alan Burns and my assessor Andy Wellings for providing me with the opportunity to work at the University of York, and for their continuing support and interest in my work. I am grateful to them and to my other colleagues in the Real-Time Research Group for many hours of fruitful discussion and for reviewing the work contained in this thesis. In particular, I would like to thank Neil Audsley for his work in implementing the DRTEE Kernel.

Finally, my thanks go to Annette for looking after me during the long dark hours of writing up.

Declaration

I declare that the research described in this thesis is original work which I undertook between October 1992 and June 1995. Certain parts of this thesis have been published previously as technical reports, conference proceedings and journal papers.

Chapter 2 draws from material previously published in sections 6-8 of *"Fixed Priority Scheduling: An Historical Perspective"* (Real-Time Systems V8 N2/3 March/May 1995)[4] and in *"On Improving the Utility of Hard Real-Time Services in Fixed Priority Pre-emptive Systems"* (Dept. Computer Science, University of York MPhil/DPhil Qualifying Dissertation 1993) [30].

Chapter 3 details work published in the paper *"Scheduling Slack Time in Fixed Priority Pre-emptive Systems"* (Proceedings IEEE Real-Time Systems Symposium, December 1993) [39].

Chapter 4 draws from the technical report *"Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems"* (Dept. Computer Science, University of York, Technical Report YCS217, 1993) [31] - later included in the paper *"Appropriate Mechanisms for the Support of Optional Processing in Hard Real-Time Systems"* (Proceedings IEEE Real-Time Operating Systems and Software, May 1994) [8].

Chapter 5 describes the research previously published in *"Dual Priority scheduling: A Means of Providing Flexibility in Hard Real-Time Systems"* (Dept. Computer Science, University of York, Technical Report YCS230 1994) [34], later versions of which have been submitted to the Real-Time Systems Journal (May 1994) and the Real-Time Systems Symposium (May 1995).

Chapter 6 comprises research previously described in the technical report *"Guaranteeing X in Y: On-line Acceptance Tests for Hard Aperiodic Tasks Scheduled by the Slack Stealing Algorithm"* (Dept. Computer Science, University of York Technical Report YCS231, 1994) [35] and later included in *"Integrating Best-Effort and Fixed Priority Scheduling"* (Proceedings IFIP Workshop on Real-Time Programming, June 1994) [9] and *"Mechanisms for Enhancing the Flexibility and*

Utility of Hard Real-Time Systems" (Proceedings 15th IEEE Real-Time Systems Symposium, December 1994) [15]. Section 6.1 is taken from the paper *"Optimal Priority Assignment for Aperiodic Tasks with Firm Deadlines in Fixed Priority Pre-emptive Systems"* (Information Processing Letters, 10th March 1995) [37].

Chapter 7 is drawn from research first published as the technical report - *"Integrating Best-Effort Policies into Hard Real-Time Systems based on Fixed Priority Pre-emptive Scheduling."* (Dept. Computer Science, University of York Technical Report YCS240, 1994) [33] and subsequently as *"Flexible Scheduling for Adaptable Real-Time Systems"* (Proceedings IEEE Real-Time Technology and Applications Symposium, 15-17th May 1995) [38].

Abstract

Complex real-time systems such as those envisaged for future autonomous vehicle control, robotics and advanced avionics applications, need to exhibit dynamic and adaptive behaviour in order to function in unpredictable environments. These systems need to be resilient to software / hardware failures and imbibed with the property of graceful degradation under overload. Cost, space and weight constraints also dictate that they must make the most effective use of limited processing / communications resources.

To realise such complex systems, two potentially conflicting objectives need to be met: first, safety critical and mission critical services must be guaranteed to provide results of a minimum acceptable quality and reliability by their deadlines. Second, the utility of the system, as determined by the frequency, timeliness, precision and confidence level of the results produced, must be maximised. The research comprising this thesis, focuses on the development of analysis and mechanisms which enable this second objective to be met.

System utility can be enhanced by the timely execution of optional components which refine or improve upon the results of their mandatory counterparts. To achieve this, a three tier strategy is proposed. Initially, algorithms are developed which identify spare capacity at run-time, enabling optional components with soft deadlines to be scheduled responsively. Second, a family of acceptance tests are introduced, which facilitate the on-line guarantee of optional components with firm deadlines. An optimal priority assignment policy is derived for such components. Third, an efficient, adaptive admission policy is developed which arbitrates between competing optional components on the basis of their value-densities, enabling system utility to be maximised.

The effectiveness of these techniques is examined in terms of their coverage, simplicity, performance and overheads, via simulation studies and via a proof of concept implementation within a hard real-time kernel.

Chapter 1

Introduction

Real-time systems are found in many diverse application areas including, engine management, process control, medical electronics, telecommunications, robotics, multi-media and avionics. These systems range in complexity from simple controllers implementing a single function, such as the ignition timing of an automobile engine, to complex sets of communicating sub-systems, each of which is responsible for performing a number of critical functions, such as navigation, engine management and flight control for a civil aircraft.

The defining characteristic of these *real-time* systems, is that they are required to respond to an external stimulus within a finite and specified time [76]. More generally, each real-time service has an associated time constraint by which a response is required. These time constraints vary considerably in their degree of severity. A service is classified as *hard* real-time, if failure to respond within its time constraints constitutes a failure of the system. In contrast, the late response of a *soft* real-time service may be tolerated, albeit at the expense of degraded system utility. Finally, a service is classified as *firm* real-time if its late delivery does not constitute a serious failure, but is of no utility. In this context, the *utility* of a service is defined as its *"usefulness as perceived from the system's operational environment"* [19]. Thus, the utility of a real-time service depends upon, the precision of the results it produces and the time at which they are delivered, (as well as other factors defining the state of the system and its external environment).

Today, hard real-time systems are often specified and designed to provide deterministic results at times which may also be pre-determined. Whilst effective in the development of small systems, this quest for predictability via determinism can lead to inefficient use of processing resources [21]; systems which lack the flexibility to provide improved performance when they are operating under favourable conditions, and which fail completely when asked to operate marginally outside their original requirements. Further, these requirements often make assumptions about the

predictability of the system's operational environment which if invalidated lead to catastrophic failure.

As an example of such drawbacks, consider a hypothetical air traffic control (ATC) system. The requirements for this system are as follows: process messages from up to 25 aircraft per second, display position, velocity and id, and give 10 seconds warning of any potential incursions into restricted airspace.

Taking a classical approach to real-time systems design and implementation, these requirements would typically result in a system comprising three tasks. The first task processes up to 25 messages. The second task predicts the precise flight path of each aircraft, determining if there are any conflicts with restricted airspace. Finally the third task displays the relevant data and warning information. The tasks are executed cyclically every second.

Whilst adequate, this approach is inefficient: when an aircraft is a long way from restricted airspace, predicting its flight path is not necessary, a simple check based on speed and distance will suffice. Further, a similar calculation based on proximity and maximum speed can be used to determine the earliest time at which a precise check is needed. This approach greatly reduces the system's processing requirements when not all aircraft are close to airspace boundaries.

The traditional approach also suffers from catastrophic degradation: increasing the number of aircraft above that specified always results in a lack of display or warning information for the additional aircraft. Further, with a reduced load, system utility is no better than in the worst case. The same quality of information is provided even though there may be sufficient processing capacity to extend the flight path check to 20 or 30 seconds, giving an advanced advisory warning to pilots.

Adopting an alternative strategy for requirement specification as advocated by Burns and McDermid in [21], the adaptability of this simple system can be markedly improved.

The fundamental requirements for this alternative ATC system are as before: the system must be guaranteed to provide precise warnings for up to 25 aircraft. However, these basic requirements are now supplemented by a set of requirements

for adaptability. The system must do the best it can to (in order of importance): (1) process and display data for all aircraft. (2) provide imprecise warnings based on aircraft speed and proximity to restricted zones. (3) give precise warnings of any potential incursions within 10 seconds. (4) provide advisory warnings of potential incursions 10-30 seconds ahead. Finally, the system must indicate if precise warning information is available for every aircraft.

The intended behaviour of this alternative ATC system resembles that of an intelligent human operator. Under conditions of light load, an advisory warning is generated 20 seconds in advance of an aircraft venturing into restricted airspace, allowing pilots more time to react and enhancing the utility of the service. Under normal conditions, more than 25 aircraft can be handled as not all of them require a complex flight path check due to their distance from airspace boundaries. Further, when resources are reduced, akin to a human operator taking over 2 control stations, an imprecise warning is given based on proximity, providing a degraded but still useful service.

We note that compared to the original, this alternative ATC system provides a higher level of utility under normal conditions and gracefully degraded performance as it is overloaded. However, to facilitate this behaviour, requires a more complex design, where functionality is decomposed into *mandatory* and *optional* components. Mandatory components provide a minimally acceptable level of performance and need to be guaranteed to meet their deadlines. Optional components represent methods of enhancing system utility by improving upon the precision or coverage of their mandatory counterparts or by providing complementary functionality. The scheduling mechanisms and policies needed to support this model must be predictable, enabling mandatory components to be guaranteed, and yet flexible enough to maximise the utility of the system, via the timely execution of optional components.

The research described in this thesis focuses on the mechanisms, policies and analysis required to provide this degree of flexibility and predictability in the next generation of real-time systems.

1.1 Background

The fundamental requirement for hard real-time systems is *predictability*. It must be possible to guarantee, within some failure model, that hard real-time services will meet their requirements in terms of both functionality and timeliness. Further, this need for predictability extends to each component of a hard real-time system: *software*, *hardware*, and the operating system *kernel*. The need for predictability does not however imply that such systems must be *deterministic*. It is sufficient to be able to predict that time constraints will be met, without necessarily being able to determine the exact time at which services will be delivered.

The software implementation of a hard real-time system typically comprises a set of processes or *tasks*. Tasks are classified in terms of their arrival pattern as *periodic*, *aperiodic* or *sporadic*. Periodic tasks are invoked cyclically, with a fixed time interval or *period* between invocations. In contrast, aperiodic tasks may be invoked at any time. Finally, sporadic tasks may also arrive at any time subject to there being a minimum *inter-arrival* time between successive invocations. Tasks may be *independent* or they may exhibit *inter-dependencies* such as precedence relationships, or requirements to access shared data or semaphores. As part of the design process, tasks are assigned *deadlines* derived from the end-to-end time constraints on the services which they implement [47].

Tasks with hard deadlines must have bounded *worst case execution times* (wcet) in order to exhibit predictable behaviour. This implies that software execution paths must not result in indefinite loops or unbounded recursion. Indeed, to enable the worst case execution time of hard tasks to be determined, it is necessary to limit the set of language constructs which may be used to those which can be analysed. Typically, the use of pointers to functions and unstructured programming (e.g. gotos) is disallowed and maximum iteration counts or times are required for all loop constructs [81, 112]. Hardware must support bounded execution times for all machine code instructions and memory accesses. For a single instruction, these bounds are often significantly higher than the average case, due to the effects of instruction pipelining and memory cacheing. However, tight bounds can be obtained for the worst case execution time of sequences of instructions on processors with pipelines [113] and caches [3]. Kernel mechanisms and policies must not lead to unbounded

delays in the execution of hard tasks. For example, the FIFO rendezvous queueing implemented in Ada83 lead to problems with unbounded delays due to priority inversion [20]. Further, if the application comprises a mixture of safety critical and non-safety critical tasks, then the kernel must also ensure that one application service cannot affect the timely and correct operation of others [24]. In particular, fault containment is required to prevent an errant task from: corrupting memory used by other (independent) tasks, overrunning its execution time budget or permanently holding software resources such as semaphores. Budget timing, memory protection and system calls are typically required to support this functionality.

The inherent concurrency of real-time services leads to contention between tasks for both physical resources (e.g. processors, communications etc.) and logical resources (e.g. semaphores). Kernel mechanisms such as task dispatch and budget timing provide a means of implementing higher level policies and protocols which arbitrate between competing demands for resources. In particular, *scheduling* policies determine, at any given time, which task is dispatched to the processor for execution.

Scheduling policies are termed *pre-emptive* if the execution of a task may be interrupted at any time. In reality, however, there are often non-pre-emptable sections within tasks, referred to as critical sections, where pre-emption could lead to problems, for example, with semaphores being left in an incorrect state. Further, some kernel operations (or system calls) are typically also non-pre-emptable.

Scheduling policies are classified as *off-line* or *on-line* depending on when scheduling decisions are made. With off-line policies (e.g. static cyclic scheduling), a schedule is pre-computed, then at run-time the decision, as to which task to execute is made by referring to this *static* schedule. In contrast, on-line policies use task *priority* to arbitrate between tasks which are ready to execute. On-line policies may be further sub-divided into *dynamic* and *fixed* priority methods. With fixed priority scheduling, priorities are statically determined by some *priority assignment policy*, such as Rate Monotonic [65]. In contrast, dynamic priorities correspond to task attributes which vary at run-time, e.g. absolute deadline in the case of Earliest Deadline First [65].

The need for predictability implies that the scheduling policies used in hard real-time systems must be *analysable*. It must be possible to determine *a priori*, by means of a *feasibility test*, if a set of tasks complying with a given computational model (e.g. bounded wcets, independent periodic tasks etc.) can be guaranteed to always meet their deadlines when scheduled according to the prescribed policy. Analysis may also be applied at run-time in the form of an on-line *acceptance test*; used to determine if a particular invocation of an aperiodic task is schedulable.

Finally, scheduling policies are required to be *efficient*: making efficient use of limited processing resources invariably reduces the cost, complexity, size and power consumption of a system. In almost all applications, the successful and / or profitable deployment of a system depends on one or more of these factors.

1.2 Motivation

It is debatable whether many of the simple real-time systems in operation today would benefit from improvements in adaptability and flexibility. Often enough control can be exerted over the operational environment to ensure that the real-time system is not asked to operate outside its deterministic requirements. In the case of air traffic control, this is achieved by diverting or stacking aircraft wishing to enter a sector, causing delays, but avoiding overloading the system. In emergency situations, it is the human operators and pilots who provide the necessary adaptability. Whilst this approach is adequate for today's systems, it is essential that the next generation of real-time systems exhibit dynamic, adaptive and intelligent behaviour if they are to function in environments which are inherently non-deterministic. For these systems [94],

> *"the typical semantics (all tasks make their deadlines 100% of the time) associated with small static real-time systems are not sufficient."*

However, critical real-time services must still be guaranteed [23].

1.2.1 Next Generation Real-Time Systems

The Pilot's Associate programme [67], is one early (c. 1986) example of an initiative to build such a "next generation" system. The aim of this programme is to enhance pilot effectiveness through increasing situational awareness and decreasing workload. To achieve this requires an intelligent advisory system which adapts to the current external environment, providing timely advice to the pilot. The functionality of this system is highly complex, with a mixture of safety and non-safety critical services, time constraints which vary from a few milli-seconds to tens of seconds, a highly dynamic task set, numerous modes of operation and extensive AI and unbounded components. A whole spectrum of real-time services are envisaged, including: managing agile sensors, fusing data from a variety of active and passive sources, combining this fused picture of the environment with database information; determining and prioritising threats due to enemy aircraft and surface to air missile systems, mission planning and re-routeing, reacting to immediate threats such as missile attack, determining countermeasures and evasion tactics and planning attack profiles and engagments, as well as more mundane functions such as navigation, fuel management and systems monitoring. The operational environment of this system is hostile, unpredictable and completely uncontrollable. The location, performance, tactics and numbers of enemy systems are unknown quantities.

Many similar systems have been proposed in areas such as autonomous vehicle control [78], robotics [77] and advanced avionics [71], although few have progressed past the prototype stage. This is due, at least in part, to the difficulties arising in providing intelligent and adaptive behaviour whilst also guaranteeing a minimum acceptable level of performance in the worst case. Indeed, in [94] Stankovic notes that,

> "*one of the most difficult aspects will be in demonstrating that these systems meet both their overall performance requirements (which are generally average case statistics but with respect to meeting deadlines and maximising the value of executed tasks), as well as specific deadline and timing requirements.*"

Historically, intelligent real-time systems have been the province of Best-Effort scheduling [51, 73]. Although providing the necessary flexibility, this approach can at best only provide probabilistic guarantees that deadlines will be met. Alternatively, fixed priority pre-emptive scheduling facilitates *a priori* guarantees that crucial application services will meet their deadlines. However, it is debatable whether it has the necessary flexibility to support dynamic and adaptive behaviour.

In this thesis, we contend that by augmenting fixed priority pre-emptive scheduling with simple on-line algorithms, acceptance tests and policies, integration with more flexible scheduling approaches can be achieved. Thus combining the benefits of guaranteeing hard time constraints with the adaptability which can be achieved via the Best-Effort paradigm.

Complex real-time applications place heavy demands on the operating system kernel and underlying scheduling theory. These demands can be expressed in terms of objectives which need to be addressed by the architects of the next generation of real-time systems. Two fundamental objectives are as follows:

1. Safety critical and mission critical services must be guaranteed (100%) to provide results of the minimum acceptable quality and reliability by their deadlines [13, 94].

2. The utility of the system, as determined by the frequency, timeliness, precision and confidence level of the results produced, should be maximised [23, 94].

Recent advances in scheduling theory have removed many of the artificial constraints which were previously needed in order to guarantee hard timing requirements. In fact, fixed priority pre-emptive scheduling is now becoming accepted as an appropriate engineering discipline for constructing hard real-time systems. However, the sole reliance placed upon off-line feasibility analysis makes pure fixed priority pre-emptive scheduling inflexible.

The second objective has been identified as one of the key challenges presented by the next generation of hard real-time systems [23]. The avenue of research explored in this thesis seeks to meet this objective by providing techniques which facilitate the timely and effective scheduling of optional components aimed at

enhancing system utility. The motivation for this line of research stems from the need to:

- improve upon the level of utility provided by analysable mandatory components.
- incorporate intelligent, dynamic and adaptive behaviour into hard real-time systems.
- provide resilience to software failures, increase application robustness and imbibe systems with the property of graceful degradation.

1.2.2 Techniques for Improving Utility

Several techniques have been proposed which exploit spare processing capacity, enabling the utility of hard real-time services to be improved. In some cases, this can be achieved by executing mandatory components at a raised priority, enabling their results to be delivered earlier. Alternatively, utility may be enhanced via the execution of optional components which improve upon the precision, reliability or confidence level of their mandatory counterparts. Paradigms which support this optional component model include; Iterative Refinement [64], Multiple Versions [94] and Approximate Processing [45].

Components which exhibit intelligent behaviour often have effectively unbounded worst case execution times. Clearly, such components cannot be guaranteed to complete by any given deadline, however, they can be integrated into hard real-time systems via the Iterative Refinement or Multiple Versions paradigms.

Iterative Refinement can be subdivided into Milestone Methods and Sieve Functions. The Milestone Method facilitates the refinement of an initial minimally acceptable result, whilst Sieve Functions refine intermediate values, improving the utility of the final output. In both cases, it is not essential that the refinement steps are completed thus unbounded techniques are permitted. Both these methods are applicable to many hard real-time services which involve numerical computation, statistical estimation and prediction, heuristic search, sorting and database query processing [66].

The Multiple Versions paradigm is a more general technique than Iterative Refinement. This approach assumes that there are a number of different algorithms available to provide essentially the same service. Typically, these algorithms make different trade-offs between result utility and execution time. Often they also emphasise different aspects of the problem. Versions with unbounded execution times can be incorporated, provided that they are supported by a bounded primary version, which is guaranteed off-line. Similarly, alternative versions may have worst case execution times which are too large to be guaranteed *a priori*, but may be given an on-line guarantee if sufficient spare capacity is available.

Examples of multiple versions can be found in the Situation Assessment (SA) subsystem of the Pilot's Associate [57]. (see section 1.2.1). In this application, the method used to answer queries regarding the status of enemy fighters is dependent upon the execution time budget available. This in turn is dependent upon the deadline imposed by the external environment: the time to intercept. *"Given a very short deadline, SA just looks up the value most recently stored in its database...With more time, SA executes algorithms which quickly extrapolate previous state data to estimate the current state. With still more time, SA uses a rule based system..."*

Similarly, the Approximate Processing paradigm exploits spare capacity to improve service utility. With this method, it is assumed that the worst case execution time of a task can be expressed as a function of a set of parameters. The idea is to set these parameters, such that a result of the maximum utility is guaranteed to be produced before the task's deadline. Thus the worst case execution time of the task is tailored to the time budget available. To achieve this, the Approximate Processing approach requires a model of how the worst case execution time of the task varies with certain controlling parameters.

A simple example of Approximate Processing can be found in the domain of route planning algorithms [50]. Here, the step size used in the search is varied to control execution time. In the case of more complex tasks, execution time may depend upon a number of parameters. For example, in the Distributed Vehicle Monitoring Test-bed [40], execution time depends upon; the level of approximation in input data clustering, the search depth and the number of knowledge sources used.

Techniques such as Iterative Refinement, Multiple Versions and Approximate Processing can be used in the implementation of adaptive components. Such components may change their timing requirements at run time within some pre-determined bounds, and have deadlines which may vary from one invocation to the next, depending upon the state of the external environment. Moreover, adaptive components may determine their next release time based on the current value and maximum rate of change, of some measured quantity, thus exhibiting a variable frequency of execution. Off-line analysis of such components must assume that they are invoked at the maximum frequency with the minimum deadline. At run-time, such conditions may seldom occur. In order to fully exploit the ability of adaptive components to modify their execution requirements and hence the quality of the result they produce, spare processing capacity must be made available prior to their deadlines. Further, an on-line acceptance test is required to guarantee the additional execution time.

Finally, resilience to software failures can be provided through time redundancy. If a software component fails the "sanity" checks on its output, then there may be sufficient spare capacity to retry the component before its deadline. Alternatively, if little spare capacity is available, there may still be time to execute a "quick and dirty" alternative which, although degrading system performance, prevents or reduces the damage caused by the failure. Again, on-line mechanisms for identifying and assigning spare capacity are required to support such behaviour.

1.3 Objectives and Approach

The research comprising this thesis is aimed at meeting the following objective for hard real-time systems: to maximise system utility, as defined by the frequency, timeliness, precision and confidence level of the results produced, whilst also ensuring that the *a priori* guarantees, made regarding the time constraints of crucial services, are honoured at run-time.

The central proposition of the thesis is that:

On-line techniques, which exploit knowledge of the run-time state of the system, can be used to integrate adaptive scheduling policies into the framework provided by fixed priority pre-emptive scheduling. Thus combining the benefits of guaranteed predictability with the flexibility necessary to attain improved system utility.

The rationale behind this idea is that in order to provide *a priori* guarantees to services with hard deadlines, off-line feasibility analysis must take account of the worst-case phasings, execution times and arrival rates of the mandatory tasks which comprise these services. In general, at run-time, such worst-case conditions seldom occur, thus spare capacity becomes available. By exploiting run-time information, about task releases, execution times and deadlines, this spare capacity can be identified and assigned to optional utility-enhancing tasks.

In this thesis, we propose a three tier approach to maximising system utility. The elements of this strategy are as follows:

1. Algorithms which identify spare capacity.

2. Acceptance tests which provide on-line guarantees to utility-enhancing tasks.

3. Admission policies which arbitrate between competing optional components.

Each layer exploits additional run-time information, allowing a greater range of utility-improving techniques to be employed and thus increasing the level of system utility which can be obtained.

1.3.1 Methodology

The methodology used in undertaking the research comprising this thesis is as follows:

1. The theoretical derivation of mechanisms, algorithms and policies corresponding to the three tiers of the strategy.

2. Evaluation of the theoretical performance of the above techniques through comparison with contemporary approaches via simulation studies.

3. Measurement of the actual performance and overheads of the various techniques as implemented in the DRTEE hard real-time system kernel.

Although in general, work progressed in the above sequence, at various points, anomalies or deficiencies highlighted by simulation or measurement provided a focus for further theoretical work. In particular, the difficultly in efficiently detecting spare capacity present due to late sporadic arrivals, highlighted in chapter 4, led to the development of the Dual Priority approach (chapter 5). Similarly, anomalies in the performance of the acceptance tests detailed in chapter 6, led to the derivation of an optimal priority assignment policy for aperiodic tasks (also presented in chapter 6). Finally, before the various techniques could be implemented in the DRTEE kernel, the underlying scheduling theory had to be tailored to reflect the particular performance characteristics of the kernel.

The final stage of our research method involved reflecting upon the degree of success in meeting the objectives set. In particular, the techniques proposed are reviewed against the following criteria:

Coverage: techniques which are only applicable to tasks sets which conform to a restrictive computational model are unlikely to be of practical use in the next generation of hard real-time systems.

Performance: the techniques proposed will only be taken up if they out-perform existing methods.

Overheads: the overheads involved in implementing a particular technique can have a significant impact on the actual performance achieved. Typically for techniques to be practical in real systems, the overheads need to be of $O(n)$ or ideally $O(1)$ complexity in the number of tasks.

Simplicity: simple intuitive methods are more likely to be taken up by industry than those with complex implementation requirements.

1.4 Research Scope

This thesis focuses on fixed priority pre-emptive scheduling as an underlying strategy. The rationale for restricting our attention to this form of scheduling is as follows: we are concerned with improving the utility of real-time systems; however, services with hard time constraints must still be guaranteed. This requirement for predictability implies that only analysable and efficient scheduling strategies need be considered. Essentially, the choice is between static cyclic, dynamic and fixed priority scheduling.

Static cyclic scheduling does not offer the flexibility required in the next generation of real-time systems. It has a number of major drawbacks when applied to complex systems [69]. The schedules produced are brittle: changing the timing characteristics of a task or adding a new task often requires that the entire schedule is re-worked. Handling tasks with a wide range of frequencies and worst case execution times is problematic: low frequency tasks have to be split up or carefully analysed to determine the time segments when they require certain resources. If this is not done, then shared resources such as data buffers or communications channels need to be reserved for the entire span of execution of the task, making them unavailable for higher frequency tasks. Finally, static cyclic scheduling is unable to deal efficiently with sporadic tasks. These tasks cannot be handled explicitly and must be accommodated by the provision of a periodic task which polls for the sporadic stimulus. If timely behaviour is to be guaranteed, then the period of the polling task must be at most the relative deadline of the sporadic service. This is particularly inefficient if the sporadic service has a deadline which is short with respect to its minimum inter-arrival time.

Dynamic priority scheduling offers the flexibility required by the next generation of real-time systems, but not the predictability. Dynamic priority scheduling is not analysable if priorities can essentially vary at random, as is the case when priority corresponds to dynamic utility values. However, for simpler attributes such as absolute deadline or laxity, analysis is possible, albeit for a constrained computational model, such as that given by Liu and Layland [65]. Unfortunately, exact analysis of dynamic priority strategies such as Earliest Deadline First appears to be intractable for more complex task models. For example, once the deadlines of sporadic tasks are permitted to be less than their periods, feasibility tests based on

utilisation become either insufficient, or if utilisation is considered as worst case execution time divided by deadline, then they are highly pessimistic. Finally, with dynamic priority scheduling, it is not in general possible to determine which tasks will miss their deadlines under conditions of transient overload.

In contrast, advances in fixed priority pre-emptive scheduling theory permit a more complex model for hard tasks and more flexible scheduling of soft tasks. In particular, hard tasks are permitted to be periodic or sporadic, to have arbitrary deadlines, which may be prior to completion, and to exhibit blocking and release jitter. Further, the response time of soft tasks may be improved by utilising spare capacity which becomes available when hard tasks require less than their worst case execution time. These advances are reviewed in the next chapter. Finally, fixed priority pre-emptive scheduling is now becoming accepted as an appropriate engineering discipline for the construction of hard real-time systems. It forms part of the Ada 95 [2] and POSIX [111] standards and has also been specified by the European Research Agency for its research contracts [17].

Note that while the focus of this research is upon fixed priority pre-emptive scheduling, where there is related work in other areas, this is also reviewed.

1.5 Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2 surveys the field of fixed priority pre-emptive scheduling; covering areas such as feasibility tests, priority assignment policies, task synchronisation and blocking, system wide application requirements and efforts to tailor analysis to the characteristics of operating system kernels. We then provide an in depth review of algorithms for identifying spare capacity. Finally, problems with two on-line acceptance tests for the static Slack Stealing algorithm [83, 84] are highlighted. Both of these acceptance tests are shown to be insufficient, that is they may give an on-line guarantee to an aperiodic task which will subsequently miss its deadline.

In chapter 3, we introduce an exact approach to identifying spare capacity. Analysis is presented which determines, at any given time, the maximum amount of interference (refered to as slack) which each hard task may be subject to without

causing its deadline to be missed. This analysis is used to formulate a dynamic Slack Stealing algorithm which we prove to be optimal. Analysis of the dynamic Slack Stealing algorithm is extended to tasks which exhibit blocking and release jitter and to tasks with arbitrary deadlines and offsets. Further, the algorithm is augmented to reclaim gain time (produced when a guaranteed task requires less than its worst case execution time at run-time), unused blocking time and unused context switch time.

Unfortunately, the run-time overheads of the exact algorithm are pseudo-polynomial in complexity, making it impractical for many real-time systems. In chapter 4, we address this problem by introducing approximate analysis which determines a lower bound on the slack available at any given time. This analysis is used to formulate various approximate slack stealing algorithms. The effectiveness of each of these algorithms is evaluated with respect to the exact approach and contemporary methods of identifying spare capacity such as the Extended Priority Exchange algorithm [92]. The mean response time of a stream of soft tasks is used as the performance metric.

Chapter 5 introduces an elegant alternative approach for identifying spare capacity: Dual Priority Scheduling. We show that this approach retains the predictability afforded to hard tasks by fixed priority scheduling. Further, we extend its applicability to tasks which exhibit blocking and release jitter and tasks which have arbitrary deadlines and offsets. This approach is also augmented to reclaim gain time. The effectiveness of the Dual Priority approach is evaluated in terms of its ability to provide responsive soft task scheduling.

Chapter 6 summarises previous work on acceptance tests, and reviews an exact test for the static Slack Stealing algorithm [96]. This test has pseudo-polynomial time and space complexity and is only applicable to hard task sets which are strictly periodic. Subsequently, we introduce more generally applicable exact and sufficient acceptance tests for use under both Slack Stealing and Dual Priority approaches. Further, we derive an optimal policy for assigning fixed priorities to aperiodic tasks which are guaranteed under the Slack Stealing algorithm.

The effectiveness of the sufficient and the exact acceptance tests are compared for Slack Stealing, Dual Priority and Background scheduling. The performance criteria used is the percentage of the total aperiodic task execution time which can be guaranteed.

Chapter 7 examines admission policies. Such policies arbitrate between competing utility-enhancing optional components, thus determining which optional components are executed and which are rejected. First, related work is reviewed. This work addresses the problem of obtaining the maximum total value or utility under conditions of overload. However, it is assumed that there are no hard tasks which must be guaranteed *a priori*.

Subsequently, we show that when optional task execution is interspersed with the execution of guaranteed hard tasks, then the value which any on-line admission policy can guarantee to obtain, is vanishingly small compared to that which can be obtained by a clairvoyant algorithm. We also investigate the average case effectiveness of various policies. First Come First Served (FCFS) and Best-Effort [68] policies are studied, together with a new Adaptive Threshold policy which we derive. These policies are combined with Slack Stealing, Dual Priority and Background scheduling methods. Further, each approach relies on using the sufficient acceptance tests (given in chapter 6) to reject or accept optional tasks. The performance of the various approaches is evaluated in terms of the fraction of an upper bound on the achievable system utility which is obtained.

Chapter 8 details the implementation of the techniques, described in chapters 4 and 5, within the framework of the DRTEE hard real-time kernel. The actual performance and overheads of Slack Stealing, Dual Priority and Background scheduling strategies are measured, for a variety of synthetic task sets.

In chapter 9, we re-examine the basic objective of the avenue of research pursued in the thesis. We comment on how well the techniques developed in previous chapters meet this objective. Finally, we conclude by outlining areas for future research.

Chapter 2

Fixed Priority Pre-emptive Scheduling: A Review

The previous chapter cited two fundamental objectives which must be met by the infrastructure and analysis supporting the next generation of real-time systems.

1. Hard real-time services must be guaranteed to meet their deadlines.

2. System utility, as defined by the frequency, precision, timeliness and reliability of the services provided, should be maximised.

In this chapter, we survey advances in fixed priority pre-emptive scheduling theory. In particular, we look at how the artificial constraints on task timing characteristics and interactions required by early work have been relaxed, allowing tasks to have more complex timing behaviour. Further, we outline how analysis has been tailored to account for the particular characteristics of actual real-time kernels and communications media. Hence we show that fixed priority pre-emptive scheduling provides an appropriate framework for the design and implementation of systems which meet the first objective.

The second objective has been identified as one of the key challenges presented by the next generation of real-time systems [23]. In the introduction, we discussed several techniques which enable the utility of a guaranteed hard real-time service to be improved by exploiting spare capacity. The facilities required to support these techniques include algorithms which identify spare capacity, making it available at a high priority level, and on-line acceptance tests which provide run-time guarantees for optional components with firm or hard deadlines. Sections 2.2 and 2.3 provide an in depth review of previous research into such algorithms and acceptance tests.

2.1 Advances in Fixed Priority Pre-emptive Scheduling Theory

To aid discussion, we first introduce a suitably general computational model and notation for tasks with *hard* deadlines.

2.1.1 Computational Model, Definitions and Notation

In fixed priority pre-emptive scheduling, each task is assigned a unique priority, then at run-time, the processor is allocated to the highest priority runnable task. Each task is assumed to have a base priority i, in the range 1 to n where n is the number of tasks. We use $hp(i)$ to denote the set of tasks with a higher base priority than i. Tasks may be *sporadic* or *periodic*. Each sporadic task gives rise to an infinite sequence of invocation requests, separated by a minimum inter-arrival time T_i. Similarly, each periodic task gives rise to an infinite sequence of invocation requests, separated by a period T_i. Thus periodic tasks may be viewed as a special case of sporadics. Each invocation of task τ_i performs an amount of computation between 0 and C_i (its bounded worst case execution time) and has a hard deadline D_i measured relative to the time of the request. Further, each task may lock and unlock semaphores according to the Priority Ceiling Protocol [88]. The worst case blocking time which an invocation of task τ_i can experience due to the operation of this protocol is denoted by B_i. Although the tasks are assigned unique static priorities, they may have their priorities temporarily increased due to priority inheritance, as part of the operation of the Priority Ceiling Protocol. Finally, tasks can arrive at any time after their minimum inter-arrival interval, but be delayed for a variable but bounded amount of time (termed *release jitter* [105], J_i) before being placed on a notional run-queue. They are then said to be released. It is assumed that for each task τ_i, C_i, T_i, D_i, B_i and J_i are known deterministic quantities.

Research into fixed priority scheduling often focuses on optimal priority assignment policies and schedulability tests. A fixed priority assignment policy P is *optimal* if no task set exists which is schedulable when priorities are assigned according to a different priority assignment policy, but not schedulable when priorities are assigned according to policy P. Schedulability tests determine if a task set is schedulable under a given fixed priority assignment. A schedulability test is *sufficient* if all task sets which pass the test are indeed schedulable. A test is

necessary if all task sets which fail the test are in fact unschedulable. Tests which are both sufficient and necessary are referred to as *exact*.

2.1.2 Early Work

Early work on fixed priority pre-emptive scheduling focused on priority assignment policies. In 1967, Fineberg and Serlin considered priority assignment for two tasks [44]. They noted that if priorities are assigned such that the task with the shorter period has the higher priority, then the least upper bound on schedulable *utilisation* is $2(\sqrt{2}-1)$ or 82.8%. Where, the utilisation U is defined as:

$$U = \sum_{\forall i} \frac{C_i}{T_i} \tag{2.1}$$

This result was later generalised by Serlin [86], and Liu and Layland [65]. Both of whom showed that Rate Monotonic priority assignment is an optimal fixed priority assignment policy for tasks which comply to the restrictive computational model detailed below. According to the Rate Monotonic policy, tasks are assigned priorities based on their period, the shorter the period, the higher the priority.

Both Serlin, and Liu and Layland also gave a simple sufficient schedulability test, which is only applicable when priorities are assigned according to the Rate Monotonic policy. This test is based on processor utilisation and is reproduced below. Provided that processor utilisation is less than the bound, then the task set is schedulable.

$$\sum_{\forall i} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right) \tag{2.2}$$

This bound corresponds to a utilisation of 82.8% for 2 tasks, 71.8% for 10 tasks and tends to $ln(2)$ = 69.3% for a large number of tasks. Unfortunately, the computational model used limits the applicability of these results to simple task sets which conform to the following artificial constraints:

1. All tasks on a single processor.

2. A fixed set of tasks.

3. All tasks are periodic.

4. All task execution times are fixed.

5. All task deadlines are equal to their periods.

6. At some point in time, all the tasks are released simultaneously.

7. All tasks are independent.

8. Tasks may not voluntarily suspend themselves.

9. There are no scheduling overheads.

Subsequent research into fixed priority pre-emptive scheduling has focused on lifting these restrictions, and thus providing optimal priority assignment policies and exact schedulability tests for more realistic computational models.

2.1.3 Optimal Priority Assignment Policies

In 1982, Leung and Whitehead showed that Rate Monotonic priority assignment is not optimal when task deadlines are less than their periods [63]. Inverse deadline or, as it is more commonly known, Deadline Monotonic priority assignment is optimal in this case. Applying this policy, priorities are assigned to tasks according to their deadlines, the shorter the deadline the higher the priority. Hence Deadline Monotonic priority assignment is equivalent to Rate Monotonic priority assignment in the limited case when for all tasks, $D_i = T_i$. However, neither Rate Monotonic nor Deadline Monotonic priority assignment policies are optimal for tasks with arbitrary deadlines, i.e. tasks with deadlines which may be greater than their periods [58]. Neither are they optimal for tasks which never share a common release time (termed a *critical instant*) [63]. In [5, 105], Audsley provides analysis which determines if a set of tasks do not share a common release time and presents an optimal priority ordering algorithm which finds a feasible priority assignment if one exists. This algorithm is

applicable to tasks with arbitrary deadlines, which may or may not share a critical instant. The algorithm requires a maximum of $\frac{1}{2}(n^2+n)$ schedulability tests in order to find a feasible priority assignment, if one exists. This is an improvement over examining all *n!* possible priority orderings. Further, the algorithm is optimal for any schedulability test where the worst case response time increases monotonically with decreasing task priority [101].

2.1.4 Schedulability Tests

Following on from early work on sufficient utilisation based tests, a number of exact tests were developed. Leung and Merrill [62] suggested that exact feasibility could be determined by simulating the schedule over the least common multiple (LCM) of task periods. This method is inefficient as the LCM may be very large.

Rate Monotonic Analysis

Sha *et al* [61] examined the characteristics of the Rate Monotonic priority assignment policy, they showed via simulation, that in the average case, the utilisation bound for a large set of tasks is 88% (compared to 69% in the worst case). Further, they derived an exact schedulability test, which is applicable to sets of independent tasks which share a critical instant and have deadlines which are less than or equal to their periods. This test is based on the assertion that: task τ_i is schedulable if and only if there is some time *t*, $t \in [0, D_i]$, when the invocations of all tasks of priority *i* and higher, released in the interval [0, *t*) are complete. The test proceeds by finding the cumulative load, $W_{i,t}$, on the processor due to level *i* and higher priority tasks,

$$W_{i,t} = \sum_{j \in hp(i) \cup i} \left\lceil \frac{t}{T_j} \right\rceil C_j \qquad (2.3)$$

If $W_{i,t} \leq t \leq D_i$, then task τ_i is schedulable. As $W_{i,t}$ only increases at the release of tasks with a higher priority than *i*, then only values of *t* corresponding to these releases need to be checked. This test is valid for any fixed priority assignment policy.

Sha *et al* [88] extended Rate Monotonic analysis to account for blocking due to the operation of the Priority Ceiling Protocol (see section 2.1.5). Furthermore, Sprunt showed how sporadic tasks with hard deadlines can be guaranteed via the operation of Sporadic Servers [90]. Finally, Lehoczky [58] extended Rate Monotonic analysis to tasks with arbitrary deadlines. The two sufficient and not necessary tests presented by Lehoczky place restrictions on task deadlines such that $D_i = kT_i$, where k is a constant across all tasks. One test restricts k to be an integer, the other does not.

Response Time Analysis

An alternative approach to feasibility analysis was developed independently by Harter [49], Joseph and Pandya [52] and Audsley *et al* [11, 10]. Here the emphasis is placed on finding the worst case *response time* of a task, defined as the longest time between its arrival and completion. Schedulability can then be determined via a simple comparison between response times and deadlines. Below, we reproduce the recurrence relation given in [10].

$$r_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \qquad (2.4)$$

This relation iteratively computes the worst case response time, r_i, of task τ_i. Iteration starts with $r_i^0 = 0$ and ends when $r_i^{n+1} = r_i^n$ or $r_i^{n+1} > D_i$, in which case the task is unschedulable. (Note, Proof of convergence for task sets with a processor utilisation ≤ 100% is given in [52]). This approach is also valid for any fixed priority assignment policy.

Subsequently, response time analysis has been extended to cater for tasks with release jitter and arbitrary deadlines [105]. This analysis uses the concept of *busy periods* [58], defined as follows: a priority level i busy period is a continuous time interval during which the notional run-queue contains one or more tasks with priority i or higher. The analysis given by Tindell (for $D > T$) [105] examines q ($q = 0, 1, 2, 3...$) level i busy periods to determine the worst case response time of task τ_i. The length of each busy period $w_i(q)$ is found according to the following recurrence relation:

$$w_i^{m+1}(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m(q) + J_j}{T_j} \right\rceil C_j \quad (2.5)$$

the response time of each invocation is then given by:

$$R_i(q) = w_i(q) - qT_i + J_i$$

Examination of increasing values of q may stop if:

$$w_i(q) \leq (q+1)T_i$$

In which case, the worst case response time of task τ_i is given by;

$$R_i = \max_{q=0,1,2,3\ldots} \left(w_i(q) - qT_i \right) + J_i$$

Unlike the tests for tasks with arbitrary deadlines mentioned earlier, the above feasibility analysis is exact and places no restrictions upon task deadlines. Finally, in [7] Audsley introduced an exact feasibility test for tasks which do not share a critical instant.

2.1.5 Task Interdependency

Allowing tasks to access shared resources which must be used in a mutually exclusive manner introduces the potential for blocking. One task may wish to access a resource, but is prevented from doing so by another task which has already obtained that resource. The first task is said to be *blocked* awaiting the release of the resource. In real-time systems, the blocking experienced by any hard task must be bounded. Arbitrary locking of resources can however lead to unbounded delays due to priority inversion or deadlock.

Priority inversion can occur when a low priority task τ_L locks a semaphore *R*, which is then required by a high priority task τ_H. This problem may be avoided by making critical sections non-preemptable [72] or by mechanisms based on priority inheritance [108]. Using priority inheritance, when task τ_H blocks because τ_L is

holding *R*, task τ_L is given the priority of τ_H (until it releases *R*), thus avoiding pre-emption by tasks of medium priority. Priority inheritance protocols prevent unbounded priority inversion, however, the system is still susceptible to deadlocks and chained blocking.

These problems may be avoided by using an idea first suggested by Lampson and Redhill in 1980 [56]. A ceiling priority is associated with each resource, equal to the highest priority of all the tasks which use that resource. Then when a task accesses a resource, its priority is temporarily increased to the ceiling priority of the resource. This approach was later formalised as the Ceiling Semaphore Protocol [82]. The Ceiling Semaphore Protocol enforces a strict ordering of critical sections. This ensures that each invocation of a high priority task can be blocked by at most, one lower priority task. This has two effects on the schedulability analysis; first, the worst case delay imposed on a high priority task is reduced to the longest critical section of any lower priority task which accesses a resource with a ceiling priority of *i* or higher. Second, no additional context switches are introduced by resource access. Tasks are only blocked prior to commencing execution, however, this blocking may sometimes be unnecessary, a high priority task which is blocked may in fact require no resources on that invocation.

This latter drawback is avoided by the Priority Ceiling Protocol [88]. Under the Priority Ceiling Protocol, a task is only allowed to claim a free resource if its priority is strictly greater than the ceiling priority of any resource currently held by another task. Further, the task holding a resource inherits the priority of the highest priority process which it is blocking. The Priority Ceiling Protocol generally results in lower mean response times when compared to the Ceiling Semaphore Protocol, however, this is at the expense of introducing two extra context switches each time a task is blocked and a more complex run-time implementation.

2.1.6 Practical Considerations

Before fixed priority scheduling theory could be considered a practical technique for use in constructing real-time systems, it was necessary for the theory to be extended to account for system-wide requirements and the characteristics of the implementation environment. In this section, we survey developments aimed at bridging the gap

between fixed priority scheduling theory and the reality of practical implementations.

System Wide Requirements

Real-time systems are often required to be stable under conditions of transient overload; that is, a set of critical tasks must be guaranteed to meet their deadlines even when the processor is overloaded. When priorities are assigned in Rate Monotonic order, tasks with longer periods miss their deadlines under overload conditions. These tasks may however, correspond to those which are most critical to the operation of the system. This stability problem was addressed by the Period Transformation Method introduced by Sha *et al* [87]. This method retains the Rate Monotonic priority ordering, however, task periods and computation times are transformed until all the critical tasks occupy priority levels at which they can be guaranteed to meet their deadlines under worst case conditions.

The problem of integrated I/O and task scheduling was also investigated by Sha *et al* [87]. They analysed the reduction in schedulability caused by insufficient priority levels and showed via simulation studies that FIFO queueing of messages also results in poor schedulability bounds. Building upon this early work, Strosnider extended Rate Monotonic analysis to the IEEE 802.5 token ring [95], enabling bus access times to be bounded. This analysis required that message deadlines be equal to their periods; a restriction which is too limiting for many applications. Corrections to this analysis were subsequently given by Pleinevaux [79], who showed that clock synchronisation messages cannot be handled within the framework given by Strosnider *et al.*

More comprehensive analysis of fixed priority communications scheduling was later given by Tindell [100]. This analysis permits messages to have arbitrary deadlines. It also enables the cost of reconstituting and delivering messages to be found, thus bounding the overheads and interference on destination processors and facilitating the guarantee of end-to-end deadlines. In [107], Tindell *et al* provided analysis, bounding the response times of messages under the Controller Area Network (CAN) protocol. This analysis was extended to account for the characteristics of actual CAN controllers, highlighting problems of priority inversion, inherent in implementations with a single transmission buffer.

In 1978, Dhall and Liu showed that optimal scheduling policies for uniprocessor systems are not optimal for multiprocessor systems: allocating tasks to processors according to the Rate-Monotonic First-fit policy results, in the worst-case, in at least twice as many processors being required as compared with an optimal allocation [42]. Recently, this result has been extended, showing that in the worst case, Rate Monotonic Next-fit, Rate Monotonic First-fit and Rate Monotonic Best-fit require respectively 2.67, 2.33 and 2.33 times as many processors as an optimal allocation [74]. Further, for tasks with up to k replicas which must be scheduled on separate processors for reasons of fault tolerance, Oh and Son [75] devised a heuristic based on Rate Monotonic First-fit which requires a maximum of $2.33N_o + k$ processors, where N_o is the number required by an optimal allocation.

In distributed systems, task allocation, task scheduling and communication scheduling are mutually dependent, NP-hard problems [99]. The problem of finding optimal static solutions is thus intractable for all but the simplest systems. However, global optimisation techniques, such as simulated annealing can be used to find good sub-optimal static allocations [99]. Simulated Annealing can also be used to minimise bus loading or balance processor loads whilst ensuring that hard time constraints are met. It can also handle other constraints such as the need to place task replicas on separate processors for fault tolerance.

Many real-time systems have a number of distinct modes of operation, for example avionics systems may have take-off, level flight and landing modes. Each mode represents a potentially different set of tasks. Thus tasks may be deleted, added, or may change their timing characteristics and priority on a mode change. In [89], Sha *et al* provided a simple mode change protocol and associated feasibility analysis. This initial analysis was subsequently shown to be insufficient: tasks which were guaranteed could miss their deadlines following a mode change. A revised protocol and analysis was given by Tindell *et al* in [103]. This revised analysis finds the worst-case response times of processes, taking account of increases in interference caused by the mode change.

Kernel Performance Characteristics

Before fixed priority feasibility theory can be applied in the engineering of an actual system, it is first necessary to tailor the theory so that the performance characteristics of the operating system kernel are adequately addressed. These characteristics include context switch times, scheduling overheads, non-pre-emptable sections, release jitter introduced by the scheduler and interference due to interrupt handling. We now discuss the advances made in this area.

In 1991, Locke *et al* described a Generic Avionics Platform (GAP) case study based on an embedded system comprising a set of Ada tasks [70]. To analyse this case study, Locke *et al* modelled timer interrupt and context switch overheads as an extra task with the highest priority and larger task execution times respectively. Initial sufficient analysis showed that less than half of the application tasks were guaranteed to meet their deadlines. The observed behaviour of the system however, suggested that only two tasks actually missed their deadlines. Subsequently, exact response time analysis was applied to the GAP task set [10]. Accounting for release jitter due to the periodic timer interrupt (tick) driven scheduler, this analysis more accurately reflected the observed behaviour of the system.

The problem of incorporating an accurate model of kernel behaviour and overheads into fixed priority scheduling theory was investigated by Katcher *et al* [53]. In the implementation of a priority pre-emptive dispatcher, there are often non-pre-emptable sections, which give rise to blocking on all application tasks. Further, the overheads of manipulating the queues used by the scheduler must be allowed for. Katcher *et al* identified two ways of implementing fixed priority pre-emptive scheduling: *event driven* and *tick driven*. In an event driven system, the scheduler is invoked each time a task arrives. Whereas in a tick driven system, there is a periodic timer interrupt which polls for task arrivals. With the latter approach, tasks can in general suffer release jitter of up to the polling period. Using Rate Monotonic analysis, Katcher *et al* accounted for this release jitter by adding a blocking factor to each task. This is a sufficient but pessimistic approach.

In [26] Burns *et al* developed a model of the scheduling overheads present in an Ada implementation of the Attitude and Orbital Control System of the Olympus satellite [17]. They recognised that there was a large variation in the execution time of the tick driven scheduler, depending upon the number of tasks moved from the delay queue to the run queue. Using this result, the analysis given by Burns *et al* determines the maximum number of tasks released in a given interval and thus enables a realistic bound on the scheduling overheads to be found.

In accounting for context switch overheads, twice the context switch time is typically added to the worst-case execution time of each task, prior to calculating response times and hence the feasibility of the task set. However, as noted by Gerber *et al*, it is only meaningful to attach a deadline to the last observable event of a task [46]. As a result, the context-switch away from a task and any internal computation occurring after the last observable event need not be completed prior to the deadline of the task. In [14] sufficient and not necessary response time analysis for processes with such internal deadlines is described, with exact analysis given in [22].

2.2 Algorithms for Providing Spare Capacity at High Priority Levels

Spare capacity: is defined as *"that processor time which is not required at run-time to meet the deadlines of hard tasks"* [12]. In spite of the advances made in both fixed priority pre-emptive scheduling theory and worst case execution time analysis, it is inevitable that spare capacity will be available at run-time. This spare capacity may be classified as follows:

1. *Extra capacity* - the processor capacity which is required to guarantee the time constraints of hard deadline tasks can be calculated using exact schedulability analysis. Any additional capacity above this level may be identified as extra capacity. Hence extra capacity is present if any hard deadline task could have its worst case execution time increased, and the task set still remain schedulable.

2. *Gain time* - which is produced when hard tasks execute in less than their worst case execution times. This may be due to software not taking worst case paths or hardware gains such as cacheing and pipelining [13]. Gain time may

be identified as the remaining execution time budget when a task completes. Alternatively, Gain Points [7] or Milestones [43] may be inserted into the code to report when a non-worst case path has been taken, and hence gain time produced.

3. *Spare time* - which is present; due to a favourable (i.e. non worst case) phasing of periodic tasks, sporadic tasks not arriving at their maximum rate, deadlines which are dependent on the external environment not being as short as those guaranteed, and tasks not being blocked when this was allowed for in the schedulability analysis.

In general, extra capacity can be identified off-line, whilst gain time and spare time can only be identified by on-line mechanisms.

Within the framework of fixed priority pre-emptive scheduling, a number of approaches have been developed which make spare capacity available for optional or soft aperiodic task execution. These include:

1. Background processing.

2. Polling server [87].

3. Priority Exchange algorithm [60].

4. Deferrable Server algorithm [60].

5. Sporadic Server algorithm [90].

6. Extended Priority Exchange algorithm [92].

7. Slack Stealing algorithm [59]. (We note that many of these techiniques were originally developed as a means of scheduling sporadic tasks prior to the development of suitable off-line feasibility analysis for such tasks).

In this section, we review the above approaches and give examples of their operation using the task set detailed below.

Hard tasks				
Name	Period	Deadline	wcet	priority
A	8	6	2	1 (high)
B	12	12	5	2 (low)

Soft tasks		
Name	Arrival time	Execution time
w	1	0.5
x	2	1.5
y	13	1.0
z	14	1.0

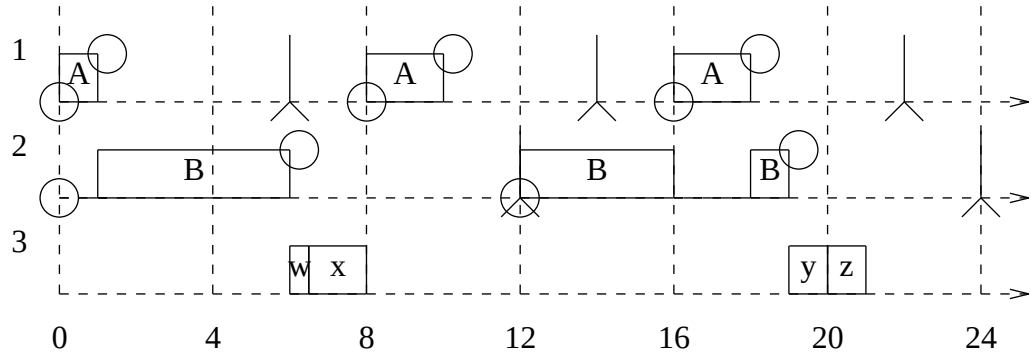
In our examples, tasks A and B are initially released at time $t = 0$. The first invocation of task A is assumed to require 1 unit of execution time, whilst all other invocations take their worst case execution time. We also assume that the soft tasks await service in order of arrival. The example task set has an extra capacity of 1 unit every 4 ticks (at the highest priority level) and gain time of 1 unit which is produced when the first invocation of task A completes early. Further, there are 2 units of spare time present in the latter half of the LCM, (which is 24 in this case).

A criteria which is often used to assess the performance of the above algorithms is the mean response time of soft tasks [90, 92, 59]. We adopt this criteria in our review, as an algorithm which results in shorter soft task response times is better able to provide spare capacity for optional computation, between the release time and deadline of a hard task.

Figures 2.1 to 2.9 illustrate the schedule produced by the various scheduling policies. In these figures, the following notation is used. Priority level is indicated on the left hand side of each time line. Execution at each unique priority level is shown on a separate time line. A circle on the time line represents the release of a hard task at that priority level. Similarly, a circle above the time line represents task

completion. The deadline of each hard task is indicated by a solid vertical line and the symbol "^". Missed deadlines are highlighted by a filled circle.

2.2.1 Background Processing



Background processing: Priority = 3.
t=0, tasks *A* and *B* are released. Task *A* executes.
t=1, soft task *w* arrives, task *A* completes and *B* commences execution.
t=2, soft task *x* arrives.
t=6, task *B* completes. As no hard tasks are pending, soft tasks may execute at priority level 3.
t=6.5, soft task *w* completes with a response time of 5.5.
t=8, soft task *x* completes with a response time of 6. Task *A* is released and executes.
t=10, task *A* completes.
t=12, task *B* is released and executes.
t=13, soft task *y* arrives.
t=14, soft task *z* arrives.
t=16, task *A* is released and pre-empts *B*.
t=18, task *A* completes, task *B* resumes.
t=19, task *B* completes. As no hard tasks are pending, soft tasks may again execute at priority level 3.
t=20, soft task *y* completes with a response time of 7.
t=21, soft task *z* completes with a response time of 7.
Mean soft task response time = 6.375.

Figure 2.1: Background Processing.

Background processing is the simplest and perhaps least effective approach. Soft tasks are assigned priority levels below all the hard tasks. Thus soft tasks may only execute when there are no hard tasks pending, i.e. when the processor would otherwise have been idle. If the processor utilisation of the hard task set is high, then processor busy periods may comprise many invocations of hard tasks. Under

these conditions, background service opportunities will be infrequent and soft task response times correspondingly long. Background processing is therefore only appropriate when the response times of soft tasks are not critical. Figure 2.1 illustrates the long response times of soft tasks *w*, *x*, *y* and *z* when they are scheduled at a Background priority level.

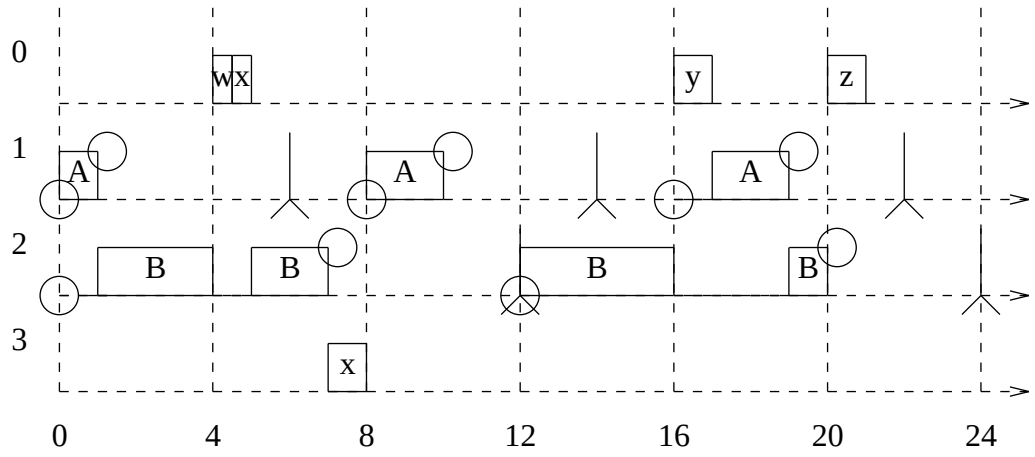
The response times of soft tasks may be improved by executing them at a higher priority under the control of a pseudo hard real-time server task, such as a simple Polling Server.

2.2.2 Polling Server

A Polling Server [87] is a periodic task with a fixed priority level (usually the highest) and an execution capacity. The capacity of the server is calculated off-line and is normally set to the maximum possible such that the hard task set, including server, remains schedulable. At run-time, the polling server is released periodically and its capacity is used to service any pending soft tasks. Once this capacity has been exhausted, execution is suspended until it can be replenished at the server's next release. If no soft tasks are pending when the polling server is allocated the processor, it must suspend execution until its next period. This wastes the high priority capacity of the server as subsequent arrivals of soft tasks must wait for background service opportunities or the next release of the server.

Nevertheless, the Polling Server offers improved soft task response times over background processing. This is illustrated in figure 2.2. Soft tasks *w*, *x*, *y* and *z* arrive mid way through the server's period and have to wait until its next release before being serviced. Note, the server is initially released at time $t = 0$ but immediately suspends as there are no soft tasks pending. This is the major drawback with the polling approach: service opportunities are not necessarily co-ordinated with the arrival of soft tasks.

The Priority Exchange, Deferrable Server and Sporadic Server algorithms avoid the above drawback. These methods are based on similar principles to the Polling Server, however, they are all able to preserve capacity if no soft tasks are pending when the server is released. Due to this property, they are termed *"Bandwidth Preserving Algorithms"*. The three algorithms differ in the ways in which the capacity of the



Polling Server: Capacity = 1, Period = 4, Priority = 0

t=0, the Polling Server is released and immediately suspends execution as there are no soft tasks pending.

t=1, soft task *w* arrives.

t=2, soft task *x* arrives.

t=4, the Polling Server is released and services the soft tasks at priority level 0

t=4.5, soft task *w* completes with a response time of 3.5.

t=5, the server's capacity is exhausted. Task *B* resumes.

t=7.5, task *B* completes, allowing soft task *x* to execute at priority 3 (background).

t=8, task *x* completes with a response time of 6. The server is released and suspends as there are no soft tasks pending.

t=12, the server is released and again suspends itself.

t=13, soft task *y* arrives and awaits service.

t=14, soft task *z* arrives.

t=16, the server is released and services task *y* at priority level 0, exhausting its capacity.

t=17, soft task *y* completes with a response time of 4.

t=20, the server is released and services soft task *z*.

t=21, task *z* completes with a response time of 7.

Mean soft task response time = 5.25.

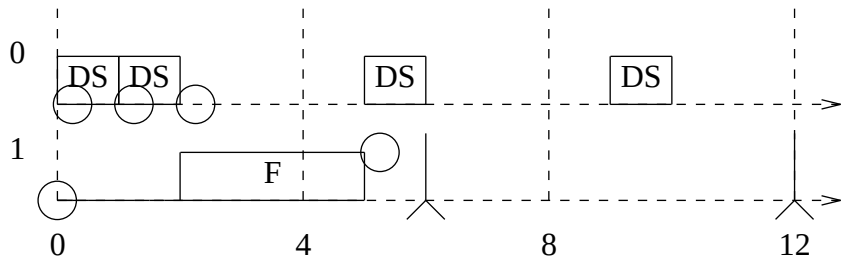
Figure 2.2: Polling Server.

server is preserved and replenished, and in the schedulability analysis required to determine their maximum capacity.

2.2.3 Deferrable Server

The Deferrable Server algorithm is similar to the polling approach, in that it makes use of a high priority periodic server task. However, unlike the Polling Server, the Deferrable Server is able to preserve its high priority capacity if there are no soft tasks pending. The Deferrable Server is therefore able to service soft tasks at a high priority level at any time during its period, provided of course, that its execution capacity has not been exhausted. Any remaining capacity is discarded at the end of the server's period, before being replenished in full at the start of its next period.

The schedulability analysis required to determine the maximum capacity of the Deferrable Server differs from that used for the Polling Server. By preserving its capacity at a high priority level throughout its period, the Deferrable Server can cause back-to-back interference on lower priority hard tasks. This is illustrated in figure 2.3.

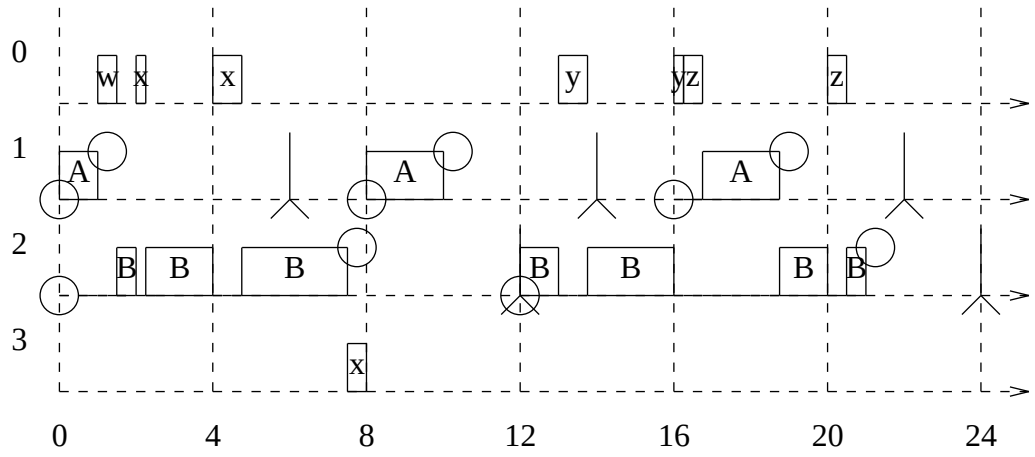


DS - Deferrable Server, period = 4, capacity = 1
F - a hard task, period = 6, deadline = 6, wcet = 3
t=0, The Deferrable Server has preserved its capacity until close to the end of its period at **t=1**.
t=1, Subsequent release of the Deferrable Server causes a back-to-back hit on task F.

Figure 2.3: Deferrable Server Interference.

Taking this extra interference into account, the maximum capacity of the Deferrable Server is smaller than that of a comparable Polling Server.

Figure 2.4 shows the example task set scheduled using the Deferrable Server algorithm. The maximum capacity of a Deferrable Server at the highest priority level (with period = 4) is 0.75, compared to 1.0 for the polling approach.



Deferrable Server: Capacity = 0.75, Period = 4, Priority = 0

t=0, the server is released and suspends execution as there are no soft tasks pending.

t=1, soft task *w* arrives and is serviced immediately in contrast with the Polling Server which is unable to preserve its capacity

t=1.5, soft task *w* completes, with a response time of 0.5, leaving 0.25 units of server capacity remaining.

t=2, soft task *x* arrives and is serviced at priority level 0, exhausting the remaining capacity of the server.

t=4, capacity is replenished, allowing task *x* to be resumed

t=4.75, the Deferrable Server's capacity is again exhausted.

t=7.5 soft task *x* resumes execution at priority 3 (background).

t=8, task *x* completes with a response time of 6. The server's capacity is replenished.

t=12, the server's capacity is discarded at the end of its period before being replenished again.

t=13, soft task *y* arrives and is serviced for 0.75 ticks, exhausting the capacity of the server.

t=14, soft task *z* arrives.

t=16, the server's capacity is replenished allowing tasks *y* and *z* to be serviced, again exhausting server capacity.

t=16.25, soft task *y* completes with a response time of 3.25

t=20, the server's capacity is replenished, allowing task *z* to resume at priority 0.

t=20.5 task *z* completes with a response time of 6.5.

Mean soft task response time = 4.0625.

Figure 2.4: Deferrable Server Algorithm.

In this example, soft task response times are improved over the polling approach in spite of the smaller size of the Deferrable Server. This is because by preserving its capacity, the Deferrable Server is better able to co-ordinate service opportunities with the arrival of soft tasks.

2.2.4 Priority Exchange Algorithm

The method of bandwidth preservation used by the Priority Exchange algorithm differs from that of the Deferrable Server: a high priority periodic server is again used to service soft tasks. However, if no soft tasks are pending when the server begins execution, then priority exchange occurs. The highest priority runnable hard task is executed at the priority level of the server and the server's capacity converted into guaranteed execution time at the lower priority level of the hard task. Soft tasks may be serviced immediately, whenever the remaining capacity of the server is at a higher priority level than any of the runnable hard tasks. Furthermore, as the objective is to reduce soft task response times, ties between server capacity and hard tasks at the same priority level are broken in favour of the server. The server's capacity may however, be degraded to priority levels which are lower than that of the highest priority runnable hard task, in which case, the server is pre-empted, no priority exchange takes place, and soft tasks cannot be serviced immediately. Note, this is in marked contrast with the Deferrable Server which preserves capacity at its original priority level and may therefore result in better soft task response times under certain circumstances.

In effect the priority exchange mechanism advances the execution of hard tasks and thus ensures that they remain schedulable. Further, the capacity of the server is preserved throughout its period albeit at decreasing priority levels. This is an improvement over the polling approach, where server capacity is immediately degraded to the background priority level if there are no soft tasks pending.

At the end of the Priority Exchange Server's period, capacity may still be present at low priority levels. This remaining capacity is not discarded and may be used in subsequent periods. The server's high priority capacity is replenished at the start of its period.

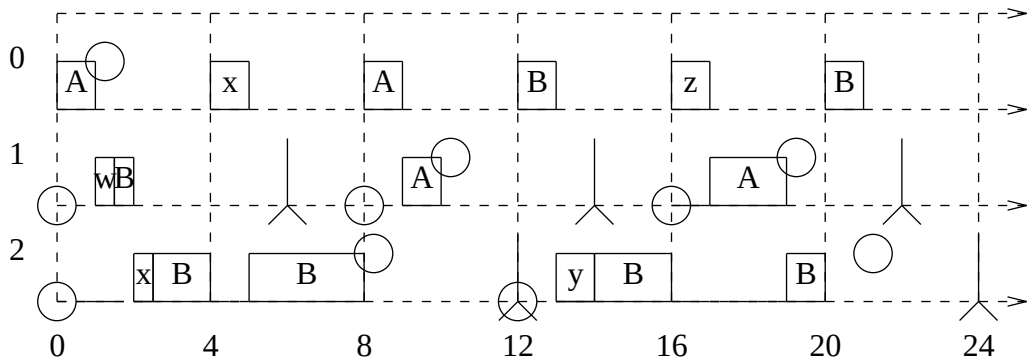
Lehoczky *et al* [60] showed that for schedulability purposes, the Priority Exchange Server may be regarded as a simple periodic task. Hence the maximum capacity of the Priority Exchange Server is the same as that of a comparable Polling Server.

Figure 2.5 describes the operation of the Priority Exchange algorithm. This example illustrates the advantages of the Priority Exchange algorithm over the polling approach, c.f. bandwidth preservation (shorter response time for task *w*), and over the Deferrable Server, c.f. server capacity (shorter response time for task *y*). However, the Priority Exchange algorithm does not always exhibit a performance advantage over the Deferrable Server approach. This is because in general, capacity is preserved at a lower priority level. For example, suppose that the first release of task *A* occurred at $t = 1$ instead of $t = 0$. Initial priority exchange with task *B* would then preserve the server's capacity at priority 2. However when task *A* (priority 1) arrived at $t = 1$ it would pre-empt the server, increasing the response time of task *w* to 1.5. In contrast, the Deferrable Server which preserves its smaller capacity at priority 0 would still be able to complete task *w* for a response time of 0.5.

2.2.5 Sporadic Server

The Sporadic Server algorithm attempts to combine the advantages of both the Priority Exchange and Deferrable Server algorithms. It is similar to the Deferrable Server algorithm in that it maintains capacity at its original priority level. Whilst, the capacity of the Sporadic Server is the same as that of a comparable Priority Exchange or Polling Server. Like the other bandwidth preserving algorithms, the Sporadic Server algorithm uses a high priority periodic server task. It differs however, in the way in which the capacity of the server is replenished. Rather than being replenished periodically, the capacity of the Sporadic Server is only replenished once some or all of it has been used.

In describing the operation of the Sporadic Server, it is instructive to use of the concept of priority *i busy* and *idle* periods [58]. These are defined as follows: a priority *i* busy period is a continuous time interval during which the processor is busy with tasks of priority *i* or higher. Similarly, a priority *i* idle period is a time interval during which no tasks execute at priority level *i* or higher. (Note, a priority *i* idle



Priority Exchange Server: Capacity = 1, Period = 4, Priority = 0

t=0, the server is released. As there are no soft tasks pending, priority exchange takes place. Task *A* executes at priority 0, converting the server's capacity of 1 unit to priority level 1.

t=1, Soft task *w* arrives and is serviced immendiately at priority 1

t=1.5, task *w* completes with a response time of 0.5. A further priority exchange then occurs. Task *B* executes for 0.5 ticks, converting the remaining server capacity (0.5) to priority 2.

t=2, Soft task *x* arrives and is serviced at priority 2. Note, there is a tie between the priority of the server and task *B* which is broken in favour of the server.

t=2.5, the server's capacity is exhausted. Task *B* resumes.

t=4, The server's priority 0 capacity is replenished, allowing the remainder of task *x* to be serviced at priority 0.

t=5 task *x* completes with a response time of 3.

t=8, the server's capacity is again replenished.

t=9, priority exchange with task *A* has converted the server's capacity to priority 1.

t=11, the server's capacity has been fully converted to priority 3 (background) and is discarded.

t=12, the server's priority 0 capacity is replenished and then exchanged with task *B*.

t=13, soft task *y* arrives and is serviced at priority 2 in preference to task *B*.

t=14, task *y* completes with a response time of 1. The server's capacity is exhausted

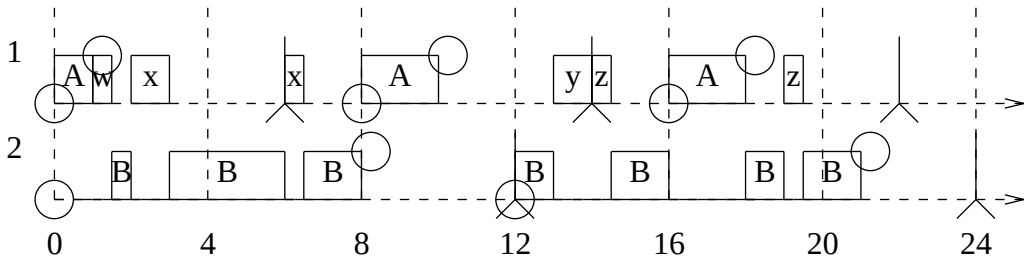
t=14, Soft task *z* arrives and awaits service.

t=16, the server's priority 0 capacity is replenished, allowing it to service task *z*.

t=17, task *z* completes with a response time of 3.

Mean soft task response time = 1.75.

Figure 2.5: Priority Exchange Algorithm.



Sporadic Server: Capacity = 1.5, Period = 6, Priority = 1

t=0, task *A* begins executing, hence **t=0** is the start of a priority 1 busy period.

t=1, soft task *w* arrives and is immediately serviced at priority 1. Replenishment of the capacity consumed (0.5) is set for **t=6**, (equal to the length of the server's period after the start of the busy period).

t=1.5, task *B* begins executing at priority 2

t=2, soft task *x* arrives and is serviced immediately

t=3, the server's capacity is exhausted. A replenishment of 1 unit is set for **t=8**, as the start of this priority 1 busy period coincided with the arrival of task *x* at **t=2**.

t=6, 0.5 units of server capacity are replenished and subsequently consumed in completing task *x*.

t=6.5, task *x* completes with a response time of 4.5. A replenishment of 0.5 units is set for **t=12**.

t=8, 1 unit of server capacity is replenished.

t=12, the server is brought up to full capacity by the replenishment of 0.5 units.

t=13, soft task *y* arrives and is serviced immediately.

t=14, task *y* completes with a response time of 1. Task *z* arrives and is serviced for 0.5 ticks, exhausting the server's capacity

t=14.5, a replenishment of 1.5 units is set for **t=19**.

t=19, the server's capacity is replenished in full, allowing the remainder of task z to be serviced.

t=19.5, *z* completes with a response time of 5.5.

Mean soft task response time = 2.875.

Note, using a Sporadic Server with priority 0, a period of 4 and a capacity of 1 results in the same response times for tasks *w* and *y* and shorter response times of 3.0 and 4.0 for tasks *x* and *z* respectively.

Mean soft task response time = 2.125.

Figure 2.6: Sporadic Server algorithm.

period may contain intervals during which tasks with a priority lower than *i* execute).

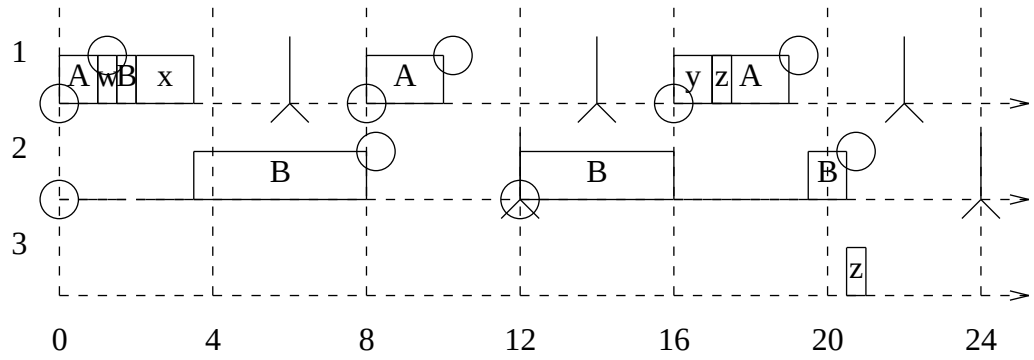
Suppose the Sporadic Server task has a priority *i* and period T_i. Soft tasks may execute under the server at priority *i*, consuming its capacity. Replenishment of this capacity is set to occur T_i after the start of the priority *i* busy period which includes the soft task execution. It is worth noting that capacity replenishment may therefore be scheduled to occur less than T_i after the start of soft task execution. This is permissible provided that the processor was previously busy with tasks of priority *i* or higher. Sprunt *et al* [90] prove that using the above method of replenishment, the interference on lower priority tasks due to the Sporadic Server is the same as a number of periodic tasks with the same period and a total execution time equal to the server's capacity. Hence, the maximum capacity of a Sporadic Server is the same as that of a comparable Polling Server.

Figure 2.6 shows the example task set scheduled under the Sporadic Server. To fully illustrate the replenishment method of the Sporadic Server algorithm, we have selected a server period of 6 and hence a capacity and priority of 1.5 and 1 respectively. Improved performance is however, obtained with a server period of 4.

Each of the server based methods described so far exploits extra capacity which is available at a high priority level to improve soft task response times. However, in systems where the load due to hard deadline tasks is high, there may not be enough extra capacity at high priority levels to include a periodic server of any useful size. Furthermore, none of the server based approaches exploits gain time produced when a hard task requires less than its worst case execution time.

2.2.6 Extended Priority Exchange Algorithm

The Extended Priority Exchange algorithm addresses the above limitations by reclaiming gain time at its original priority level. In describing the operation of this algorithm, it is instructive to view the priority levels assigned to hard tasks as having a maximum capacity. The maximum capacity available at each priority level may be found by examining the hard task set in priority order. First the execution budget of the task at priority level 1 is increased to the maximum possible such that the hard task set remains schedulable. The difference between this maximum priority 1



Extended Priority Exchange: Priority 1 extra capacity = 1.5.

t=1, task *A* completes early, producing 1 unit of gain time at priority level 1. This increases the server's capacity to 2.5 at priority 1. Task *w* arrives and is serviced immediately.

t=1.5, task *w* completes, with a response time of 0.5, leaving a capacity of 2 remaining at priority 1. Priority exchange then occurs with task *B* executing at priority 1. This leaves the server with a capacity of 1.5 at priority 1 and 0.5 at priority 2.

t=2, task *x* arrives and is serviced at priority level 1.

t=3.5, task *x* completes with a response time of 1.5. This represents an improvement over all the algorithms which do not exploit gain time.

t=8, task *A* is released and the server's priority 1 capacity is incremented by 1.5. Note, this is in addition to the remaining priority 2 capacity of 0.5.

t=12, the server's capacity has been completely converted to the background priority level and is therefore discarded.

t=13, task *y* arrives.

t=14, task *z* arrives.

t=16, task *A* is released and the server's priority 1 capacity incremented by 1.5, allowing tasks *y* and *z* to be serviced.

t=17, task *y* completes with a response time of 4.

t=17.5, priority 1 (and 2) capacity is exhausted

t=21, task *z* completes with a response time of 7, by utilising a background service opportunity

Mean soft task response time = 3.25.

Note, representing extra capacity as a server with priority 0, capacity 1 and period 4, results in soft task execution in the same intervals as the Priority Exchange algorithm; except for task *x* which completes at **t=3.5**.

Mean soft task response time = 1.375.

Figure 2.7: Extended Priority Exchange algorithm.

capacity and the task's worst case execution time is stored as priority 1 extra capacity. Similarly, the extra capacity present at subsequent priority levels is found by assuming that all higher priority tasks use the maximum capacity available at their priority levels.

At run-time, the Extended Priority Exchange algorithm operates according to the priority exchange mechanism described previously. However, unlike the Priority Exchange algorithm, the extended approach replenishes capacity at each release of a hard task. Server capacity at priority *i* is thus incremented by the priority *i* extra capacity at each release of hard task τ_i. Furthermore, on completion of each invocation of task τ_i, the actual execution time is compared to the worst case time. Any gain time so identified is then reclaimed by adding it to the priority *i* server capacity.

Figure 2.7 illustrates the operation of the Extended Priority Exchange algorithm. The extra capacity at priority 1 is 1.5. There is no extra capacity at priority level 2.

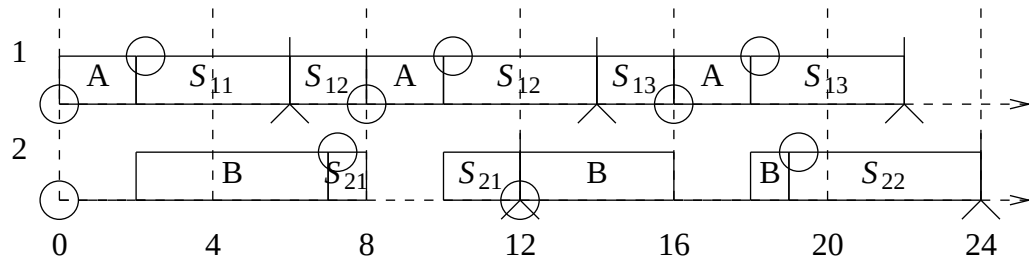
All the bandwidth preserving algorithms can under certain circumstances offer significant improvements over the polling approach. There are however, still disadvantages with these more complex algorithms. They tend to degrade to providing essential the same performance as the polling server at high loads. Moreover, none of the approaches described so far, are able to exploit spare time (present due to a favourable phasing of hard tasks) as anything other than a background service opportunity.

2.2.7 Slack Stealing Algorithm

The Slack Stealing algorithm of Lehoczky and Thuel [59] suffers from none of the above disadvantages. It services soft task requests by making any spare processing time available as soon as possible. In doing so, it effectively steals slack from the hard periodic tasks. A means of determining the maximum amount of slack which may be stolen, without jeopardising the hard timing constraints, is thus key to the operation of the algorithm.

In [59], Lehoczky and Thuel describe how the slack available can be found. This is done by mapping out the processor schedule for the hard periodic tasks over the LCM of their periods. The mapping is then inspected to determine the total amount of priority *i* idle time present between the deadline on one invocation of task τ_i and the deadline on the next. The values found for each task are stored in a table. At run-time, a set of counters are used to keep track of the slack at each priority level. These counters are decremented depending on which tasks, if any, are executing. For example, if hard task τ_i executes during the time interval $[t_a, t_b)$, then the slack at all priority levels higher than *i* is reduced by $t_b - t_a$. Similarly, if the processor is idle, or executing soft tasks during the interval $[t_a, t_b)$, then the slack at all priority levels is reduced by $t_b - t_a$. At the completion of each task τ_i, the counter for priority *i* slack is incremented by the value stored in the table corresponding to the next invocation of task τ_i. Furthermore, if gain time g_i is identified at priority level *i*, it may be exploited by incrementing the slack at level *i* and all lower priority levels by g_i. Whenever there is slack available at *all* priority levels, then soft tasks may be executed at the highest priority.

Unfortunately, the need to map out the LCM restricts the applicability of the Slack Stealer: slack can only be stolen from hard tasks which are strictly periodic and have no release jitter [10] or synchronisation. Realistically, it is also limited to task sets with a manageably small LCM.



S_{ik} = slack on the *k*th invocation of task τ_i.

Figure 2.8: Slack at each Priority Level.

Figure 2.8 shows the priority 1 slack which corresponds to the first (S_{11}), second (S_{12}) and third (S_{13}) invocations of task *A* and similarly, the priority 2 slack corresponding to the first (S_{21}) and second (S_{22}) invocations of task *B*. The slack

on each invocation of a task over the LCM is given in the table below.

Hard task slack			
Task	Invocation		
	1st	2nd	3rd
A	4	6	6
B	3	5	-

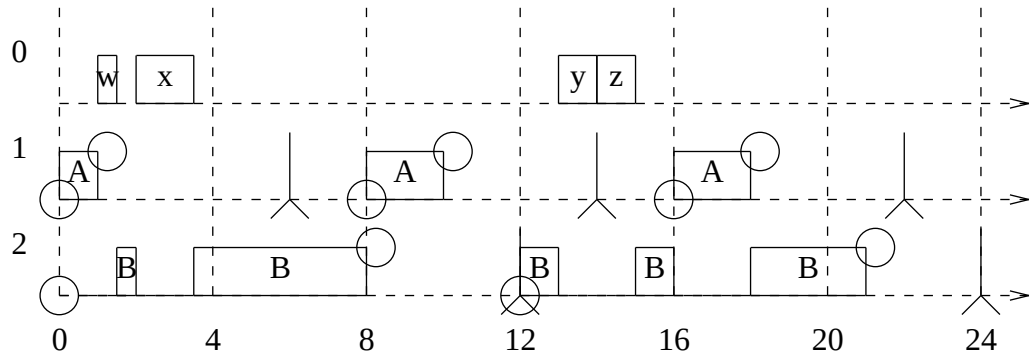
Figure 2.9 illustrates the operation of the Slack Stealing algorithm.
Note, by making use of spare time present in the latter half of the LCM, the Slack Stealer is able to improve the response time of soft task *z* over all the other algorithms discussed.

2.2.8 Critique

In this section, we review the seven algorithms described previously against the following criteria: performance, overheads, simplicity and coverage.

Performance

If spare capacity is to be used to improve the utility of hard tasks, then it generally needs to be made available at a high priority level. This can be seen by considering the scheduling of an optional soft task *x* in support of a high priority hard task *A*. For *x* to improve the utility of *A*, it must execute before *A*'s deadline. To facilitate this, spare capacity has to be provided at a high priority level. If it is not, then *x* will be pre-empted by another hard task *B* and therefore be unable to complete until after *A*'s deadline. (Note, this point is graphically illustrated by comparing the execution of task *x* in figures 2.1 and 2.9). Further, enough spare capacity must be made available at a high priority to ensure that soft or optional tasks can be completed without relying upon a background service opportunity. (For example, compare the execution of task *x* in figures 2.4 and 2.5). Performance depends upon both the amount of spare capacity provided and the priority level at which it is made available.



Slack Stealer.

t=0, task *A* is released with an initial priority 1 slack (S_1) of 4 (i.e., task *A* could be delayed by upto 4 ticks and still meet its deadline). Task *B* is also released and has an initial priority 2 slack (S_2) of 3.

t=1, task *A* completes producing gain time of 1 tick which is added to the slack available at priorities 1 and 2. In addition, as task *A* has completed, the priority 1 slack is incremented by the value stored in the table corresponding to the second invocation of task *A*. Thus at **t=1**, S_1 = 11 and S_2 = 4.

t=1, task *w* arrives and is serviced immediately as there is slack available at all priority levels.

t=1.5, task *w* completes with a response time of 0.5. (S_1 = 10.5, S_2 = 3.5) Task *B* executes for 0.5 ticks, leaving the priority 2 slack unchanged, but reducing S_1 to 10.

t=2, task *x* arrives and is serviced immediately.

t=3.5, task *x* completes with a response time of 1.5, (S_1 = 8.5, S_2 = 2). Task *B* executes for 4.5 ticks, reducing the priority 1 slack to 4.

t=8, task *B* completes. S_2 is therefore incremented by the value corresponding to the second invocation of task *B*. Hence S_2 = 7. Task *A* is released.

t=10, task *A* completes and the priority 1 slack is incremented (S_1 = 10). The processor is then idle for 2 ticks, reducing the slack at all priority levels by 2.

t=12, task *B* is released and executes for 1 tick, further reducing the priority 1 slack to 7.

t=13, task *y* arrives and is serviced immediately.

t=14, task *y* completes with a response time of 1, (S_1 = 6, S_2 = 4). Task *z* arrives and is serviced immediately.

t=15, task *z* completes with a response time of 1, (S_1 = 5, S_2 = 3).

Mean soft task response time = 1.

Figure 2.9: Slack Stealing algorithm.

With background processing, spare capacity is only ever made available at a background priority level. This is insufficient to ensure that it can be used to improve the utility of hard tasks. The Polling Server initially makes capacity available at a high priority level, however, if no tasks are pending which wish to use this capacity, then it is immediately degraded to a later background service opportunity. The Priority Exchange and Extended Priority Exchange algorithms also make capacity available at an initially high priority level, however, if it is not needed immediately, then capacity is fragmented by the priority exchange mechanism to ever decreasing priority levels. This is an improvement over the polling approach, although, it is not as effective as the Deferrable Server and Sporadic Server algorithms which preserve capacity at its original high priority level. The Slack Stealing algorithm provides the optimal approach by making spare capacity available at the highest possible priority level.

Background processing makes no spare capacity available at a high priority level. The Polling Server, Deferrable Server, Priority Exchange and Sporadic Server algorithms can all utilise any extra capacity present in the system, although in general this may require the use of more than one server task. The high priority capacity provided under the Deferrable Server is less than with the other algorithms due to the possibility of back to back interference on lower priority tasks. The Extended Priority Exchange algorithm has a capacity advantage over the server based approaches. This is because it is able to reclaim gain time at a high priority level. In [92], Sprunt *et al* show that mean hard task execution times need only differ from the worst case by a few percent for the Extended Priority Exchange algorithm to significantly out perform the Priority Exchange and Deferrable Server approaches. The Slack Stealing algorithm again provides the most efficient approach by reclaiming both gain time and spare time. The table below summarises the performance of each method, given our example task set.

Performance	
Method	Mean soft task response time
Background	6.4
Polling	5.3
Deferrable Server	4.1
Sporadic Server	2.1
Priority Exchange	1.8
Extended Priority Exchange	1.4
Slack Stealing	1.0

Overheads / Simplicity

Overheads include processor time and memory requirements, which could potentially outweigh the capacity and priority advantages of more complex algorithms. In particular, any execution time overhead reduces the spare capacity available for improving hard task utility.

Background processing has the simplest implementation and no overheads above that of basic fixed priority pre-emptive scheduling. Of the other approaches, the Polling Server and Deferrable Server algorithms have the lowest run-time overheads - $O(1)$, and simple implementations, as capacity is used, and replenished periodically at only one priority level. The overheads of the Sporadic Server algorithm are dependent on the soft task arrival pattern, as it needs to keep track of busy period start times and a number of replenishment times and amounts. The run-time overheads and complexity of the Priority Exchange, Extended Priority Exchange and Slack Stealing algorithms are $O(n)$ as capacity or slack needs to be managed at every priority level.

The memory requirements of all the algorithms are insignificant, with the exception of the Slack Stealer. The Slack Stealing algorithm uses a table which contains a value for every invocation of a task over the LCM of task periods. The memory requirements of this table effectively limit the applicability of the Slack

Stealer to task sets with a manageably small LCM.

Coverage

The next generation of real-time systems are expected to include hard sporadic and periodic tasks which may exhibit release jitter, blocking and stochastic execution times, as well as variable but lower bounded deadlines. Algorithms which make spare capacity available need to be able to accommodate and ideally exploit such complex characteristics.

As outlined in section 2.1, scheduling theory now extends to guaranteeing hard tasks with the complex timing characteristics listed above. These advances can be incorporated into the analysis which determines the maximum capacity of a server task used by the Polling, Deferrable Server, Sporadic Server and Priority Exchange algorithms. Similarly, the extra capacity available at each priority level can be found for the Extended Priority Exchange algorithm. The Slack Stealing algorithm however, relies on prior knowledge of the exact release times and phasings of hard tasks. This is only generally possible in the case of strictly periodic task sets.

2.3 Acceptance Tests: Guaranteeing Spare Capacity

In the context of fixed priority pre-emptive scheduling, acceptance tests have been developed by Thuel and Lehoczky [83]. They focus on using the Slack Stealing algorithm [59] to provide aperiodic service opportunities whilst ensuring that hard periodic tasks do not miss their deadlines. The Slack Stealing algorithm is combined with an acceptance test which enables on-line guarantees to be given to aperiodic tasks with hard deadlines. The acceptance test operates as follows: first, the aperiodic processing time available between the release and the deadline of a hard aperiodic task is calculated. This is then compared to the execution requirements of the task to determine if it can be guaranteed.

Unfortunately, the analysis given by Thuel and Lehoczky [83] leads to an acceptance test which is insufficient: hard aperiodic tasks may be guaranteed and yet still miss their deadlines. This problem was pointed out in a private communication [36], enabling Thuel and Lehoczky to formulate a correction which was presented at the 1993 Real-Time Systems Symposium [84]. However, subsequent investigation of

the revised acceptance test revealed that it too is insufficient. Detailed analysis of these two acceptance tests is given in appendix A.

In December 1994, Thuel and Lehoczky presented an exact on-line acceptance test [96]. Note, this acceptance test builds upon analysis given in the next chapter and previously published in [39]. We therefore defer review of this test until chapter 6.

2.4 Summary

Analysis of fixed priority scheduling has now progressed to the point where it provides an appropriate framework for the construction of many of today's hard real-time systems. The assumptions and restrictions imposed on hard tasks by early work have been lifted permitting tasks to have realistic timing characteristics such as sporadic arrival patterns, arbitrary deadlines and offsets and allowing interaction via shared resources. Analysis can be tailored to account for the overheads of operating system kernel implementations. Further, scheduling theory has been extended to communications protocols, enabling end-to-end deadlines to be guaranteed in distributed systems with a static allocation of tasks. We therefore conclude that fixed priority pre-emptive scheduling forms a suitable basis for research into the flexible scheduling techniques needed to facilitate the dynamic, adaptive and intelligent behaviour required in the next generation of real-time systems.

Much of this adaptive behaviour can be integrated into real-time systems through the use of techniques aimed at improving the utility of hard real-time services. These techniques require that optional components are executed between the release and the deadline of the hard real-time service which they support. The timely execution of these components relies on the early identification of spare capacity and its provision at a high priority level.

A number of techniques exist which address this problem, including the Polling Server, Deferrable Server, Priority Exchange, Sporadic Server, Extended Priority Exchange and Slack Stealing algorithms. The static analysis underlying the Polling Server and the bandwidth preserving algorithms enables extra capacity to be made available at a high priority level. The Extended Priority Exchange algorithm builds

upon this static analysis by dynamically reclaiming gain time, enabling it to outperform earlier methods. However, all these algorithms are significantly outperformed, under conditions of high load at least, by the Slack Stealing algorithm. This is because it is also able to make spare time (present due to favourable task phasings) available at the highest priority level. This spare time may represent up to 31% [65] of the processors capacity, although nearer 12% could be expected in the average case [61]. Spare time is however, a dynamic quantity which can only be predicted for hard task sets comprising independent periodic tasks. By using a static approach to identifying spare time, the applicability of the Slack Stealing algorithm is limited to such simple task sets. A dynamic approach to Slack Stealing is presented in the next chapter.

To make effective use of spare capacity, the Multiple Versions and Approximate Processing paradigms require that optional computation is guaranteed to complete before a given deadline. An efficient on-line acceptance test is required to furnish such guarantees. Two approximate acceptance tests [84, 83] have been presented for aperiodic tasks with hard deadlines scheduled under the Slack Stealing algorithm. We showed that these tests are insufficient: they may guarantee aperiodic tasks which subsequently miss their deadlines. Exact and sufficient tests are presented in chapter 6.

Chapter 3

Identifying Spare Capacity: An Exact Approach

In this chapter, we focus on the problem of jointly scheduling tasks with hard and soft time constraints. This is an important issue in real-time scheduling, due to the tension between the scheduling requirements of the two types of task: soft tasks typically benefit from being completed as soon as possible, whilst hard tasks must be guaranteed to meet their deadlines. To address this problem, spare capacity needs to be identified and made available as early as possible.

In the previous chapter, we classified spare capacity as *extra capacity*, *gain time* and *spare time*. Recall that extra capacity is defined as that processor time which is not required by the set of hard tasks, even assuming worst case arrival patterns and worst case execution times. In contrast, gain time represents processor time initially assigned to hard tasks but not utilised at run-time, due to invocations of these tasks taking less than their worst case execution times. Finally, spare time corresponds to processor time which is available due to favourable arrival patterns, for example, when not all the hard tasks arrive simultaneously, or when sporadic tasks do not arrive at their maximum rate.

In general, only extra capacity can be identified *a priori*. Both gain time and spare time must be identified on-line. Gain time can be identified at the completion of each hard task, or earlier via Gain Point kernel calls [7]. Efficiently identifying spare time is however, significantly more difficult. Nevertheless, spare time can account for a significant proportion of spare capacity, particularly in systems with sporadic components. Thus algorithms which enable spare time to be utilised as soon as possible hold the promise of significantly improved response times for soft tasks.

In the previous chapter, we reviewed the static Slack Stealing algorithm [59]. This algorithm enables all three forms of spare capacity to be identified for task sets complying with a simple computational model. The performance of the Slack Stealing algorithm in providing responsive soft task scheduling, illustrates the

improvements which can be obtained when full advantage is taken of spare time as well as other forms of spare capacity. Unfortunately, by virtue of using a static method to determine slack, the coverage of this algorithm is limited to hard tasks sets which are strictly periodic, do not exhibit blocking or release jitter and have a manageably small least common multiple of task periods. These restrictions are too severe for next generation real-time systems.

In this chapter, we extend the response time analysis, given by Audsley *et al* in [11], to determine the maximum processing time which may be stolen from periodic or sporadic tasks with hard deadlines, without jeopardising their timing constraints. This new analysis forms the basis for a dynamic Slack Stealing algorithm, which is proved to be optimal, in the sense that, at any given time, it determines the maximum contiguous amount of processing time which can be used immediately by soft tasks. However, allowing a soft task to use this spare capacity in a greedy manner (i.e. as soon as it is available) does not necessarily result in the minimum response time for the soft task [97]. Indeed, we show that when soft or hard task execution times are stochastic, only a clairvoyant algorithm can guarantee to minimise soft task response times.

The dynamic Slack Stealing algorithm derived in this chapter is equivalent to the static Slack Stealing approach of Lehoczky and Thuel, in the limited case of independent periodic tasks. However, by virtue of computing the slack at run-time, the dynamic algorithm is applicable to a more general class of scheduling problems, including hard task sets which contain sporadics and tasks which exhibit release jitter and synchronisation. Further, the dynamic algorithm is able to improve the response times of soft tasks by exploiting run-time information about hard task execution requirements and deadlines, as well as blocking and context switch times.

3.1 Computational Model and Assumptions

The computational model used in the rest of this chapter is the same as that introduced in Section 2.1.1. In addition, the analysis presented in the next section uses the concept of *busy* and *idle periods* [58], (defined in section 2.2.5).

In subsequent sections, the following assumptions apply:

- The hard task set is assumed to be schedulable using fixed priority pre-emptive dispatch with a priority ordering determined by some means, such as Deadline Monotonic priority assignment [63].
- Tasks cannot voluntarily suspend themselves.

3.2 Schedulability Analysis

In this section, we determine the maximum amount of processing time which may be stolen from an invocation of a hard task without causing its deadline to be missed.

For clarity, we initially assume that the task set exhibits no synchronisation or release jitter and that each invocation of a task takes its worst case execution time. Further, we assume that the deadline of each task is less than or equal to its minimum inter-arrival time and the overheads due to context switching and scheduling are zero. In later sections, we relax these assumptions.

Our formulation stems from considering the schedulability of each hard task at some arbitrary time *t*. We assume that at time *t*, the following data is available, for each task, via the operating system, (typically derived from data stored in a task control block):

$l_i(t)$ - The time at which task τ_i was last released.

$x_i(t)$ - The earliest possible next release of task τ_i. Typically $x_i(t) = l_i(t) + T_i$.

$d_i(t)$ - The next deadline of an invocation of task τ_i. (Note, if the current invocation of task τ_i is complete, then $d_i(t) = x_i(t) + D_i$, i.e. $d_i(t)$ is the deadline following the next release).

$c_i(t)$ - The remaining execution time budget for the current invocation of task τ_i. Note, $c_i(t)$ is typically found by subtracting the execution time used from the worst case execution time, C_i. (If at time *t* task τ_i is complete and thus awaiting release, then $c_i(t) = 0$).

Note, $l_i(t)$, $x_i(t)$ and $d_i(t)$ are all measured relative to time *t*.

Initially, we focus on finding the maximum amount of slack time, $S_i(t)$, which may be stolen at priority level *i*, during the interval $[t, t+d_i(t))$, whilst guaranteeing that task τ_i meets its deadline. (Note, $S_i(t)$ may not actually be available for soft task processing due to the constraints on hard deadline tasks with priorities lower than *i*. We return to this point in section 3.2.1.) To guarantee that task τ_i will meet its deadline, we need to analyse the worst case scenario from time *t* onwards. We therefore assume that all tasks τ_j are re-invoked at their earliest possible next release $x_j(t)$ and subsequently with a period of T_j.

In attempting to determine the maximum guaranteed slack, $S_i(t)$, it is instructive to view the interval $[t, t+d_i(t))$ as comprising a number of level *i* busy and idle periods. Any level *i* idle time between the completion of task τ_i and its deadline could be swapped for task τ_i computation without causing the deadline to be missed. Hence the maximum slack which may be stolen is equal to the total level *i* idle time in the interval. We use this result to calculate $S_i(t)$.

Our method for finding the priority *i* idle time relies on two equations: equation (3.1), determines $w_i(t)$, the length of a priority level *i* busy period which starts at time *t*. Equation (3.2) determines the length of a priority level *i* idle period given its start time. By combining these two equations, we are able to iterate over the interval $[t, t+d_i(t))$, totalling up all the idle time, and hence find $S_i(t)$.

We first derive equation (3.1), two components determine the extent of the busy period:

1. The level *i* or higher priority processing outstanding at time *t* given by:

$$\sum_{\forall j \in hp(i)+i} c_j(t)$$

2. The level *i* or higher priority processing released during the busy period, given by:

$$\sum_{\forall j \in hp(i)+i} \left\lceil \frac{w_i(t) - x_j(t)}{T_j} \right\rceil_0 C_j$$

(Note, $(x)_0$ is notational shorthand for $\max(x, 0)$, i.e. the minimum value of

$(x)_0$ is zero).

The second component implies a recursive definition. As the processing released increases monotonically with the length of the busy period, a recurrence relation can be used to find $w_i(t)$:

$$w_i^{m+1}(t) = S_i(t) + \sum_{\forall j \in hp(i)+i} \left(c_j(t) + \left\lceil \frac{w_i^m(t) - x_j(t)}{T_j} \right\rceil_0 C_j \right) \tag{3.1}$$

The term $S_i(t)$ represents priority i soft task processing released at time t and executing in slack time. We return to this shortly.

The recurrence relation begins with $w_i^0(t) = 0$ and ends when $w_i^{m+1}(t) = w_i^m$ or $w_i^{m+1}(t) > d_i(t)$. Proof of convergence follows from analysis of similar recurrence relations by Joseph and Pandya [52] and Audsley *et al* [10]. The final value of $w_i(t)$ defines the length of the busy period. Alternatively, we may view $t + w_i(t)$ as defining the start of a priority level i idle period.

Given the start of a level i idle period, within the interval $[t, t + d_i(t))$, the end of the idle time, which may be converted to slack, occurs either at the next release of a task of priority i or higher (given by the second component of equation 3.2 below) or at the end of the interval (the first component of equation 3.2). Equation (3.2) gives the length, $v_i(t, w_i(t))$, of the priority i idle period.

$$v_i(t, w_i(t)) = \min \left(\begin{array}{l} \left(d_i(t) - w_i(t) \right)_0, \\ \min\limits_{\forall j \in hp(i)+i} \left(\left\lceil \frac{w_i(t) - x_j(t)}{T_j} \right\rceil_0 T_j + x_j(t) - w_i(t) \right) \end{array} \right) \tag{3.2}$$

Combining equations (3.1) and (3.2), our method for determining the maximum slack, $S_i(t)$, proceeds as follows:

1. The slack which may be stolen, $S_i(t)$, is initially set to zero.

2. Equation (3.1) is used to compute the end of a busy period in the interval $[t, t+d_i(t))$.

3. The end of the busy period is used as the start of an idle period by equation (3.2) which returns the length of contiguous idle time.

4. The slack processing, $S_i(t)$ is incremented by the amount of idle time found in step 3.

5. If the deadline of task τ_i has been reached, then the maximum slack which can be stolen is given by $S_i(t)$. Otherwise, we repeat steps 2 to 5.

This method can be implemented as detailed in the algorithm below (3.3).

Algorithm for determining the slack at priority *i*

$S_i(t) = 0$
$w_i^{m+1}(t) = 0$
do while $w_i(t)^{m+1} \le d_i(t)$
$w_i^m(t) = w_i^{m+1}(t)$

$$w_i^{m+1}(t) = S_i(t) + \sum_{\forall j \in hp(i) \cup i} \left(c_j(t) + \left\lceil \frac{w_i^m(t) - x_j(t)}{T_j} \right\rceil C_j \right)$$

if $w_i^m(t) = w_i^{m+1}(t)$
$gap = v_i(t, w_i^m(t))$
$S_i(t) = S_i(t) + gap$
$w_i^{m+1}(t) = w_i^{m+1}(t) + gap + \varepsilon$
end if
enddo
return $S_i(t)$ (3.3)

Note: ε, set to the granularity of time, is a mathematical device used to force the recurrence relation to continue.

It is important to note that this formulation is applicable regardless of whether each hard task is periodic or sporadic. Further, it can be readily extended to handle more complex task characteristics.

The complexity of this approach is O(Kn) where K is the number of iterations and n is the number of hard deadline tasks. Hence, computing the exact slack available at all n priority levels can be done in O(Kn^2) time. We note that the number of iterations, K, depends on the periods and deadlines of the hard tasks, thus the complexity of algorithm (3.3) is pseudo-polynomial.

The operation of algorithm (3.3) is illustrated by the example given in Figure 3.1, using the task set detailed in the table below:

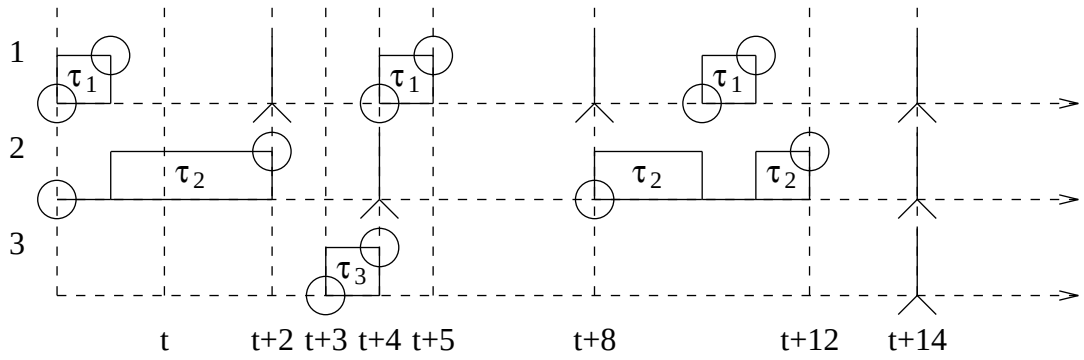
Hard tasks				
Priority	Period	Deadline	Offset	WCET
1	6	4	0	1
2	10	6	0	3
3	12	11	5	1

Time t is taken to be 2 ticks after the initial release of tasks τ_1 and τ_2. Figure 3.1 shows that the exact priority level 3 slack available at time t is 6.

3.2.1 Dynamic Slack Stealing Algorithm

In this section, we use the above analysis as the basis for a dynamic Slack Stealing algorithm. We require that soft tasks are executed as soon as possible, such that the deadlines on all hard real-time tasks are still guaranteed to be met. In the case of strictly periodic tasks as discussed by Lehoczky and Thuel [59], this can be achieved by servicing soft tasks at the highest priority, when there is slack available at *all* priority levels. However, when hard sporadic tasks are considered, there are problems with this approach.

Suppose, at time t there are soft tasks pending and the highest priority hard task in the run-queue is task τ_k. Further, suppose a hard sporadic task with priority higher than k has zero slack (say D = C for this task) and could arrive at any time. This



Calculation of priority level 3 slack at time t
At time t, we have outstanding computation: $c_1(t)=0$, $c_2(t)=2$, $c_3(t)=0$. The next releases of tasks τ_1, τ_2 and τ_3 (relative to t) are at: $x_1(t)=4$, $x_2(t)=8$, $x_3(t)=3$
The next deadline of task τ_3 (relative to t) is at $d_3(t)=14$
Applying algorithm (3.3) to determine the priority 3 slack:
Iteration 1: $w_3^1(t)=c_2(t)=2$
Iteration 2: $w_3^2(t)=2$. Identifying the end of a priority 3 busy period at $t+2$. $v_3(t,2)=x_3(t)-w_3^2(t)=1$, corresponding to the idle period $[t+2,t+3)$. Converting this idle period to slack, we have $S_3(t)=1$ and $w_3^2(t)=3+\varepsilon$.
Iteration 3: $w_3^3(t)=S_3(t)+c_2(t)+C_3=4$
Iteration 4: $w_3^4(t)=4$, identifying the end of a busy period at $t+4$. $v_3(t,4)=x_1(t)-w_3^4(t)=0$, corresponding to an idle period of length zero prior to the release of task τ_1 at $t+4$. $S_3(t)=1$ and $w_3^4(t)=4+\varepsilon$
Iteration 5: $w_3^5(t)=S_3(t)+c_2(t)+C_3+C_1=5$.
Iteration 6: $w_3^6(t)=5$, indicating the end of a busy period at $t+5$. $v_3(t,5)=x_2(t)-w_3^6(t)=3$ corresponding to the 3 idle ticks prior to the release of task τ_2 at $t+8$. $S_3(t)=4$ and $w_3^6(t)=8+\varepsilon$.
Iteration 7: $w_3^7(t)=S_3(t)+c_2(t)+C_3+C_1+C_2=11$.
Iteration 8: $w_3^8(t)=S_3(t)+c_2(t)+C_3+2.C_1+C_2=12$.
Iteration 9: $w_3^9(t)=12$, hence $t+12$ is the end of a busy period. $v_3(t,12)=d_3(t)-w_3^9(t)=2$, corresponding to 2 ticks of idle time prior to the end of the interval at $t+14$. $S_3(t)=6$ and $w_3^9(t)=14+\varepsilon>d_3(t)$, hence iteration is complete. The exact priority level 3 slack at time t is 6.

Figure 3.1: Example calculation of slack

sporadic task may never arrive, preventing slack from ever being available at *all* priority levels.

To avoid the above problem, we use a different criteria for determining when soft tasks may execute. Our analysis guarantees that, provided priority k soft task processing is limited to $S_k(t)$ in the interval $[t,t+d_k(t))$ then hard tasks at priority

levels *k* and higher will meet their deadlines. As we require that the deadlines of *all* hard real-time tasks are met, soft task processing is only permissible at priority *k* whilst there is slack present at level *k* and *all* lower priority levels:

$$\min_{\forall j \in lp(k)} S_j(t) > 0 \tag{3.4}$$

Note, for completeness, when there are no hard tasks runnable, we regard there as being infinite slack available at priority level *n*+1. Provided inequality (3.4) is true, soft tasks can execute at priority *k*, (in preference to hard task τ_k) even though higher priority sporadic tasks apparently have zero slack. If such a sporadic became runnable, it would immediately pre-empt the soft task.

Using the above result, the dynamic Slack Stealing algorithm is formulated as follows: at each time increment when there are soft tasks pending, algorithm (3.3) is used to find the slack available at each priority level lower than or equal to *k*. Where *k* is the priority level of the highest priority runnable hard task. Inequality (3.4) is then used to determine if soft task processing can proceed immediately in preference to task τ_k.

We note that the dynamic algorithm, described above, potentially requires the slack at each priority level to be re-computed at each time increment. We explore this problem further in section 3.2.4.

3.2.2 Optimality of the Dynamic Algorithm

In this section, we prove that the dynamic Slack Stealing algorithm described above is optimal.

Theorem: *For any hard deadline task set scheduled according to a fixed priority scheme and any valid schedule of hard and soft task execution up to an arbitrary time t, from time t onwards, the dynamic Slack Stealing algorithm provides the maximum contiguous amount of processing time which can be made available for soft task processing, amongst all algorithms which are guaranteed to meet all hard task deadlines. (Note, we exclude clairvoyant algorithms which may use prior knowledge of sporadic arrivals and task execution times).*

Proof: We prove the theorem by showing that any alternative algorithm, *A*, which provides a larger amount of soft task processing time cannot guarantee that the deadlines of all the hard tasks will be met.

Let time t^1: $t^1 \geq t$, be the earliest time at or after *t*, when the dynamic Slack Stealing algorithm cannot permit soft task processing to continue. Suppose at time t^1, algorithm *A* permits one further time unit of soft task processing. Let τ_k be the highest priority runnable task at time t^1. The dynamic Slack Stealing algorithm uses the exact upper bound on the slack which may be stolen, this is zero at time t^1. Hence at least one of the hard tasks with priority equal to or lower than *k* has zero slack at time t^1, let this be task τ_i. In continuing to permit soft task processing at time t^1 algorithm *A* has therefore lengthened the worst case priority *i* busy period culminating in the completion of task τ_i beyond its deadline. Algorithm *A* cannot therefore guarantee that the deadline of task τ_i will be met.
□

We note that the dynamic algorithm is optimal for stealing slack from both periodic and sporadic tasks with hard deadlines. In the case of task sets comprising only periodic tasks, its behaviour is equivalent to the static, table driven algorithm of Lehoczky and Thuel [59].

3.2.3 Non-Optimality w.r.t. Minimising Soft Task Response Times

As noted by Tia [97], the optimality of the Slack Stealing algorithm in making spare capacity available does not imply that using slack time immediately will result in minimising soft task response times. In figure 3.2, we give an example of this effect, showing that if soft or hard task execution times are stochastic, then only a clairvoyant algorithm can minimise the response time of even a single soft task.

The example is similar to the one given in Appendix A, figure A.2. The hard task set is defined in the table below. In addition, a soft task, *A* is assumed to arrive at time $t = 0$.

Hard periodic tasks				
Priority	Period	Deadline	Offset	WCET
1	10	5	2	2
2	10	6	0	2

From figure 3.2, it is clear that in order to minimise the response time of soft task *A*, it is necessary to know the exact future execution times of each hard and soft task. Hence only a clairvoyant algorithm can guarantee to minimise soft task response times.

3.2.4 Feasibility of the Dynamic Algorithm

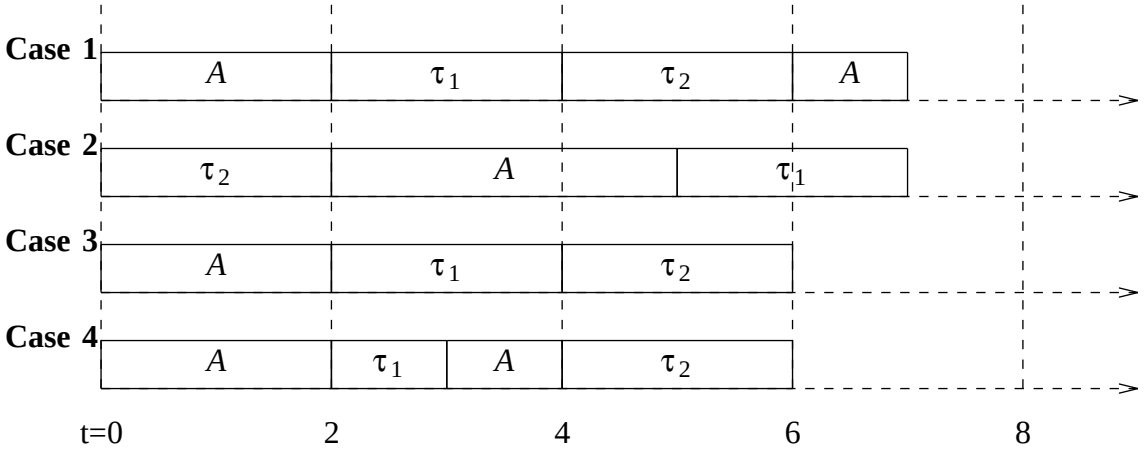
We now seek to reduce the run-time overheads of the dynamic algorithm. To do this, we examine the feasibility of deriving the slack available at some later time t' from the values computed at time t. First, we assume that no hard tasks complete during the interval $[t,t')$ and that each task τ_i is released at $t+x_i(t)$ and subsequently with a period of T_i. By considering the priority level i idle time in the intervals $[t,t+d_i(t))$ and $[t',t+d_i(t))$, it can be seen that, if the processor serviced soft tasks or was idle between t and t' then slack is consumed at all priority levels:

$$\forall j \in lp(1) \ : \quad S_j(t') = S_j(t) - (t'-t) \tag{3.5}$$

Whereas, if the processor was busy with hard deadline task τ_i, then slack is consumed at all priority levels higher than i:

$$\forall j \in hp(i) \ : \quad S_j(t') = S_j(t) - (t'-t) \tag{3.6}$$

Next, we consider the effect of task τ_i completing on the level i slack. The next deadline which task τ_i could miss increases by at least T_i when it completes. As any level i idle time in the interval $[t,t+d_i(t))$ must also be present in $[t,t+d_i(t)+T_i)$, the level i slack cannot be reduced when task τ_i completes. On the contrary, it may well be increased. Hence, for stealing slack from strictly periodic tasks we need only



The slack available at time $t=0$ at priority levels 1 and 2 is $S_1(0)=5$ and $S_2(0)=2$ respectively. At time 0, soft task A arrives with a worst case execution time of 3.

Case 1:
Allowing task A to execute immediately, reduces the slack at priority level 2 to zero at time $t=2$. Task τ_1 must then execute followed by task τ_2, which just meets its deadline at time $t=6$. The soft task can then be permitted to continue executing and completes with a response time of 7.

Case 2:
If task τ_2 is allowed to execute first, completing at time $t=2$, soft task A can then be accomodated in the next 3 time units, completing at time $t=5$. Thus the greedy policy of using slack time as soon as possible results in a longer response time for the soft task (7 v. 5).

Case 3:
However, if the soft task actually only required 2 units of computation time, the greedy policy would result in a shorter response time (2 v. 4).

Case 4:
Similarly, if task τ_1 actually requires only 1 unit of computation (and the soft task requires 3), then the greedy policy would again result in a shorter response time (4 v. 5).

Figure 3.2: Non-optimality of greedy slack assignment.

re-compute the level i slack each time task τ_i completes. (Note, the level i slack present immediately prior to completion provides a lower bound which can be used as an initial value for $S_i(t)$ in algorithm (3.3), reducing the computation required.)

Finally, we consider the effect of sporadic tasks not arriving at their maximum rate. Suppose a sporadic task τ_i is not released at its earliest possible next release time, t^1, but at some later time t^2. Equations (3.5) and (3.6) provide a lower bound on the level i slack at time t^2. However, the deadline of the request at t^2 is later than it would have been had the request occurred at time t^1, potentially increasing the slack. Therefore to ensure that we have the exact upper bound on the slack available, we are compelled to re-evaluate S_i at each time increment in the interval $[t^1, t^2]$. Again, this may be done using the lower bound given by equations (3.5) and (3.6) as an initial value. In addition to the possible increase in priority level i slack, the interference on lower priority tasks may be reduced. (This can be seen by examining the effect of increasing the values of $x_j(t)$ in algorithm (3.3)). To retain optimality, we are therefore compelled to also re-evaluate the slack available at all priority levels lower than i.

With a large number of sporadic tasks, the exact slack available at each priority level can potentially change on each processor clock tick. Thus the slack at each priority level may still need to be re-evaluated at each clock tick. Clearly, this is infeasible in practice. It does, however, provide us with an optimal algorithm for stealing slack from both hard deadline periodic and sporadic tasks. Moreover, it forms the basis for research into approximate slack stealing algorithms with practical utility (see chapter 4).

3.3 Stochastic Timing Attributes

The analysis given in section 3.2 extends the scope of slack stealing algorithms to hard task sets comprising sporadic as well as periodic tasks. In this section, we augment this analysis further, to handle tasks which exhibit stochastic execution times and release jitter.

3.3.1 Reclaiming Gain-Time

Gain time is generated when a guaranteed hard task requires less than its worst case execution time. Gain time may be identified at various points during the execution of a task: for example, at the start by inspecting the input parameters of the task. These can have a critical influence on the execution path, by determining the number of times around a loop etc. Using this run-time information, a less pessimistic, 'specific' worst case execution time can be calculated [93]. Alternatively, gain time may be identified part way through task execution via Gain Points [7] or Milestones [43]. The Milestones separate the task into sections each of which has a worst case execution time. This enables gain time to be identified when a section completes in less than its worst case time. Finally, gain time can be identified when the entire task completes in less than its worst case time.

The analysis given in section 3.2 requires no modification to reclaim gain time. The worst case execution time remaining, $c_i(t)$, is normally found by subtracting the execution time used from the worst case execution time, C_i. However, the methods of identifying gain time described above may be used to provide less pessimistic values for $c_i(t)$, enabling algorithm (3.3) to reclaim gain time as slack. In fact, gain time may be added directly to the slack available without recourse to algorithm (3.3): Suppose gain time g_i is identified at priority level i, then the slack available at level i and all lower priority levels is increased by g_i [59].

Additional slack may also be generated when it is known that the next release of a sporadic task will occur after a time greater than its minimum inter-arrival time. An example of this is a task which carries out some operation when a measured value reaches a certain level. It is assumed that the maximum rate of change of the value is known. Once a measurement has been taken, it is often possible to determine when the task should be next released, so that it is guaranteed not to miss the value reaching its specified level. Thus the task operates with a guaranteed minimum period when the value is close to its critical level and less frequently otherwise.

In the above scenario, an extended earliest possible release, $x_i(t)$, of an invocation may result in increased slack time. Similarly if the deadline on a hard real-time service is dependent on the external environment, it will be guaranteed at a

minimum acceptable level, however, at run-time, the deadline on specific invocations may be increased. Again this is potentially a source of slack time.

It is interesting to note the effect of executing an independent hard deadline task τ_i in slack time, at a raised priority *j*. This results in gain time being generated at level *i*, whilst slack time is reduced at all priority levels $\in lp(j)$. Further, the gain time produced is transformed into slack at priority level *i* and below, illustrating the equivalence between slack and gain time.

$$\forall k : k \in hp(i) \cap lp(j) \quad S_k(t) = S_k(t) - g_i \tag{3.7}$$

Where g_i is the time for which τ_i executes at priority *j*.

3.3.2 Release Jitter

When a task is subject to a bounded delay between its arrival and release, it is said to exhibit release jitter [10]. To incorporate release jitter into our analysis, we need only consider the effect on the earliest possible next release of each task. We also assume that there may only be one invocation of each task present at any given time, hence $D_i + J_i \leq T_i$. For each task τ_i with release jitter, we modify the calculation of $x_i(t)$ as follows:

$$x_i(t) = \left(l_i(t) + T_i - J_i \right)_0 \tag{3.8}$$

3.4 Synchronisation

In this section, we examine how synchronisation between hard tasks and between hard and soft tasks can be incorporated into our analysis.

3.4.1 Resource Sharing between Hard Tasks

First we relax the assumption that the hard tasks are independent. We assume that each invocation of a task τ_i may lock and unlock semaphores according to the Priority Ceiling Protocol [88]. Each invocation of task τ_i may therefore be blocked for at most B_i, the worst case blocking time. Where B_i is equal to the longest

critical section of any lower priority task which accesses a semaphore with a ceiling priority of *i* or higher [88]. For the moment, we assume that for Sha *et al's* analysis of the Priority Ceiling Protocol to hold, we require that no semaphores, used by hard tasks, may be accessed by any task executing in slack time. We return to this point in section 3.4.3.

We now consider incorporating blocking into the dynamic slack stealing algorithm. One simple approach is to assume that each invocation of task τ_i will be blocked for a time B_i. To account for this, we modify the criteria given in (3.4) for stealing slack as follows, (note: *k* is the highest base priority of any task with outstanding computation time):

$$\min_{\forall i \in lp(k)} \left(S_i(t) - B_i \right) > 0 \tag{3.9}$$

This is sufficient to ensure that tasks which are subject to blocking will still meet their deadlines under the dynamic Slack Stealing algorithm. However, this approach is pessimistic in that it assumes task τ_i will always be blocked for the worst case time B_i.

3.4.2 Reclaiming Unused Blocking Time

Below, we describe how less pessimistic blocking factors can be maintained at run-time. We then show how the dynamic Slack Stealing algorithm can be modified to use these blocking factors and thus reclaim unused blocking time.

In the following analysis, we use $b_i(t)$ to denote the worst case time for which task τ_i could be blocked in the interval over which slack is calculated i.e. $[t, t + d_i(t))$. At run-time, prior to the release of an invocation of task τ_i, we assume that $b_i(t) = B_i$. However, when task τ_i is released, it is possible to determine a less pessimistic value for $b_i(t)$. This can be done as follows:

1. First, the system ceiling is inspected. If it is lower than *i*, then the invocation of task τ_i will not be subject to blocking (by lemma 7 in [88]). Hence $b_i(t) = 0$.

2. If the system ceiling is greater than or equal to *i*, then we must determine if there exists a task τ_j with base priority lower than *i* which holds a semaphore with ceiling *i* or higher (by theorem 12 in [88], there is at most one such task). If no such task is present, as is the case when the only task holding a semaphore has a base priority higher than *i*, then the invocation of task τ_i will not be subject to blocking, hence $b_i(t)=0$.

3. However, if there is such a task τ_j, then the invocation of task τ_i could be blocked for at most the worst case remaining execution time of the critical section which task τ_j is in. We denote the remaining execution time of this critical section by $z_j(t)$. Hence $b_i(t)=z_j(t)$.

Execution of task τ_j, at raised priority, from time t^1 to t^2, whilst task τ_i is blocked or pre-empted reduces the worst case remaining blocking time:

$$b_i(t^2)=b_i(t^1)-(t^2-t^1) \tag{3.10}$$

Using the above conditions, dynamic blocking factors can be maintained at run-time as follows: initially, $b_i(0)$ is set to B_i. At each release of task τ_i, $b_i(t)$ is modified according to the three conditions set out above. During the interval between each release and completion of task τ_i, $b_i(t)$ is decremented by an amount of time corresponding to the execution of the blocking task τ_j. Finally, at the completion of each invocation of task τ_i, the interval, over which the priority level *i* slack is calculated, is extended to include the next invocation. Clearly the invocation of task τ_i which has just completed cannot be subject to further blocking, however, the next, as yet unreleased, invocation could be blocked for at most B_i. Hence $b_i(t)$ is reset to B_i each time task τ_i completes.

We now modify the criteria which determines when slack may be stolen to enable unused blocking time to be reclaimed. Slack may be stolen whilst:

$$\min_{\forall i \in lp(k)} \left(S_i(t)-b_i(t) \right) > 0 \tag{3.11}$$

Where, *k* is again the highest base priority of a task with outstanding computation.

Stealing slack according to the criteria given in (3.11) ensures that tasks which are subject to blocking are guaranteed to meet their deadlines under the dynamic Slack Stealing algorithm. This can be demonstrated as follows. For a feasible task τ_i, $S_i(t^c) \geq b_i(0) = B_i$, (where time t^c is a critical instant). Hence initially, the slack which may be stolen according to (3.11) is greater than or equal to zero. An upper bound on the time for which task τ_i may be blocked is provided by $b_i(t)$. This upper bound decreases monotonically, except at each completion of task τ_i. However, at each completion, there can be no outstanding computation of higher priority than *i*. Thus the worst case interference which the next release of task τ_i can be subject to, corresponds to a critical instant (all higher priority tasks released simultaneously). For a feasible task, the additional slack present in the interval from completion to the task's next deadline must be $\geq S_i(t^c) \geq B_i$, hence the slack available can not decrease at the completion of task τ_i. Stealing slack according to the criteria given in (3.11) therefore ensures that there is always sufficient slack on task τ_i to accommodate any actual blocking which takes place.

We note that the dynamic Slack Stealing algorithm is not optimal for task sets which exhibit blocking. This is due to the pessimism inherent in the dynamic blocking factors between the completion and release of each task. The dynamic Slack Stealing algorithm is, however, able to effectively reclaim blocking time. This is illustrated by figure 3.3 which provides an example of how reclaiming unused blocking time improves the response times of soft tasks. The example is based on the task sets detailed below. All three hard tasks are assumed to share a resource R1 which has a ceiling priority of 1. The maximum time for which R1 may be held by each task is given in the column marked *Z*. Further the values in column *R* are the worst case response times for each task, whilst the values in column *O* are the offsets.

Hard tasks							
Priority	O	T	D	Z	C	B	R
1	2	10	6	1	3	3	6
2	7	15	12	3	5	2	10
3	0	22	22	2	4	0	15

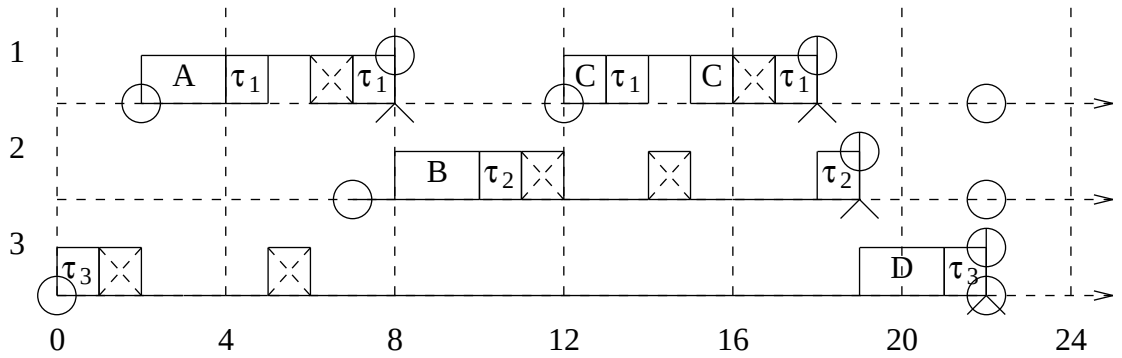
Soft tasks		
Name	Arrival time	Execution time
A	2	2
B	7	2
C	12	2
D	17	2

This example illustrates two important points. First, if a low priority task has already executed part of its critical section when it is pre-empted then the blocking factor is reduced, leading to increased slack on higher priority tasks. Second, the completion of a critical section in less than its worst case time increases the slack on all tasks with priorities less than the active priority of the task executing its critical section.

The analysis detailed in this section can also be applied to the Ceiling Semaphore Protocol [82]. In contrast with the Priority Ceiling Protocol, under the Ceiling Semaphore Protocol, the active priority of a task is raised immediately, on locking a semaphore, to the priority ceiling of that semaphore. This results in larger response times for soft tasks than the Priority Ceiling protocol, however the Ceiling Semaphore Protocol has the advantage that fewer context switches are required.

3.4.3 Resource Sharing between Hard and Soft Tasks

We now consider the situation where a soft task shares one or more semaphores with the hard tasks. Suppose that the soft task wishes to lock a semaphore with ceiling priority *j*. The deadlines of tasks at priority levels *j* or lower could be jeopardised by



t=0: $S_1(0)=5$, $b_1(0)=3$, $S_2(0)=8$, $b_2(0)=2$, $S_3(0)=7$, $b_3(0)=0$.
No soft tasks are pending. Task τ_3 executes, requesting resource R1 at **t=1**.

t=2: Task τ_1 is released. The system ceiling is 1. Task τ_1 can be blocked for at most the remainder of task τ_3's critical section.
$S_1(2)=3$, $b_1(2)=1$, $S_2(2)=6$, $b_2(2)=2$, $S_3(2)=7$, $b_3(2)=0$.
Soft task A arrives and is executed in preference to task τ_1.

t=4: $S_1(4)=1$, $b_1(4)=1$, $S_2(4)=4$, $b_2(4)=2$, $S_3(4)=5$, $b_3(4)=0$.
Task τ_1 executes and requests R1 at **t=5**. The request is denied. Task τ_1 is blocked by task τ_3. τ_3 then executes, completing its critical section at **t=6**.

t=6: $S_1(6)=0$, $b_1(6)=0$, $S_2(6)=3$, $b_2(6)=2$, $S_3(6)=5$, $b_3(6)=0$.
Task τ_1 is granted R1 and executes inside its critical section.

t=7: Task τ_2 is released, as no resources are locked, $b_2(7)=0$. Soft task B arrives. $S_1(7)=0$, $b_1(7)=0$, $S_2(7)=3$, $b_2(7)=0$, $S_3(7)=5$, $b_3(7)=0$.

t=8: Task τ_1 completes. $S_1(8)=7$, $b_1(8)=3$. Soft task B executes.

t=10: $S_1(10)=5$, $b_1(10)=3$, $S_2(10)=1$, $b_2(10)=0$, $S_3(10)=3$, $b_3(10)=0$.
Task τ_2 executes, requesting and being granted R1 at **t=11**.

t=12: Task τ_1 is released. The system ceiling is 1. Task τ_1 may be blocked for the remainder of task τ_2's critical section.
$S_1(12)=3$, $b_1(12)=2$, $S_2(12)=1$, $b_2(12)=0$, $S_3(12)=3$, $b_3(12)=0$.
Soft task C arrives and executes for 1 tick in preference to task τ_1.

t=13: $S_1(13)=2$, $b_1(13)=2$, $S_2(13)=0$, $b_2(13)=0$, $S_3(13)=2$, $b_3(13)=0$.
τ_1 commences execution and requests R1 at **t=14**. The request is denied.

t=14: Task τ_2 executes, completing its critical section 1 tick early at **t=15**.

t=15: The gain time produced by task τ_2 increases the slack on tasks τ_2 and τ_3 and reduces the blocking of task τ_1. Soft task execution is again permissible
$S_1(15)=1$, $b_1(15)=0$, $S_2(15)=1$, $b_2(15)=0$, $S_3(15)=3$, $b_3(15)=0$.

t=16: $S_1(16)=0$, $b_1(16)=0$, $S_2(16)=0$, $b_2(16)=0$, $S_3(16)=2$, $b_3(16)=0$.
Task τ_1 resumes and is granted R1.

t=17: Soft task D arrives. Task τ_1 executes to completion followed by task τ_2.

t=18: $S_1(18)=7$, $b_1(18)=3$, $S_2(18)=0$, $b_2(18)=0$, $S_3(18)=2$, $b_3(18)=0$.

t=19: $S_1(19)=6$, $b_1(19)=3$, $S_2(19)=5$, $b_2(19)=2$, $S_3(19)=2$, $b_3(19)=0$.
Soft task D executes for 2 ticks in preference to task τ_3.

t=22: Task τ_3 completes.

Figure 3.3: Reclaiming blocking time

being blocked by the soft task. To ensure that this cannot happen, the soft task must be guaranteed sufficient processing time to complete its critical section, prior to the continued execution of any tasks of priority *j* or lower. Only then can it be allowed to lock the semaphore. If the length of the critical section of the soft task is bounded by *c*, the ceiling priority of the semaphore is *j* and the time of the request is *t*, then the soft task is only allowed to succeed in locking the semaphore if:

$$\min_{\forall i \in lp(j)} \left(S_i(t) - b_i(t) \right) \geq c \tag{3.12}$$

Provided the above condition holds, only hard tasks with priority higher than *j* can pre-empt execution of the soft task's critical section. From the definition of the priority ceiling, none of these higher priority tasks use the semaphore and hence none of them can be blocked by the soft task. Further, the soft task is guaranteed execution time of at least *c* in preference to hard tasks of priority *j* and lower. It is therefore able to complete its critical section before being pre-empted by any task of priority *j* or lower. Thus using the criteria given in (3.12) resources may be shared between hard and soft tasks without increasing the worst case blocking of any hard task.

Alternatively, the worst case blocking time which hard tasks may be subject to due to soft tasks locking semaphores could be included in the appropriate blocking factors B_i used in the determination of hard task feasibility. This may however adversely affect the feasibility of the hard task set.

3.5 Context Switches

In this section, we examine how context switch overheads can be incorporated into our analysis of slack time. We also detail how context switches accounted for in off-line feasibility analysis but not executed at run-time can be reclaimed. Further, we discuss using the dynamic Slack Stealing algorithm to reduce the number of context switches which take place.

3.5.1 Incorporating Context Switch Overheads

Overheads due to context switching between hard tasks may be accounted for by sub-summing the cost of two context switches into the worst case execution time of each hard task [26]. Assuming that Ω is the worst case context switch time and Ψ_i is the computed worst case execution time of task τ_i, then:

$$C_i = 2\Omega + \Psi_i \tag{3.13}$$

To incorporate context switch overheads into the dynamic Slack Stealing algorithm, we must also account for context switches between hard and soft tasks. Two situations need to be considered.

1. Hard task τ_j is pre-empted by soft task processing and then resumes.

2. Hard task τ_j is pre-empted by soft task processing. A higher priority hard task τ_k is then released. Soft task processing is eventually curtailed and task τ_k begins executing.

In the first case, there must be at least sufficient slack available at priority levels j and below to allow for two context switches: from hard task τ_j to the soft task and back again, (see figure 3.4, case 1).

$$\min_{\forall i \in lp(j)} \left(S_i(t) - b_i(t) \right) \; > \; 2\Omega \tag{3.14}$$

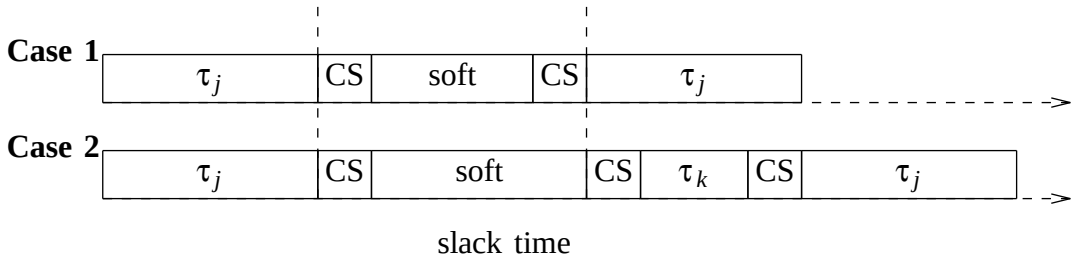
Note, in practice, the switch to soft task processing would not be made unless there was also sufficient slack to perform a useful amount of work.

Once soft task processing has commenced, then it may continue until,

$$\min_{\forall i \in lp(j)} \left(S_i(t) - b_i(t) \right) \; = \; \Omega \tag{3.15}$$

leaving sufficient slack for the context switch back to task τ_j.

In the second case, the context switch between the soft task and task τ_k is already included in the worst case computation time of task τ_k. Thus even if the slack available at priority level *k* is zero when task τ_k is released, the context switch away from soft task processing is accounted for. Hence, whenever a hard task τ_k is released and is the highest priority runnable task, but the processor is busy with soft tasks, then one context switch can be reclaimed at priority *k* (see figure 3.4, case 2).

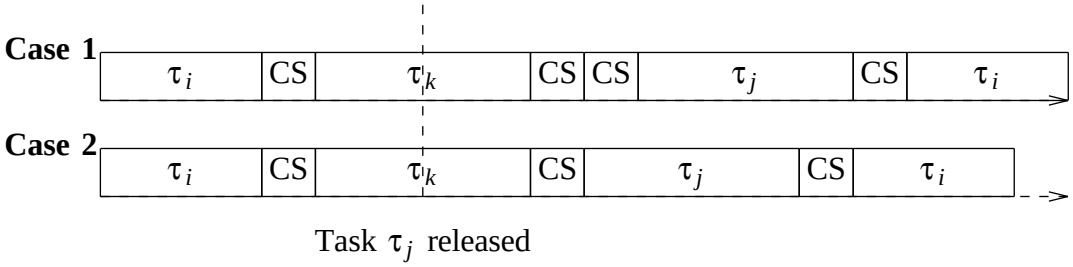


CS = Context switch
Case 1:
Execution of hard task τ_j is interrupted by soft task processing.
Two context switches take place in slack time.
Case 2:
The release of a higher priority task τ_k whilst soft task processing is proceeding allows one context switch to be reclaimed. Only one context switch takes place in slack time.

Figure 3.4: Context switches in slack time.

3.5.2 Reclaiming Unused Context Switch Time

At run-time, the worst case number of context switches (two per hard task) does not always occur. In fact we may identify an unused context switch each time a hard task τ_j is released and is *not* the highest priority runnable task. Consider the time budget C_j, for task τ_j, as comprising three components; a context switch to task τ_j from a lower priority task, the worst case execution time of task τ_j and a context switch back to a lower priority task. When task τ_j is released and is not the highest priority runnable task, the first of these two context switches is no longer required. The context switch to task τ_j is accounted for in the time budget of a higher priority task τ_k which proceeds the execution of task τ_j. See figure 3.5.
Effectively, gain time, equivalent to the context switch time Ω, is produced at priority level *j*.



CS = Context switch
Case 1:
Two context switches per task are assumed by off-line analysis.
Case 2:
At run-time, the context switch from task τ_k to task τ_j is accounted for in C_k. The unused context switch may be reclaimed.

Figure 3.5: Reclaiming unused context switches.

When task τ_j is released at time t and is not the highest priority runnable task at that time, then the slack at priority levels j and lower is increased:

$$\forall i \in lp(j) \qquad S_i(t) = S_i(t) + \Omega \tag{3.16}$$

and the worst case outstanding computation time for task τ_j is decreased:

$$c_j(t) = C_j - \Omega \tag{3.17}$$

Thus reclaiming the unused context switch time.

3.5.3 Reducing the Number of Context Switches

In some hard real-time systems, the cost of context switching is high. For example, in the domain of hard real-time disk scheduling [98], the context switch time represents the time taken to move the disk head to the required track plus the rotational delay before the desired data block can be read. Typically, the context switch time may be 25ms [28]. By comparison, reading a data block takes 1ms. In such systems, it is desirable to minimise the number of context switches and hence

increase the spare capacity available for soft disk accesses. Using the dynamic Slack Stealing algorithm, this may be achieved as follows. Suppose a hard stream τ_j is accessing disk blocks when a read request from a higher priority stream τ_k arrives. Normally, stream τ_j would be pre-empted. However, if there is enough slack available, the request of stream τ_j can be completed prior to servicing stream τ_k, saving one context switch. The time required to complete the request of stream τ_j is given by $c_j(t)$. Noting that servicing stream τ_j reduces the slack at all priority levels higher than *j*, the disk drive may continue to service stream τ_j in preference to stream τ_k provided:

$$\min_{\forall i \in lp(k) \cap hp(j)} S_i(t) \geq c_j(t) - \Omega \tag{3.18}$$

We note that even if a request from stream τ_m (of higher priority than *k*) arrives whilst stream τ_j is being serviced at priority *k* in slack time, causing stream τ_j to be pre-empted, one context switch has still been saved.

In general, the context switch or seek time for a disk drive depends on the relative track positions of the data blocks requested. Using this information, context switch overheads may be further reduced by servicing requests, in slack time, by track number rather than priority.

3.6 Arbitrary Deadlines

In this section, we extend our analysis to tasks with arbitrary deadlines (i.e. $D_i > T_i$), by combining the analysis given by Tindell *et al* [102, 105] with that presented in section 3.2. To find the worst-case response time of a task τ_i the arbitrary deadline analysis examines a number of priority *i* busy periods and takes the largest response time corresponding to each of these. To apply the arbitrary deadline analysis to the algorithm for determining slack (3.3), we must check when evaluating a priority *i* busy period to see if the end of the busy period exceeds the subsequent release of task τ_i, denoted by $x_i^*(t) = d_i(t) - D_i + T_i$. (Note, if task τ_i is complete at time *t*, then $x_i^*(t) = x_i(t) + T_i$, otherwise, $x_i^*(t) = x_i(t)$). If $W_i(t) > x_i^*(t)$ then we must determine if the level *i* slack is limited by the invocation of task τ_i released at $x_i^*(t)$. This is done by examining another priority *i* busy period starting at time *t* but

including additional computation time C_i. If this new busy period exceeds the subsequent invocation of task τ_i (i.e. $w_i(t) > x_i^*(t) + T_i$), then we must check if the slack is limited by the invocation of task τ_i released at $x_i^*(t) + T_i$. To do this, we examine a further busy period, again starting at time *t*, but including additional computation time $2C_i$. In general, if the $(q-1)$th busy period exceeds $x_i^*(t) + qT_i$ we need to check if the level *i* slack is limited by the invocation of task τ_i released at $x_i^*(t) + qT_i$. The slack available at priority level *i* is then the minimum of the slack on each of the *q* invocations examined. This value is used as before, in inequality (3.4) to determine whether soft task processing may take place.

In determining the slack available, the level *i* busy period is extended to $d_i(t) + qT_i$ which given $D_i > T_i$, always exceeds the next release of task τ_i. The sequence of busy periods which require examination is therefore potentially infinite. We address this problem in appendix B.

3.7 Summary

This chapter extended previous research into Slack Stealing algorithms. Exact analysis was presented which allows the slack available on both hard periodic and sporadic tasks to be calculated. This analysis forms the basis of a dynamic slack stealing algorithm which was proved to be optimal. Further, we extended our analysis to cater for tasks which have stochastic execution times and release jitter, tasks which exhibit blocking and finally tasks with arbitrary deadlines. We also incorporated context switch overheads into our model. Moreover, we augmented the dynamic slack stealing algorithm to reclaim unused execution time, blocking time and context switch time.

The exact approach to identifying spare capacity described in this chapter has pseudo polynomial complexity. It is therefore impractical for use at run-time in real systems. We address this problem in the next chapter.

Chapter 4

Identifying Spare Capacity: Approximate Slack Stealing

Theoretically, optimal Slack Stealing algorithms offer significant performance improvements over previously state of the art techniques such as the Extended Priority Exchange algorithm [92]. However, in practice this performance advantage may not be realised. The optimal static Slack Stealing algorithm [59] relies on a pre-computed table to define the slack present on each invocation of a hard task. Hence the application of this approach is limited to systems comprising strictly periodic tasks. In contrast, the optimal dynamic Slack Stealing algorithm developed in the previous chapter, calculates the available slack at run-time. Hence it is valid for a wider range of scheduling problems, including systems containing hard sporadic tasks. However, the run-time overheads of this approach are pseudo-polynomial in complexity, making it infeasible in practice.

In this chapter, we present two approximate methods of determining slack, one static, the other dynamic. These methods address the space and time complexity problems inherent in the optimal algorithms. Moreover, they form the basis for approximate Slack Stealing algorithms which offer close to optimal performance with practical utility.

The performance of the approximate Slack Stealing approaches is compared to that of background, Extended Priority Exchange and optimal Slack Stealing methods via simulation. Comparisons are made using hard task sets with a range of utilisation levels, different proportions of periodic, sporadic and adaptive tasks, and varying levels of gain time. Finally, we discuss overheads and implementation issues.

4.1 Approximate Slack Stealing Algorithms

The computational model and notation used in this chapter is the same as that introduced in section 2.2.1. In addition, we use the notation for run-time variables introduced in chapter 3. In particular, recall that:

$l_i(t)$ - is the time, relative to t, at which task τ_i was last released.

$x_i(t)$ - is the earliest possible next release of task τ_i.

$d_i(t)$ - is the next deadline on an invocation of task τ_i.

$c_i(t)$ - is the remaining execution time budget for the current invocation of task τ_i.

$S_i(t)$ - is the slack at priority level i, corresponding to the maximum additional interference which task τ_i may be subject to without jeopardising its next deadline.

The following assumptions also apply:

- The hard deadline task set is assumed to be schedulable using fixed priority pre-emptive dispatching with a priority ordering determined by some means such as Deadline Monotonic priority assignment [63].

- Tasks cannot voluntarily suspend themselves.

- Task deadlines are assumed to be less than or equal to their periods.

Further, for the sake of clarity, we assume that tasks do not exhibit blocking or release jitter and that context switch times are zero. These characteristics may however be taken into account by all the approximate Slack Stealing algorithms discussed, using the techniques given in the previous chapter. The results presented in this chapter, are therefore applicable to systems with these characteristics.

4.1.1 Generic Slack Stealing Algorithm Formulation

Below, we formulate the generic part of a run-time Slack Stealing algorithm which may be combined with either static or dynamic, exact or approximate methods of calculating slack to form various optimal or approximate Slack Stealing algorithms.

1. If the processor executes task τ_i during the interval $[t^a, t^b)$ then the slack at all priority levels higher than i is reduced:

$$\forall j \in hp(i) \ : \ S_j(t^b) = S_j(t^a) - (t^b - t^a) \tag{4.1}$$

2. If the processor is idle or executes soft tasks during the interval $[t^a, t^b)$ then the slack at all priority levels is reduced:

$$\forall j \in lp(1) \ : \ S_j(t^b) = S_j(t^a) - (t^b - t^a) \tag{4.2}$$

3. If gain time g_i is identified at priority level i at time t^b then the slack at priority i and all lower levels is increased:

$$\forall j \in lp(i) \ : \ S_j(t^b) = S_j(t^b) + g_i \tag{4.3}$$

4. If task τ_k is the highest priority runnable task, then soft task processing may take place in preference to τ_k provided that:

$$\min_{\forall j \in lp(k)} S_j(t) > 0 \tag{4.4}$$

Equations 4.1 to 4.3 represent a generic set of methods for maintaining the slack at each priority level. These methods accurately maintain the priority level i slack provided that task τ_i does not complete during the interval $[t^a, t^b)$ and that all tasks of priority i or higher are released at their earliest next release time and periodically thereafter. If these conditions do not hold, then the above equations still maintain valid lower bounds on the slack available at each priority level, although the degree of pessimism in these bounds is increased. Hence, for strictly periodic hard task sets, optimal Slack Stealing can be achieved by re-calculating the exact priority level i slack each time task τ_i completes, whilst using equations 4.1, 4.2 and 4.3 to maintain the slack counters at other times. In contrast, for task sets containing hard deadline sporadics, optimal Slack Stealing is only possible if the exact slack at all priority levels is recalculated every clock tick.

We now derive approximate methods of calculating the available slack. Our aim is to use these approximations to produce Slack Stealing algorithms which are applicable to a wide range of task sets, offer significantly better performance than previously published methods such as the Extended Priority Exchange algorithm (summarised in chapter 2), and have low memory and execution time overheads.

First, we derive a static approximation which uses the time between the completion of a task and its next deadline to find a lower bound on the available slack. We then describe a dynamic approximation which requires $O(n)$ time to calculate a lower bound on the slack at each priority level. Further, we show how the accuracy of this approximation can often be improved through the computation of effective deadlines. Finally, we outline how our approximations can be used to construct three approximate slack stealing algorithms.

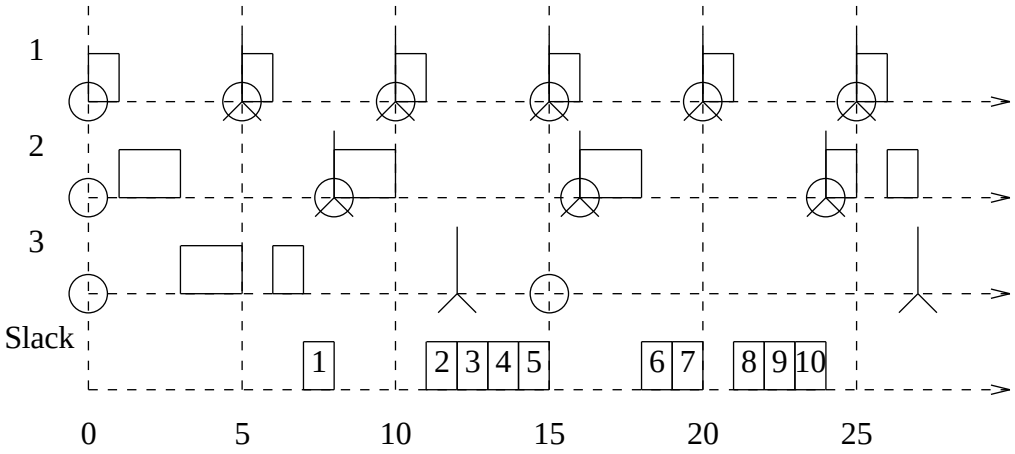
4.1.2 Static Approximation

Recall that for hard task sets comprising only periodic tasks, the slack available at priority level i only needs to be recalculated each time task τ_i completes. We therefore seek to find a lower bound on the level i slack (denoted by S_i^C) available immediately after completion of task τ_i. This is equivalent to the priority level i idle time present in the interval between the completion of task τ_i and the deadline on its next invocation. The lower bound on level i idle time in this interval is dependent on the length of the interval, δ. We use $S_i^C(\delta)$ as notation for a function of δ which returns this lower bound. Note that $D_i + T_i \geq \delta \geq T_i$, corresponding to task τ_i finishing as early or as late as possible. At each completion of task τ_i, no tasks of higher priority than i can be runnable, hence the least level i idle time in the interval occurs when all tasks of higher priority than i are released immediately task τ_i completes. We use this result to construct a table off-line, containing the values of $S_i^C(\delta)$ for each possible value of δ.

$S_i^C(\delta)$ is equivalent to the level i idle time in the interval $[0,\delta)$, assuming a critical instant for all tasks of higher priority than i at time $t=0$ and including precisely one invocation of task τ_i. A variation of algorithm 3.3 (chapter 3) can be used to calculate $S_i^C(\delta)$ and hence construct the table.

Figure 4.1 illustrates the construction of a table of slack values for priority level 3, given the task set detailed in the table below.

Hard tasks			
Priority	Period	Deadline	wcet
1	5	5	1
2	8	7	2
3	15	12	3



To construct a table of priority level 3 slack, we examine the above schedule, starting at a critical instant, extending to $T_3 + D_3$ (= 27) and including a single invocation of task τ_3.
The slack available at the completion of task τ_3, δ ticks prior to the deadline of the next invocation can be read off for the range of values which δ can take i.e. T_3 to $T_3 + D_3$ (15 to 27).
The table of values represented as a set of pairs $(\delta , S_3^C(\delta))$ is as follows. {(15,5),(16,5),(17,5),(18,5),(19,6),(20,7),(21,7),(22,8), (23,9),(24,10),(25,10),(26,10),(27,10)}.
At run-time, if task τ_3 completes say 20 ticks before the deadline on its next invocation, then the appropriate table entry is accessed and the priority level 3 slack set to 7.

Figure 4.1: Constructing a table of slack values

We note that the table of values required for this approximation differs from that used by the static Slack Stealing algorithm [59]: only a limited number of table entries are required per hard task (see section 4.3.2). By comparison, the table used

by the static Slack Stealing algorithm requires entries for each invocation of each hard task over the LCM of task periods.

4.1.3 Dynamic Approximation

To calculate a lower bound on the level *i* slack available at some arbitrary time *t*, we modify the sufficient but not necessary off-line feasibility test given by Audsley [7] for use at run-time. Recall that the exact level *i* slack available at time *t* is equivalent to the priority *i* idle time present in the interval $[t, t+d_i(t))$. We now calculate a lower bound on this priority *i* idle time. Consider a task τ_j of higher priority than *i*. The maximum interference, $I_j(t, d_i(t))$, due to task τ_j which could fall into the interval $[t, t+d_i(t))$ is given by:

$$I_j(t,e) = c_j + f_j(t,e)\,C_j + \min\left(C_j, (e - x_j(t) - f_j(t,e)\,T_j)_0\right) \tag{4.5}$$

Where *e* is the extent of the interval ($e = d_i(t)$) and $f_j(t,e)$ is the number of complete invocations of task τ_j which fall into the interval.

$$f_j(t,e) = \left\lfloor \frac{e - x_j(t)}{T_j} \right\rfloor_0 \tag{4.6}$$

Thus the interference $I_j(t,e)$ generally comprises three components; task τ_j computation outstanding at time *t*, $f_j(t,e)$ complete invocations of task τ_j and a partially complete final invocation.

A lower bound on the level *i* slack is given by the length of the interval less the maximum interference from all tasks of priority *i* and higher falling into that interval.

$$S_i(t) = \left(e - \sum_{\forall j \in hp(i)+i} I_j(t,e)\right)_0 \tag{4.7}$$

This approximation is pessimistic in that it assumes that all the execution time of each task τ_j which would fall into the interval, if only task τ_j were present, will

actually fall within the interval, when all the hard tasks are considered. The degree of pessimism inherent in this approximation can often be reduced through the calculation of an *effective deadline* [7]. An effective deadline (e) is a point in time within the interval $[t, t+d_i(t))$ after which, the processor is continuously busy with level i or higher priority computation until the end of the interval. Hence there can be no level i idle time present during $[t+e, t+d_i(t))$. Note, we deliberately overload the symbol e as the effective deadline is synonymous with the end of the interval over which slack need be determined.

An effective deadline may be calculated by iterating over all tasks of priority i and higher as follows: initially, e is set to $d_i(t)$. Each task of priority i or higher is then inspected in turn. If the last release of task τ_j is within C_j of the end of the interval, then the last release time of task τ_j forms an effective deadline on the execution of task τ_i.

$$
\begin{aligned}
&e = d_i(t) \\
&\textbf{loop} \; \forall j \in hp(i) \\
&\textbf{if} \quad x_j(t) + f_j(t,e)\,T_j + C_j \geq e \quad \textbf{then} \quad e = x_j(t) + f_j(t,e)\,T_j \quad \textbf{endif} \\
&\textbf{endloop}
\end{aligned}
\tag{4.8}
$$

The effective deadline calculated above is then used as the extent of the interval in equation (4.5) to give an improved estimate of the slack available.

Figure 4.2 illustrates the approximate calculation of slack at priority level 4 at some arbitrary time t for the task set given in the table below. First slack is calculated using the next deadline of task τ_4 as the end of the interval. This gives a lower bound of 5. An effective deadline is then calculated and substituted in the approximate slack calculation yielding an improved lower bound of 6. The exact priority level 4 slack at time t is in fact 7. We note, that task execution intervals are shown as they are considered by the approximation, not as the tasks would be scheduled.

Hard tasks			
Priority	Period	Deadline	wcet
1	8	8	3
2	19	19	2
3	23	23	2
4	28	28	2

We now present three approximate stealing algorithms which make use of the methods of calculating slack detailed above.

4.1.4 SASS Algorithm

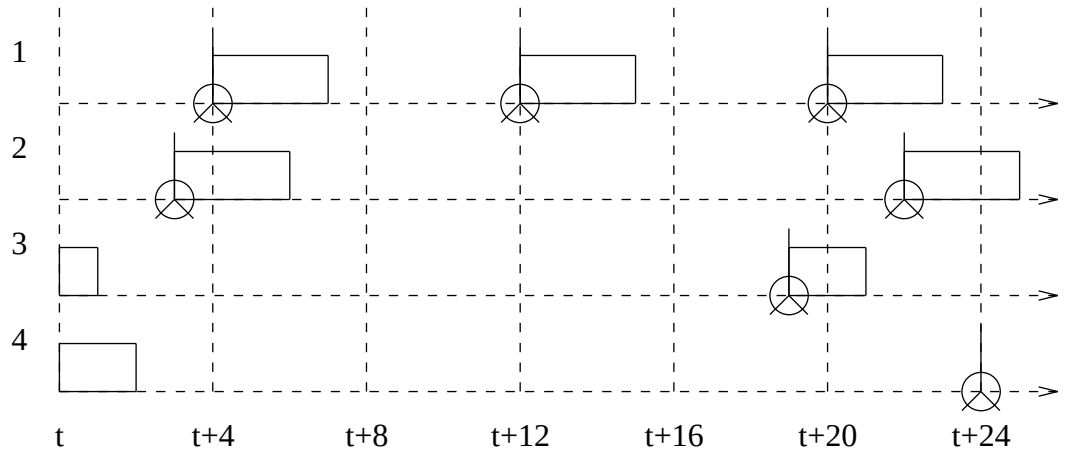
The SASS (Static Approximate Slack Stealing) algorithm uses the static approximation given in section 4.1.2. The run-time operation of the SASS algorithm is based on the generic methods described in section 4.1.1. At each completion of a hard task τ_i, the time to the deadline $d_i(t)$, of the next invocation of task τ_i is used to look up the value of $S_i^C(d_i(t))$ stored in the appropriate table. The level i slack counter is then set to this value.

4.1.5 DASS Algorithm

The DASS (Dynamic Approximate Slack Stealing) algorithm is similar to the SASS algorithm. Again, the level i slack is re-calculated each time task τ_i completes, however, this time it is done using the dynamic approximation described in section 4.1.3.

4.1.6 PASS Algorithm

The PASS (Periodic Approximate Slack Stealing) algorithm combines both the static and dynamic methods of calculating slack. The static method is used to calculate slack at each task completion, whilst the dynamic approximation is used to periodically re-evaluate the slack at every priority level. Recall that to maintain the



Approximate calculation of priority level 4 slack at time t
At time t, we have outstanding computation: $c_1(t)=0$, $c_2(t)=0$, $c_3(t)=1$. and $c_4(t)=2$.
The next releases of tasks τ_1, τ_2, τ_3 and τ_4 (relative to t) are at: $x_1(t)=4$, $x_2(t)=3$, $x_3(t)=19$, $x_4(t)=24$
The next deadline of task τ_4 (relative to t) is at $d_4(t)=24$, hence the extent of the interval over which slack is calculated: $e=24$.
$f_1(t,24)=2$, hence $I_1(t,24)=c_1+2C_1+\min(C_1,4)=9$.
$f_2(t,24)=1$, hence $I_2(t,24)=c_2+C_2+\min(C_2,2)=5$.
$f_3(t,24)=0$, hence $I_3(t,24)=c_3+\min(C_3,5)=3$.
$f_4(t,24)=0$, hence $I_4(t,24)=c_4+\min(C_4,0)=2$.
Lower bound on the priority level 4 slack at time t is (24 - 9 - 5 - 3 - 2) = 5.
Improved approximation using effective deadlines.
$e_1=24$, $e_2=22$, $e_3=22$, $e_4=22$.
$f_1(t,22)=2$, hence $I_1(t,22)=c_1+2C_1+\min(C_1,2)=8$.
$f_2(t,22)=0$, hence $I_2(t,22)=c_2+\min(C_2,19)=3$.
$f_3(t,22)=0$, hence $I_3(t,22)=c_3+\min(C_3,3)=3$.
$f_4(t,22)=0$, hence $I_4(t,22)=c_4+\min(C_4,0)=2$.
Lower bound on the priority level 4 slack at time t is (22 - 8 - 3 - 3 - 2) = 6.
Note, the exact priority level 4 slack at time t is 7.

Figure 4.2: Approximate calculation of slack

exact slack at all priority levels in a system containing hard sporadic tasks potentially requires that the slack at every priority level be re-evaluated every clock tick. Clearly this is infeasible in practice. However, the PASS algorithm approximates to this optimal approach. Varying the period of the PASS algorithm enables the overhead of slack calculation to be traded off against a decrease in the responsiveness of soft tasks. A very short period minimises the theoretical response times of soft tasks at the expense of a large overhead. Whilst a very long period minimises the overhead

and increases response times. This trade off is examined further in the next two sections.

4.2 Performance Evaluation

In this section, we compare the performance of the above approximate algorithms to that of the optimal dynamic Slack Stealing algorithm, Background scheduling and Extended Priority Exchange algorithms. The Background and optimal Slack Stealing algorithms were chosen as they effectively bound the range of expected performance. The Extended Priority Exchange algorithm represents the most effective algorithm previously developed, which is applicable to task sets which are not strictly periodic.

First we outline the assumptions made in our scheduling simulation and discuss the generation of the hard and soft task sets used as test data. We then present results for hard task sets comprising only periodic tasks. These are followed by results for tasks sets containing sporadic and adaptive tasks, and tasks which exhibit stochastic execution times.

4.2.1 Simulation

The criteria used to evaluate the performance of the various scheduling algorithms is the mean response time of soft tasks. This criteria is used as it gives a measure of how early spare capacity can be made available: a key performance criteria, which reflects an algorithm‘s effectiveness.

In our simulations, the soft task load was represented by a FIFO queue of tasks. The execution requirements of soft tasks were exponentially distributed from 1 to 16 units of processing time. The arrival times of soft tasks followed a uniform distribution over the duration of each experiment - 100000 ticks (i.e. a Poisson arrival process). The number of soft tasks was varied to produce a range of mean processor loadings (plotted on the x-axis of the graphs).

The hard task load was simulated using groups of task sets with utilisation levels of 30, 50, 70 and 90%. The results presented in subsequent sections are the averages over a group of ten task sets. Each hard task set had an *exponential* composition of task periods in the range 40 to 2560 ticks. Each hard task set was

generated as follows. First, the periods of the tasks were chosen at random from the desired range. Deadlines were again randomly chosen, but constrained to be less than or equal to the task's period. The tasks were then sorted into deadline monotonic priority order. Next, random computation times were assigned, highest priority first. The computation times were constrained such that the partial task set remained feasible according to an exact schedulability test. Finally tasks sets with a utilisation level differing by more than 1% from that required were discarded.

To simulate sporadic task sets, the same hard task sets were used, however, certain tasks were now designated sporadic. Each task τ_i designated as sporadic had a probability of arrival per tick of $1/T_i$ once its minimum inter-arrival time had expired. The mean inter-arrival time of each sporadic task was thus $2T_i$ compared to a minimum inter-arrival time of T_i.

To simulate adaptive tasks, the same hard task sets were again used. In this case, each task designated as adaptive, determined its next release time at completion, allowing this information to be used by the Slack Stealing algorithms. The inter-arrival time of the adaptive tasks followed a uniform distribution from T_i to $2T_i$.

Tasks with stochastic execution times were simulated as producing gain time upon completion. The amount of gain time produced by each task was uniformly distributed from 0-25% and 0-50% of the task's worst case execution time.

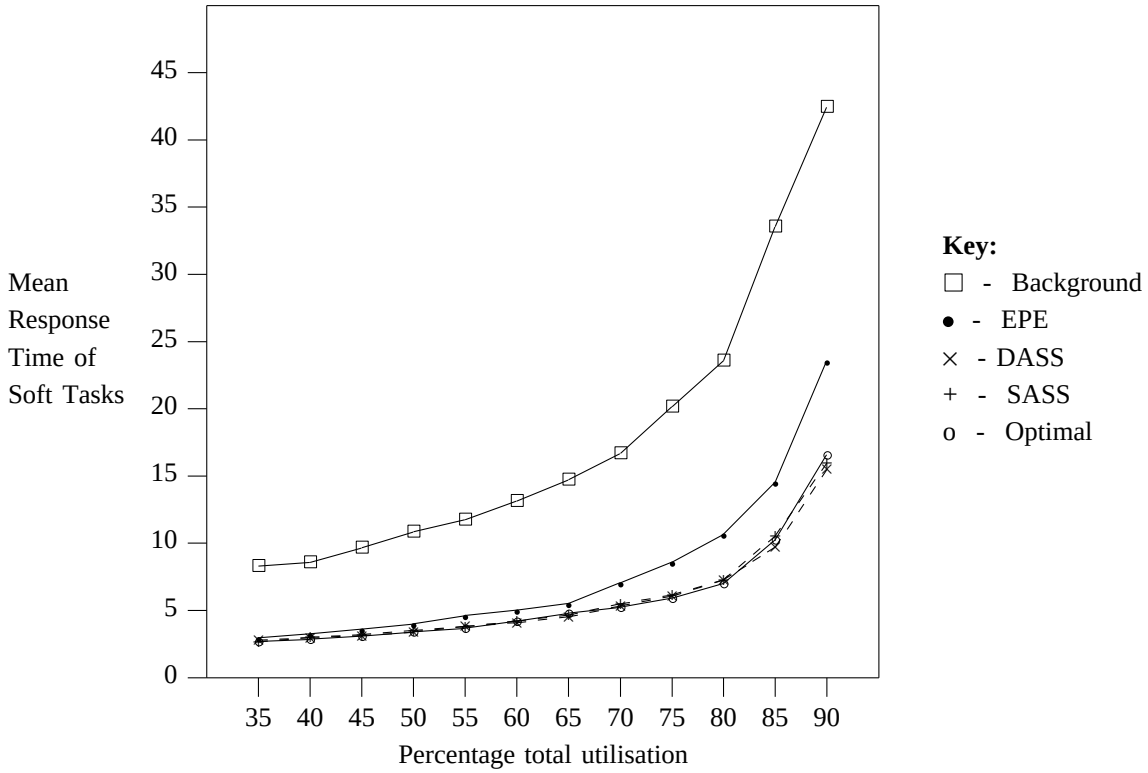
The overheads involved in scheduling, maintaining counters for extra capacity or slack, and calculating slack were assumed to be zero. We return to this point in section 4.3.

We note that in some of the experiments (e.g. 4.14), the "optimal" Slack Stealing algorithm was marginally outperformed by an approximate algorithm. This an be explained as follows: the optimal algorithm is optimal in the sense that it determines the maximum amount of soft task processing which may take place immediately without causing the deadlines of hard tasks to be missed. However, as shown in section 3.2.3, this does not necessarily imply that the mean response time of soft tasks will be minimised.

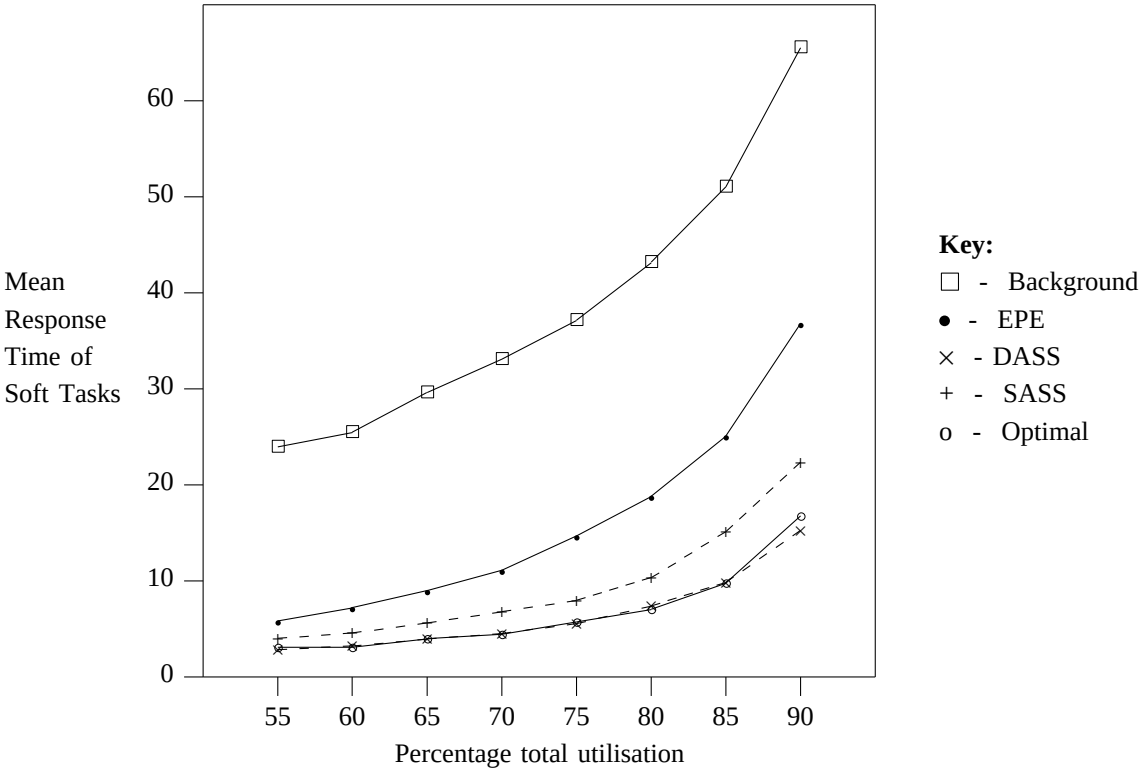
4.2.2 Periodic Tasks: Experiments 4.1-4.4

Experiments 4.1 to 4.4 examined the scheduling of hard task sets with utilisation levels of 30, 50, 70 and 90% respectively. In these experiments, all the hard tasks were periodic and every invocation required its worst case execution time. In each case, the soft task load was varied to give the total utilisation levels shown in the graphs.

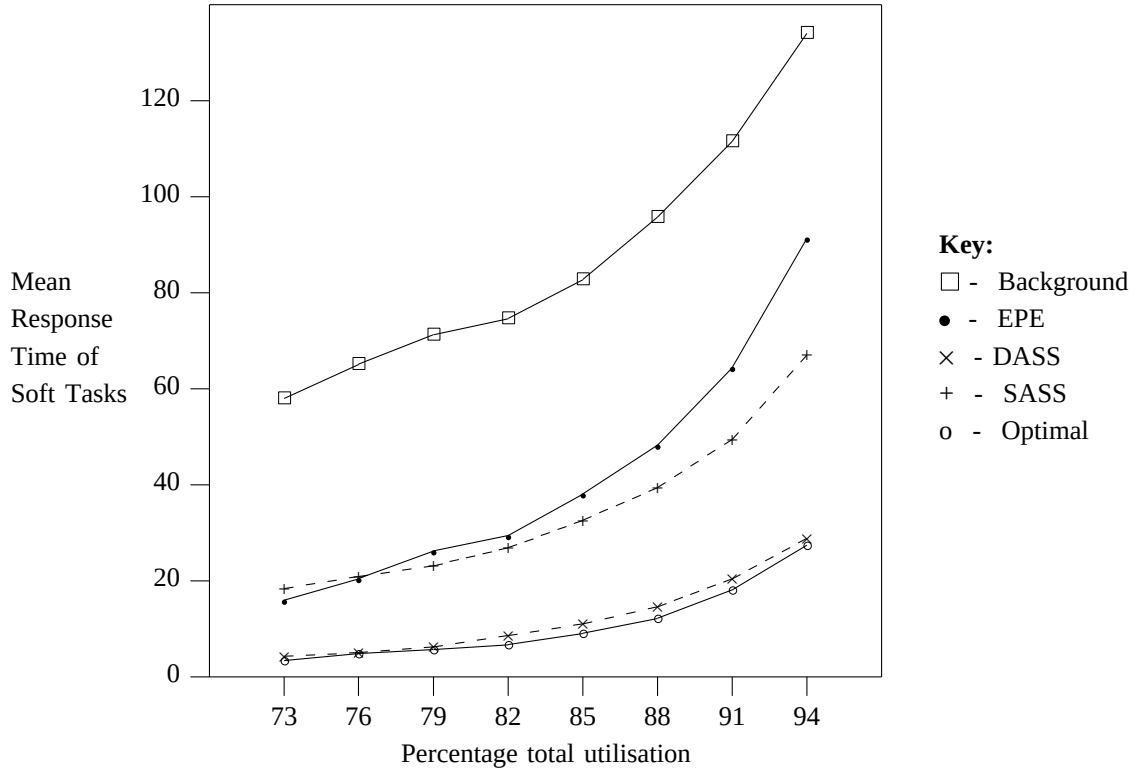
In these experiments, the performance of the DASS algorithm was close to optimal over the entire range of hard and soft task loads examined. The SASS algorithm similarly out-performed Extended Priority Exchange for hard task sets with a utilisation of less than 70%.



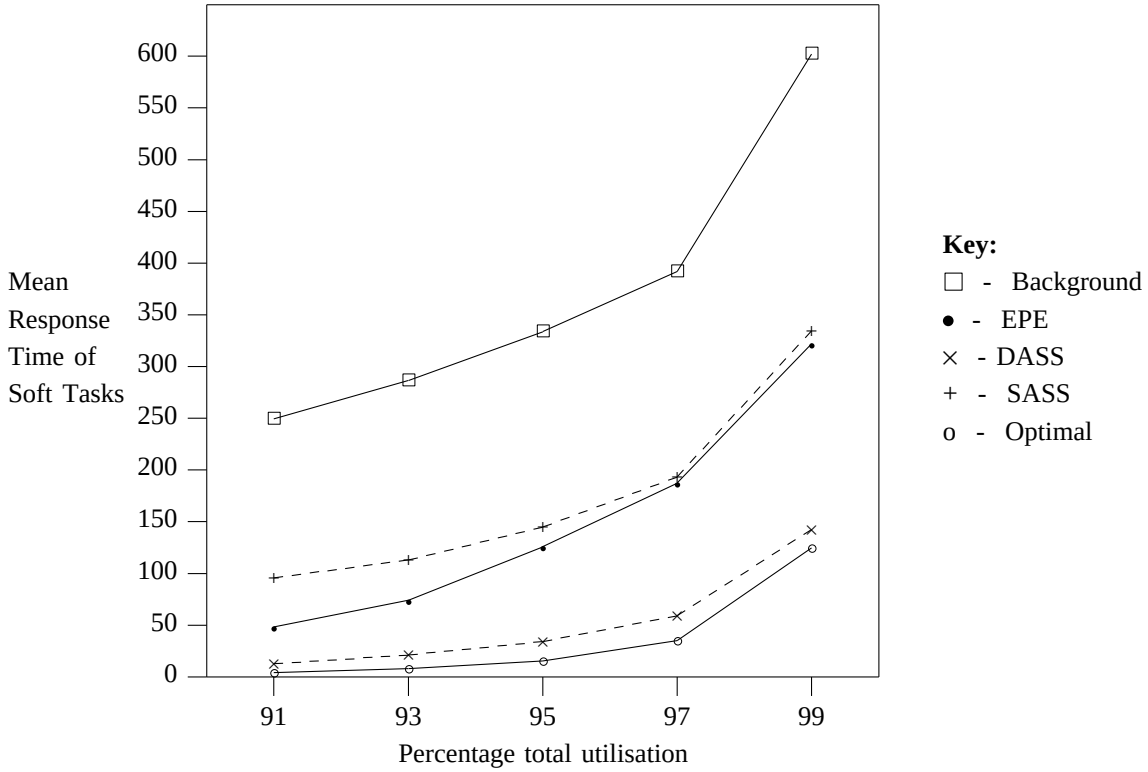
Expt. 4.1: Periodic tasks, 30% utilisation



Expt. 4.2: Periodic tasks, 50% utilisation



Expt. 4.3: Periodic tasks, 70% utilisation

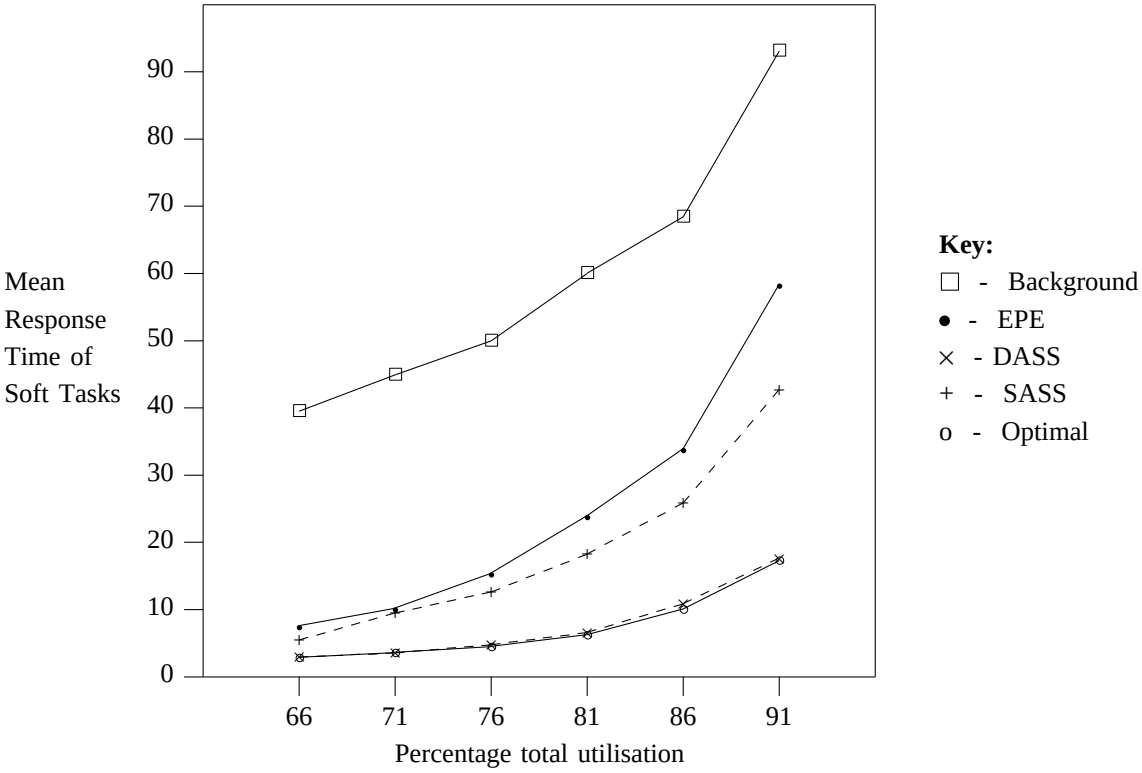


Expt. 4.4: Periodic tasks, 90% utilisation

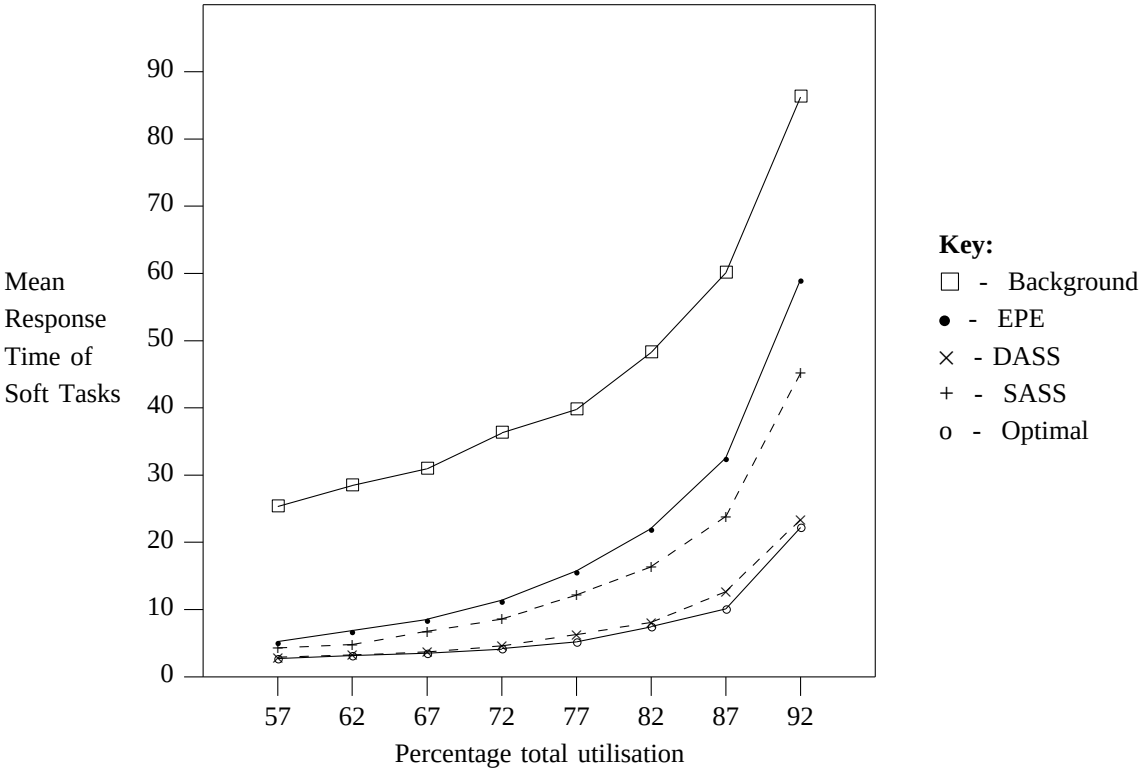
4.2.3 Gain Time: Experiments 4.5-4.8

The second set of experiments, investigated the response times of soft tasks given that the hard task set exhibited stochastic execution times, varying from 75-100% (experiments 4.5 and 4.7) and 50-100% (experiments 4.6 and 4.8) of their worst case execution times.

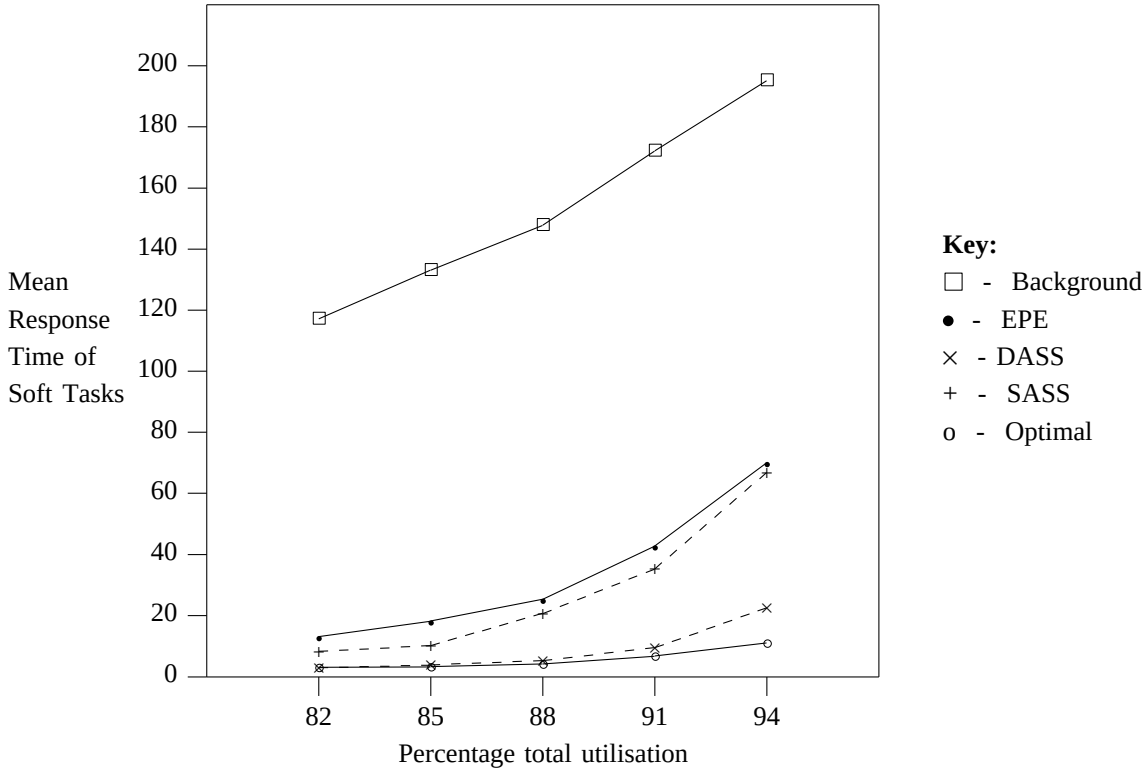
In each of these experiments, the DASS algorithm again provided close to optimal performance. Moreover, the SASS algorithm out-performed the Extended Priority Exchange approach. All the algorithms studied in these experiments, except Background scheduling, were shown to be highly effective at reclaiming gain time.



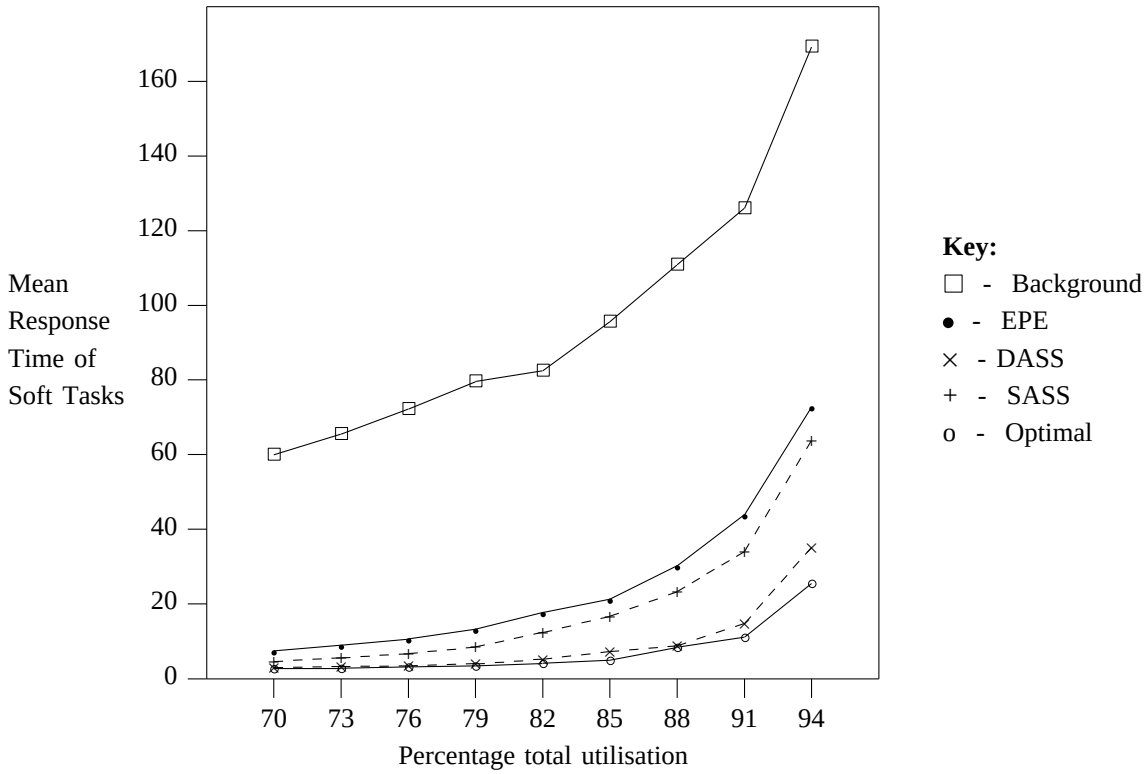
Expt. 4.5: Periodic tasks, 70% utilisation, 0-25% gain time



Expt. 4.6: Periodic tasks, 70% utilisation, 0-50% gain time



Expt. 4.7: Periodic tasks, 90% utilisation, 0-25% gain time



Expt. 4.8: Periodic tasks, 90% utilisation, 0-50% gain time

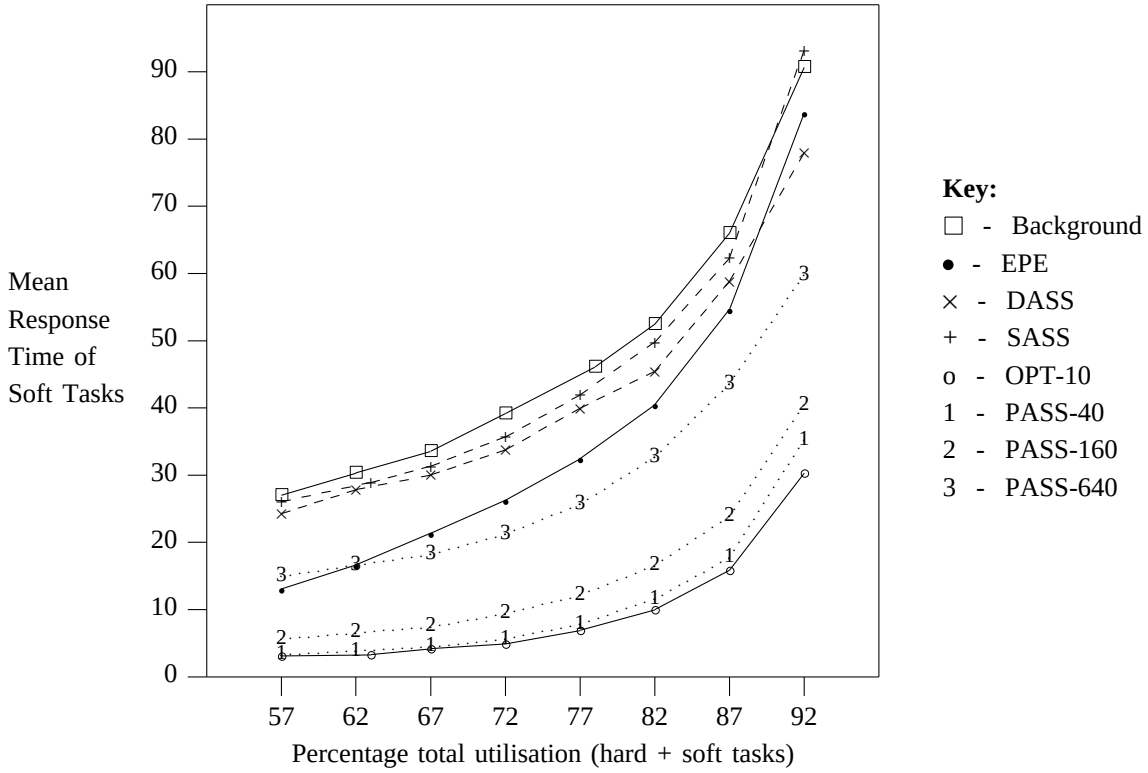
4.2.4 Sporadic Tasks: Experiments 4.9 - 4.12

In this series of experiments, we evaluated the performance of the various algorithms for hard task sets with a worst case utilisation of 70% or 90% and different proportions of periodic and sporadic tasks. In each case, the soft task load was varied to give the *mean* combined loads shown in the graphs. Note in many cases, the worst case combined load exceeded 100%. For comparison purposes, we simulated a close to optimal Slack Stealing algorithm referred to as OPT10, which calculates the exact slack at every priority level every 10 ticks. (Evaluating the exact slack every tick is prohibitively expensive even for simulation purposes).

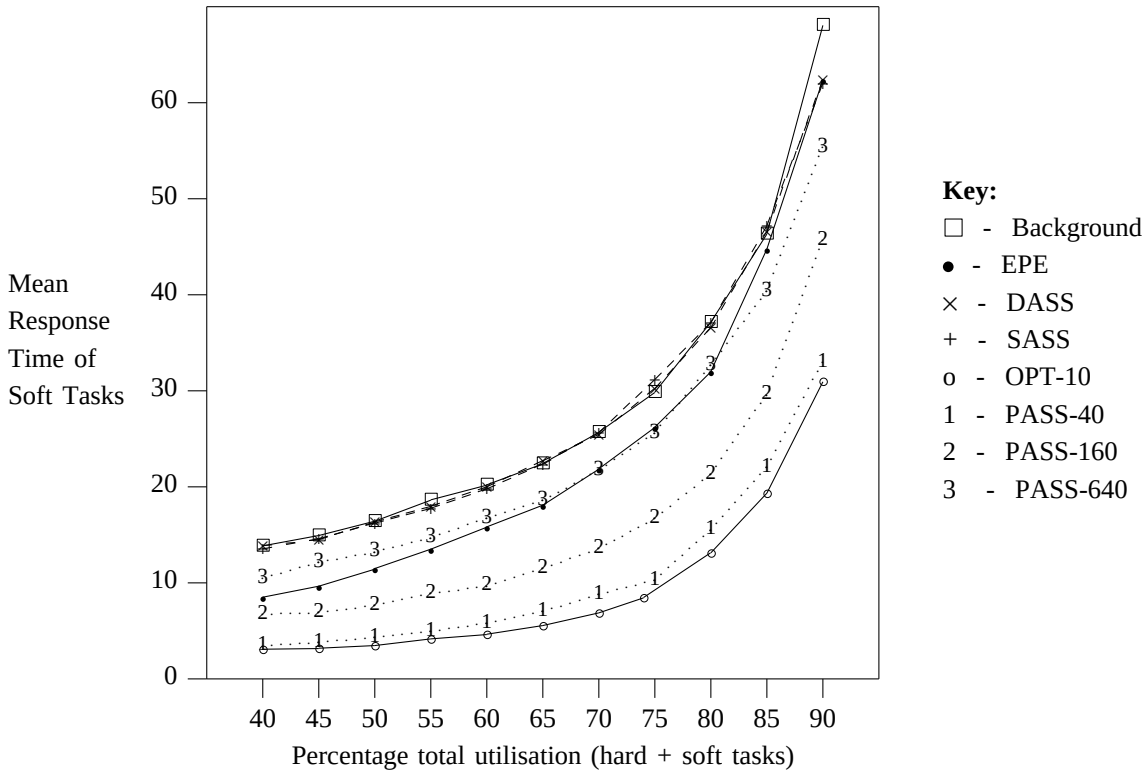
In experiments 4.9 and 4.11, each task with an even priority was designated sporadic, thus half of the tasks followed a periodic arrival pattern and the other half, a sporadic pattern.

In experiments 4.9 to 4.12, the performance of the SASS and DASS algorithms was degraded to that of Background scheduling. This is perhaps not surprising as these algorithms were designed for strictly periodic task sets. In effect, once the minimum inter-arrival time of a task τ_i has expired, then the slack at priority level i rapidly reduces to zero. As the slack at this priority level is not re-evaluated until the next time that task τ_i completes, there are long periods when one or more low priority tasks have zero slack; thus preventing soft tasks from executing at a high priority level.

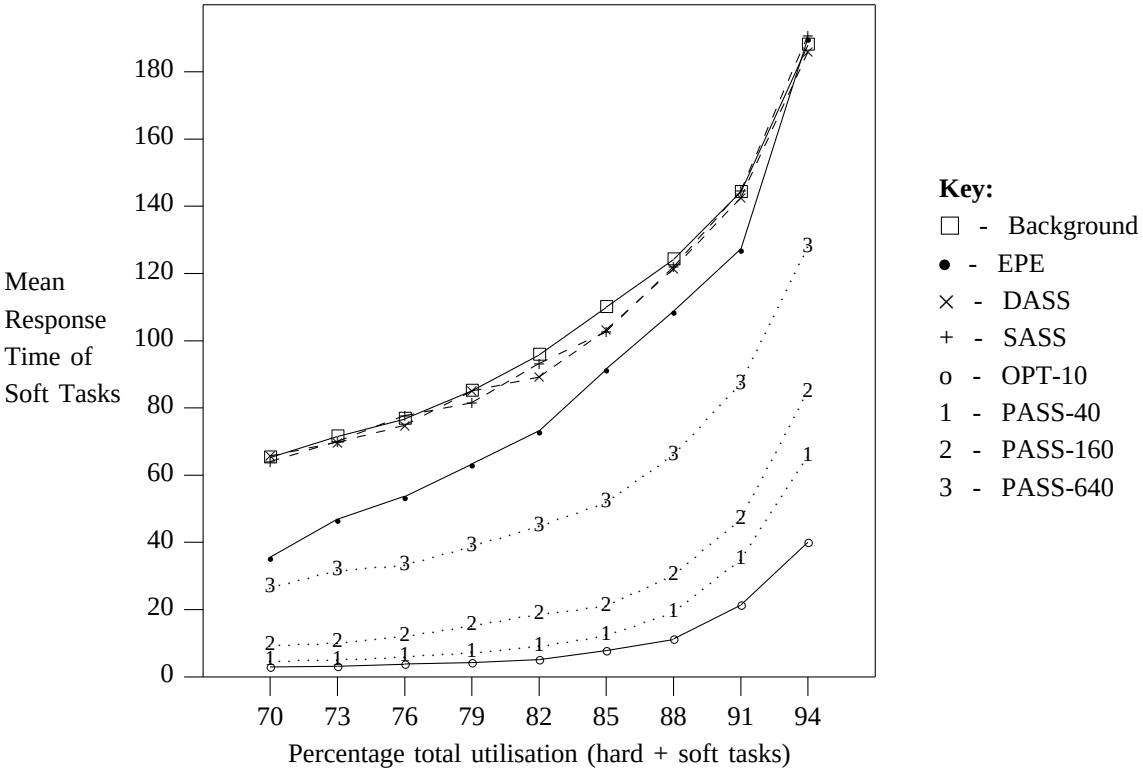
This problem is addressed by the PASS algorithm which periodically re-calculates the slack available at each priority level (for example, PASS-40 recalculates the slack every 40 ticks). It is clear from the experimental results, that in order to provide close to optimal performance, the period of the PASS algorithm must be of the same order as the shortest hard task inter-arrival time. Increasing the period of the PASS algorithm above this level increased the response time of soft tasks, as the spare time brought about by late sporadic arrivals went undetected for a longer period. When a period of 640 was used, the slack was not re-calculated for many invocations of higher priority tasks (with periods less than 640) resulting in these tasks executing in preference to the soft tasks, significantly increasing their response times.



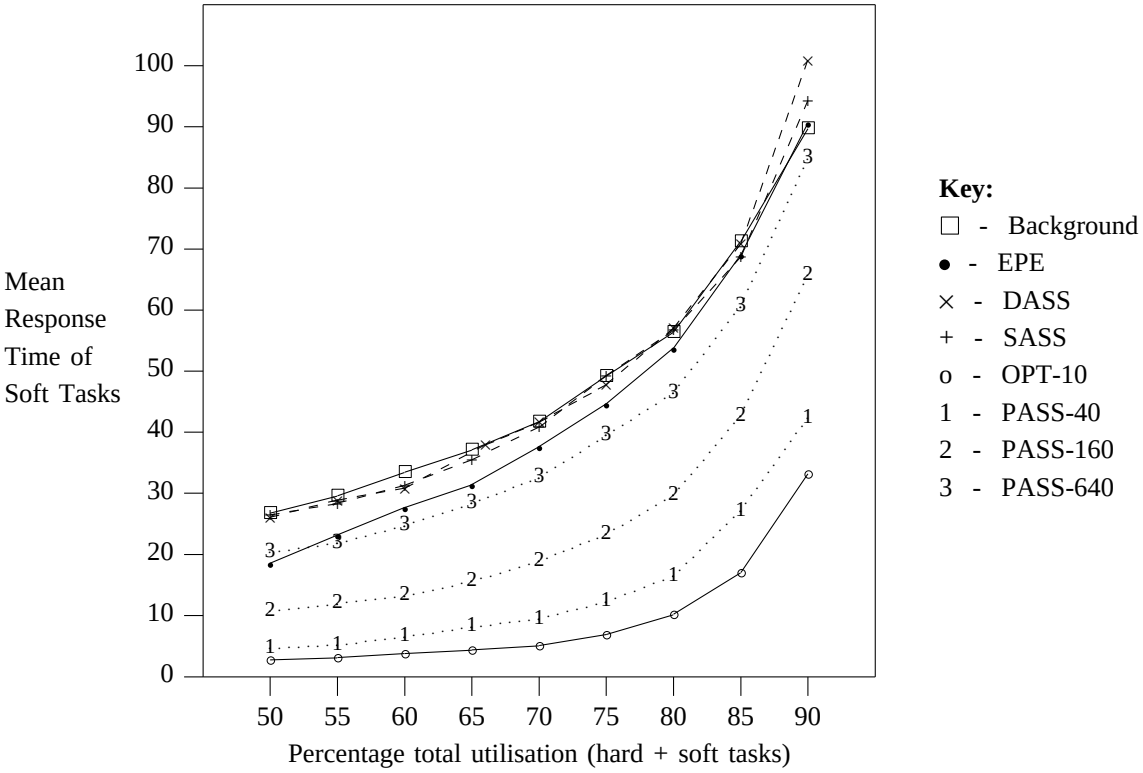
Expt. 4.9: Half sporadic, half periodic tasks, 70% utilisation



Expt. 4.10: All sporadic tasks, 70% utilisation



Expt. 4.11: Half sporadic, half periodic tasks, 90% utilisation



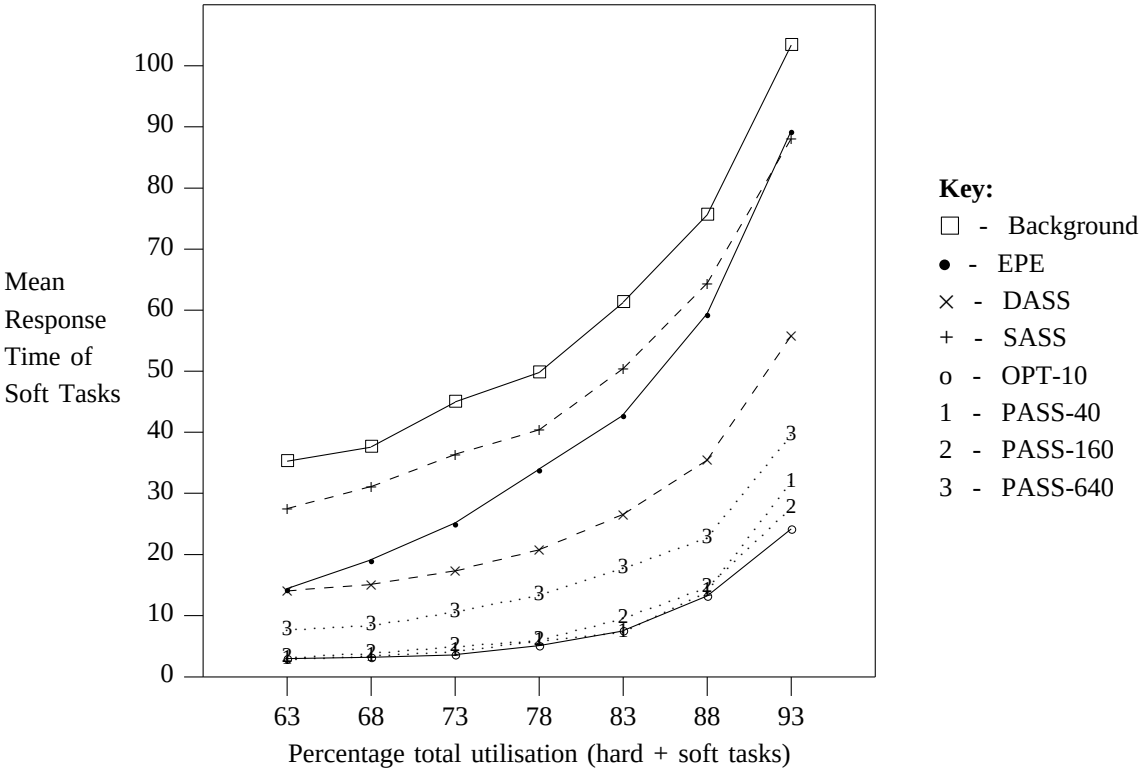
Expt. 4.12: All sporadic tasks, 90% utilisation

4.2.5 Adaptive Tasks: Experiments 4.13 - 4.16

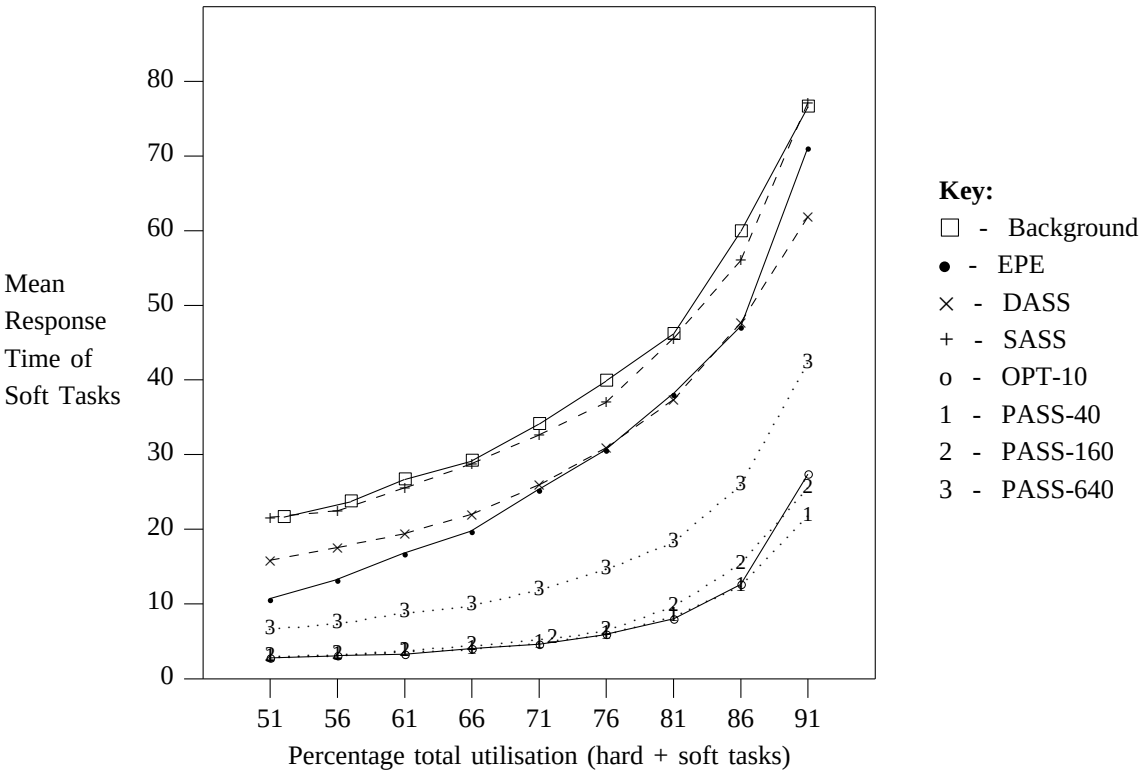
In experiments 4.13 to 4.16, we examined the scheduling of hard task sets containing tasks which exhibit adaptive behaviour. In particular, at each adaptive task completion, the next release time was set to some randomly determined value between T_i and $2T_i$ since the previous release time.

In these experiments, the SASS and DASS algorithms faired somewhat better than in similar experiments using hard sporadic tasks (4.9 - 4.12). This is because at completion, the next release and hence next deadline of an adaptive task τ_i is known. The slack available at priority i can therefore be accurately calculated at this point. However, neither the SASS or DASS algorithms take into account the fact that the late release of task τ_i increases the slack available at lower priority levels.

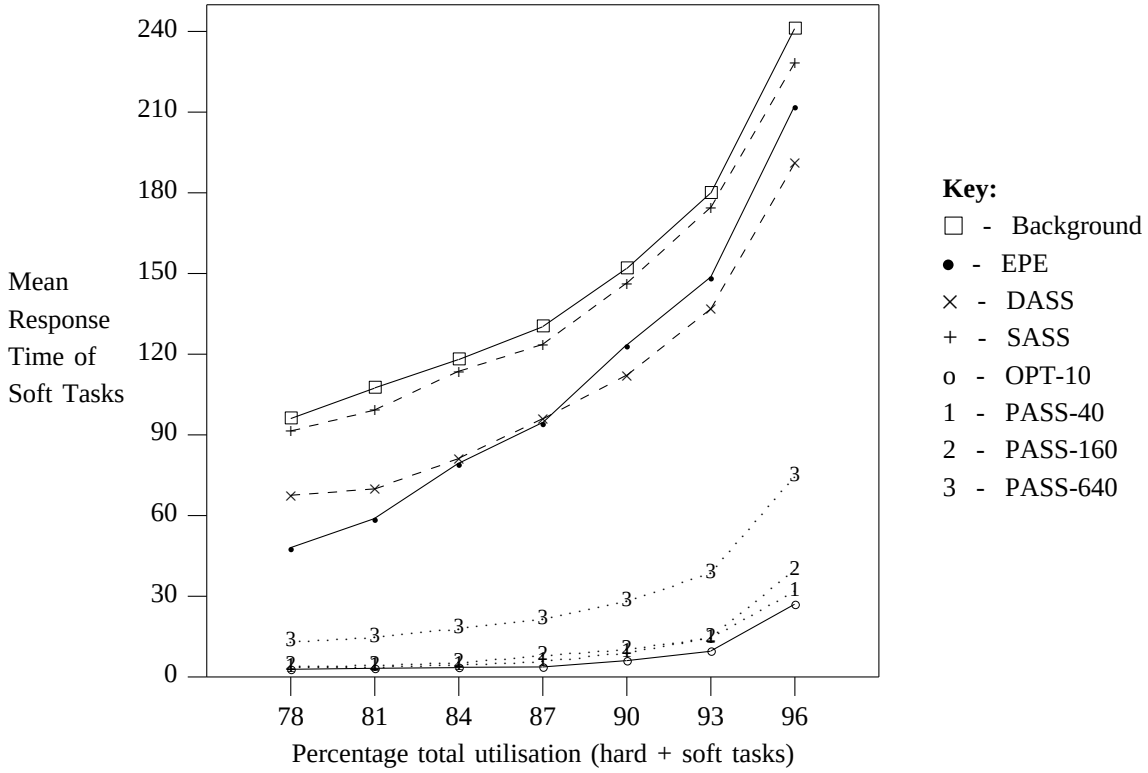
In experiments 4.13 to 4.16, the period of the PASS algorithm needed to achieve close to optimal performance was around 160 ticks; a considerable increase over that required for similar sets of sporadic tasks. This is because the slack at high priority levels is maintained reasonably accurately by making use of known next release times. The slack at lower priority levels is however affected by higher priority tasks not arriving at their maximum rate. To achieve good performance, it is therefore necessary to re-evaluate the slack at these lower priority levels periodically.



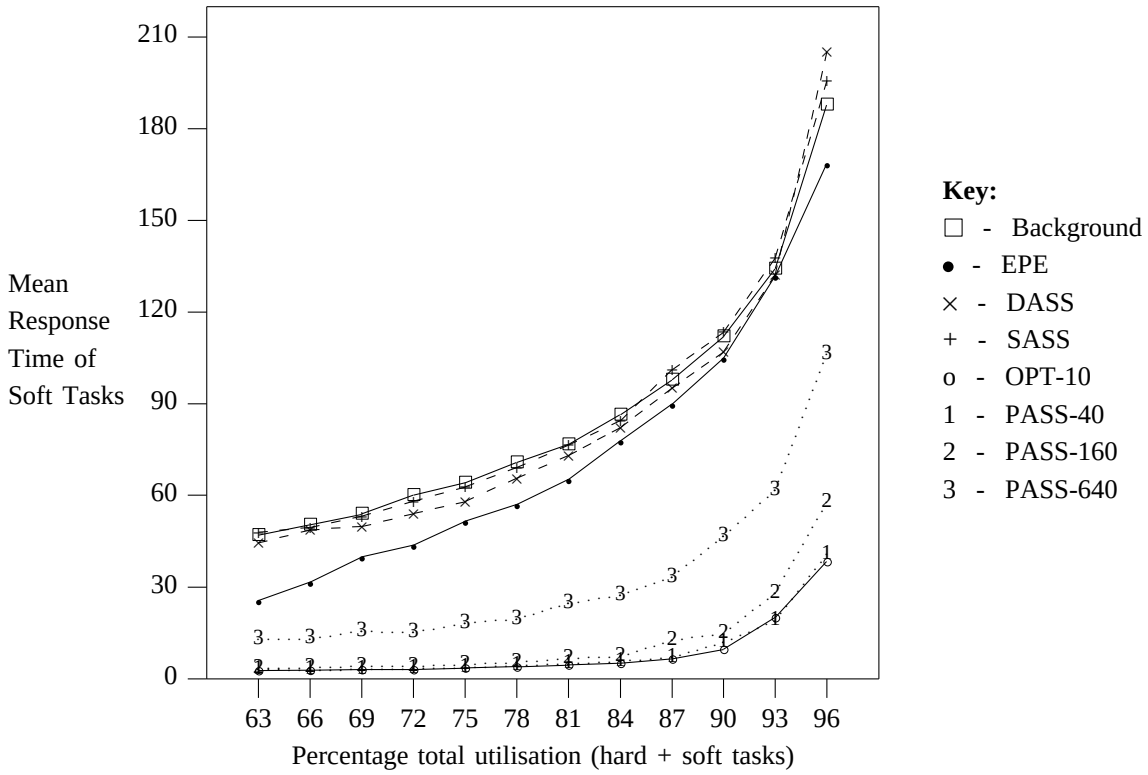
Expt. 4.13: Half Adaptive, half periodic tasks, 70% utilisation



Expt. 4.14: All Adaptive tasks, 70% utilisation



Expt. 4.15: Half Adaptive, half periodic tasks, 90% utilisation

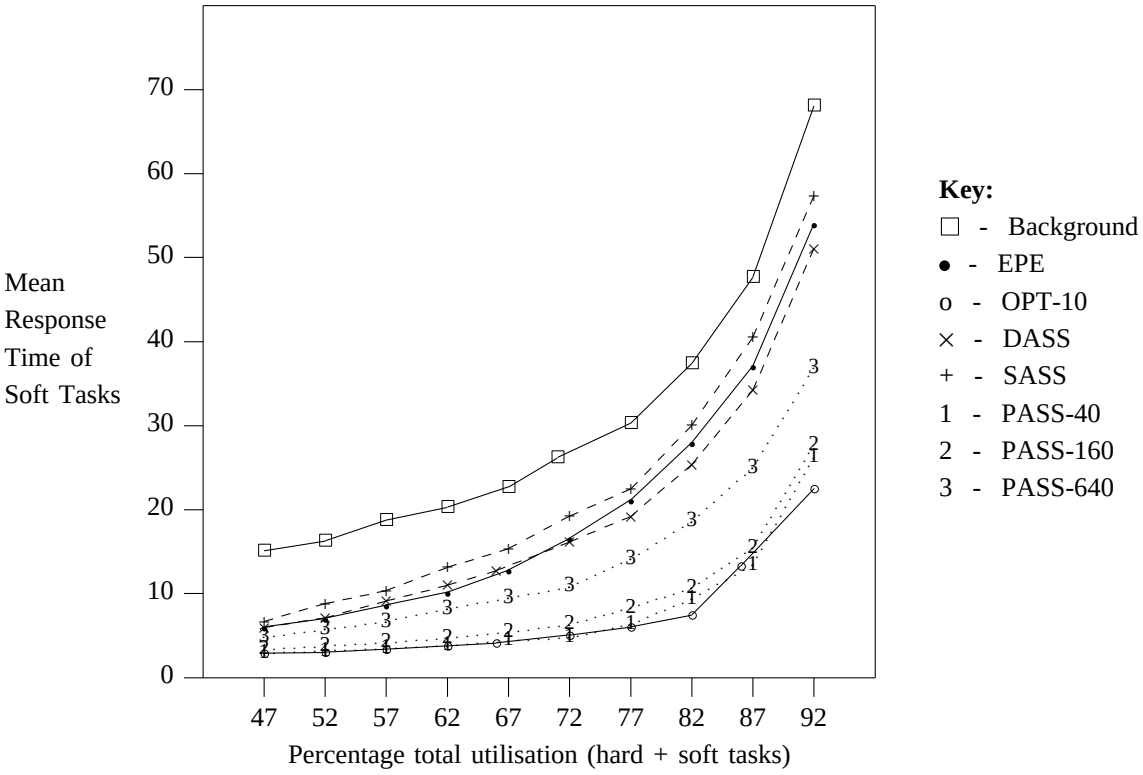


Expt. 4.16: All Adaptive tasks, 90% utilisation

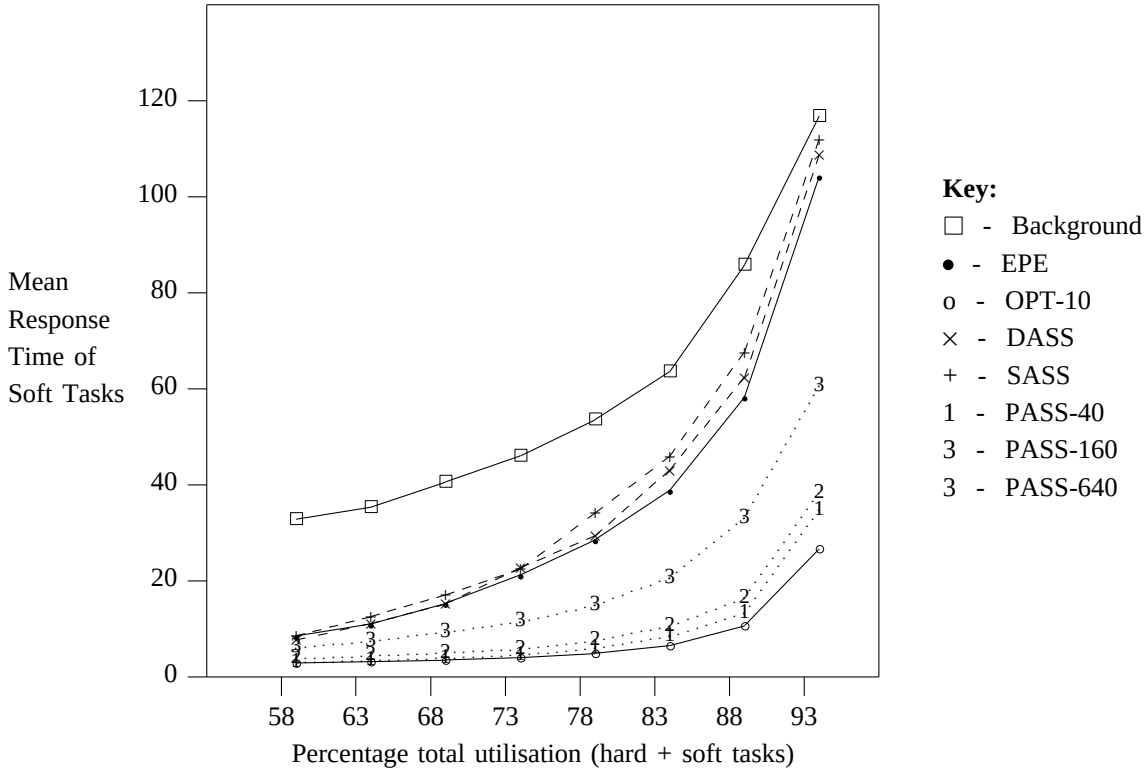
4.2.6 Mixed Task Attributes: Experiments 4.17 - 4.18

We evaluated the various algorithms for hard task sets exhibiting sporadic, adaptive and periodic release, and stochastic execution times. Tasks τ_1, τ_5 and τ_9 were designated as sporadic, tasks τ_3, τ_7 and τ_{11} were adaptive, with the remainder strictly periodic. Each invocation of a task executed for 50-100% of its wcet. Any gain time produced was detected at task completion.

In these experiments, the DASS and SASS algorithms exhibited similar performance to the Extended Priority Exchange approach. All three of these methods reclaim gain time at a high priority level, but are unable to make spare time available as anything other than a background service opportunity. In contrast, the PASS algorithm is able to reclaim spare time brought about by the late arrival of adaptive and sporadic tasks, leading to significantly improved soft task response times.



Expt. 4.17: Mixed periodic, sporadic, adaptive tasks, 70% utilisation, 0-50% gain time



Expt. 4.18: Mixed periodic, sporadic, adaptive tasks, 90% utilisation, 0-50% gain time

4.3 Overheads and Implementation Issues

In this section, we discuss the implementation of the approximate Slack Stealing algorithms. We then assess the performance and memory overheads associated with these algorithms. In particular, we present the results of simulations investigating the effect which dynamically calculating slack has on performance.

4.3.1 Data Consistency

To ensure that data synchronisation problems do not arise on the data structures describing slack, the approximate computation of slack used in the DASS and PASS algorithms proceeds as follows: First a consistent snap shot of all timing data (i.e. $S_i(t)$, $c_i(t)$, $x_i(t)$ and $d_i(t)$) is taken at time t. Next the approximate slack at time t is calculated for the required priority levels. The extra slack found at each priority level i, corresponds to additional pessimism in the lower bound on level i slack at time t. During the computation of slack, the lower bound on slack at each priority level continues to be maintained according to the generic methods given in section

4.1.1. Once the extra slack has been computed, it is added, indivisibly, to the slack counters. This ensures that the data structures describing slack remain consistent.

All three approximate slack stealing algorithms incur a performance overhead maintaining *n* slack counters in accordance with the generic methods outlined in section 4.1.1. In addition, there are execution time and memory overheads involved in re-calculating the available slack, these are discussed in detail below.

4.3.2 Slack Stealing: Static Approximation

The memory requirements of the static approximation are due to the tables of slack values. We note that this storage overhead may be limited by restricting the number of values stored in each table. For example, assuming the table storing the values of $S_i^C(\delta)$ is of size *m* and has an index *l* ($l = 0,1,2,3...m-1$). Values of slack are stored corresponding to:

$$\forall l = 0,1,2...m-1 \ : \ \delta_l = T_i + l\left\lceil \frac{D_i}{m} \right\rceil \tag{4.9}$$

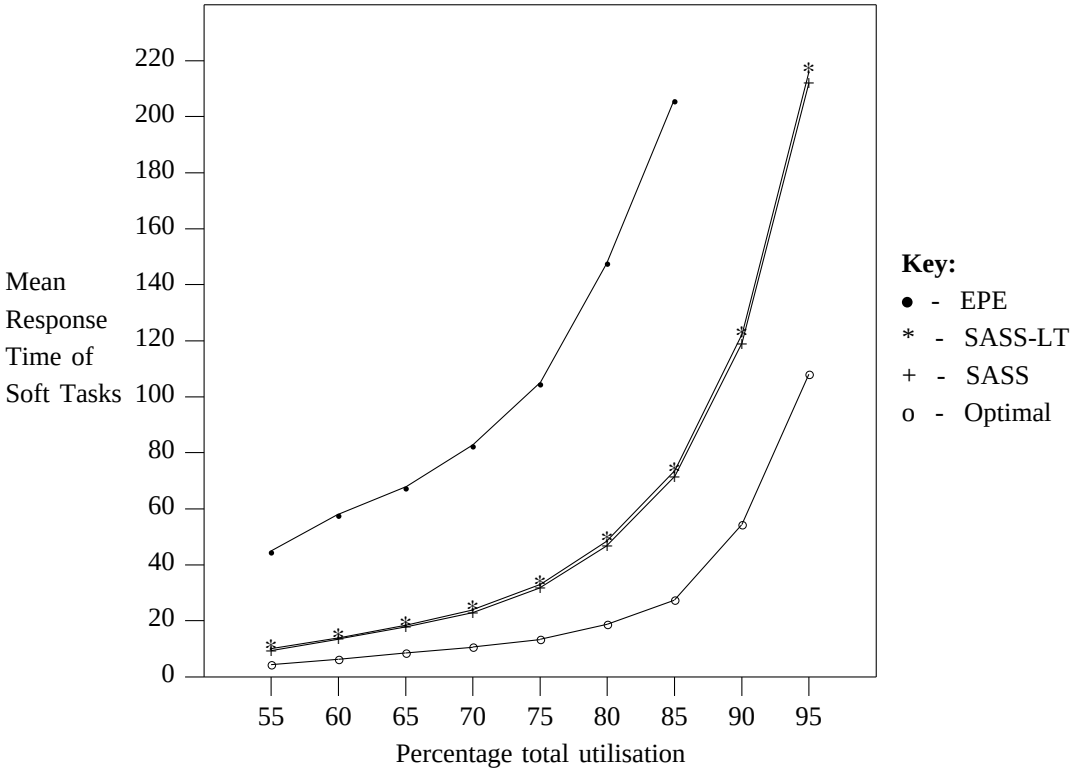
At run-time, at the completion of task τ_i, the appropriate array index *l* is defined by:

$$l = \left\lfloor \frac{d_i(t) - T_i}{\left\lceil \frac{D_i}{m} \right\rceil} \right\rfloor \tag{4.10}$$

Thus ensuring that the value of slack retrieved from the table is a lower bound on $S_i^C(d_i(t))$.

In experiment 4.19, we examined the degradation in performance of the SASS algorithm caused by limiting the number of table entries to 100 per task (SASS-LT). In this experiment, we used periodic hard task sets with 50% utilisation level and periods chosen at random in the range 1000 to 10000 ticks. The number of table entries which would have otherwise been required varied from 34 (the shortest task deadline) to 8851 (the longest task deadline). This experiment showed that the degradation in performance caused by limiting the number of table entries was so

small as to be insignificant.



Expt. 4.19: Hard task set, 10 tasks, 50% utilisation

Task periods in the range 1000 to 10000.

By restricting the size of the table used, the memory overheads of the SASS algorithm can be greatly reduced with only a marginal reduction in performance. In fact, the version of the SASS algorithm reported in earlier experiments used this method of limiting the number of table entries.

4.3.3 Slack Stealing: Dynamic Approximation

The memory requirements of the approximate dynamic algorithms are very low. Storage is required for the n values of S_i, c_i, x_i and d_i. The execution time overheads are however, more significant. We investigated the execution time overheads of the PASS and DASS algorithms via worst case execution time analysis [113]. First, the dynamic approximation was implemented in C, the resulting code was then compiled using the Zortech C++ compiler (v3.0) and the object code converted to i486, 32 bit assembler. The worst case execution time was then found using the techniques given in [113]. The following assumptions were made in

calculating the worst case execution time.

- Both instructions and data were assumed to need fetching for each instruction (i.e. no cacheing or pipelining).
- The time to fetch each instruction was 2 clock cycles.

Information on instruction times was taken from the Intel i486 Microprocessor reference manual [1].

4.3.4 DASS Algorithm Overheads

In the case of the DASS algorithm, the approximate slack at priority level *i* is calculated each time task τ_i completes. The worst case number of clock cycles needed to accomplish this is given by the following function of *m*, where *m* is the number of hard tasks with priority *i* or higher.

$$cycles = 94 + 355m \tag{4.11}$$

Equation (4.11) gives the number of clock cycles which need to be added to the worst case execution time of each task τ_i to account for the dynamic calculation of slack. Assuming an execution environment comprising an Intel i486 Microprocessor running at 33MHz, the following table gives this overhead for the *m*th hard task.

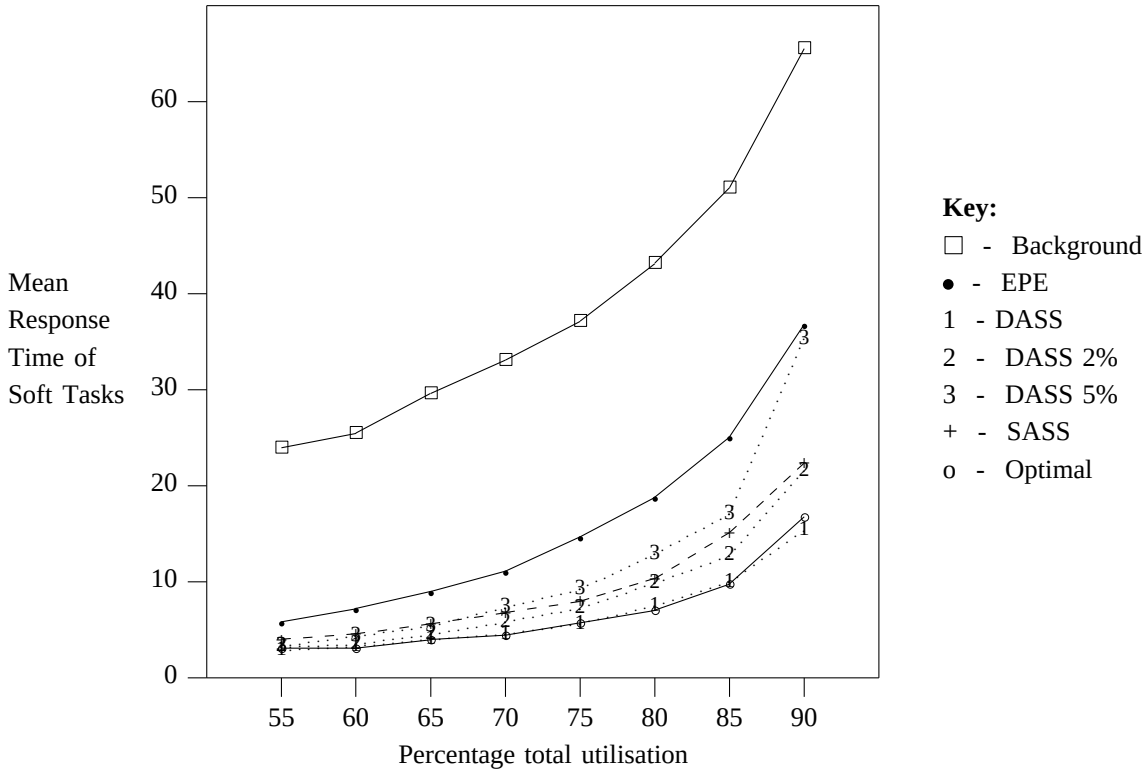
DASS algorithm overheads		
*m*th task	Cycles	Time (ms)
10	3644	0.11
20	7194	0.22
30	10744	0.33
40	14294	0.43
50	17844	0.54

We note that the effect which the overheads of the DASS algorithm have on soft task response times is dependent upon the timing attributes of the hard task set. Typically, for hard task sets comprising a modest number of tasks (e.g. 20) with long periods (e.g. 1 second), the overheads will be insignificant and the performance close

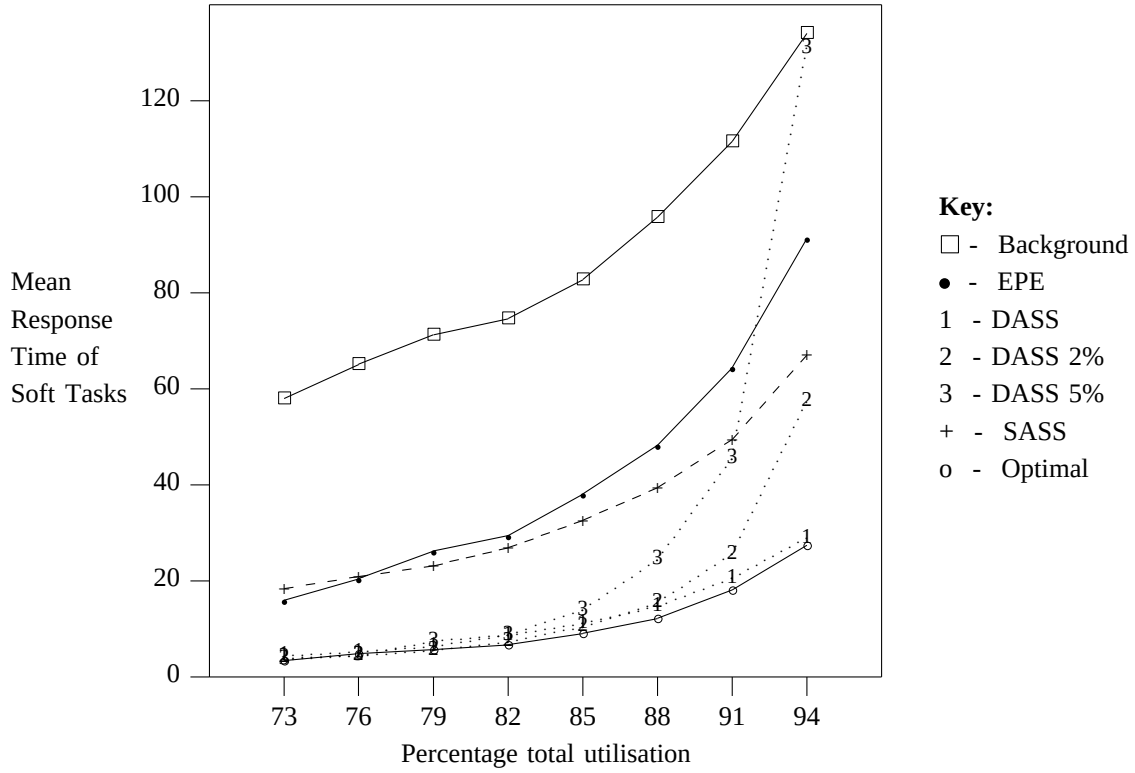
to that illustrated in section 4.2. However, for large task sets with short periods, the overheads of dynamically calculating slack may make the hard task set infeasible.

We now examine to what extent the overhead of dynamically calculating slack can be tolerated and still provide close to optimal scheduling of soft tasks. Note, we assume that the increase in hard task worst case execution times caused by the addition of overheads does not cause the hard task set to become infeasible.

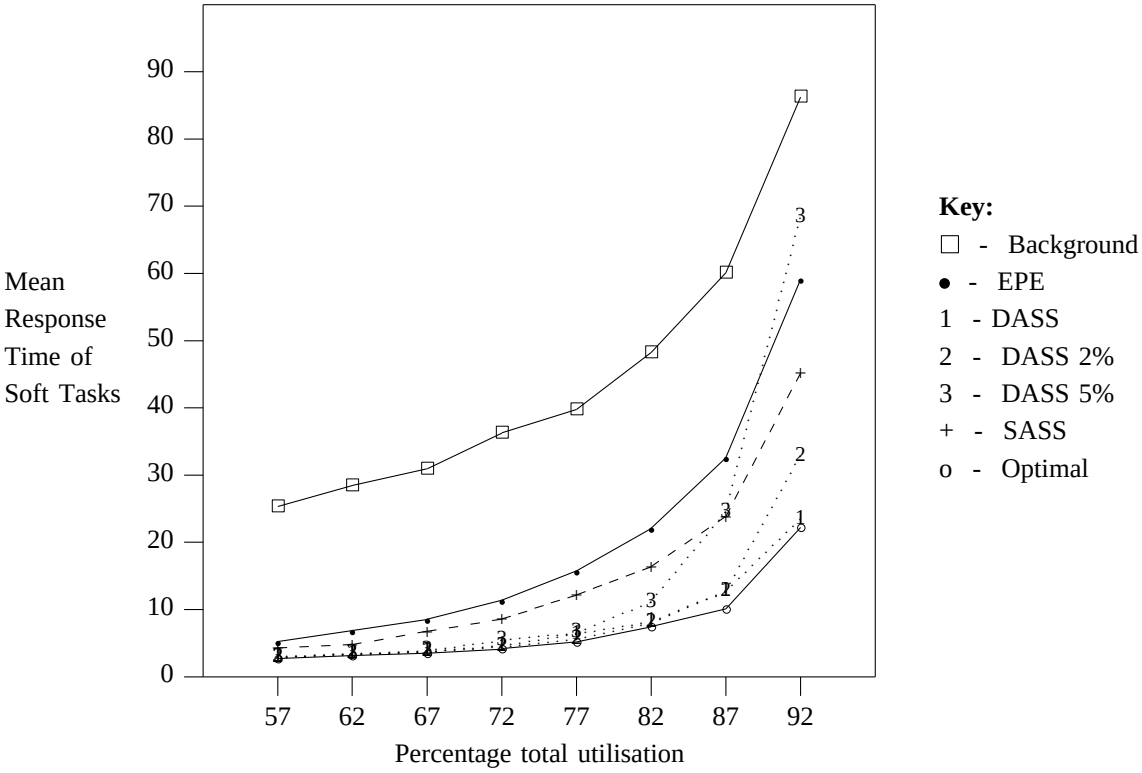
Experiments 4.20 to 4.23 examined the reduction in performance of the DASS algorithm caused by a 2% or 5% overhead in calculating slack. With no overheads, the DASS algorithm offers close to optimal performance over the entire range of soft task loads. With a 2% or 5% overhead, performance is close to optimal until the total load (hard + soft + overheads) reaches 85 - 90%, soft task response times then rapidly increase as the processor becomes fully utilised.



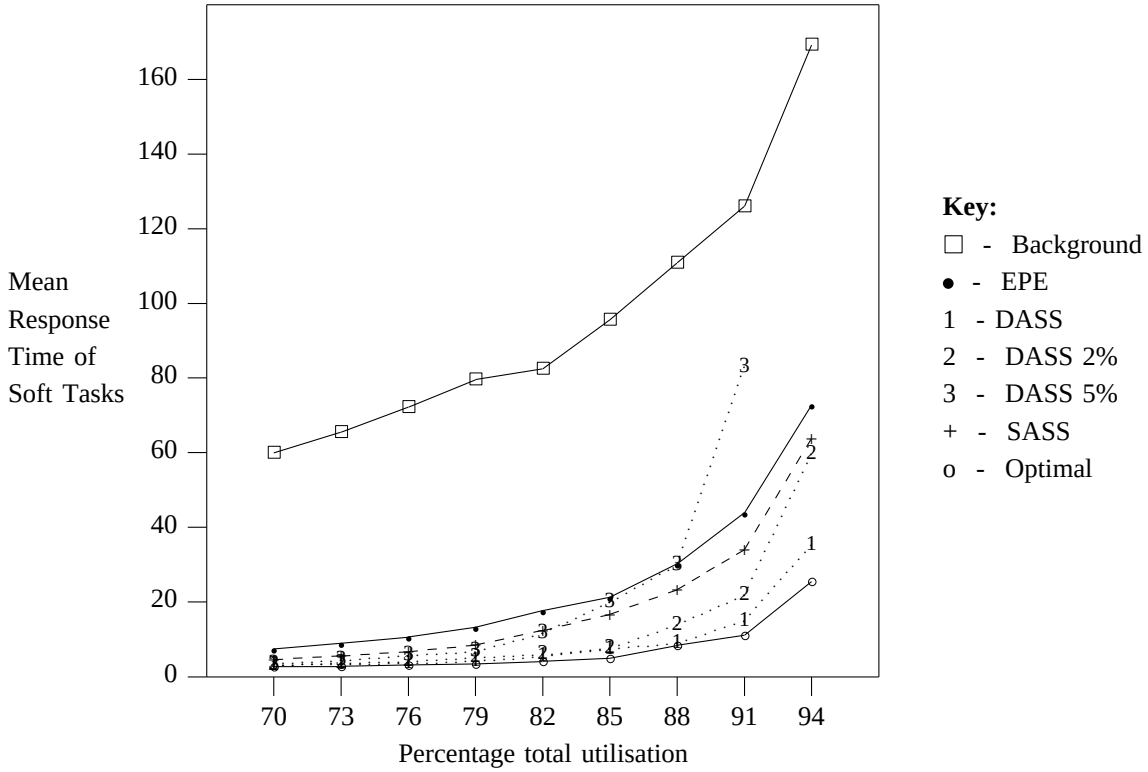
Expt. 4.20: Periodic tasks, 50% utilisation



Expt. 4.21 :Periodic tasks, 70% utilisation



Expt. 4.22: Periodic tasks, 70% utilisation, 0-50% gain time



Expt. 4.23: Periodic tasks, 90% utilisation, 0-50% gain time

Assuming that the sets of 12 hard tasks simulated in tests 4.1 to 4.23 are realistic with 1 tick = 1ms, then the overhead of calculating slack on a 33MHz i486 is given in the table below.

DASS algorithm overheads	
n tasks	% overhead
10	0.2
20	0.8
30	1.8
40	3.2
50	5.0

The results of our simulations show that for strictly periodic task sets, the overhead of calculating slack dynamically is justified, provided that this overhead remains less than approx 3%. In this case, the DASS algorithm provides improved

soft task response times with respect to static approaches such as the Extended Priority Exchange and SASS algorithms.

4.3.5 PASS Algorithm Overheads

For the PASS algorithm, the worst case number of clock cycles needed to calculate the approximate slack at all *n* priority levels is given by the following function of *n*.

$$cycles = 91 + 56n + \frac{n(n+1)}{2}343 \tag{4.23}$$

The table below gives the worst case number of clock cycles and the corresponding time in ms for a 33MHz i486 Microprocessor to calculate the approximate slack at *all* priority levels.

PASS algorithm overheads		
n tasks	Cycles	Time (ms)
10	19516	0.6
20	73241	2.2
30	161266	4.9
40	283591	8.6
50	440216	13.3

The overheads of the PASS algorithm are clearly dependent on its period and the cardinality of the hard task set. By dynamically calculating slack on a separate scheduling co-processor, these overheads can be mitigated. Assuming of course that the co-processor has sufficient processing capacity to calculate slack at the required rate. Under these assumptions, the overhead of calculating slack can be taken as zero and hence the performance of the PASS algorithm will be close to the theoretical levels illustrated in section 4.2.

We now consider the performance of the PASS algorithm when the calculation of slack is performed on the processor which is executing the hard and soft tasks. This requires the addition of a special high priority task which performs the necessary slack calculations. Unfortunately, the presence of such a task may render the hard task set infeasible. We therefore modify the PASS algorithm so that the calculation of slack has no effect on the processing capacity available for hard tasks. Note, we refer to the new algorithm so formed as the HASS (Hyperperiodic Approximate Slack Stealing) algorithm.

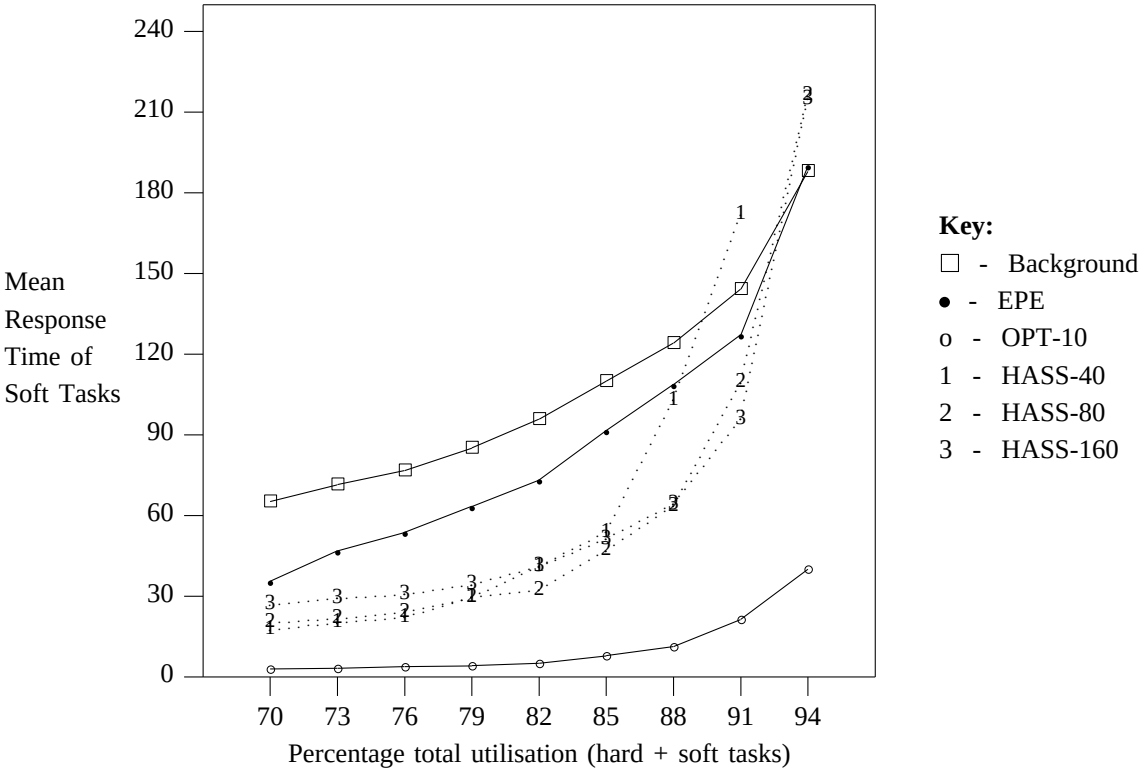
Under the HASS algorithm, the dynamic calculation of slack is only ever performed in slack time. A special soft task SC is used to calculate the slack at all priority levels. Upon release, SC is placed at the head of the queue of soft tasks awaiting execution. When there is slack available, SC executes re-calculating the slack at each priority level. Finally, when SC completes at time t, its next release is set for $t + T_{HASS}$, where T_{HASS} is the period of the HASS algorithm. This approach guarantees that the calculation of slack cannot interfere with the execution of hard tasks whilst ensuring that it is performed as promptly as possible.

The performance of the HASS algorithm is inferior to the theoretical performance of the PASS algorithm for two reasons. First the calculation of slack consumes valuable processing time which could otherwise be used to execute soft tasks. Second, as the HASS algorithm relies on slack time to calculate the available slack, when one or more of the slack counters reaches zero, the soft task SC has to wait for a background service opportunity before re-calculating the slack.

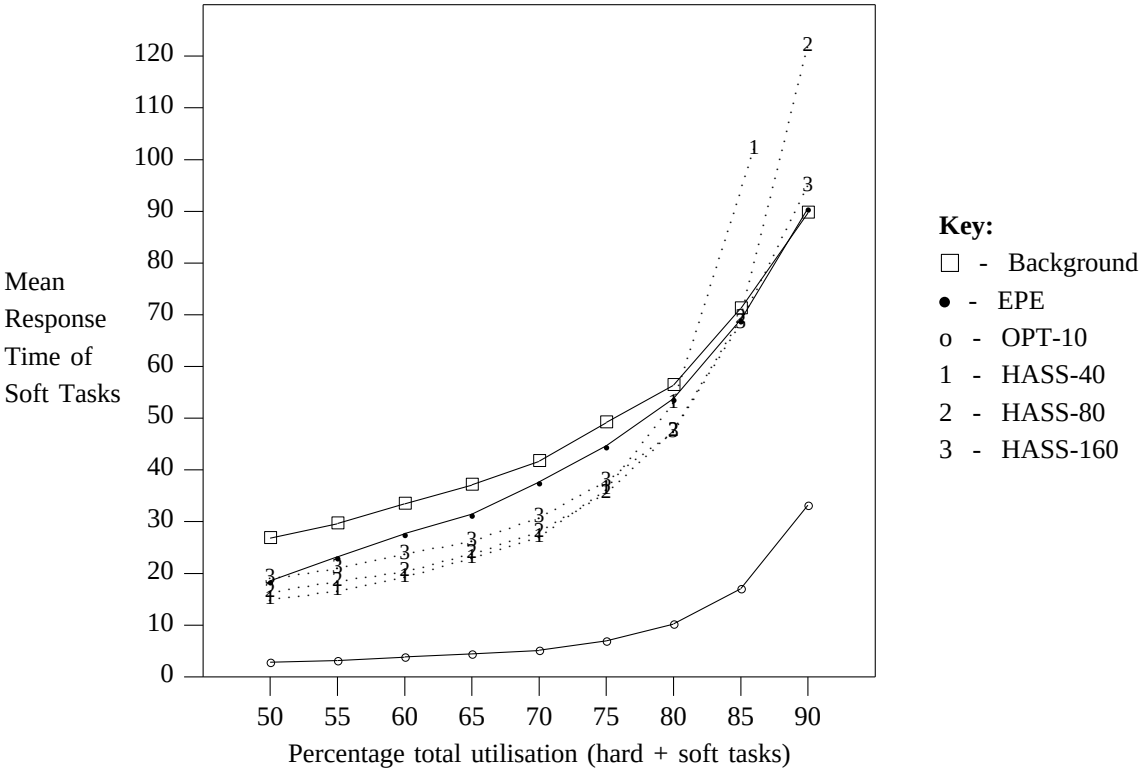
4.3.6 HASS Algorithm Overheads

We investigated the performance of the HASS algorithm in experiments 4.24 - 4.29. In each of these experiments, the soft task SC, which calculates the slack at all priority levels, was assumed to require 5 ticks of execution time (corresponding to 5ms for a task set of cardinality 30).

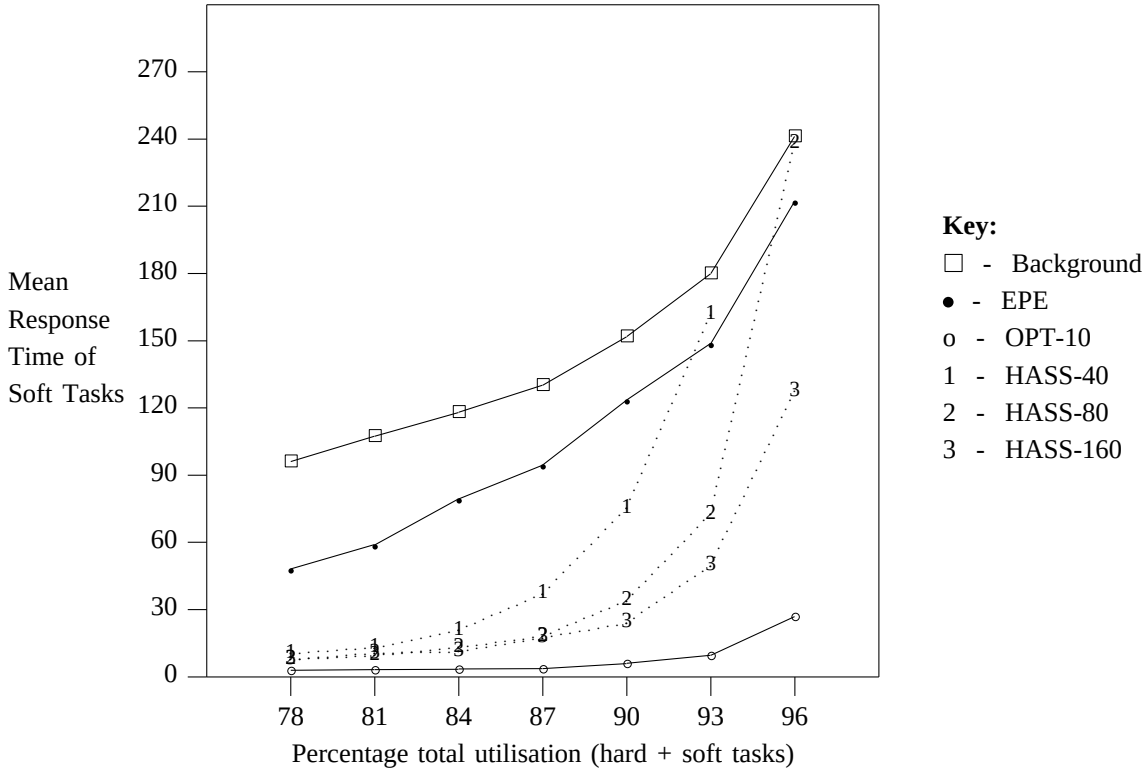
The simulation results show that the HASS algorithm is effective provided that the total utilisation is less than approximately 90%, above this level, its performance rapidly degrades as the processor becomes fully utilised.



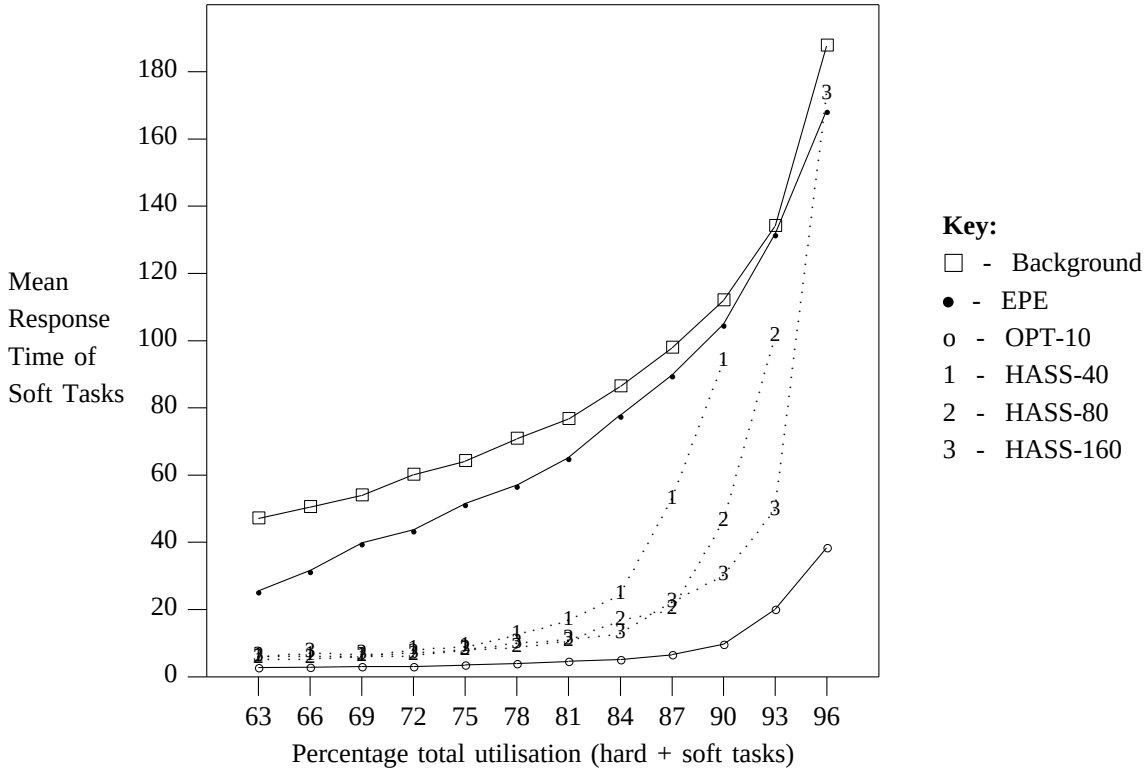
Expt. 4.24: Half sporadic, half periodic tasks, 90% utilisation



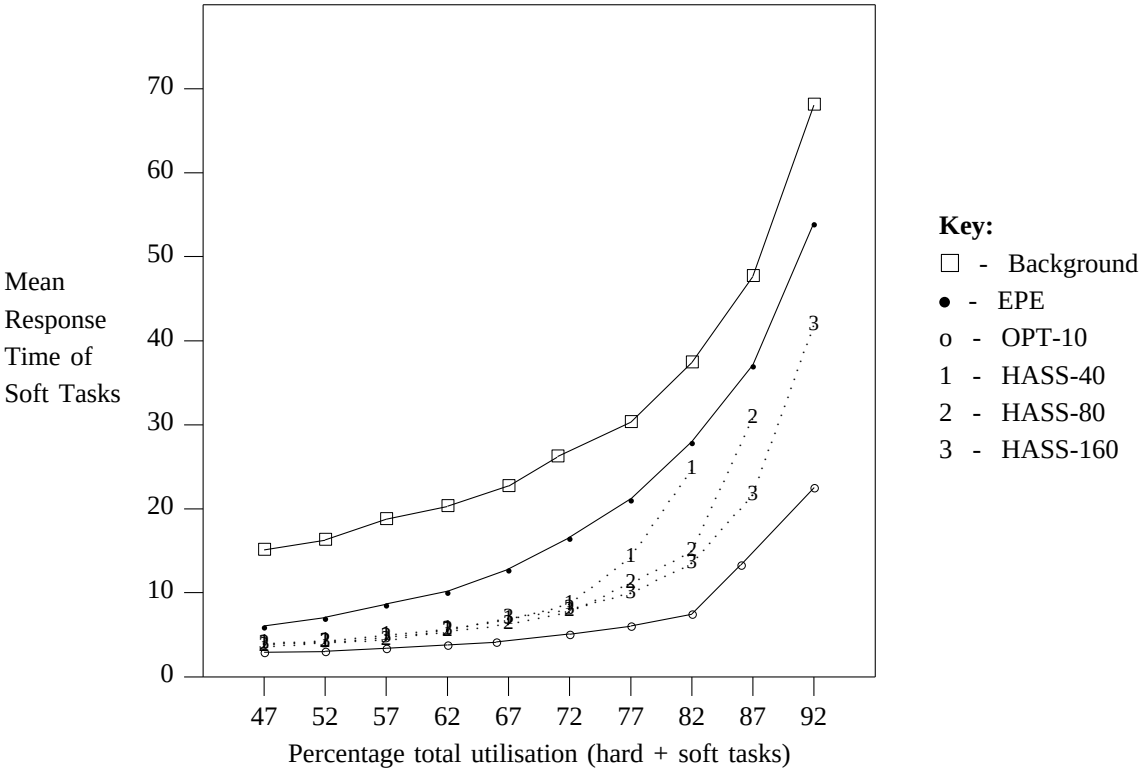
Expt. 4.25: All sporadic tasks, 90% utilisation



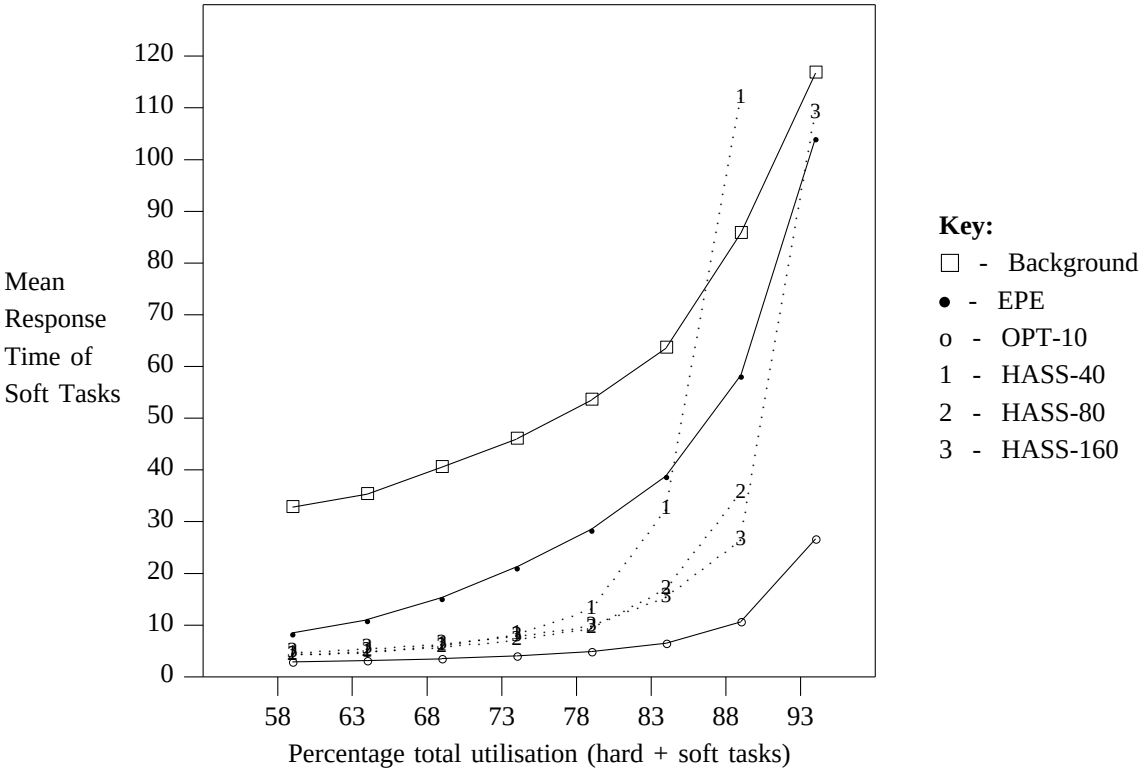
Expt. 4.26: Half Adaptive, half periodic tasks, 90% utilisation



Expt. 4.27: All Adaptive tasks, 90% utilisation



Expt. 4.28: Mixed periodic, sporadic, adaptive tasks, 70% utilisation, 0-50% gain time



Expt. 4.29: Mixed periodic, sporadic, adaptive tasks, 90% utilisation, 0-50% gain time

4.4 Summary

In this chapter we developed two approximate methods of calculating slack. These approximate methods were then used to construct various approximate Slack Stealing algorithms.

We examined the performance of the above algorithms with respect to responsively scheduling soft tasks. Comparisons were made with Background, Extended Priority Exchange and optimal Slack Stealing methods. Simulation studies indicated that for hard periodic task sets, the DASS algorithm provides significant performance improvements over the Extended Priority Exchange algorithm provided that the additional overheads incurred represented less than approximately 3% processor utilisation. For task sets resembling those simulated (i.e. with periods in the range 40ms to 2s), executing on a i486 processor, this limits the DASS algorithm to task sets of cardinality less than 30.

For hard task sets with a sizable sporadic or adaptive component, we showed that the PASS algorithm can provide close to optimal performance under certain conditions. These conditions are that the period of the PASS algorithm is the same order of magnitude as the shortest hard task period and that the overheads of calculating slack are low. Low overheads may be achieved by using a scheduling co-processor. However, without such a co-processor, a special high priority periodic task would be needed to calculate slack. Unfortunately, the presence of such a task may render the hard task set infeasible. We addressed this problem by developing the HASS algorithm which calculates slack in slack time. Again, simulation studies showed that this algorithm can provide effective soft task scheduling provided its period is short with respect to the majority of hard tasks, and that the overhead involved in calculating slack is low (less then approx. 5%). For hard task sets containing a large number of tasks, the overheads of dynamically calculating slack clearly become prohibitive. However, our results show that such techniques are viable for many real-time systems containing less than 30 hard deadline tasks.

Chapter 5

Identifying Spare Capacity: Dual Priority Scheduling

In the previous chapter, we saw that in general, the effective identification of spare capacity requires computationally expensive periodic re-evaluation of the slack, when dynamic Slack Stealing algorithms are employed.

In this chapter, we introduce an elegant alternative method of identifying spare capacity: Dual Priority scheduling. This minimally dynamic approach retains the predictability afforded to hard tasks by fixed priority scheduling, whilst facilitating the responsive scheduling of soft tasks. By comparison with Slack Stealing, the Dual Priority approach represents a more efficient means of identifying spare time which becomes available due to sporadic or adaptive tasks not being released at their maximum rate.

Under Dual Priority scheduling, hard tasks execute at either an upper or lower band priority level. Upon release, each hard task assumes its lower band priority, however, at a fixed time offset from release, the priority of the task is promoted to the upper band level. At run-time, other tasks with soft deadlines are assigned priorities in the middle band. Thus soft tasks execute in preference to hard tasks which are yet to undergo priority promotion.

Off-line analysis is given which determines the latest priority promotion time for each hard task. Further, we extend the applicability of Dual Priority scheduling to tasks which exhibit blocking and release jitter and have arbitrary deadlines and offsets. Finally, we augment the Dual Priority approach to reclaim gain time.

The performance of the Dual Priority approach, (in terms of responsively scheduling soft tasks) is compared to that of Background, Extended Priority Exchange and optimal Slack Stealing methods via simulation. Comparisons are made using a similar approach to the previous chapter. Finally, we discuss overheads and implementation issues.

5.1 Dual Priority Scheduling

The Dual Priority scheduling strategy, was inspired by previous work on dual priorities by Burns and Wellings [25] and the Earliest Deadline Last algorithm of Chetto and Chetto [29]. Harbour *et al* [48] have also considered the scheduling of tasks comprising precedence constrained subtasks with multiple priorities. However, the approach presented here differs, from that investigated by Harbour *et al*: tasks have their priority promoted a fixed interval of time after their release, as in [25]. In essence, the Dual Priority scheme facilitates the responsive execution of soft tasks by running all hard tasks immediately when there are no soft tasks to execute, or as late as possible, if there are soft tasks present.

5.1.1 Computational Model, Assumptions and Notation

The Dual Priority model considered in this chapter is similar to the fixed priority model detailed in section 2.2.1. We assume that there is a range of unique priorities split into three bands: *Upper*, *Middle* and *Lower*. Any priority level in the upper band is considered higher than any in the middle or lower bands. Hard tasks are assigned two priorities, one each from the upper and lower bands. At run-time, other tasks, typically with firm or soft deadlines, are assigned priorities in the middle band.

Each hard task has an *initial* priority in the lower band and a unique *promoted* priority i where $1 \leq i \leq n$, in the upper band. Thus 1 is the highest and n the lowest priority level in the upper band. Note that the lower band also comprises n priority levels, however, assignment of lower band priorities may be arbitrary: it need not reflect the upper band priority order. For notational convenience, we refer to hard tasks by their upper band priority level, thus task τ_i is the hard task with promoted priority i. We use $hp(i)$ to denote the set of tasks with a higher promoted priority than i and $lp(i)$ to denote the tasks with promoted priority i or lower.

Each task has a priority promotion delay Y_i, $(0 \leq Y_i < D_i)$ measured relative to its release. Upon release, each task τ_i assumes its initial priority (in the lower band), however after Y_i time units, its priority steps up to its promoted priority i (in the upper band).

5.1.2 Basic Analysis

Analysis developed for fixed priority scheduling is now applied to the basic Dual Priority model. We show that the worst case response times of hard tasks scheduled using dual priorities can be bounded and their time constraints guaranteed. Initially, we assume that all tasks are independent, do not exhibit release jitter and have deadlines which are less than or equal to their periods. These restrictions are subsequently relaxed, permitting tasks to synchronise with other tasks on the same processor, exhibit release jitter and to have arbitrary deadlines.

First we give a brief description of Dual Priority scheduling. When a hard task τ_i is first released, it has a priority which is in the lower band. It may be pre-empted by other tasks which have higher, lower band priorities. It may be pre-empted by any soft or optional task executing at a priority level in the middle band or finally by any hard task executing at an upper band priority. Once time Y_i has elapsed, measured from the release of task τ_i, its priority is promoted to the upper band. Once this has occurred, task τ_i can only be pre-empted by other hard tasks which have also had their priorities promoted and have a higher upper band priority than *i*.

To provide an *a priori* guarantee for task τ_i, we seek to obtain its worst case response time. We assume that in the worst case, task τ_i will be unable to execute any computations at its lower band priority, (perhaps due to the execution of soft tasks in the middle priority band). Once promoted to its upper band priority level, task τ_i will only be subject to interference from tasks $\tau_j \in hp(i)$ which have also had their priorities promoted. As priority promotion for each task τ_j occurs Y_j time units after its release, the worst case response time R_i of task τ_i may be calculated, using analysis derived for fixed priority scheduling [10].

We assume that task τ_i was released at time $-Y_i$ and has its priority promoted at time 0. Further, we consider all tasks $\tau_j \in hp(i)$ to have been released at their respective values of $-Y_j$, hence these tasks also undergo priority promotion at time 0, giving rise to a critical instant for task τ_i. The recurrence relation given below iteratively computes the length w_i, of the priority level *i* busy period starting at time 0 and culminating in the completion of task τ_i.

$$w_i^{m+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil C_j \qquad (5.1)$$

Iteration starts with $w_i^0 = 0$ and ends when $w_i^{m+1} = w_i^m$ or $w_i^{m+1} > D_i - Y_i$, in which case task τ_i is unschedulable. Assuming that it is schedulable, the worst case response time of τ_i is given by:

$$R_i = w_i + Y_i$$

5.1.3 Synchronisation

We now extend analysis of the Dual Priority model to permit hard tasks to lock and unlock semaphores according to the Ceiling Semaphore Protocol [82], commonly referred to as the Immediate Priority Ceiling Protocol. First, we show how this protocol can be applied to tasks with dual priorities. We then present analysis which bounds the worst case response time of each task.

Under the Immediate Priority Ceiling Protocol, each semaphore has an associated ceiling priority which is equal to the highest priority of any task which accesses that semaphore. With the Dual Priority approach, ceiling priorities are assigned on the basis of promoted (upper band) priorities. At run-time, when a process is granted a semaphore, its priority is immediately increased to the ceiling priority associated with the semaphore. Under the Immediate Priority Ceiling Protocol, the worst case blocking time which an invocation of task τ_i, executing at its promoted priority, can experience due to tasks in the set $lp(i)$ is denoted by B_i. Where B_i is the longest time any task of lower promoted priority than i can lock a semaphore with a ceiling priority of i or higher. Note, initially, we assume that no optional or soft tasks executing at priorities in the middle band share any semaphores with the hard tasks.

In deriving the worst case response time of task τ_i, we need to consider two cases.

Case 1: Immediately before the priority of task τ_i is promoted, task $\tau_k \in lp(i)$, executing at either its upper or lower band priority, locks a semaphore with a ceiling priority higher than *i*. Note, again we assume that task τ_i has been unable to carry out any computations at its lower band priority level. In this case the worst case response time of task τ_i follows directly from analysis of fixed priority scheduling.

$$w_i^{m+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil C_j \tag{5.2}$$

Note, the iteration start and completion conditions are the same as equation (5.1). The worst case response time of task τ_i remains:

$$R_i = w_i + Y_i$$

Case 2: Immediately before the priority of task τ_i is promoted, task $\tau_j \in hp(i)$, executing at its low band priority, locks a semaphore with a ceiling priority higher than *i*. In this case, task τ_i is apparently subject to blocking not accounted for in the derivation of equation (5.2). However, we can show that equation (5.2) does in fact still give us the worst case response time of task τ_i.

Due to the operation of the Immediate Priority Ceiling Protocol, only one task which has not yet reached its priority promotion time, can have locked any semaphores. We refer to this task as τ_j. To find the worst case response time of task τ_i, we consider the situation where τ_j locks a semaphore immediately before the priority promotion of task τ_i at time 0. Let y_j (≥ 0) be the time at which task τ_j is promoted to its upper band priority. Further, we assume that z_j is the length of time for which task τ_j will continue to hold any semaphores. The worst case time in a priority level *i* busy period w_i, during which task τ_j prevents task τ_i from executing is given below:

$$I_j(w_i) = \begin{pmatrix} C_j & \textbf{if} \quad w_i > y_i \\ z_i & \textbf{otherwise} \end{pmatrix} + \left\lceil \frac{w_i - y_j - T_j}{T_j} \right\rceil_0 C_j \tag{5.3}$$

As $z_j \leq C_j$ and $y_j \geq 0$, the maximum value of $I_j(w_i)$ occurs when $y_j = 0$, and the

above equation reduces to:

$$I_j(w_i) = \left\lceil \frac{w_i}{T_j} \right\rceil C_j \tag{5.4}$$

The worst case response time of task τ_i can therefore again be found via equation (5.2).

Resource Sharing between Hard and Soft Tasks

Resource sharing between hard and soft tasks can be facilitated by including in the calculation of B_i, the longest critical section of any task (hard or soft), of lower priority than i which may lock a resource of ceiling priority i or higher. Then at run time, the operation of the Immediate Priority Ceiling Protocol ensures that the blocking which a hard task τ_i is subject to is at most B_i irrespective of whether it is blocked by a hard or soft task.

5.1.4 Release Jitter

We now relax the restriction that all tasks are released as soon as they arrive. We assume that tasks can arrive at any time after their minimum inter-arrival interval, but may then be delayed for a variable time, bounded by the maximum release jitter J_i, before being released.

As the priority promotion point for each invocation of a task τ_i occurs a fixed time Y_i after its release, the analysis given in [10] is directly applicable. The maximum interference $I_j(w_i)$ occurs when an invocation of task τ_j with maximum release jitter J_j has a priority promotion point corresponding to the start of the priority i busy period and subsequent invocations of task τ_j have no release jitter. Thus the worst case response time of task τ_i is calculated as follows:

$$w_i^{m+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m + J_j}{T_j} \right\rceil C_j \tag{5.5}$$

Iteration again starts with $w_i^0 = 0$ and ends when $w_i^{m+1} = w_i^m$ or $w_i^{m+1} > D_i - Y_i - J_i$.

The worst case response time of task *i*, measured from its arrival, is given by:

$$R_i = w_i + Y_i + J_i$$

5.1.5 Arbitrary Deadlines

Here we relax the assumption that task deadlines are no greater than their periods, thus permitting tasks to have arbitrary deadlines ($D_i \leq T_i$ or $D_i > T_i$). Again, by virtue of considering priority promotion at a fixed offset from task release, we are able to directly apply the analysis given by Tindell *et al* in [105]. We assume that earlier invocations of a given task τ_i are executed in preference to later invocations of the same task, even though they share the same upper and lower band priority levels. Note that as earlier invocations always have their priority promoted in advance of later invocations this assumption remains valid with priority promotion.

The analysis given by Tindell, which is modified below, examines q ($q = 0,1,2,3...$) level i busy periods to determine the worst case response time of task τ_i. The length of each busy period $w_i(q)$ is found according to the following recurrence relation:

$$w_i^{m+1}(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m(q) + J_j}{T_j} \right\rceil C_j \qquad (5.6)$$

the response time of each invocation is then given by:

$$R_i(q) = w_i(q) - qT_i + J_i + Y_i$$

Examination of increasing values of q may stop if:

$$w_i(q) \leq (q+1)T_i$$

In which case, the worst case response time of task τ_i is given by;

$$R_i = \max_{q=0,1,2,3...} \left(w_i(q) - qT_i \right) + J_i + Y_i$$

5.1.6 Choosing Priority Promotion Times

Our motivation in developing the idea of Dual Priority scheduling is to provide a strategy which enables system utility to be increased whilst also ensuring that crucial hard tasks are guaranteed to meet their deadlines. To achieve this, we need to facilitate the responsive scheduling of soft tasks and the on-line guarantee of optional tasks with firm deadlines, both of which are assigned priorities in the middle band. (Note, on-line acceptance tests are elaborated in the next chapter). We therefore choose to set the values of Y_i as large as possible. To achieve this requires that the worst case response times R_i^{FP}, are calculated assuming all priority promotion times are zero (i.e. using exact analysis of fixed priority scheduling). The largest priority promotion times which still result in a feasible task set are then given by:

$$Y_i = D_i - R_i^{FP} \tag{5.7}$$

We note that using these values of Y_i potentially leads to the situation where a small computational overrun by a task τ_j with a high promoted priority may cause its deadline to be missed. However, we argue that this is indicative, not of a problem with Dual Priority scheduling but either, inaccurate worst case execution time analysis for tasks of promoted priority *j* or higher, or inaccurate analysis of overheads. In any case, the value of Y_j could be reduced to provide an engineered margin for error. Further, the Dual Priority approach retains the stability property of fixed priority scheduling: computational overrun by a task of low promoted priority (outside of a critical section) cannot cause tasks with higher upper band priorities to miss their deadlines.

It is interesting to note that we have not placed any restrictions on the priority assignments used within the lower and middle bands. Within the framework of the Dual Priority model, it is therefore possible to schedule tasks in these priority bands according to dynamic value density or best effort policies [68], we return to this issue in chapter 7.

5.1.7 Run-Time Operation of Dual Priority Scheduling

The basic operation of the algorithm is as follows: At each release of a hard task τ_i, its priority promotion time is set:

$$y_i = t^{REL} + Y_i \tag{5.8}$$

Where t^{REL} is the time of release. When time y_i is reached, the task's priority is promoted to its upper band level: *i*. Prior to time y_i, task τ_i executes at its lower band priority and may therefore be pre-empted by soft tasks executing at middle band priority levels.

Reclaiming Spare Time

The Dual Priority approach provides an efficient means of reclaiming spare time created when sporadic or adaptive tasks do not arrive at their maximum rate. Given that the earliest next release, $x_i(t)$, of such tasks is known, (typically, $x_i(t) = 0$ for a sporadic task τ_i which last arrived more than its minimum inter-arrival time ago), the Dual Priority strategy recognises that the sporadic task will not be promoted to the upper band until at least $y_i = x_i(t) + Y_i$, continually identifying spare time.

Reclaiming Gain Time

Let $y_i(t)$ be the priority promotion time for the current invocation of task τ_i at time *t*. Given that τ_i executes in the interval $[t, t')$, then its priority promotion time is extended:

$$y_i(t') = y_i(t) + (t' - t) \tag{5.9}$$

In essence, if the computation time of a task is reduced by $(t' - t)$ then its response time is reduced by at least $(t' - t)$ hence its priority promotion time may be increased by $(t' - t)$. It is therefore never necessary to interrupt a task to promote its own priority. Moreover, if task τ_i produces gain time g_i at time *t*, then

$$y_i(t) = y_i(t) + g_i \tag{5.10}$$

Furthermore, if task τ_i is executing at its promoted priority, when gain time is identified, then the priority promotion times of all tasks with lower (promoted) priorities are also potentially extended:

$$\forall j \in lp(i) \ : \ y_j(t) = \max(y_j(t), t + c_i(t) + g_i) \tag{5.11}$$

Effectively reclaiming the priority level i execution, corresponding to gain time g_i, for soft task execution.

5.1.8 Example

We now give an example of Dual Priority scheduling using the same hard and soft task sets used to illustrate the operation of various approaches to identifying spare capacity reviewed in chapter 2. The timing characteristics of the tasks are detailed in the tables below. Hard tasks *A* and *B* are initially released at time $t = 0$. Figure 5.1 shows the scheduling of these tasks under the Dual Priority scheduling algorithm. In this example, the Dual Priority approach allows soft tasks *w*, *x*, *y* and *z* to be serviced immediately, giving a mean soft task response time of 1.0; which compares favourably with other methods (see section 2.2.8).

Hard tasks					
Name	Priority	T	D	C	Promotion delay (Y)
A	4 then 1	8	6	2	4
B	5 then 2	12	12	5	3

Soft tasks		
Name	Arrival Time	Execution Time
w	1	0.5
x	2	1.5
y	13	1.0
z	14	1.0

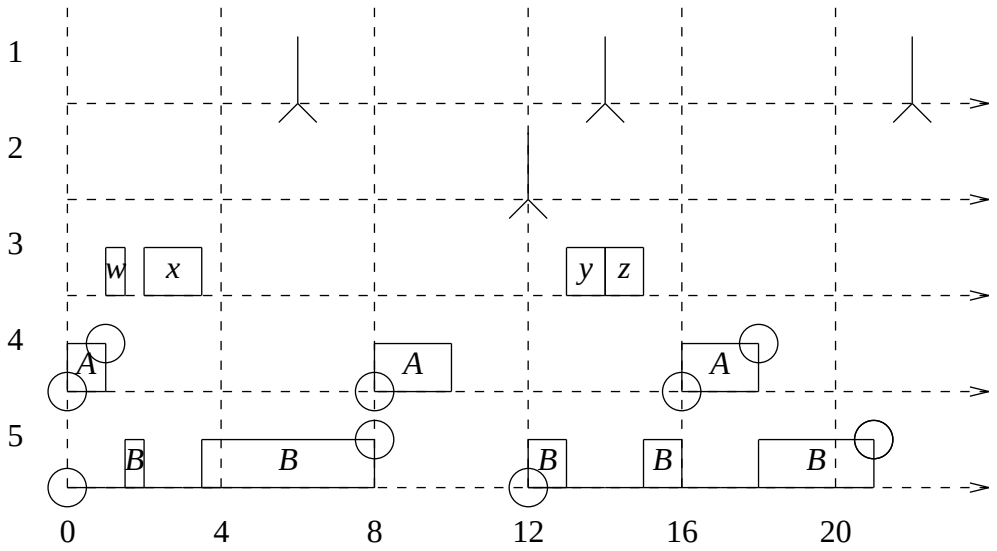
5.2 Performance Evaluation

In this section, we compare the performance of the Dual Priority approach to that of the optimal Slack Stealing algorithm, Background scheduling and Extended Priority Exchange algorithms. The sets of hard and soft tasks used in these simulations are the same as those examined in chapter 4. The criteria used to evaluate performance was the mean response time of soft tasks. The overheads involved in scheduling, maintaining counters for extra capacity, priority promotion times or slack, and calculating slack were assumed to be zero.

5.2.1 Periodic Tasks: Experiments 5.1-5.4

Experiments 5.1 to 5.4 examined the scheduling of hard task sets with utilisation levels of 30, 50, 70 and 90% respectively. In these experiments, all the hard tasks were periodic and every invocation required its worst case execution time. In each case, the soft task load was varied to give the total utilisation levels shown in the graphs.

In these experiments, the performance of the Dual Priority approach, although non-optimal was significantly better than that of the Extended Priority Exchange algorithm.



Dual Priority Scheduling:
1 and 2 are upper band priority levels, 3 is a middle band priority and 4 and 5 are lower band priority levels.
t=0, hard tasks *A* and *B* are released. The initial priority promotion times of tasks *A* and *B* are $y_1(0)=4$ and $y_2(0)=3$ respectively.
Task *A* begins executing at its initial priority of 4.
t=1, task *A* completes. soft task *w* arrives and begins executing at priority 3.
t=1.5, task *w* completes with a reponse time of 0.5. Task *B* executes at priority 5.
t=2. task *x* arrives. Task *B* has completed one unit of execution, thus its priority promotion time is extended to **t=4**. Task *x* executes.
t=3.5, task *x* completes, task *B* is resumed.
t=8, task *B* completes. Task *A* is released and executes at priority 4.
t=10, task *A* completes.
t=12, task *B* is released. Its priority promotion time is set to **t=15**.
t=13, task *y* arrives. As task *B* has executed for one unit, its priority promotion time is advanced to **t=16**. Task y executes.
t=14, task *y* completes with a response time of 1.0, task *z* arrives and executes.
t=15, soft task *z* completes in a response time of 1.0. Task *B* is resumed.
t=21, task *B* completes.
Mean Soft Task Response Time = 1.0.

Figure 5.1: Dual Priority Scheduling.

5.2.2 Gain Time: Experiments 5.5-5.8

The second set of experiments, investigated the response times of soft tasks given that the hard task set exhibited stochastic execution times, varying from 75-100% (expts. 5.5 and 5.7) and 50-100% (expts. 5.6 and 5.8) of their worst case execution times.

In each of these experiments, the Dual Priority approach again out-performed the Extended Priority Exchange algorithm.

5.2.3 Sporadic Tasks: Experiments 5.9 - 5.12

In this series of experiments, we evaluated the performance of the Dual Priority approach for hard task sets with a worst case utilisation of 70% or 90% and different proportions of periodic and sporadic tasks.

In experiments 5.9 and 5.11, each task with an even priority was designated sporadic, thus half of the tasks followed a periodic arrival pattern, the other half a sporadic pattern. In experiments 5.10 and 5.12, all the hard tasks were designated as sporadic.

In this series of experiments, the Dual Priority approach efficiently identified spare time and thus significantly out-performed the Extended Priority Exchange algorithm: soft task response times were 2 to 4 times shorter. Moreover, the performance of the Dual Priority approach was close to that of the PASS algorithm assuming that the latter had a short period of 40 - 160 ticks (See Experiments 4.9 - 4.12).

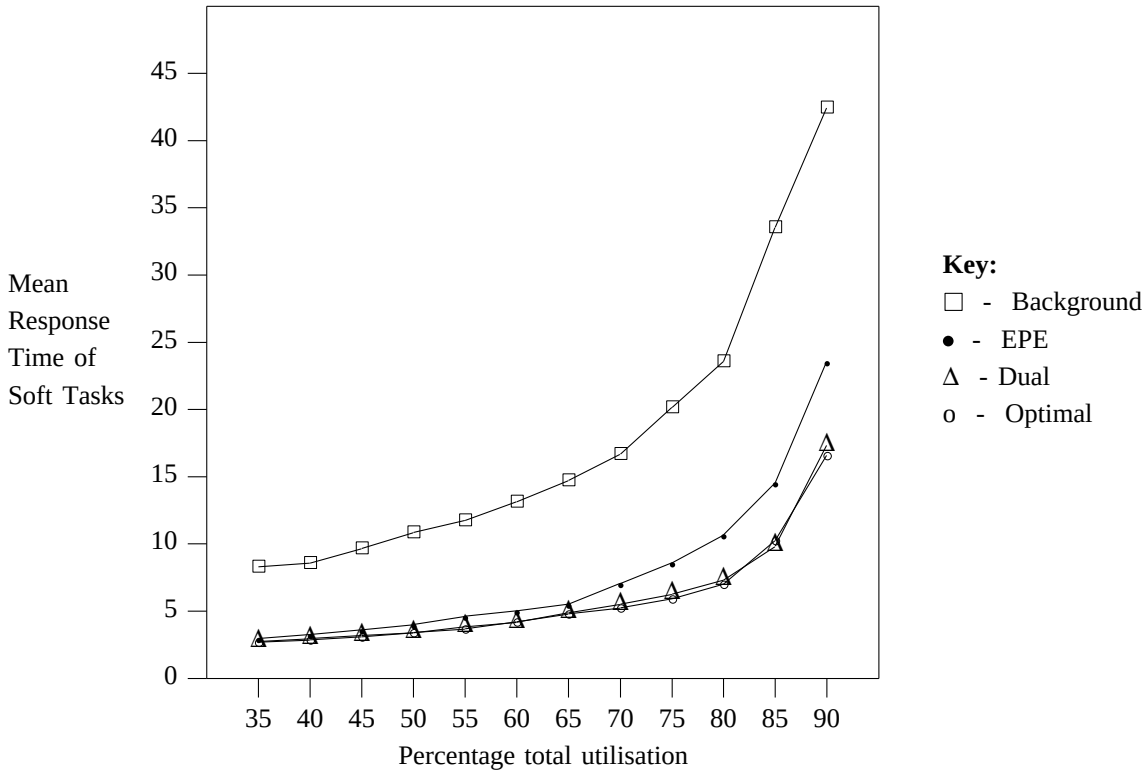
5.2.4 Adaptive Tasks: Experiments 5.13 - 5.16

In experiments 5.13 to 5.16, we examined the scheduling of hard task sets containing tasks which exhibited adaptive behaviour. In particular, at each adaptive task completion, the next release time was set to some randomly determined value between T_i and $2T_i$ since the last release.

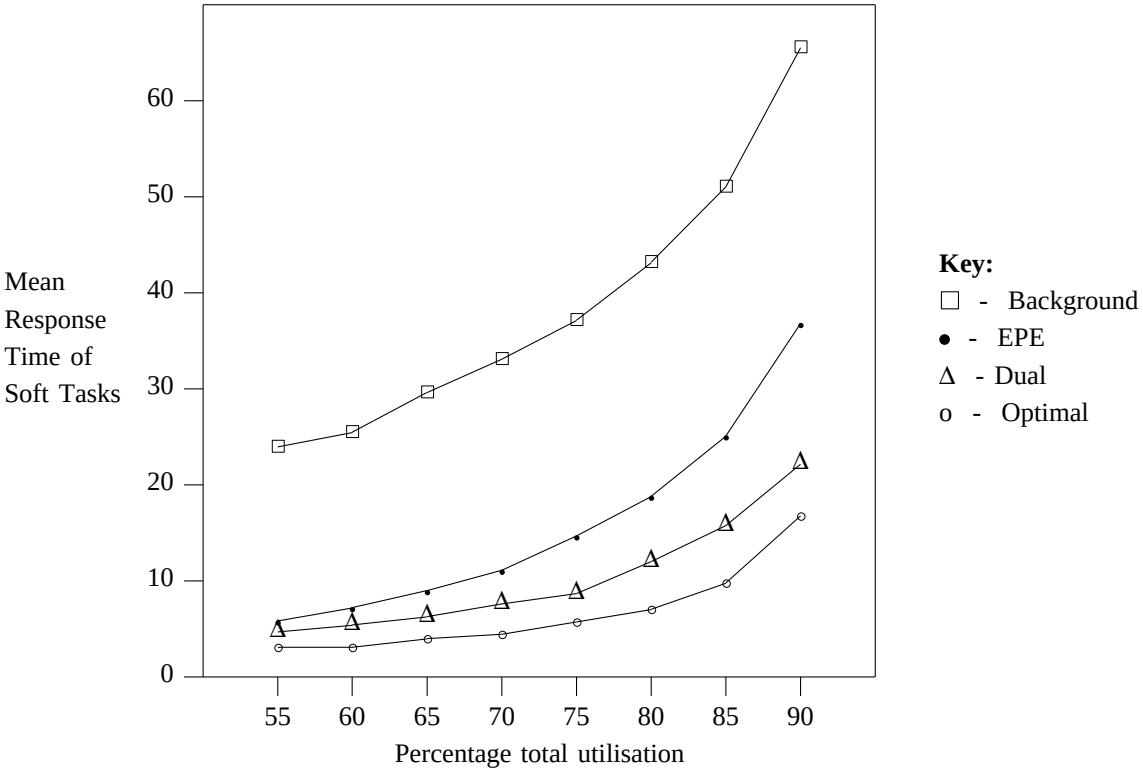
In these experiments, the performance of the Dual Priority approach far exceeded that of the Extended Priority Exchange algorithm. This is due to the ability of the former approach to reclaim spare time which becomes available when adaptive tasks do not arrive at their maximum rate.

5.2.5 Mixed Task Attributes: Experiments 5.17 - 5.18

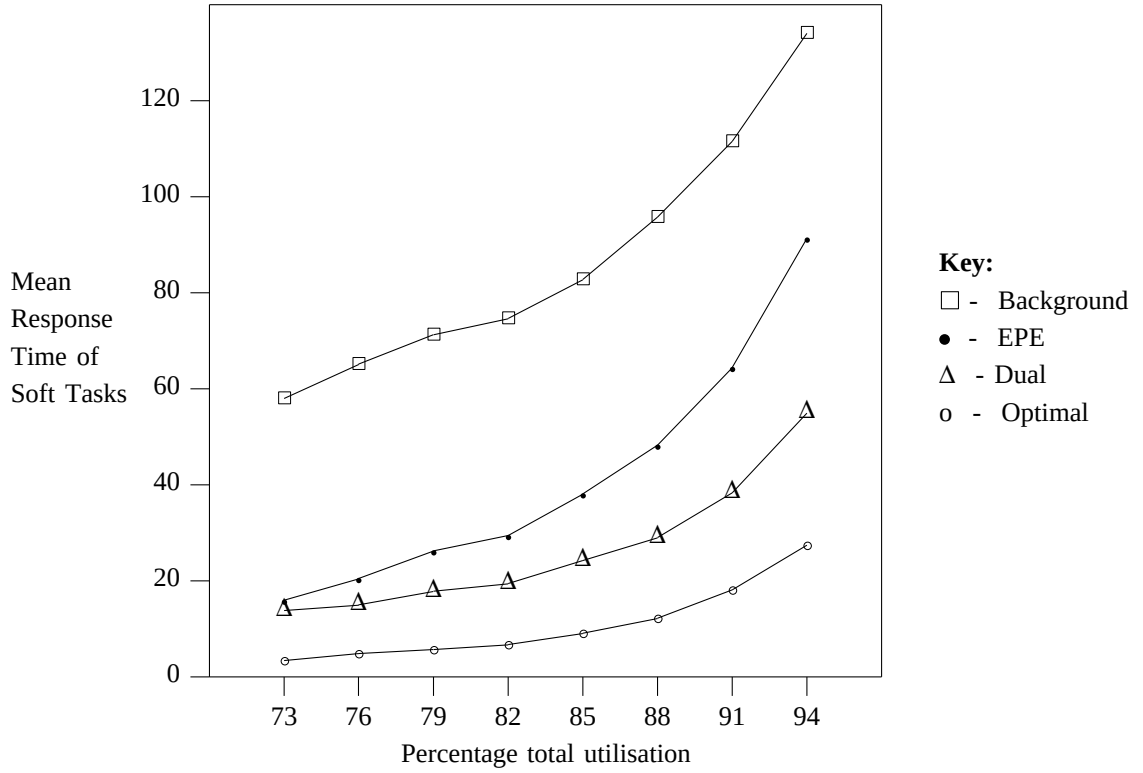
These experiments evaluated the performance of the Dual Priority approach for hard task sets exhibiting sporadic, adaptive and periodic release, and stochastic execution times. In these experiments, the Extended Priority Exchange algorithm is unable to make spare time available as anything other than a background service opportunity. In contrast, the Dual Priority approach is able to reclaim spare time brought about by the late arrival of adaptive and sporadic tasks, leading to significantly improved soft task response times.



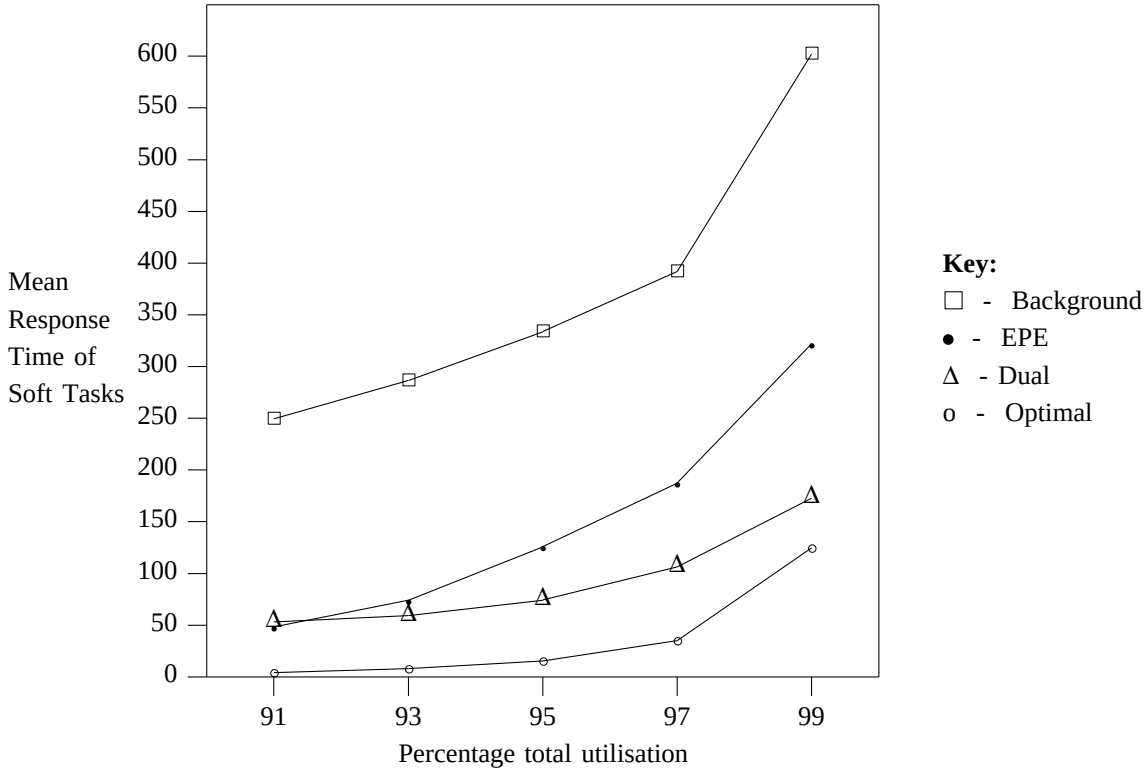
Expt. 5.1: Periodic tasks, 30% utilisation



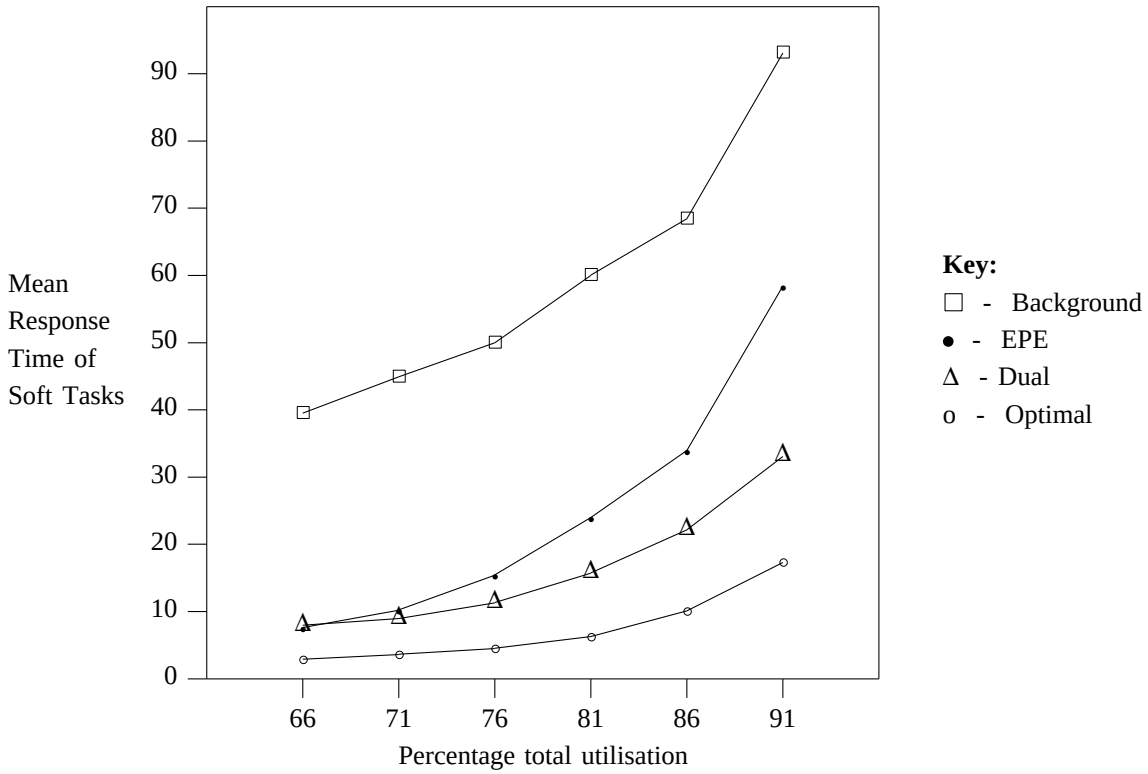
Expt. 5.2: Periodic tasks, 50% utilisation



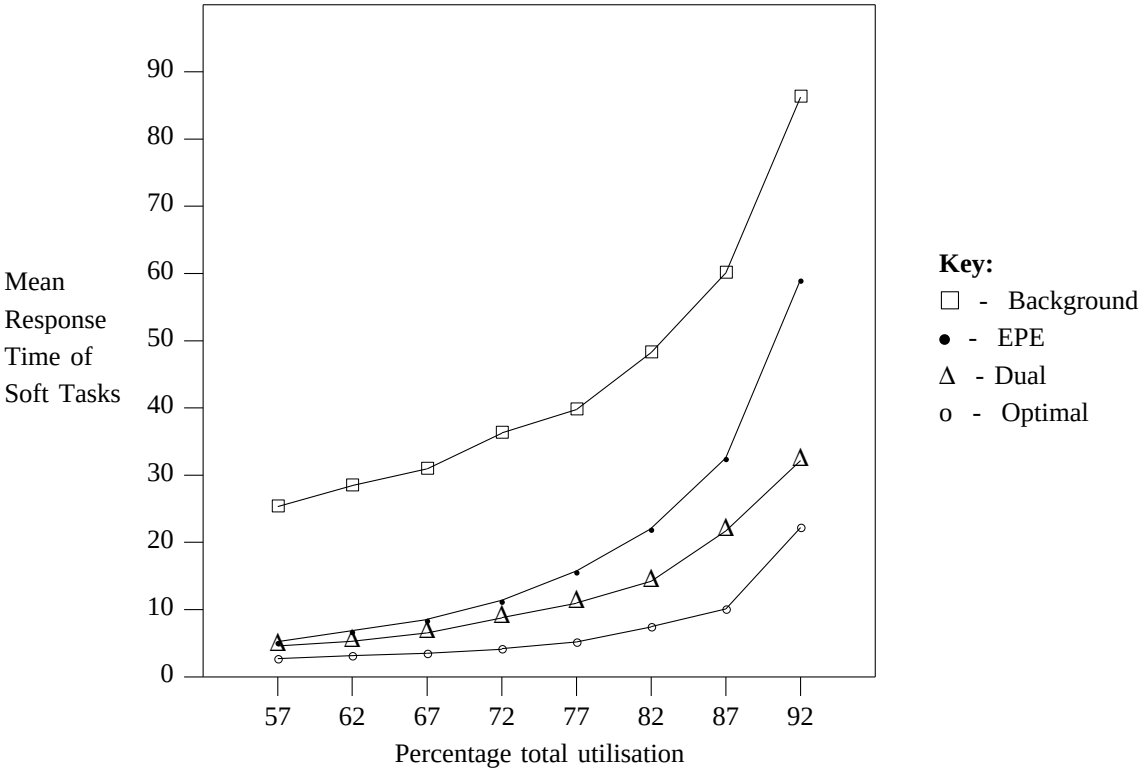
Expt. 5.3:Periodic tasks, 70% utilisation



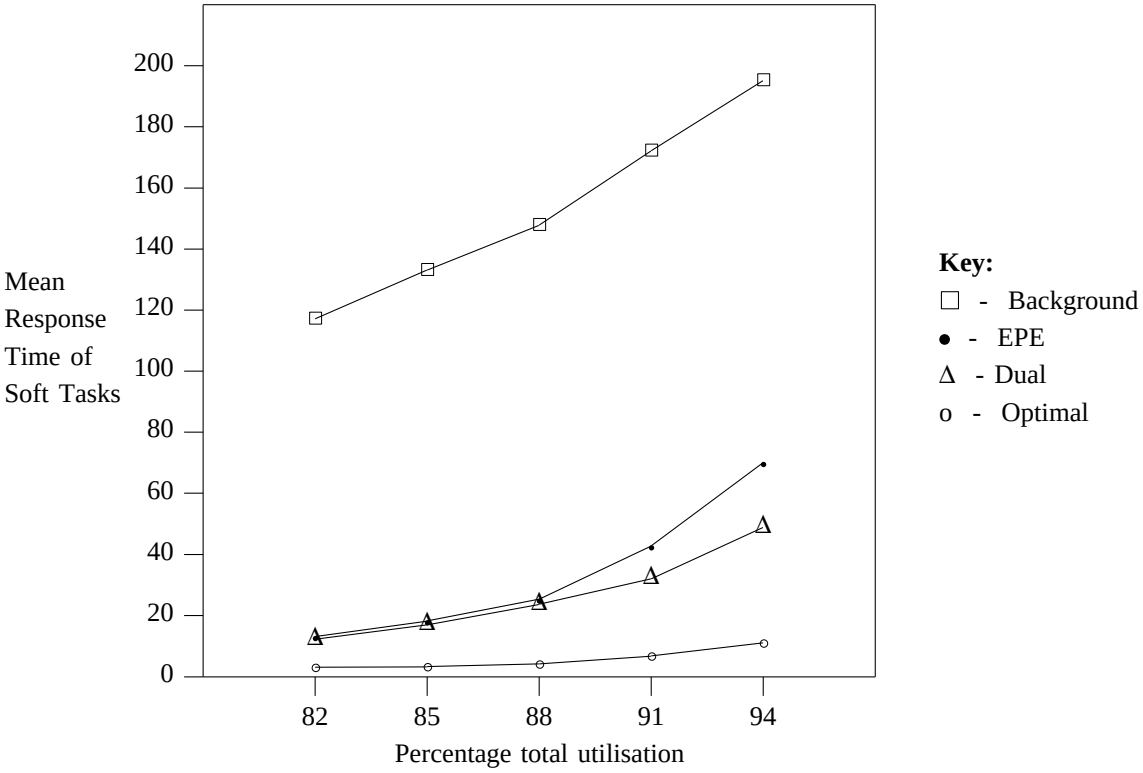
Expt. 5.4: Periodic tasks, 90% utilisation



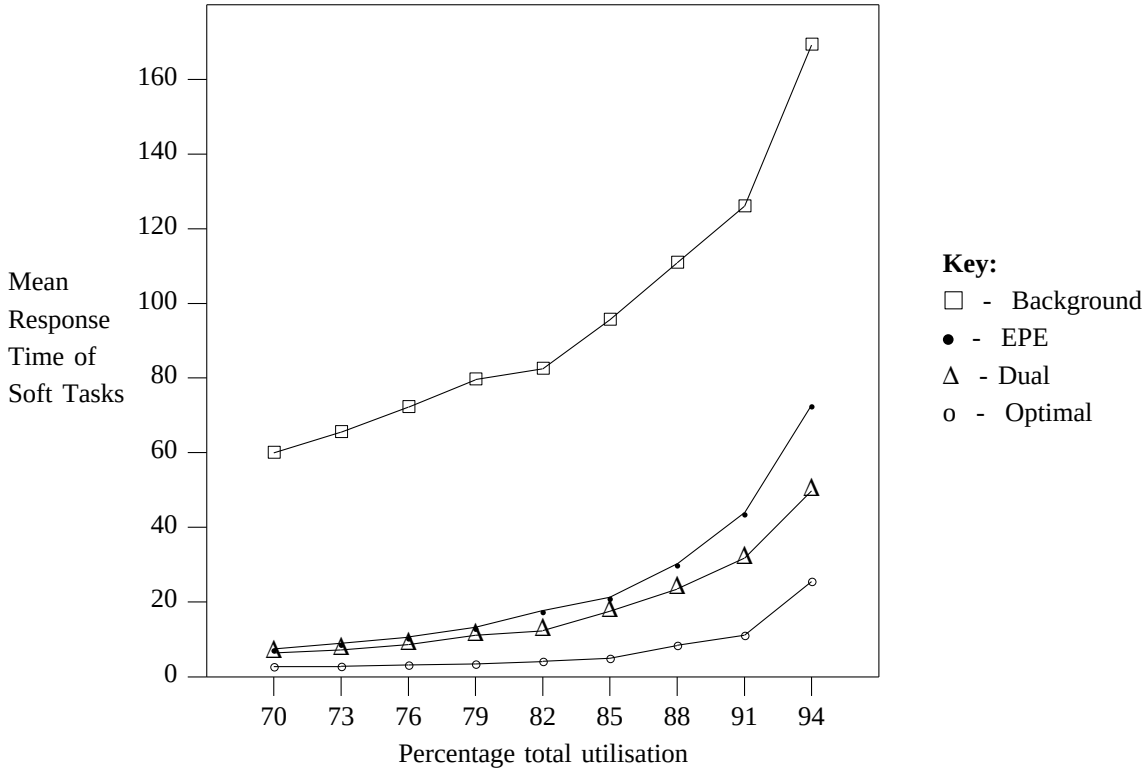
Expt. 5.5: Periodic tasks, 70% utilisation, 0-25% gain time



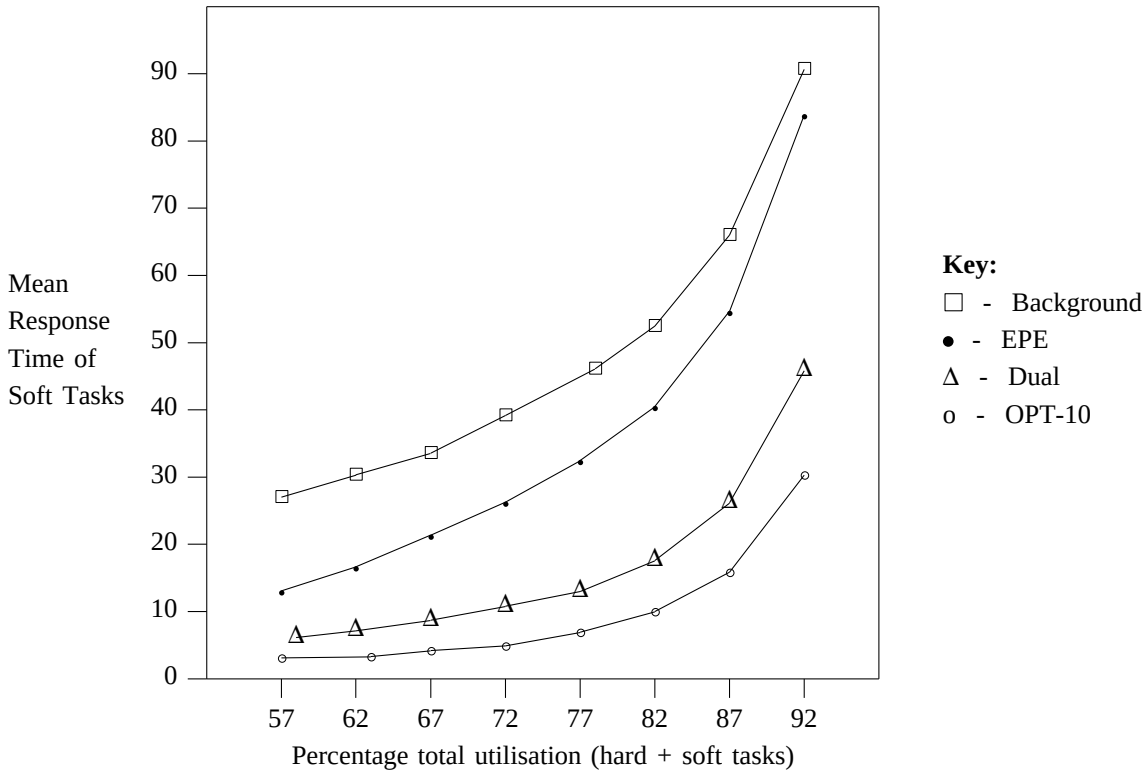
Expt. 5.6: Periodic tasks, 70% utilisation, 0-50% gain time



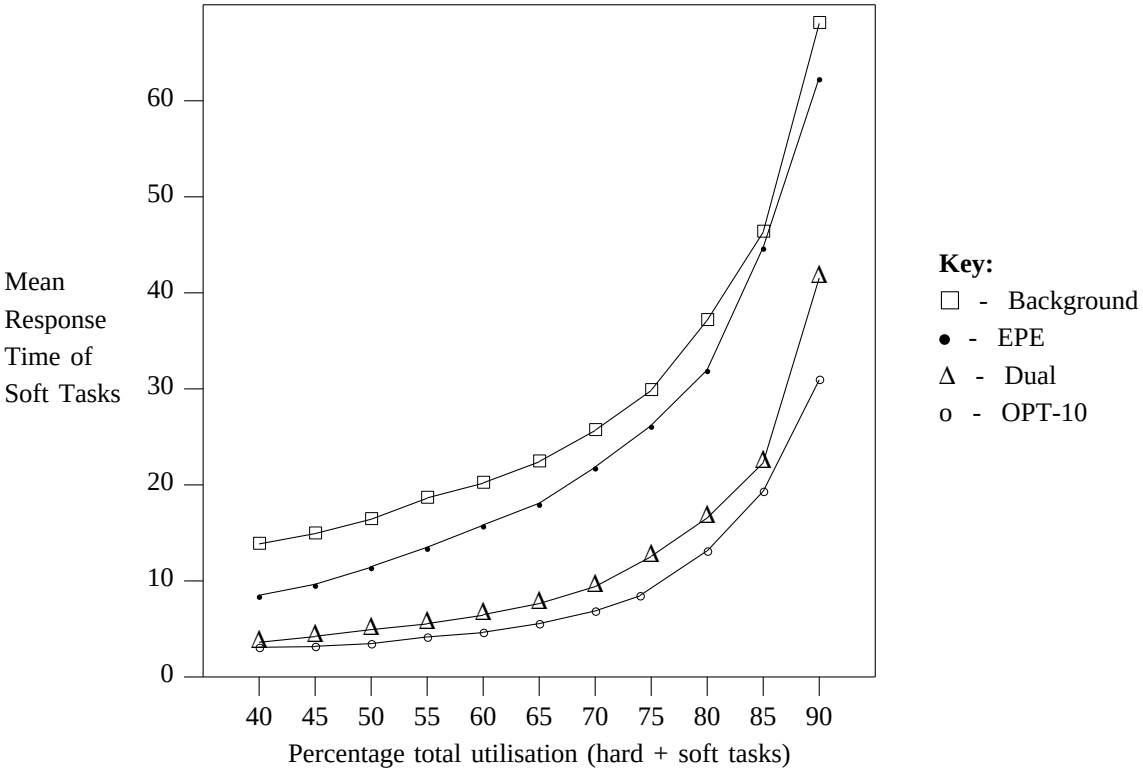
Expt. 5.7: Periodic tasks, 90% utilisation, 0-25% gain time



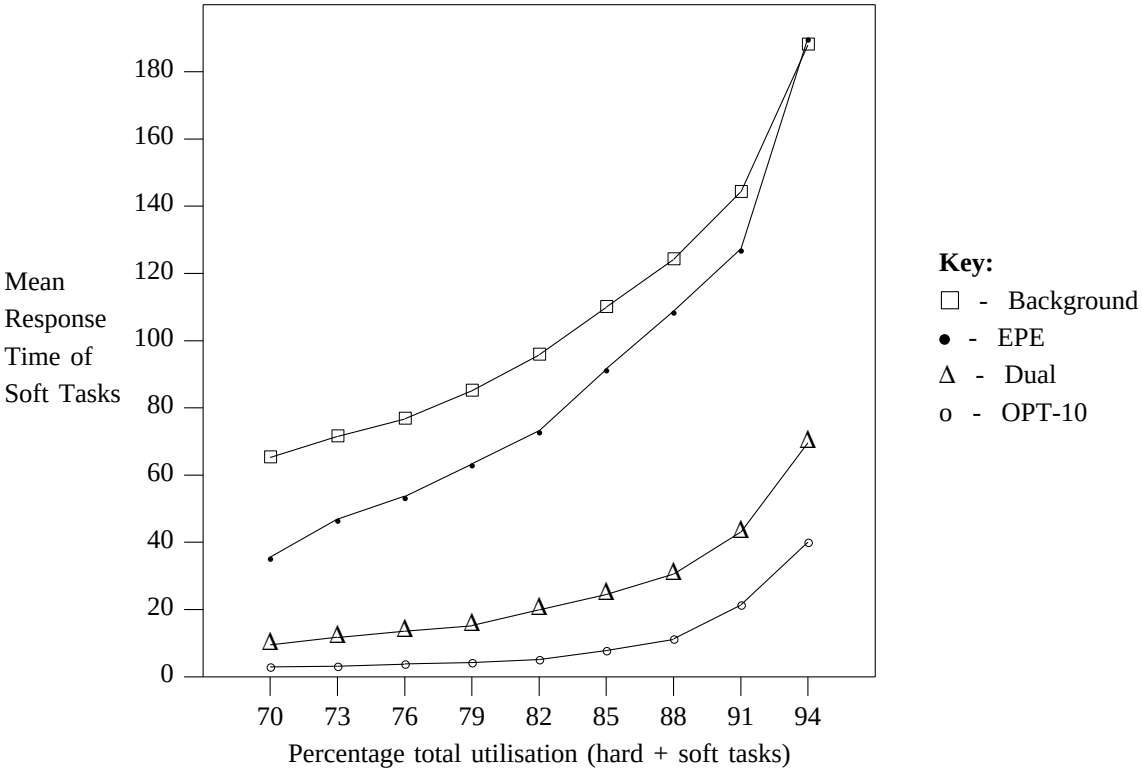
Expt. 5.8: Periodic tasks, 90% utilisation, 0-50% gain time



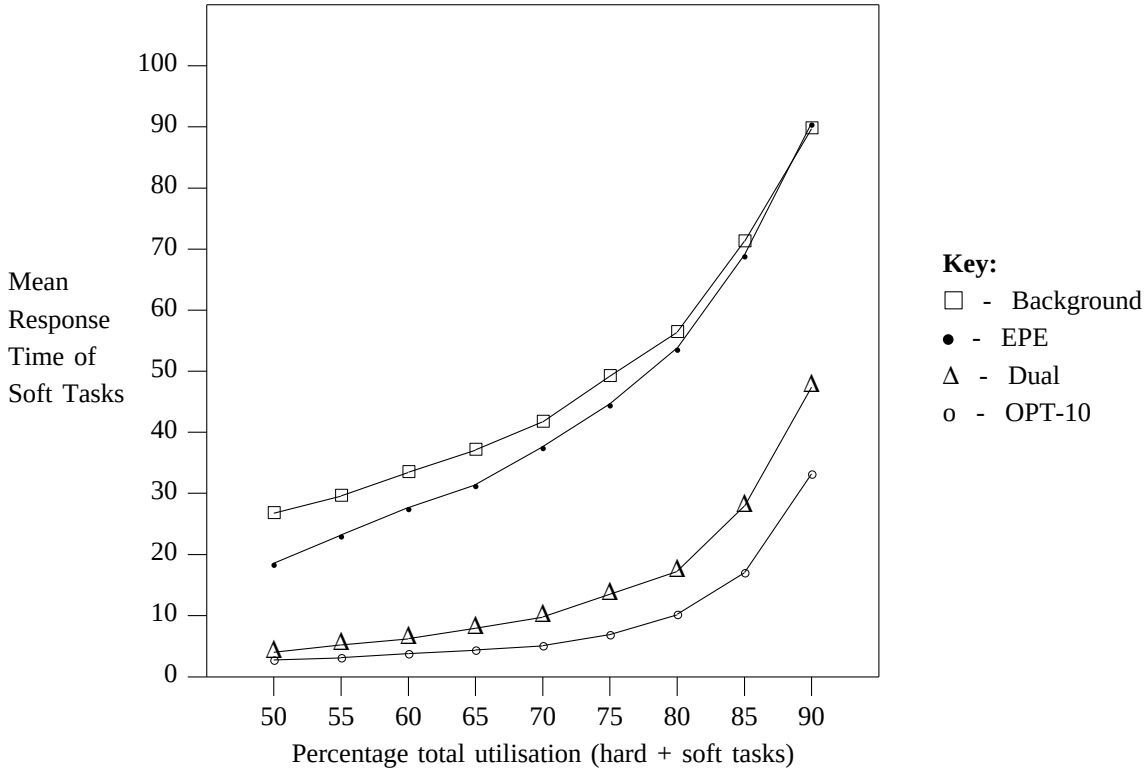
Expt. 5.9: Half sporadic, half periodic tasks, 70% utilisation



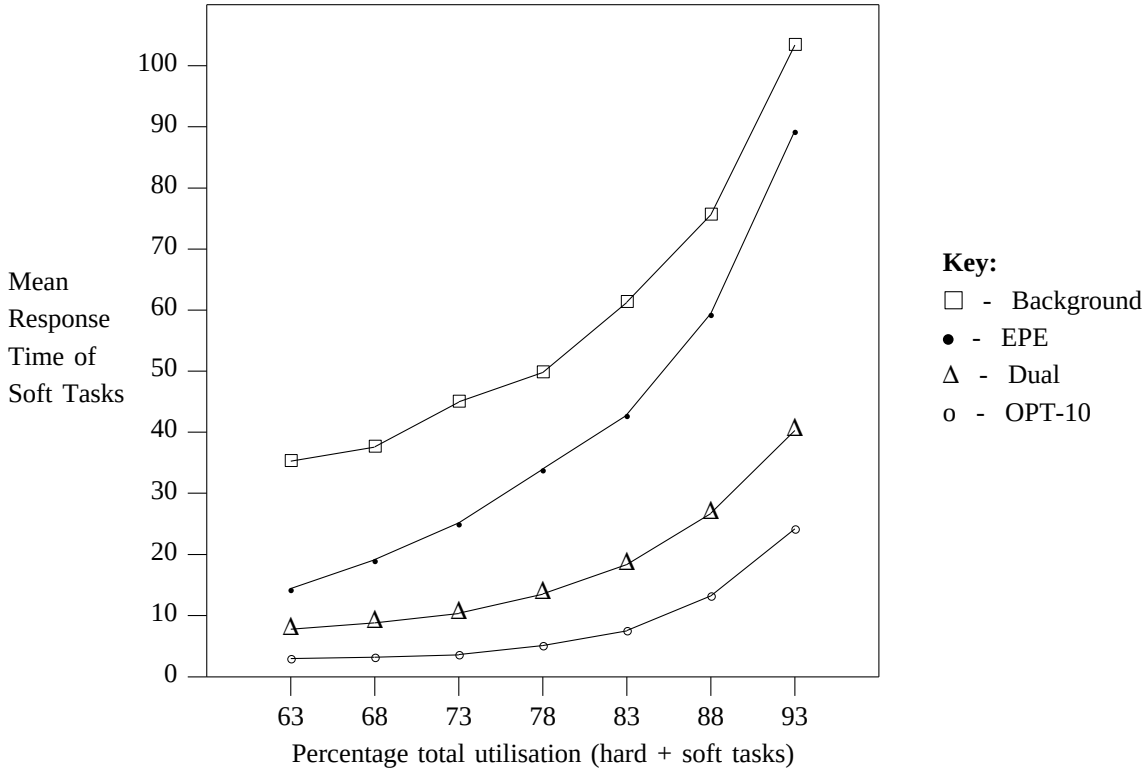
Expt. 5.10: All sporadic tasks, 70% utilisation



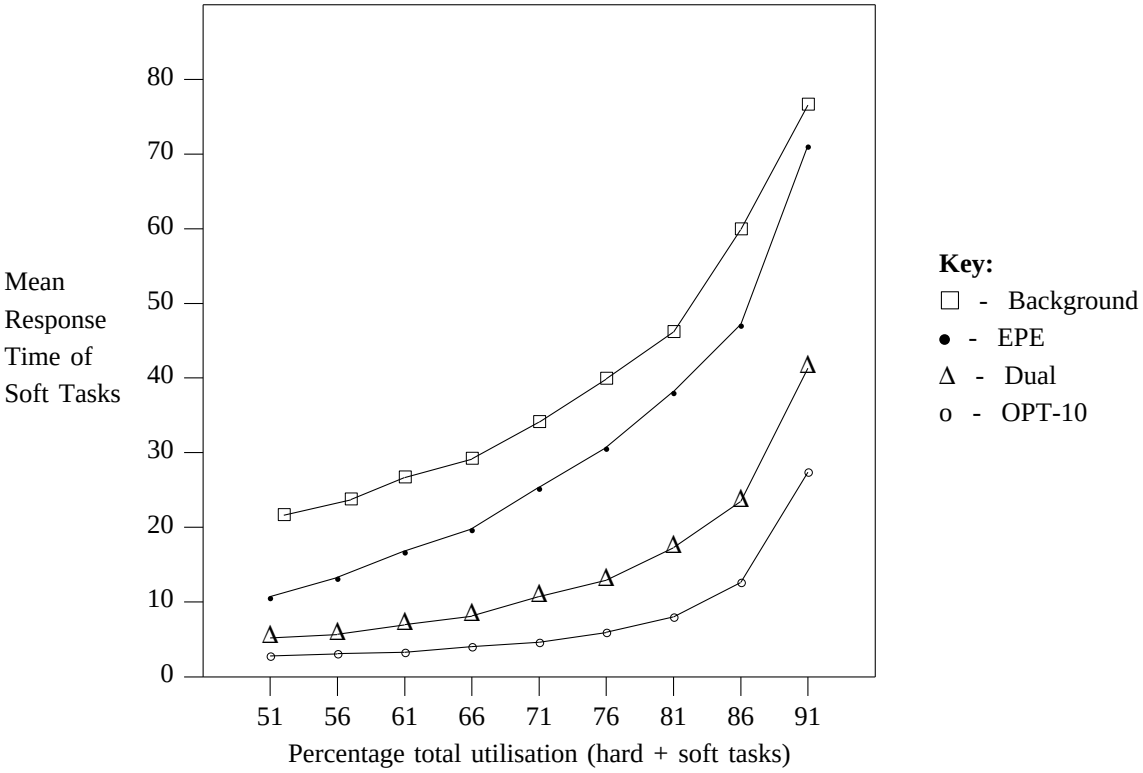
Expt. 5.11: Half sporadic, half periodic tasks, 90% utilisation



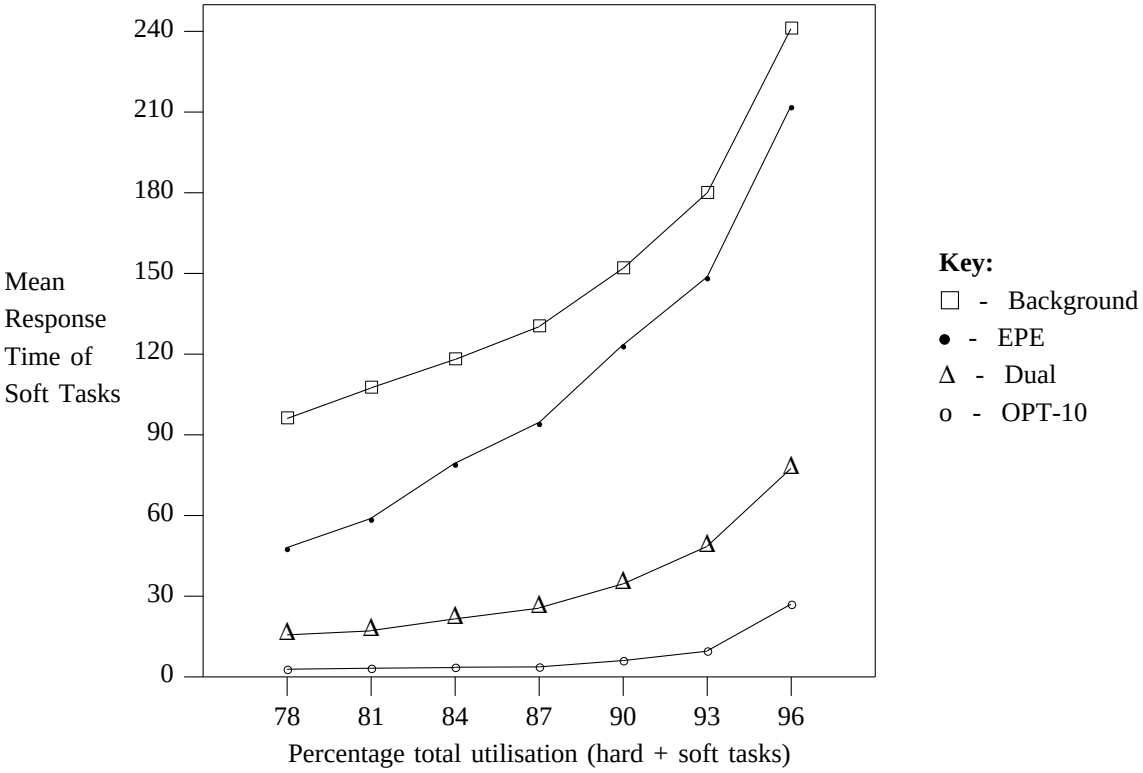
Expt. 5.12: All sporadic tasks, 90% utilisation



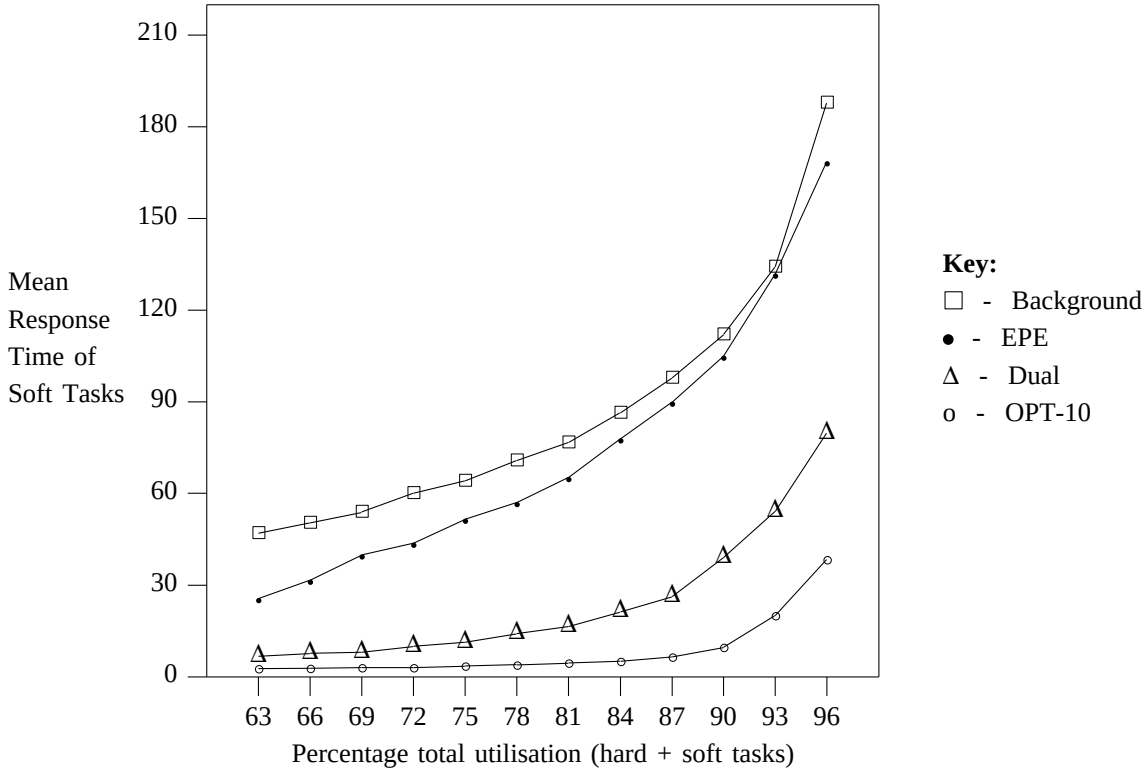
Expt. 5.13: Half Adaptive, half periodic tasks, 70% utilisation



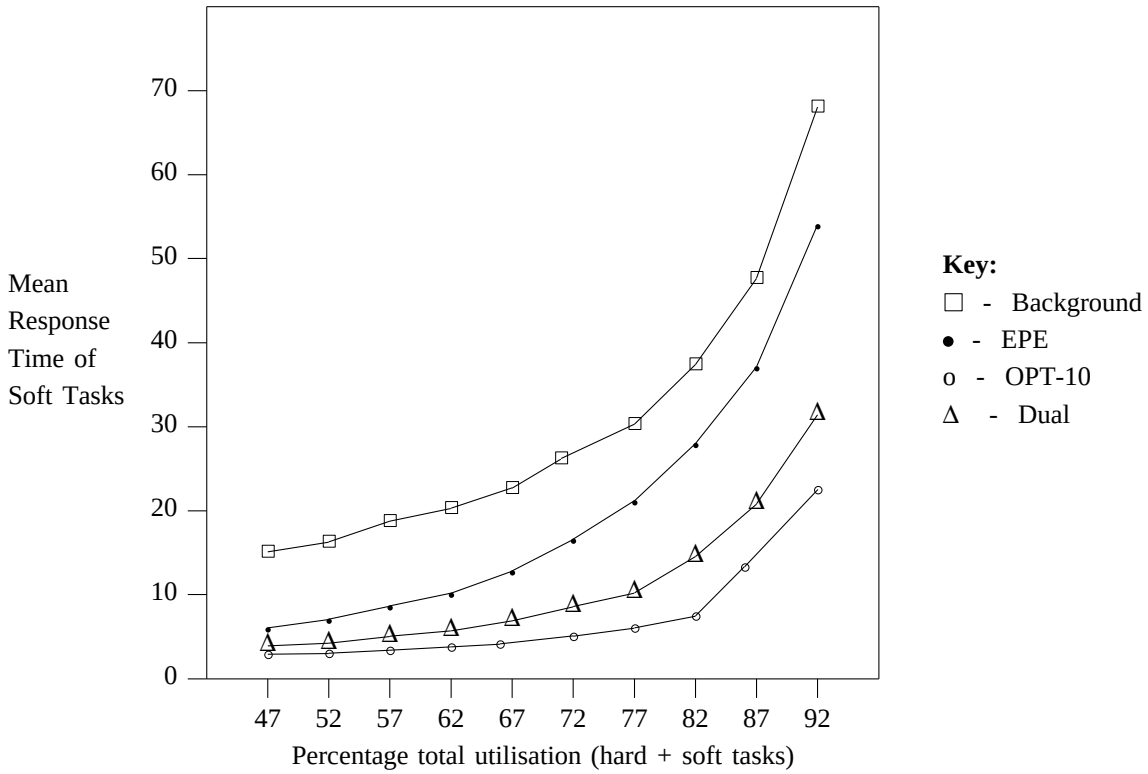
Expt. 5.14: All Adaptive tasks, 70% utilisation



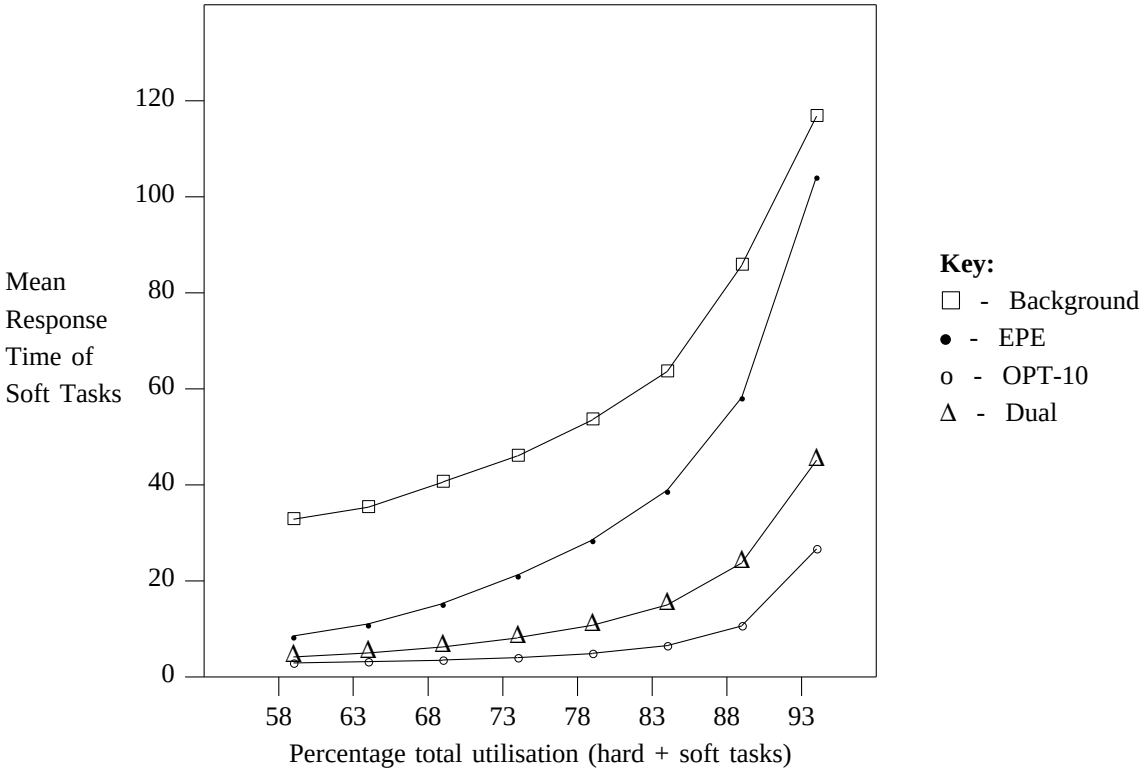
Expt. 5.15: Half Adaptive, half periodic tasks, 90% utilisation



Expt. 5.16: All Adaptive tasks, 90% utilisation



Expt. 5.17: Mixed periodic, sporadic, adaptive tasks, 70% utilisation, 0-50% gain time



Expt. 5.18: Mixed periodic, sporadic, adaptive tasks, 90% utilisation, 0-50% gain time

5.3 Overheads and Implementation Issues

Both the memory and execution time overheads of the Dual Priority scheduling strategy are low. Storage is required for n sets of the variables Y_i, y_i, $l_i(t)$ and $x_i(t)$. Where Y_i is the priority promotion delay calculated off-line, y_i is the priority promotion time for the current release of task τ_i, and $l_i(t)$ and $x_i(t)$ are the last release and earliest next release times respectively for task τ_i.

The execution time overheads of the Dual Priority approach stem from the need to promote the priorities of hard tasks. In addition to hard task release events, there is an equal number of priority promotion events. Inserting these extra events into the event queue represents, in the worst case, an $O(logn)$ overhead at each scheduling point (or release event). Note, this compares favourably with the dynamic slack stealing algorithms, investigated in the previous chapter, which typically require $O(n)$ time to recalculate the slack at each priority level. Further, in common with both Slack Stealing and Extended Priority Exchange algorithms, the worst case overhead of reclaiming gain time is $O(n)$. The issue of overheads is revisited in chapter 8.

5.4 Summary

In this chapter we introduced an alternative method of identifying spare capacity: Dual Priority Scheduling. In common with the dynamic Slack Stealing approach, Dual Priority scheduling is applicable to tasks sets with a wide range of timing characteristics, including sporadic or adaptive arrival patterns, task which exhibit blocking, release jitter and have stochastic execution times. Further, the Dual Priority approach has the advantage of low overheads and simple implementation.

We examined the performance of the Dual Priority approach with respect to responsively scheduling soft tasks. Comparisons were made with Background, Extended Priority Exchange and optimal Slack Stealing methods. The Dual Priority approach provided better performance than the Extended Priority Exchange algorithm in all our experiments. In particular, the Dual Priority approach is highly effective at reclaiming spare time produced when sporadic or adaptive tasks do not arrive at their maximum rate. For task sets with a significant sporadic component, the mean response time of soft tasks is significantly less (2-4 times better) than under the Extended Priority Exchange algorithm. Moreover, the run-time overheads of the Dual Priority approach are low: effectively $O(logn)$ at each hard task release.

Chapter 6

Guaranteeing Spare Capacity: Acceptance Tests

Under the Approximate Processing and Multiple Versions paradigms, optional computation aimed at improving the utility of hard real-time services needs to be guaranteed to complete by a given deadline. Similarly, alternative versions used in fault recovery also need to be guaranteed to complete by the deadline of the task which they support.

In this chapter, we introduce a family of acceptance tests which facilitate the on-line guarantee of optional components, represented by aperiodic tasks with firm deadlines. Furthermore, we present an optimal priority assignment policy for aperiodic tasks with arbitrary ready times and firm deadlines, scheduled along with a set of hard periodic / sporadic tasks. This priority assignment policy is shown to be optimal both in terms of maximising the computation time which can be made available before a given aperiodic deadline and with respect to guaranteeing subsequent aperiodic arrivals.

Acceptance tests are presented for aperiodic tasks scheduled at a fixed priority level under the Slack Stealing algorithm. The first test is valid for aperiodic tasks which share a deadline with one of the hard periodic / sporadic tasks (as is often the case when the aperiodic task represents optional computation aimed at improving upon the results of a mandatory task). Subsequently, more general acceptance tests are given for aperiodic tasks with arbitrary deadlines. These latter tests are extended to aperiodic tasks scheduled according to Background or Dual Priority strategies. Finally, we compare the performance of the various exact and sufficient acceptance tests for Background, Slack Stealing and Dual Priority scheduling methods. The performance metric used is the percentage of the total requested aperiodic task execution time which can be guaranteed. The affects which acceptance test overheads have on performance are also considered.

6.1 Overview of Previous Research

Research into on-line acceptance tests has been carried out by Chetto and Chetto [91] Schwan and Zhou [85] and Kim [54] with respect to Earliest Deadline scheduling. However, when applied to the case of mixed task sets, (aperiodic and periodic) the tests presented in [91] and [85] are pseudo-polynominal in complexity: in the worst case the time taken to perform the tests depends upon the number of task invocations within the least common multiple of the hard task periods. Further, the model used by Schwan and Zhou requires that all task invocations, both periodic and aperiodic, undergo an on-line acceptance test, leading to unnecessarily high overheads. The test presented by Kim also requires that all task invocations undergo acceptance testing. Of greater consequence however, is the implicit assumption that tasks with an earlier ready time are of greater importance and are therefore accepted in preference to those with a later ready time. When mixed task sets are considered this assumption no longer holds and it is difficult to see how the algorithm can be applied without jeopardising the hard deadlines of periodic tasks.

Within the context of fixed priority scheduling, Thuel and Lehoczky have developed two approximate acceptance tests [83, 84]. These tests assume that tasks are scheduled using the static Slack Stealing algorithm [59]. Unfortunately, as shown in section 2.3.1, both of these tests are insufficient: guarantees may be given to aperiodic tasks which will then miss their deadlines.

Subsequently, Thuel and Lehoczky developed a revised acceptance test [96] which makes use of the algorithm for calculating slack given in section 3.2. When an aperiodic task arrives, this acceptance test evaluates the amount of processing time which the static Slack Stealing algorithm can make available for aperiodic processing prior to the aperiodic deadline. This is achieved by enacting the operation of the Slack Stealing algorithm up to the aperiodic's deadline as follows:

1. An aperiodic task τ_A arrives at time a_A with an absolute deadline of d_A. The aperiodic processing time available is initially zero. Let time $s = a_A$.

2. The aperiodic processing time available $A^*(a_A, d_A)$ is incremented by A' - the minimum of the interval from s to the aperiodic's deadline and the least slack at any priority level at time s:

$$A' = \min\left((d_A - s), \min_{\forall j \in lp(1)} S_j(s) \right)$$

3. The task invocation with the least slack (and the lowest priority, in the case of a tie) is identified. Let this be invocation τ_{jk} (i.e. task τ_j invocation k). The latest finish time F_{jk} for τ_{jk} is found from a table.

4. If $F_{jk} \geq d_A$ then iteration is complete and $A^*(a_A, d_A)$ gives the available aperiodic execution time. Otherwise, the slack available at each priority level is updated to reflect the execution of tasks $\tau_i \in hp(j)+j$ up to time F_{jk} and the aperiodic execution time A' identified in step 2: Initially,

$$\forall i : S_i(F_{jk}) = S_i(s) - A'$$

Then for each invocation τ_{il} of task τ_i which will execute in the interval $[s, F_{jk})$ and each priority level, m:

$$S_m(F_{jk}) = \begin{cases} S_m(F_{jk}) + C_{il}(s) & \forall m \in hp(i) \\ S_m(F_{jk}) + S(F_{il}) & \text{if } m = i \end{cases}$$

Where $S(F_{il})$ is the increment in priority i slack at the completion of invocation τ_{il}, assuming that it completes at its latest finish time F_{il}. The value $S(F_{il})$ is found by reference to a table, with entries of the form $\{F_{il}, S(F_{il})\}$ created off-line using a variant of algorithm 3.3 (previously published in [39]). $C_{il}(s)$ is the outstanding computation time of invocation τ_{il} at time s. (Note, if τ_{il} has not been released at time s, then $C_{il} = C_i$). Finally, s is set to F_{jk} and steps 2 - 4 are repeated.

Provided $A^*(a_A, d_A)$ is greater than the execution requirements of the aperiodic task then it can be guaranteed.

6.1.1 Example

Figure 6.1 gives an example of the operation of the above acceptance test, for an aperiodic task τ_A arriving at time *t*, with an absolute deadline, d_A of 15 and an execution requirement C_A of 6, given the hard task set detailed in the table below.

Hard tasks					
Priority	WCET	Period	Deadline	Next Release	Next Deadline
1	2	5	5	t+3	t+8
2	3	10	10	t+3	t+13
3	1	21	21	t+1	t+22

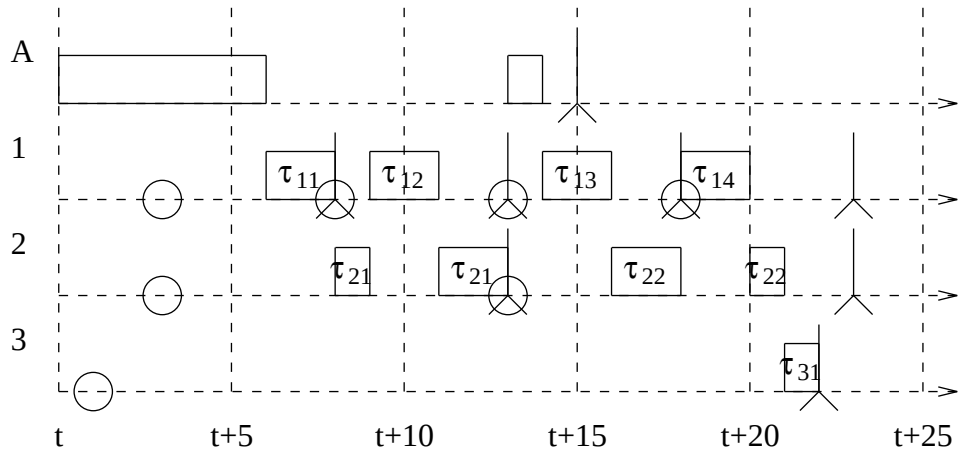
$$S_m(F_{jk}) = \begin{cases} S_m(F_{jk}) + C_{il}(s) & \forall m \in hp(i) \\ S_m(F_{jk}) + S(F_{il}) & \text{if } m = i \end{cases}$$

The latest finish times of invocations τ_{11}, τ_{12}, τ_{13}, τ_{14}, τ_{21}, τ_{22} and τ_{31} are given below, along with the corresponding slack increments.

$$F_{11} = t + 8 \quad S(F_{11}) = 3 \quad F_{12} = t + 13 \quad S(F_{12}) = 3$$
$$F_{13} = t + 18 \quad S(F_{13}) = 3 \quad F_{14} = t + 23 \quad S(F_{14}) = 3$$
$$F_{21} = t + 13 \quad S(F_{21}) = 3 \quad F_{22} = t + 23 \quad S(F_{22}) = 3$$
$$F_{31} = t + 22 \quad S(F_{31}) = 6$$

The aperiodic processing time available before the deadline of τ_A is 7, hence τ_A is schedulable.

The pseudo-polynomial time and space complexity of the above test severly limits its applicability as an on-line acceptance test in real systems. In particular, the table used to store latest finish times and slack increments requires an entry for each task invocation in the LCM of task periods. Hence this test places the same restrictions upon the hard task set as the static Slack Stealing algorithm [59]: all hard tasks must be strictly periodic and the LCM must be manageably small. In section 6.3, we address these issues by introducing an exact and a sufficient acceptance test, both of which are applicable to a more general computational model. First, however, we present a complementary priority assignment policy.



Time available for aperiodic task execution is shown on timeline A. Note, task execution is as scheduled by the Slack Stealing algorithm.

At time *t*, aperiodic task τ_A arrives, with a deadline d_A of $t+15$.
Acceptance test: $s=t$, $S_1(s)=6$, $S_2(s)=6$, $S_3(s)=7$ and $d_A-s=15$.
Hence the first interval of aperiodic processing A^1 is of length 6.
$A^*(t,d_A)=6$. τ_2 will be the lowest priority task with zero slack once $A^1=6$ units of aperiodic processing have taken place. $F_{21}=t+13$.
Invocations τ_{11}, τ_{12} and τ_{21} complete before F_{21}, hence:
$S_1(F_{21})=S_1(s)-A^1+S(F_{11})+S(F_{12})-C_{21}(s)=6-6+3+3-3=3$
$S_2(F_{21})=S_2(s)-A^1+S(F_{21})=6-6+3=3$
$S_3(F_{21})=S_3(s)-A^1=7-6=1$
Let $s=F_{21}=t+13$. $S_1(s)=3$, $S_2(s)=3$, $S_3(s)=1$ and $d_A-s=2$.
Hence the second interval of aperiodic execution A^2 is of length 1, limited by the slack at priority level 3. $F_{31}=22$.
τ_{13}, τ_{14}, τ_{22} and τ_{31} all complete in the interval $[s,F_{31})$.
Hence:
$S_1(F_{31})=S_1(s)-A^2+S(F_{13})+S(F_{14})-C_{22}(s)-C_{31}(s)$
$=3-1+3+3-3-1=4$
$S_2(F_{31})=S_2(s)-A^2+S(F_{22})-C_{31}(s)=3-1+3-1=4$
$S_3(F_{31})=S_3(s)-A^2+S(F_{31})=1-1+6=6$
Let $s=F_{31}=t+22$. Iteration is now complete as $s>d_A$.
Aperiodic processing time available $A^*(t,d_A)=7$**.**

Figure 6.1: Static Slack Stealing: Acceptance test.

6.2 Priority Assignment

Below, we introduce an optimal priority assignment policy for aperiodic tasks with firm deadlines, given a set of previously guaranteed hard periodic / sporadic tasks. The latter tasks are assumed to have been assigned unique fixed priorities according to some arbitrary policy and guaranteed, via off-line feasibility analysis, to meet their deadlines. In contrast, priority assignment and acceptance testing of aperiodic tasks is carried out on-line.

The following strategy is assumed for handling aperiodic tasks with firm deadlines. Upon arrival, each aperiodic task is assigned a unique priority before being acceptance tested. The acceptance test is in two parts: first the schedulability of the new aperiodic task is determined. Second, it is necessary to check that admitting the aperiodic task will not cause any previously guaranteed tasks to miss their deadlines. Provided that all deadlines can be met, then the aperiodic task is guaranteed and executed at its assigned priority level. This strategy has the advantage of simple implementation: once guaranteed, aperiodic tasks are handled in exactly the same way as hard periodic or sporadic tasks: only fixed priority pre-emptive dispatch is required. (Note, this strategy differs from that used by Thuel and Lehoczky [96], where aperiodic tasks are serviced in processing time made available by the Slack Stealing algorithm at the highest priority level, as illustrated in figure 6.1).

Assuming the scheduling strategy described above, an optimal priority assignment policy is defined which maximises the computation time that can be afforded to an aperiodic task without causing any deadlines to be missed. Further, we show that this priority assignment policy is also optimal in terms of being able to guarantee subsequent aperiodic arrivals.

6.2.1 Computational Model and Notation

We consider a set of hard sporadic / periodic tasks which comply with the computational model given in section 2.1.1. Further, the run-time variables introduced in chapter 3 are also used, in particular, $c_i(t)$ is the computation due to hard task τ_i, outstanding at time t and $d_i(t)$ is its next deadline. We also consider an aperiodic

task τ_A which may arrive and be ready to execute at any arbitrary time *t*. We assume that τ_A requires C_A units of computation to be completed by its firm deadline at $d_A(t) = t + D_A$, where D_A is its relative deadline. If accepted, τ_A will be assigned some unique priority *j*, and executed in preference to hard tasks in the set $lp(j)$. We assume that acceptance testing of τ_A is performed upon arrival, i.e. at time *t*. We also assume that the tasks do not exibit blocking and do not voluntarily suspend themselves.

6.2.2 Optimal Priority Assignment

Given the above assumptions, we seek to find the optimal priority level, P_A^{OPT}, at which to schedule aperiodic task τ_A. The definition of optimality is as follows: if τ_A is schedulable at any priority level, then it is also schedulable at priority level P_A^{OPT}. Equivalently, P_A^{OPT} is the priority level at which τ_A can be given the maximum computation time consistent with the task set, including τ_A, remaining schedulable.

We denote the maximum permissible computation time at priority level *i*, which may be released at time *t* and complete by time $d_A(t)$, as A_i. (Note, under certain circumstances, an increased amount of execution time may be available in the interval $[t, d_A(t))$ by delaying acceptance testing and priority assignment. See [97] and section 6.2.4). A_i is constrained first by interference due to higher priority tasks: all A_i units of computation must be completed at priority level *i*, by $d_A(t)$. Secondly, A_i is constrained by the slack at lower priority levels: executing τ_A must not cause a lower priority task to miss its deadline. Hence:

$$A_i = \min\left((D_A - I_{hp(i)}(t, D_A)), \min_{\forall j \in lp(i)} S_j(t) \right) \tag{6.2}$$

Where $I_{hp(i)}(t, D_A)$ is the interference from tasks in the set $hp(i)$ which falls into the interval $[t, t + D_A)$ and $S_j(t)$ is the slack at priority level *j*, corresponding to the maximum additional interference which task τ_j may be subject to without causing its next deadline to be missed. (Note, both $I_{hp(i)}(t, D_A)$ and $S_j(t)$ can be calculated from the next release, next deadline and remaining execution time budget of each task, see section 6.3.4).

Definition : The priority level P_A^{OPT} for an aperiodic arrival, τ_A, is the highest priority level such that every task of lower priority than P_A^{OPT} has an absolute deadline later than $d_A(t)$.

Theorem 6.1: Assigning τ_A priority P_A^{OPT}, results in the maximum execution time being available to τ_A, before its deadline, consistent with no deadlines being missed.

Proof : The proof is in two parts; first we show that assigning τ_A a higher priority than P_A^{OPT} cannot result in a greater available execution time. Second, we show that neither can assigning τ_A a lower priority.

Case 1 : τ_A is assigned a priority level k, where $k \in hp(P_A^{OPT})$.

Let $A_{P_A^{OPT}}$ be the maximum permissible execution time for τ_A when assigned priority P_A^{OPT}. There are two sub-cases to consider:

Sub-case 1.1 : $A_{P_A^{OPT}}$ is limited by interference due to tasks in the set $hp(P_A^{OPT})$.

Let j be the priority level immediately higher than P_A^{OPT}. From the definition of P_A^{OPT}, $d_j(t) \le d_A(t)$. As $d_j(t) \le d_A(t)$, at least $S_j(t)$ units of computation must be available at priority level P_A^{OPT} prior to $d_A(t)$. Hence $A_{P_A^{OPT}} \ge S_j(t)$. Now if τ_A is assigned a priority $k \in hp(j)$, then A_k is limited by $S_j(t)$, hence:

$$\forall k \in hp(P_A^{OPT}) \ : \ A_k \le S_j(t) \le A_{P_A^{OPT}} \tag{6.3}$$

Sub-case 1.2 : $A_{P_A^{OPT}}$ is limited by the slack $S_j(t)$ at some lower priority level j.

Assigning τ_A a priority level $k \in hp(P_A^{OPT})$ does not change the set of tasks $hp(j)$, thus the slack at priority level j remains the same and hence:

$$\forall k \in hp(P_A^{OPT}) \ : \ A_k \le S_j(t) = A_{P_A^{OPT}} \tag{6.4}$$

Case 2 : τ_A is assigned a priority level k, where $k \in lp(P_A^{OPT})$.

Again there are two sub-cases to consider:

Sub-case 2.1 : $A_{P_A^{OPT}}$ is limited by interference due to tasks in the set $hp(P_A^{OPT})$.

Reducing the priority of τ_A from P_A^{OPT} to k can only result in additional interference from tasks in the set $lp(P_A^{OPT}) \cap hp(k)$, thus:

$$\forall k \in lp(P_A^{OPT}) \ : \ A_k \leq A_{P_A^{OPT}} \tag{6.5}$$

Sub-case 2.2 : $A_{P_A^{OPT}}$ is limited by the slack, $S_j(t)$ at a lower priority level j.

First, A_k, where $k \in lp(P_A^{OPT}) \cap hp(j)$ cannot exceed the slack on τ_j and hence it cannot exceed $A_{P_A^{OPT}}$. (Note, $S_j(t)$ is unaffected by re-ordering the priorities of tasks in the set $hp(j)$). Second, consider τ_A assigned a priority level $k \in lp(j) - j$. The computation due to task τ_j and tasks in the set $hp(j)$ excluding τ_A, results in slack time $S_j(t)$ being available before $d_j(t)$. Assigning τ_A a priority level k, lower than j, constrains the aforementioned computation to the shorter interval $[t, d_A(t))$, as $d_j(t) > d_A(t)$. Hence:

$$\forall k \in lp(j) \ : \ A_k \leq S_j(t) = A_{P_A^{OPT}} \tag{6.6}$$

□

We note that the above proof also extends to the slightly more complex problem where a number of aperiodic tasks have previously been accepted. Once accepted, and until completed, these earlier aperiodic arrivals are treated in an identical manner to the hard tasks. Further, if no hard tasks are present at all, then the priority assignment policy reduces to the well known Earliest Deadline first policy, which is optimal for scheduling a stream of feasible aperiodic tasks [41].

Example

Below, we give a simple example illustrating the optimal priority assignment policy and showing how the maximum available execution time decreases as the assigned priority level departs from the optimal.

The example is based upon the hard task set detailed in the table below.

Hard tasks						
P_i	D_i	T_i	$d_i(t)$	C_i	$c_i(t)$	$S_i(t)$
1	4	10	t+4	2	2	2
2	18	18	t+18	2	2	12
3	20	20	t+12	3	2	4
4	25	25	t+25	1	1	9
5	30	30	t+30	3	3	10

In addition to the hard tasks given in the table, we also consider an aperiodic task τ_A which arrives at time t and has an absolute deadline $d_A(t)$ of $t+17$. Note, we assume that if τ_A is assigned the same priority as a hard task, then if both tasks are runnable, τ_A is executed first. Figure 6.2 illustrates the fixed priority schedule for the hard task set in the absence of τ_A and gives a graphical illustration of how the available aperiodic processing time varies with priority level.

6.2.3 Optimality w.r.t Scheduling Subsequent Aperiodic Arrivals

Next, we consider how the priority assigned to an aperiodic task τ_A affects the schedulability of subsequent aperiodic arrivals.

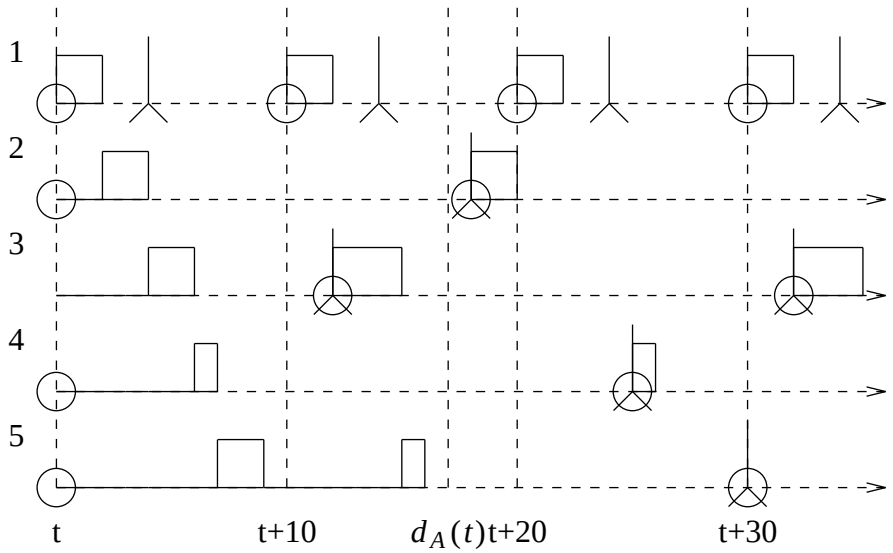
Theorem 6.2 : If an aperiodic arrival τ_B is schedulable with a previous aperiodic arrival τ_A scheduled at any arbitrary priority level, then it is also schedulable with τ_A assigned to its optimal priority level, P_A^{OPT}.

Proof: The proof is in two parts; first we consider the case where τ_B has a later deadline than τ_A, and second the case where it has an earlier deadline.

Case 1 : τ_B has a later deadline than τ_A.

Sub-case 1.1 : τ_A is assigned a priority level k, where $k \in hp(P_B^{OPT})$.

From Theorem 6.1, the optimal priority level for an aperiodic task is lower than that of any task with an earlier deadline. Hence when τ_A is assigned its optimal priority, it is a member of the set of tasks with a higher priority than P_B^{OPT}. Now, assuming



The optimal priority level for τ_A is 4, corresponding to the highest priority level such that all tasks of a lower priority have a later deadline than $d_A(t) = t + 17$.

The aperiodic processing time available before $d_A(t)$ at each priority level is as follows:
Priority 1: $A_1 = 2$, limited by the slack at priority level 1.
Priority 2: $A_2 = 4$, limited by the slack at priority level 3.
Priority 3: $A_3 = 4$, again limited by the slack at priority level 3.
Priority 4: $A_4 = 6$, limited by interference from tasks τ_1, τ_2, τ_3.
Priority 5: $A_5 = 5$, again limited by higher priority interference.
Priority 6: $A_6 = 2$, also limited by higher priority interference.
In accordance with theorem 6.1, the optimal priority level for τ_A is 4.

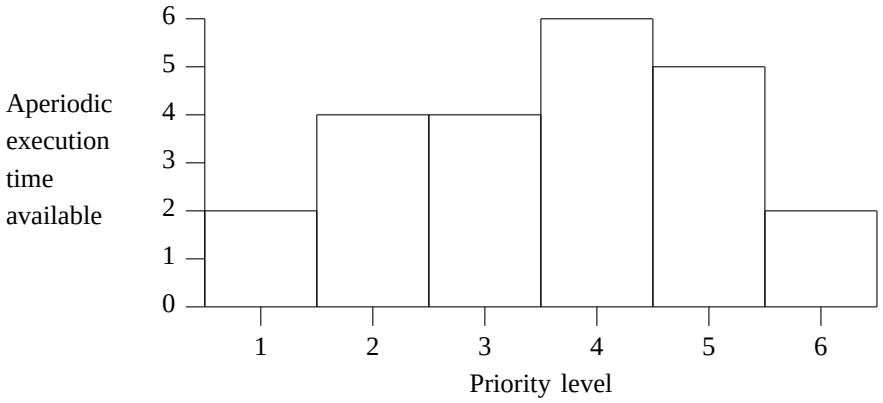


Figure 6.2: Aperiodic processing time at each priority level.

that τ_A is assigned a priority k, rather than P_A^{OPT}, where $k \in hp(P_B^{OPT})$, then the set of tasks $hp(P_B^{OPT})$ is unchanged. Hence the interference which τ_B is subject to is independent of the priority level to which τ_A is assigned, (provided that it is higher than P_B^{OPT}). Further, the set of tasks $lp(P_B^{OPT})$ is also unchanged, hence, any constraint due to slack at these lower priority levels is also unaltered. The maximum permissible execution time for τ_B is therefore constant for τ_A assigned any priority level, including P_A^{OPT}, which is higher than P_B^{OPT}.

Sub-case 1.2 : τ_A is assigned a priority level k, where $k \in lp(P_B)$ and P_B is the optimum priority level for τ_B when τ_A is assigned its optimum priority.

If τ_A is assigned a priority level k, which is lower than P_B, then the optimal priority level for τ_B, P_B^{OPT} is also lowered. In fact P_B^{OPT} is immediately below k (i.e. $k+1$). This is equivalent to case 2 in the proof of Theorem 6.1. Hence the maximum permissible execution time for τ_B at priority $k+1$ can be no greater than it is at priority P_B when τ_A is assigned priority P_A^{OPT}. Hence scheduling τ_A at priority level P_A^{OPT} or indeed any priority level higher than P_B^{OPT} results in the maximum possible execution time becoming available for τ_B.

Case 2 : τ_B has a earlier deadline than τ_A.

From Theorem 6.1, the optimal priority level for τ_B, P_B^{OPT}, is independent of the priority level to which τ_A is assigned.

Sub-case 2.1 : τ_A is assigned a priority level k, where $k \in hp(P_B^{OPT})$

Let A_B^* be the maximum permissible execution time for τ_B when both τ_A and τ_B are assigned their respective optimal priority levels. Now, if A_B^* is limited by interference from higher priority tasks, then assigning τ_A a priority $k \in hp(P_B^{OPT})$, can only serve to increase this interference.

Alternatively, A_B^* may be constrained by the slack at some lower priority level j, then if P_A^{OPT} is a higher priority level than j, assigning τ_A priority $k \in hp(P_B^{OPT}) \subset hp(j)$ has no effect on the slack at priority j, whilst if P_A^{OPT} is a lower priority level than j, assigning τ_A priority $k \in hp(P_B^{OPT})$ can only reduce the slack at priority j. Either way, the permissible computation for τ_B cannot exceed A_B^*.

Sub-case 2.2 : τ_A is assigned a priority level k, where $k \in lp(P_B^{OPT})$

Again let A_B^* be the maximum permissible execution time for τ_B when both τ_A and τ_B are assigned their respective optimal priority levels. Now, if A_B^* is limited by interference from higher priority tasks, then assigning τ_A any priority level $k \in lp(P_B^{OPT})$ has no effect on this interference. As P_A^{OPT} is a lower priority level than P_B^{OPT}.

Alternatively, A_B^* may be constrained by the slack, $S_j(t)$ at some lower priority level j. Assuming that P_A^{OPT} is a higher priority level than j, then assigning τ_A to any priority level $k \in hp(j)$ has no effect on the slack at priority level j. Further, assigning τ_A a priority level lower than j, results in the execution time available for task τ_A becoming less than or equal to the original slack at priority level j (Case 2.2 of Theorem 6.1). If P_A^{OPT} is a lower priority level than j, then assigning τ_A to any priority level $k \in hp(j)$ can only serve to reduce $S_j(t)$ and hence also reduce the computation time available to τ_B. Further, assigning τ_A to any priority level $k \in lp(j)$ has no effect on $S_j(t)$.

Finally, if the slack on task τ_A at priority level P_A^{OPT} constrains A_B^*, then assigning τ_A any other priority level cannot result in a greater amount of slack (Theorem 6.1). Hence scheduling τ_A at a higher or lower priority level than P_A^{OPT} cannot result in more execution time becoming available for τ_B.

□

6.2.4 When to Perform an Acceptance Test?

As a corollary to Theorem 6.1, with aperiodic task τ_A scheduled at its optimal priority level, all outstanding computation with an earlier deadline than $d_A(t)$ must complete before τ_A can execute. Carrying out this computation before acceptance testing τ_A cannot reduce the maximum time which can be made available for τ_A. On the contrary, these higher priority tasks may not take their worst case execution time, increasing the time available for τ_A. It is therefore advisable to delay priority assignment and acceptance testing until all outstanding computation with an earlier deadline has completed.

We note that this delay in acceptance testing is not necessarily optimal in achieving the maximum available execution time for a given aperiodic task as shown by the following simple example (derived from a similar example given by Tia *et al* [97] illustrating soft aperiodic task scheduling).

Example : Consider a single hard periodic task τ_1, characterised by $D_1 = 10$, $T_1 = 10$, $C_1 = 5$, which is first released at time $t = 2$. Further, consider an aperiodic task τ_A with a relative deadline of 17 which arrives at time $t = 0$. At $t = 0$ scheduling τ_A at priority 0 (i.e. at a higher priority than τ_1), results in a maximum permissible execution time of 7 units for the aperiodic task: limited by the slack on the first invocation of τ_1. Similarly, applying the optimal priority assignment policy, and assigning τ_A a priority of 2 (lower than τ_1), 7 units of computation time are available, limited by interference from the first two invocations of τ_1. However, if acceptance testing and priority assignment is delayed until $t = 7$ when the first invocation of τ_1 completes, the aperiodic task may be scheduled at what is then the optimal priority level, priority 0, with a maximum of 10 units of computation time permissible.

In the general case, determining the optimal delay requires prior knowledge of actual task execution times and therefore clairvoyancy.

6.3 Acceptance Tests for the Slack Stealing Algorithm

In this section, we present a family of acceptance tests, which facilitate the on-line guarantee of aperiodic tasks with firm deadlines. First we give a simple test for aperiodic tasks which share a deadline with one of their hard sporadic or periodic counterparts. Exact and sufficient acceptance tests are then given for aperiodic tasks with arbitrary deadlines, scheduled at a fixed priority level. Subsequent sections present similar acceptance tests for Background and Dual Priority Scheduling.

6.3.1 Exact Test: Aperiodics with Specific Deadlines

Optional computation supporting a hard real-time task typically inherits the same deadline as the mandatory computation. In this case, a very simple acceptance test is possible. Assuming that the aperiodic task τ_A, representing optional computation, is

assigned the same priority as its mandatory counterpart τ_i, then τ_A can be guaranteed to complete by its deadline, provided that:

$$\min_{\forall j \in lp(i)} S_j(t) \geq C_A \tag{6.7}$$

In which case, τ_A is accepted and the slack at all priority levels *i* or lower is reduced accordingly:

$$\forall j \in lp(i) \ : \ S_j(t) := S_j(t) - C_A \tag{6.8}$$

Note, this is the exact opposite of task τ_i producing gain time.

6.3.2 Generic Acceptance Test Procedure and Computational Model

When the deadline of an aperiodic task does not correspond to that of any of the hard tasks, then a more complex acceptance test is required. The basic operation of such exact and sufficient acceptance tests, detailed in subsequent sections, is as follows: when a firm aperiodic task arrives, it is first given a unique priority commensurate with its absolute deadline, according to the optimal priority assignment policy described in section 6.1. That is the highest priority such that all the guaranteed tasks (i.e. hard periodic / sporadic tasks and uncompleted firm aperiodic tasks) with lower priorities have later absolute deadlines. The acceptance test then proceeds by determining if the new aperiodic task and all other tasks of lower priority are schedulable. If this is the case, then the new aperiodic is afforded an on-line guarantee. Further, it is given an execution time budget and a slack budget, which represents the amount of extra interference which the task could be subject to without causing its deadline to be missed. It is then executed whenever there are no higher priority tasks runnable.

In formulating each acceptance test, we assume that the set of hard tasks comply with the computational model outlined in section 2.2.1. Further, we make use of the run-time variables introduced in chapter 3: recall that $c_i(t)$ is the remaining execution time of hard task τ_i at time *t*, $x_i(t)$ is its earliest next release time, $d_i(t)$ its next deadline and $S_i(t)$ is the maximum additional interference which task τ_i may

be subject to without causing its next deadline to be missed. (Note, $x_i(t)$ and $d_i(t)$ are both measured relative to t).

We also consider a set of aperiodic tasks. Each aperiodic task τ_A arrives at some arbitrary time a_A and requires C_A units of computation time before its deadline at $d_A = a_A + D_A$. Further, let $Ahp(A)$ and $Alp(A)$ be the sets of previously guaranteed but as yet uncompleted aperiodic tasks with priorities higher than and lower than A, respectively. Note, in determining whether aperiodic task τ_A can be accepted, we must account for interference from previously guaranteed aperiodics in the set $Ahp(A)$ and check that tasks in the set $Alp(A)$ remain schedulable. To achieve this, we make use of the following run-time information: $c_X(t)$ - the remaining execution time of aperiodic task τ_X at time t and $S_X(t)$ - the slack at priority X, defined as the maximum amount of additional interference that τ_X may be subject to without causing its deadline to be missed.

6.3.3 Exact Test: Aperiodics with Arbitrary Deadlines

Using the above information, the exact feasibility of a new aperiodic arrival τ_A may be determined as follows: First the aperiodic processing time available, $A_A(t, d_A)$ at priority level A in the interval $[t, d_A)$ is calculated using a variant of algorithm 3.3: (Note, as yet τ_A is not part of the task set and thus its execution time is not included in this calculation).

Exact processing time available at priority level A in the interval $[t,d_A)$

$A_A(t,d_A) := 0$

$w_A^{k+1}(t) := 0$

do while $w_A(t)^{k+1} \le d_A(t)$

$w_A^k(t) := w_A^{k+1}(t)$

$$w_A^{k+1}(t) := A_A(t,d_A) + \sum_{\forall j \in hp(A)} \left(c_j(t) + \left\lceil \frac{w_A^k(t) - x_j(t)}{T_j} \right\rceil C_j \right) + \sum_{\forall X \in Ahp(A)} c_X(t)$$

if $w_A^k(t) = w_A^{k+1}(t)$

then $$gap = \min \left(\begin{array}{l} D_A - w_A^k \\ \min\limits_{\forall j \in hp(A)} \left\lceil \frac{w_A^k(t) - x_j(t)}{T_j} \right\rceil_0 T_j + x_j(t) - w_A^k(t) \end{array} \right)$$

$A_A(t,d_A) := A_A(t,d_A) + gap$

$w_A^{k+1}(t) := w_A^{k+1}(t) + gap + \varepsilon$

end if

enddo

return $A_A(t,d_A)$ (6.9)

Note, ε set to the granularity of time is a mathematical device used to force iteration to continue.

The basic operation of the above algorithm is as follows: first, the length $w_A^{m+1}(t)$ of a priority level A busy period starting at time t is computed. The end of this busy period is found when $w_A^m(t) = w_A^{m+1}(t)$. In which case, the *gap* to either the end of the interval or the next release of a higher priority task is identified as aperiodic processing time. This newly identified aperiodic processing time is assumed to be released at time t and is added to the total aperiodic execution time found so far - $A_A(t,d_A)$. Iteration then continues, computing the length of a priority level A busy period including aperiodic processing $A_A(t,d_A)$ and subsequently identifying further priority A idle periods which can be reclaimed. Iteration terminates when the end of the interval is reached. Whereupon, $A_A(t,d_A)$ gives the maximum amount of aperiodic processing at priority A, guaranteed to be possible during $[t,d_A)$.

If $A_A(t) \geq C_A$ then the new aperiodic task is feasible. However, before it can be accepted, we must first check that all previously guaranteed but uncompleted aperiodic tasks and all hard tasks remain schedulable. The execution time budget C_A afforded to aperiodic task τ_A reduces the slack budget of all lower priority tasks, thus provided:

$$\forall X \in Alp(A) \ : \ S_X(t) \geq C_A$$

and

$$\forall i \in lp(A) \ : \ S_i(t) \geq C_A \tag{6.10}$$

then task τ_A can be accepted. In which case, the slack budgets of all lower priority tasks are reduced:

$$\forall X \in Alp(A) \ : \ S_X(t) := S_X(t) - C_A$$

and

$$\forall i \in lp(A) \ : \ S_i(t) := S_i(t) - C_A \tag{6.11}$$

Finally, the slack budget of the new aperiodic τ_A is set.

$$S_A(t) = A_A(t, d_A) - C_A \tag{6.12}$$

6.3.4 Sufficient Test: Aperiodics with Arbitrary Deadlines

The sufficient feasibility of aperiodic task τ_A may be found by deriving a lower bound on the aperiodic processing time available to task τ_A in the interval $[t, d_A)$. To achieve this, we modify the sufficient and not necessary off-line schedulability test given by Audsley in [7] for use at run-time.

Consider a hard periodic task τ_j with a higher priority than aperiodic task τ_A (i.e. $j \in hp(A)$). The maximum interference $I_j(t, d_A)$, due to task τ_j which could fall into the interval $[t, d_A)$ is given by:

$$I_j(t,d_A) \;=\; c_j(t) + f_j(t,d_A)\,C_j + \min\left(C_j, \left(D_A - x_j(t) - f_j(t,D_A)\,T_j\right)_0\right) \tag{6.13}$$

Where $f_j(t,d_A)$ is the number of complete invocations of task τ_j which fall within the interval.

$$f_j(t,d_A) = \left\lfloor \frac{D_A - x_j(t)}{T_j} \right\rfloor_0 \tag{6.14}$$

Thus the interference generally comprises three components: computation due to the current invocation of task τ_j, $f_j(t,d_A)$ complete invocations of task τ_j and a potentially only partially complete final invocation.

A lower bound on the execution time available for aperiodic task τ_A is given by the length of the interval less the upper bound on interference due to all tasks with a priority higher than *A*.

$$A_A(t,d_A) = \left(D_A - \sum_{\forall j \in hp(A)} I_j(t,D_A) - \sum_{\forall X \in hp(A)} c_X(t)\right)_0 \tag{6.15}$$

Once a value of $A_A(t,D_A)$, has been calculated, then task τ_A will be able to meet its deadline provided:

$$A_A(t,D_A) \geq C_A \tag{6.16}$$

If τ_A is feasible, then examination of the schedulability of lower priority tasks and subsequent adjustments to the slack available at each priority level proceeds in the manner detailed previously.

The time complexity of the above acceptance test is $O(n+m)$ where n is the number of hard periodic / sporadic tasks (with off-line guarantees) and m is the number of previously guaranteed but as yet uncompleted aperiodic tasks. By comparison, the complexity of the exact test is $O(Kn+m)$ where K is the number of iterations required, which depends upon the ratio of aperiodic deadline to hard task minimum inter-arrival times. Thus the exact test has pseudo-polynomial complexity.

6.3.5 Slack Stealing

In order for the acceptance tests derived above to function correctly, we require that the slack available at each priority level is maintained according to the principles of the Slack Stealing algorithm [59]. In effect, once guaranteed, firm aperiodic tasks are treated in exactly the same manner as hard periodic / sporadic tasks. For example if between times t and t', the processor is busy with priority level j computation, then the slack is reduced at all higher priority levels:

$$\forall A \in Ahp(j) \ : \ S_A(t) := S_A(t) - (t' - t)$$

and

$$\forall i \in hp(j) \ : \ S_i(t) := S_i(t) - (t' - t) \tag{6.17}$$

Similarly, if gain time is produced by a guaranteed task (periodic, sporadic or aperiodic) completing in less than its worst case execution time, then the slack available at that and all lower priority levels is increased accordingly. Further, soft tasks may be scheduled at priority level k provided that there is slack available at all lower priority levels. That is if:

$$\forall i \in lp(k) \ : \ S_i(t) \geq 0$$

and

$$\forall A \in Alp(k) \ : \ S_A(t) \geq 0 \tag{6.18}$$

Thus both responsive soft task scheduling and an efficient means of providing guarantees for hard aperiodic tasks can be achieved using the same model.

6.3.6 Incorporating Blocking

We now relax the assumption that the hard tasks are independent, however, we still require that all aperiodic tasks are independent and as such do not share any resources with the hard tasks. Each hard task τ_i may be subject to a bounded delay of up to B_i waiting for a resource which is shared with lower priority tasks. We assume that resource access is controlled by the Immediate Priority Ceiling Protocol [82]. Under this protocol, at any given time, only one task with priority lower than *i* can have locked a resource with a ceiling priority greater than *i*.

Assuming that a firm aperiodic task τ_A arrives at time *t*, let $b_A(t)$ be the remaining execution time of task τ_k ($k \in lp(A)$) before it releases all resources of ceiling priority greater than *A*. (Note, it may be the case that no such resources are held by any lower priority task τ_k, in which case, $b_A(t)=0$). Taking account of the additional interference which aperiodic task τ_A is subject to, the execution time available at priority level *A* is reduced:

$$A_A(t,d_A) \;:=\; A_A(t,d_A) - b_A(t) \tag{6.19}$$

Provided that:

$$A_A(t,d_A) \geq C_A \tag{6.20}$$

then τ_A is schedulable. However, it can only be accepted if all lower priority hard tasks and previously guaranteed aperiodic tasks remain schedulable i.e. provided that:

$$\forall i \in lp(A) \;:\; S_i(t) - b_i(t) \geq C_A$$

and

$$\forall X \in Alp(A) \;:\; S_X(t) - b_X(t) \geq C_A \tag{6.21}$$

Where $b_i(t)$ is defined similarly to $b_A(t)$ as the longest delay which the current invocation of task τ_i may be subject to due to blocking by a lower priority task. Note, if, at time *t*, task τ_i is awaiting release, then $b_i(t)=B_i$ (see section 3.4.2).

6.3.7 Resource Sharing between Aperiodics and Hard Tasks

We now consider the situation where aperiodic tasks share resources with hard sporadic / periodic tasks. Again, we assume that resource access is controlled according to the Immediate Priority Ceiling Protocol. Allowing such resource sharing raises two issues. First, the acceptance test must account for blocking which aperiodic task τ_A is subject to. This is readily achieved using the analysis given above. Second, the feasibility of tasks in the set $hp(A) \cap lp(j)$, (where j is the highest ceiling priority of any resource accessed by τ_A) must be determined, as the blocking which they may be subject to is potentially increased. Note, task τ_A cannot affect higher priority aperiodic tasks, as these are runnable at time t and therefore complete before task τ_A can execute and lock a resource.

For the set of hard tasks only, let $S_i^C(D_i)$ be the slack available at priority level i at a critical instant for all hard tasks of priority i and higher (see section 4.1.2). $S_i^C(D_i)$ is equivalent to the maximum blocking which task τ_i may be subject to and yet still be guaranteed to meet its deadline. Further, at each completion of task τ_i, the slack available at priority level i increases by at least $S_i^C(D_i)$. Making use of this information, we now determine the schedulability of subsequent invocations of each task $\tau_i \in hp(A) \cap lp(j)$. Let z_{Aj} be the longest time for which aperiodic task τ_A locks a resource of ceiling priority j or higher. We must check that the blocking caused by task τ_A cannot cause the next or subsequent deadlines of each task $\tau_i \in hp(A) \cap lp(j)$ to be missed. Task τ_i will meet its next deadline provided that:

$$S_i(t) - b_i(t) \geq 0 \qquad (6.22)$$

Where $b_i(t)$ is defined as follows:

1. If task τ_i is runnable at time t, then the system ceiling is inspected, if it is less than i then $b_i(t) = 0$. If the system ceiling is greater than or equal to i, then we must determine if there is a task of lower base priority than i which has locked a resource with ceiling priority i or higher. There is at most one such task τ_k. If there is no such task, then again $b_i(t) = 0$, otherwise, $b_i(t)$ is equal to the worst case remaining execution time of the critical section which task τ_k is in. Note, if τ_i is runnable at time t its current invocation cannot be blocked

by aperiodic task τ_A as it is of higher priority.

2. If task τ_i is not runnable at time *t*, then we must assume that the next invocation could be blocked by any lower priority task, including the aperiodic τ_A. Hence $b_i(t) = \max(B_i, z_{Ai})$.

In addition to the next invocation of each task $\tau_i \in hp(A) \cap lp(j)$, we must also check that subsequent invocations will also be schedulable when subject to blocking from aperiodic τ_A, this is the case, provided that:

$$S_i^C(D_i) \geq z_{Ai} \tag{6.23}$$

Finally, if aperiodic task τ_A is accepted, then the blocking factors $b_i(t)$ for each hard and each aperiodic task must be maintained according to 1. and 2. above. Thus at each release of τ_i, $b_i(t)$ is set according to 1. During the interval between release and completion, $b_i(t)$ is decremented by an amount of time corresponding to the execution of the blocking task τ_k. Finally, at each completion, $b_i(t)$ is set according to 2. When aperiodic task τ_A completes, there can be no runnable tasks with priorities higher than *A*, thus the system is subsequently free of any direct or push through blocking effects due to task τ_A.

6.4 Acceptance Tests for Background Scheduling

Extension of the above acceptance tests to Background scheduling is straightforward: all aperiodic tasks are given priorities below those of hard sporadic / periodic tasks. In this case, the optimal priority assignment policy reduces to ordering the aperiodic tasks according to Earliest Deadline First, i.e. the aperiodic task with the earliest absolute deadline has the highest priority amongst the background priority levels occupied by aperiodics.

Acceptance testing of an aperiodic task τ_A then reduces to determining if it is schedulable according to equation 6.20, that it does not cause any previously guaranteed but as yet uncompleted lower priority aperiodic task to miss its deadline (inequality 6.21), and that accessing resources with a ceiling priority of at most *j* does not cause any tasks in the set $hp(A) \cap lp(j)$ to miss their deadlines. In the case

of Background scheduling, the latter check reduces to testing that:

$$\forall i \in hp(A) \cap lp(j) : S_i^C(D_i) \geq z_{iA} \tag{6.24}$$

Again, higher priority aperiodic tasks cannot be blocked as they are already runnable when aperiodic τ_A arrives and therefore complete before τ_A gets a chance to execute.

6.5 Acceptance Tests for Dual Priority Scheduling:

Using the approach taken in section 6.2, we now derive exact and sufficient acceptance tests for aperiodic tasks with firm deadlines scheduled under the Dual Priority paradigm.

Recall that under the Dual Priority paradigm, there are three priority bands, upper, middle and lower. Each hard task τ_i is assigned two priority levels, one in each of the lower and upper bands. Upon release, τ_i initially executes at its lower band priority level. However, once the priority promotion delay Y_i has elapsed, its priority is promoted to its upper band level, *i*. Aperiodic tasks, such as τ_A, are assigned priorities in the middle band. Thus the optimal priority assignment policy again reduces to assigning priorities to aperiodic tasks such that a task with an earlier absolute deadline is assigned a higher middle band priority than any such task with a later deadline. Let $Mhp(A)$ be the set of tasks in the middle band with priorities which are higher than *A*. Similarly, $Mlp(A)$ denotes the set of tasks in the middle band with priorities lower than *A*. Note, we initially assume that the set of hard tasks may share resources according to the Immediate Priority Ceiling Protocol [82], but that the aperiodic tasks are independent.

Analysis of the execution time afforded to an aperiodic task τ_A under the Dual Priority model requires the following run-time information. (Note, we define the current invocation of a hard task τ_i as that invocation which will next utilise the processor).

$c_i^*(t)$ - the remaining execution time budget for the current invocation of task τ_i. (Note, if τ_i is awaiting release, then $c_i^*(t) = C_i$, otherwise $c_i^*(t) = c_i(t)$).

$x_i^*(t)$ - the earliest possible subsequent release of task τ_i. (If τ_i is awaiting release, then, $x_i^*(t) = x_i(t) + T_i$, otherwise, $x_i^*(t) = x_i(t)$.

$y_i(t)$ - the earliest time at which the current invocation of task τ_i will be promoted to the upper priority band. If at time *t*, task τ_i is awaiting release, then $y_i(t)$ typically corresponds to Y_i time units after its earliest possible next release $x_i(t)$. Note that $y_i(t)$ is measured relative to *t*, thus if the current invocation of τ_i has already been promoted to the upper band, then $y_i(t) = 0$.

$z_i(t)$ - the worst case remaining execution time of any critical section which task τ_i is in at time *t*. Note, $z_i(t)$ is zero if task τ_i is not within a critical section.

In order to provide an acceptance test for aperiodic tasks with fixed priorities in the middle band, we need to derive an exact or upper bound on the execution at upper band priority levels during some arbitrary interval. Invocations of a hard task τ_i can only normally interfere with aperiodic task τ_A once their priority has been promoted to the upper band. However, the current invocation of task τ_i may also block task τ_A by executing a critical section at a raised priority under the Immediate Priority Ceiling Protocol. Recall that the Immediate Priority Ceiling Protocol ensures that only one hard task which has not yet had its priority promoted may be in a critical section.

6.5.1 Exact Acceptance Test: Aperiodics with Arbitrary Deadlines

An exact bound on the amount of aperiodic processing time available to τ_A in the interval $[t, d_A)$ is given by the algorithm below, again derived from algorithm 3.3.

Exact priority level A processing time available in an interval $[t, d_A)$

$$A_A(t, d_{A)} := 0$$

$$w_A^{k+1}(t) := 0$$

do while $w_A(t)^{k+1} \le d_A(t)$

$$w_A^k(t) := w_A^{k+1}(t)$$

$$w_A^{k+1}(t) := A_A(t, d_A) + \sum_{\forall j \in hp(A)} \begin{cases} c_i^*(t) & \textbf{if} \quad w_A^k(t) > y_i(t) \\ z_i(t) & \textbf{otherwise} \end{cases}$$

$$+ \sum_{\forall j \in hp(A)} \left\lceil \frac{w_A^k(t) - x_j^*(t) - Y_j}{T_j} \right\rceil C_j + \sum_{\forall X \in Ahp(A)} c_X(t)$$

if $w_A^k(t) = w_A^{k+1}(t)$

then $gap = \min \begin{cases} D_A - w_A^k(t) \\ \min\limits_{\forall j \in hp(A)} \begin{cases} y_j(t) - w_A^k(t) & \textbf{if} \quad y_j(t) \ge w_A^k(t) \\ \infty & \textbf{otherwise} \end{cases} \\ \min\limits_{\forall j \in hp(A)} \left\lceil \frac{w_A^k(t) - x_j^*(t) - Y_j}{T_j} \right\rceil_0 T_j + x_j^*(t) + Y_j - w_A^k(t) \end{cases}$

$$A_A(t, d_A) := A_A(t, d_A) + gap$$

$$w_A^{k+1}(t) := w_A^{k+1}(t) + gap + \varepsilon$$

end if

enddo

return $A_A(t, d_A)$ (6.25)

The above calculation of available aperiodic processing time proceeds in a similar manner to algorithm (6.9). Due regard is however given to the fact that hard tasks only execute at a priority higher than *A*, either when they are accessing resources or once they have had their priorities promoted to the upper band. Again calculation proceeds by determining the extent of priority level *A* busy and idle periods, claiming the idle periods as aperiodic execution time.

Once a value of $A_A(t,d_A)$, has been calculated, then task τ_A can be guaranteed provided:

$$A_A(t,d_A) \geq C_A \tag{6.26}$$

and all lower priority, previously guaranteed aperiodic tasks remain schedulable:

$$\forall X \in Mlp(A) \quad : \quad S_X(t) \geq C_A \tag{6.27}$$

If aperiodic τ_A is accepted, then its slack budget S_A is set to $A_A(t,d_A) - C_A$ and the slack budgets of all lower priority aperiodic tasks are reduced by C_A. Note, typically, no maintenance of these slack budgets is required save at the acceptance of further aperiodic tasks. This is because, up to the completion of τ_A, the only execution which takes place is of higher priority than *A*. Further, as there is only one invocation of τ_A, once it completes, it is of no further concern.

6.5.2 Sufficient Acceptance Test: Aperiodics with Arbitrary Deadlines

An upper bound on the interference $I_i(t,d_A)$, which aperiodic task τ_A is subject to, due to the execution of task τ_i, in the interval $[t,d_A)$, is given by:

$$\begin{aligned} I_i(t,d_A) = & \min\left(z_i(t), D_A\right) + \\ & \min\left(D_A - y_i(t), c_i^*(t) - z_i(t)\right) + \\ & f_i(t,D_A)\, C_i + \\ & \min\left(\left(D_A - x_i^*(t) - Y_i - f_i(t,d_A)\, T_i\right)_0, C_i\right) \end{aligned} \tag{6.28}$$

Where, $f_i(t,D_A)$ is the number of complete invocations of task τ_i, excluding the current one, which may complete before the end of the interval.

$$f_i(t,D_A) = \left\lfloor \frac{D_A - x_i^*(t) - Y_i}{T_i} \right\rfloor_0 \tag{6.29}$$

Thus the interference due to each hard task generally comprises four components: the remainder of the critical section which task τ_i is in, if any; the computation due to the current invocation of task τ_i once its priority has been promoted; $f_i(t, D_A)$ further invocations of task τ_i which undergo priority promotion and complete before the end of the interval; and a potentially partially complete final invocation of task τ_i.

A lower bound on the execution time available for aperiodic task τ_A is given by the length of the interval less the upper bound on interference due to all hard tasks which may execute at an upper band priority level and the interference due to higher priority aperiodic tasks which have not yet completed.

$$A_A(t, d_A) = \left(D_A - \sum_{\forall j \in hp(X)} I_j(t, d_A) - \sum_{\forall X \in Mhp(A)} c_X(t) \right)_0 \tag{6.30}$$

Once the lower bound has been calculated, then the acceptance test proceeds by checking if all lower priority aperiodic tasks will remain schedulable in the presence of task τ_A. If so, then the slack budgets of these tasks are adjusted accordingly and a slack budget of $A_A(t, d_A) - C_A$ is assigned to τ_A.

The complexity of the sufficient acceptance test formulated above is $O(n+m)$ where n is the number of hard tasks and m is the number of previously accepted but as yet uncompleted aperiodic tasks. By comparison, the complexity of the exact test is $O(Kn+m)$ where K depends upon the ratio of the aperiodic deadline to hard task minimum inter-arrival times.

6.5.3 Resource sharing

To permit resource sharing between hard tasks and aperiodics under the Dual Priority paradigm, it is necessary to take account of the additional blocking which hard tasks may be subject to due to aperiodic tasks accessing resources. To achieve this requires that this additional source of blocking is accounted for in the term B_i when calculating the priority promotion delay Y_i for each hard task (see section 4.2.3).

6.5.4 Soft Tasks

We now consider adding soft tasks to our model. As with firm deadline aperiodic tasks, we assume that soft tasks are assigned priorities in the middle band. Typically, soft tasks are of most value to the system if they are executed responsively. However, we must also ensure that they do not cause firm tasks with on-line guarantees to miss their deadlines. To achieve this, soft tasks are normally assigned a priority below those of tasks with on-line guarantees. If more responsive execution is required, then soft tasks may be given an execution time budget at a high priority level (in the middle band). This budget is enforced by the kernel and must not exceed the slack on previously guaranteed aperiodic tasks. Affording soft tasks such a budget reduces the slack budgets of aperiodic tasks in the manner described previously. When considering the acceptance of new aperiodics however, the remaining execution time budget of the soft tasks may be rescinded increasing the slack available and the probability of guaranteeing a new task.

6.5.5 Scheduling Policies

There are many trade-offs in choosing which particular, hard, firm or soft aperiodic tasks to schedule. Once a hard aperiodic task has been accepted, then the semantics of such tasks often dictate that the 100% guarantee afforded at acceptance must not be rescinded. In the case of firm aperiodic tasks, any guarantee could be overturned in order to accommodate a more valuable alternative. Finally, soft tasks may be required to execute at high priority deferring the execution of aperiodic tasks with firm or hard deadlines, provided their guarantees are not violated. In the next chapter, we examine admission policies which arbitrate between competing requests for spare capacity.

6.6 Performance Evaluation

In this section, we evaluate the performance of the acceptance tests derived earlier. The percentage of total aperiodic execution time requested which comprises aperiodic tasks afforded a 100% guarantee is used as the performance metric. This criteria is dependent on both the ability of the underlying scheduling policy to defer the execution of hard tasks (which we examined in detail in chapters 4 and 5) and the

efficiency of the acceptance test used.

During each simulation run, the appropriate acceptance test was used to determine if each aperiodic task could be guaranteed. Only aperiodic tasks which were given a guarantee were allowed to execute.

The hard task sets used in these simulations were the same as those described in chapter 4. That is, they had an exponential distribution of periods between 40 and 2560 ticks and a worst case utilisation of 30, 50, 70 or 90%.

The firm task load was simulated by a queue of aperiodic tasks the arrival times of these tasks followed a uniform distribution over the test duration of 100000 ticks (i.e a Poisson arrival process). The execution requirements of the aperiodic tasks were varied to examine the effect on the percentage of tasks accepted. In each case the number of aperiodic tasks in the queue was set such that the combined utilisation of all hard sporadic / periodic and all firm aperiodic tasks was 100%. The queue of aperiodic tasks was ordered by arrival time prior to commencing the simulation. During each simulation run, once the arrival time of the aperiodic task at the head of the queue had been reached, it was removed from the pending queue and subject to an acceptance test. If accepted, the task was placed in the aperiodic task run-queue and executed under the appropriate scheduling algorithm.

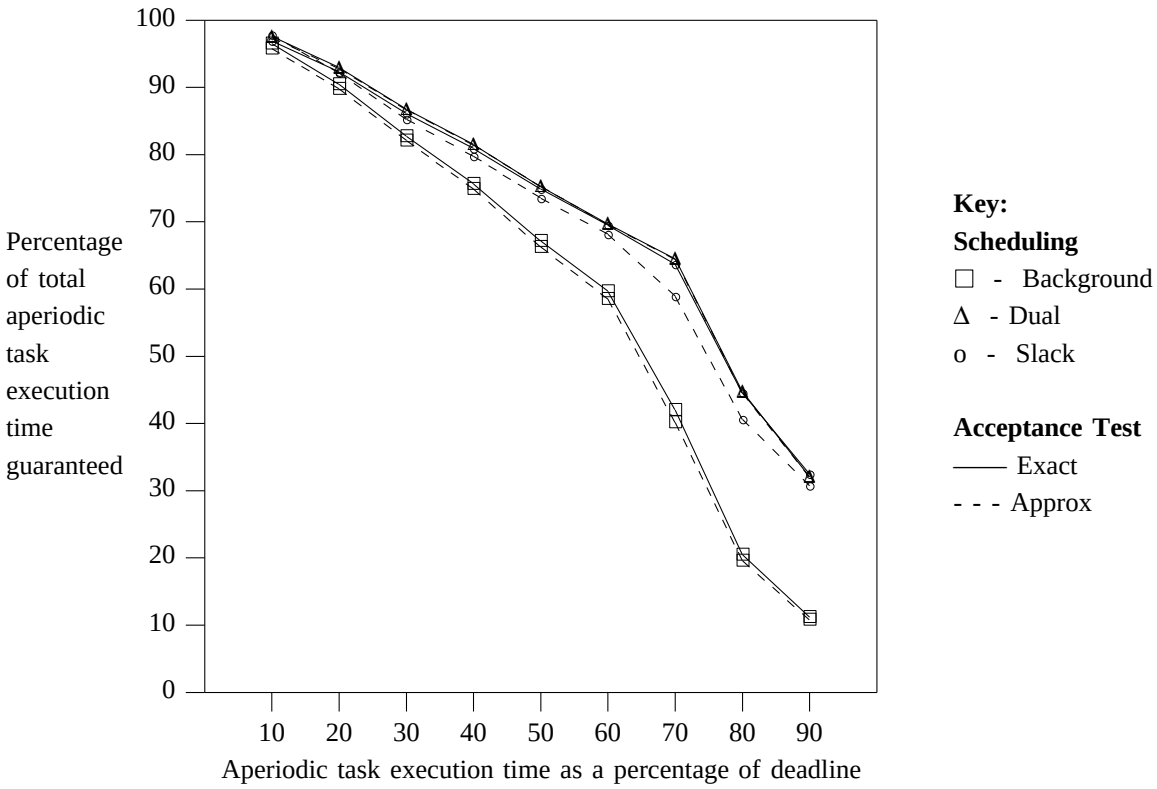
In experiments 6.1 to 6.4, scheduling and acceptance test overheads were assumed to be zero. The issue of overheads is examined in section 6.5.3.

6.6.1 Results

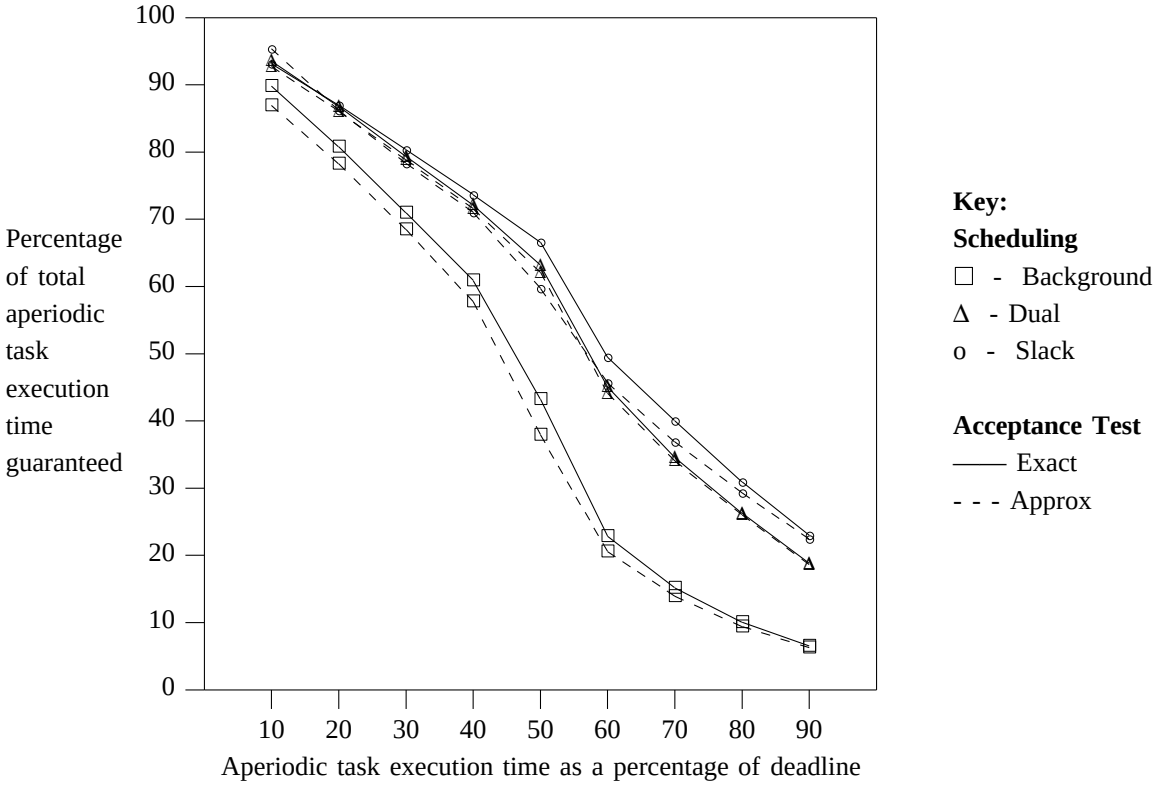
Experiments 6.1 to 6.4 examined the percentage of aperiodic tasks with an exponential distribution of deadlines in the range 10 to 2560 ticks (mean 60) which were accepted and scheduled by the various approaches. The execution time of each aperiodic task corresponded to a fixed proportion of its deadline (10 - 90%). This proportion was varied to examine the effect on the number of aperiodic tasks guaranteed and is plotted on the *x*-axis of the graphs. The percentage of total aperiodic task execution time guaranteed is plotted on the *y*-axis.

In these experiments, the Slack Stealing algorithm supported by an exact test gave the best theoretical performance over the entire range of hard periodic and firm aperiodic task loads examined.

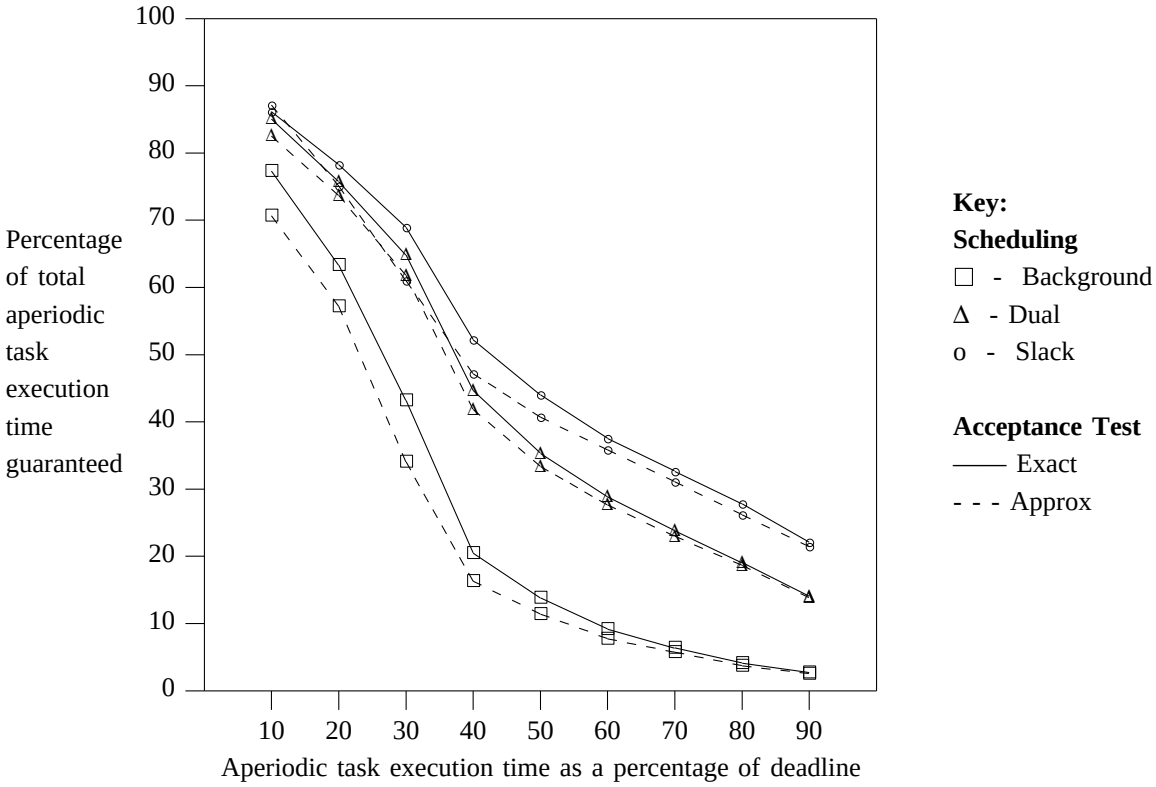
The approximate acceptance tests simulated for Background, Dual Priority and dynamic Slack Stealing approaches, typically resulted in 3-10% fewer aperiodic tasks being accepted than their exact counterparts. In each experiment (6.1 - 6.4), the percentage of total aperiodic execution time guaranteed drops sharply as the utilisation of each aperiodic task reaches X%, where X% is equivalent to 100% minus the hard task set utilisation. This is because on average, in any given interval, the aperiodic execution time available is X%. For aperiodic tasks with a utilisation greater than this level, the differential between Background and Dual Priority or Background and Slack Stealing increases as hard task execution must typically be deferred to accommodate the aperiodic tasks.



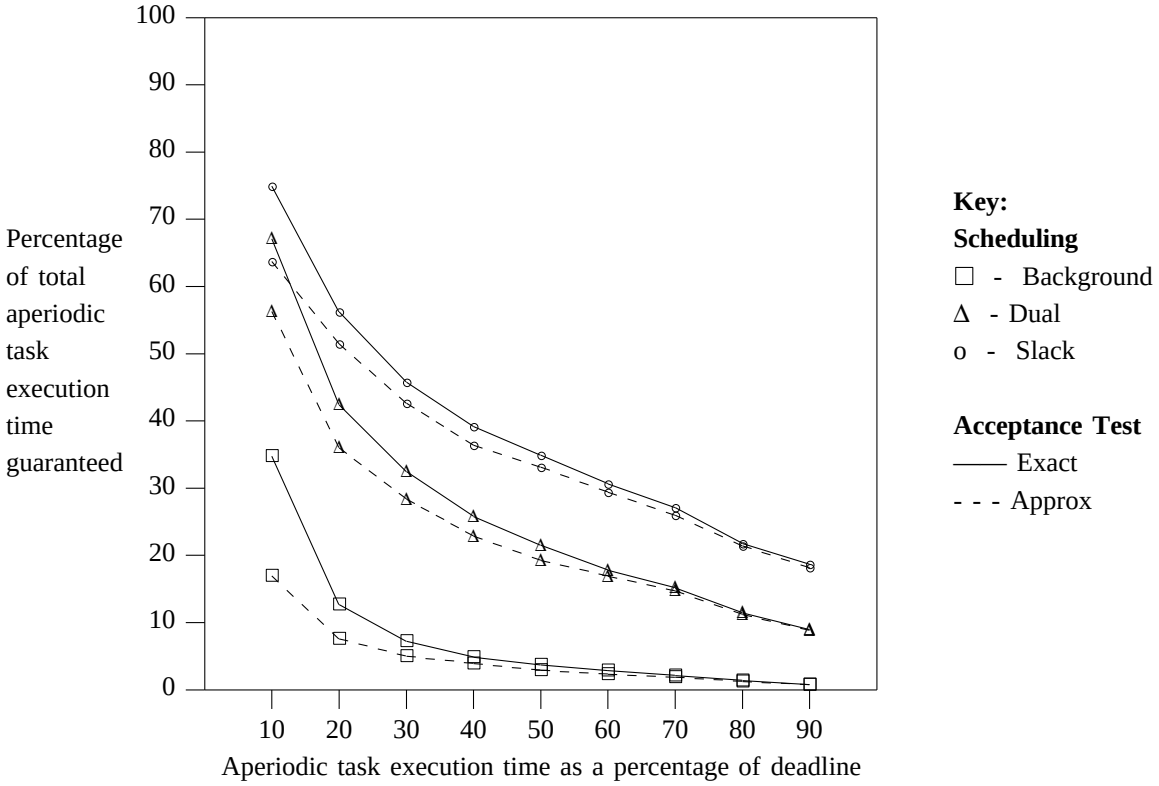
Expt 6.1: Periodic Tasks 30% utilisation, Aperiodics 70%



Expt 6.2: Periodic Tasks 50% utilisation, Aperiodics 50%



Expt 6.3: Periodic Tasks 70% utilisation, Aperiodics 30%



Expt 6.4: Periodic Tasks 90% utilisation, Aperiodics 10%

From the results of experiments 6.1 - 6.4, it is clear that the Slack Stealing algorithm in conjunction with its associated exact acceptance test, can theoretically guarantee the largest percentage of aperiodic task execution. This is perhaps not unexpected, as the Slack Stealing algorithm has been shown to be optimal in terms of making spare capacity available (see section 3.2.2) and is thus an optimal method of deferring hard task execution.

In the simulations reported so far, it was assumed that all scheduling overheads, including the performance of acceptance tests and the calculation and maintenance of data describing extra capacity or slack, were zero. However in practice these overheads have an impact on the effectiveness of a particular technique. These issues are discussed in the next section.

6.6.2 Practical Considerations

In terms of overheads, Dual Priority scheduling results in an extra event (similar to task release) at the promotion point of each invocation of a hard task. This event requires that the priority of the task is increased and hence its position in the run-queue altered accordingly. In the worst case, this operation takes $O(logN)$ time, where N is the number of tasks in the run-queue. The acceptance testing of a task also incurs an overhead which is $O(n+m)$, where n is the number of hard tasks and m the number previously guaranteed but as yet uncompleted aperiodic tasks. Thus the run-time overheads involved in Dual Priority scheduling are relatively low, making the approach suitable for many practical systems.

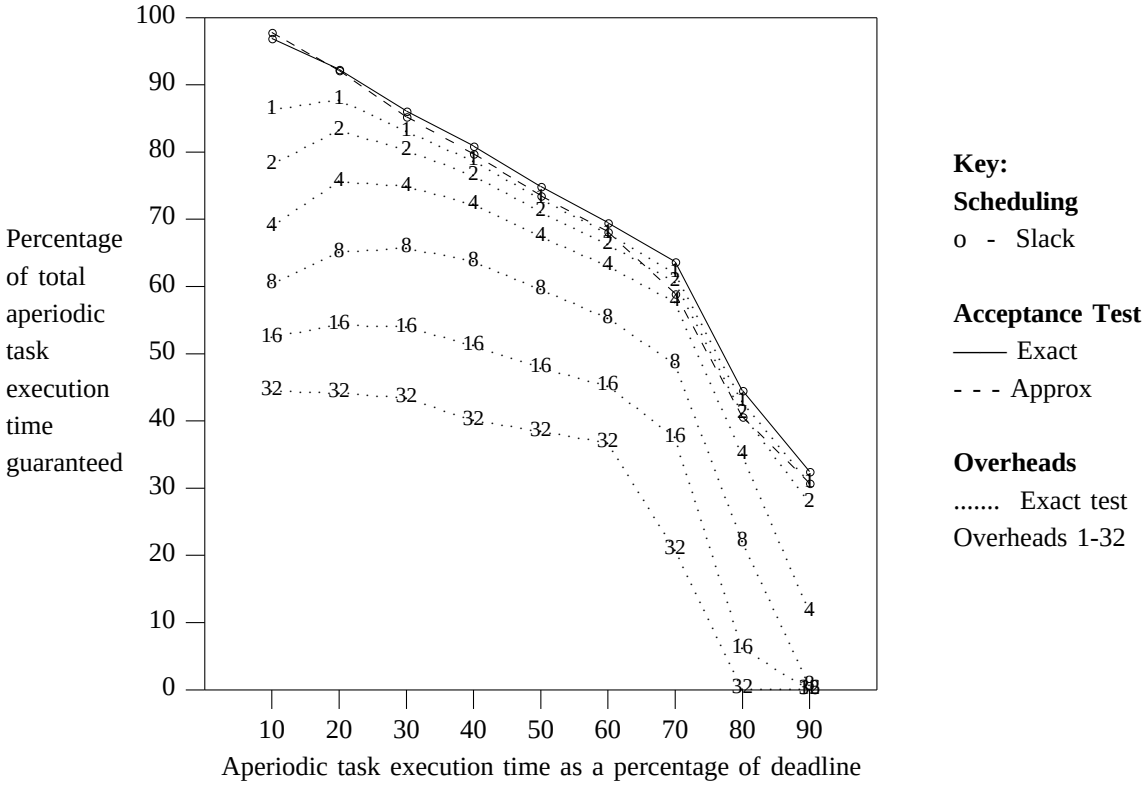
Although offering higher theoretical levels of performance, the dynamic Slack Stealing approach incurs higher overheads. Typically, the PASS and HASS algorithms, introduced in chapter 4, periodically require $O(n^2)$ time to calculate the slack. In addition, $O(n)$ time is required at each context switch to maintain slack counters. Finally, the sufficient acceptance test used has $O(n+m)$ complexity.

By comparison, Background scheduling has no overheads save those of the acceptance test - $O(n+m)$, however, its performance is poor compared with that of Slack Stealing or Dual Priority scheduling.

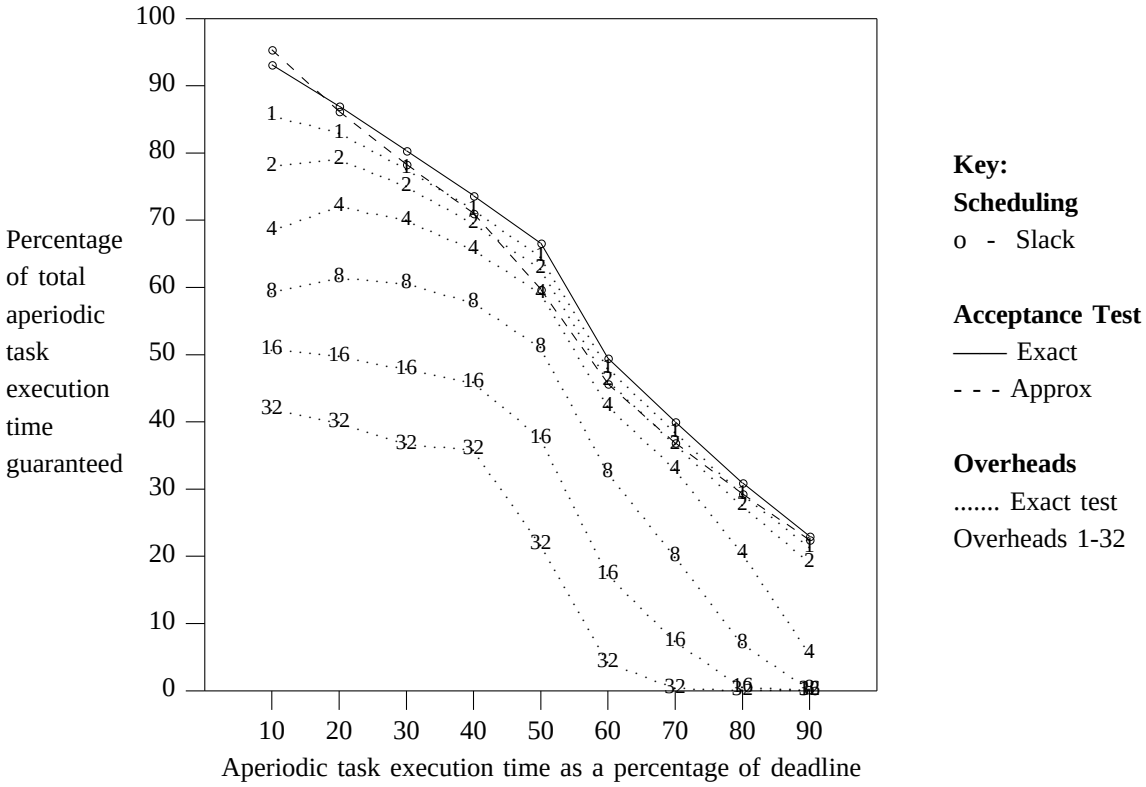
6.6.3 Overheads: Simulation

To determine the effect of overheads on the percentage of aperiodic task execution guaranteed, we carried out further simulations using only the Slack Stealing approach but including various levels of overhead for the acceptance test. Note, these overheads were only incurred in accepting or rejecting aperiodic tasks which, at the time of acceptance testing, could have any chance of being accepted. Thus aperiodic tasks with less than their execution time to go before their deadline were assumed to be rejected immediately without incurring any overhead.

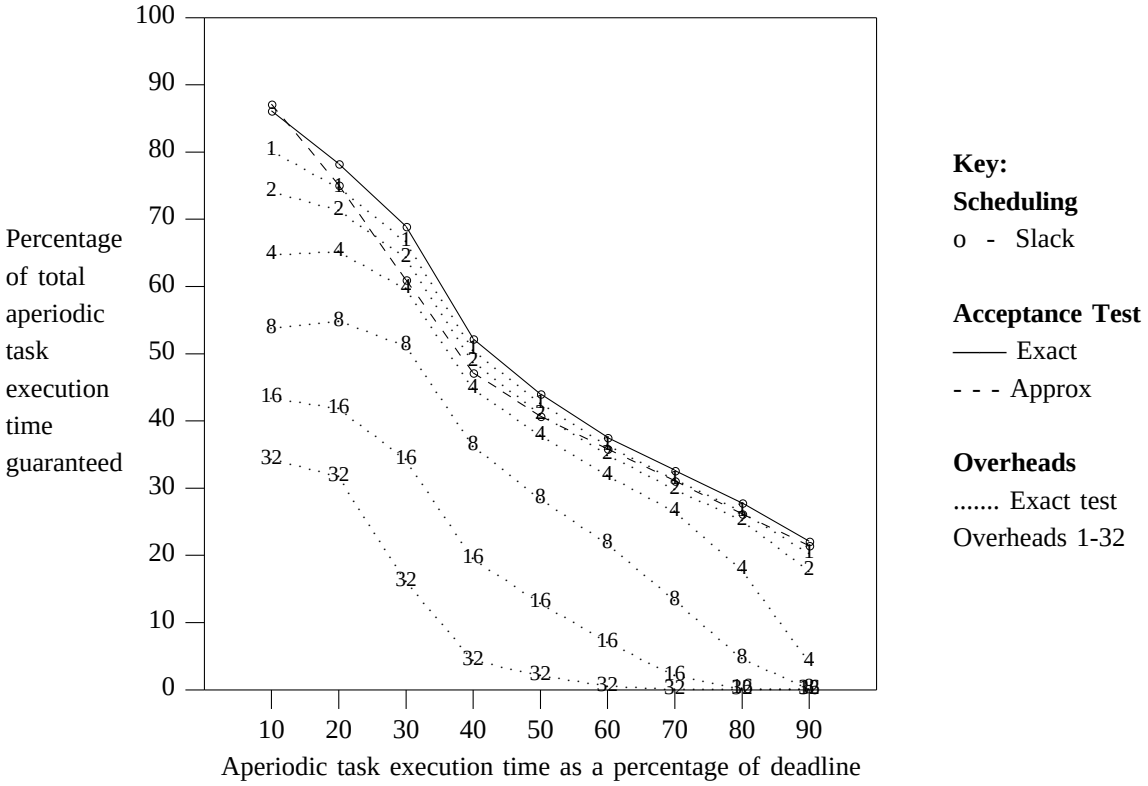
Experiments 6.5 to 6.8 illustrate the effects of overheads which were varied from 0 to 32 ticks per acceptance test. In these simulations, the same exponential distributions of hard periodic and aperiodic tasks were used as in experiments 6.1 to 6.4.



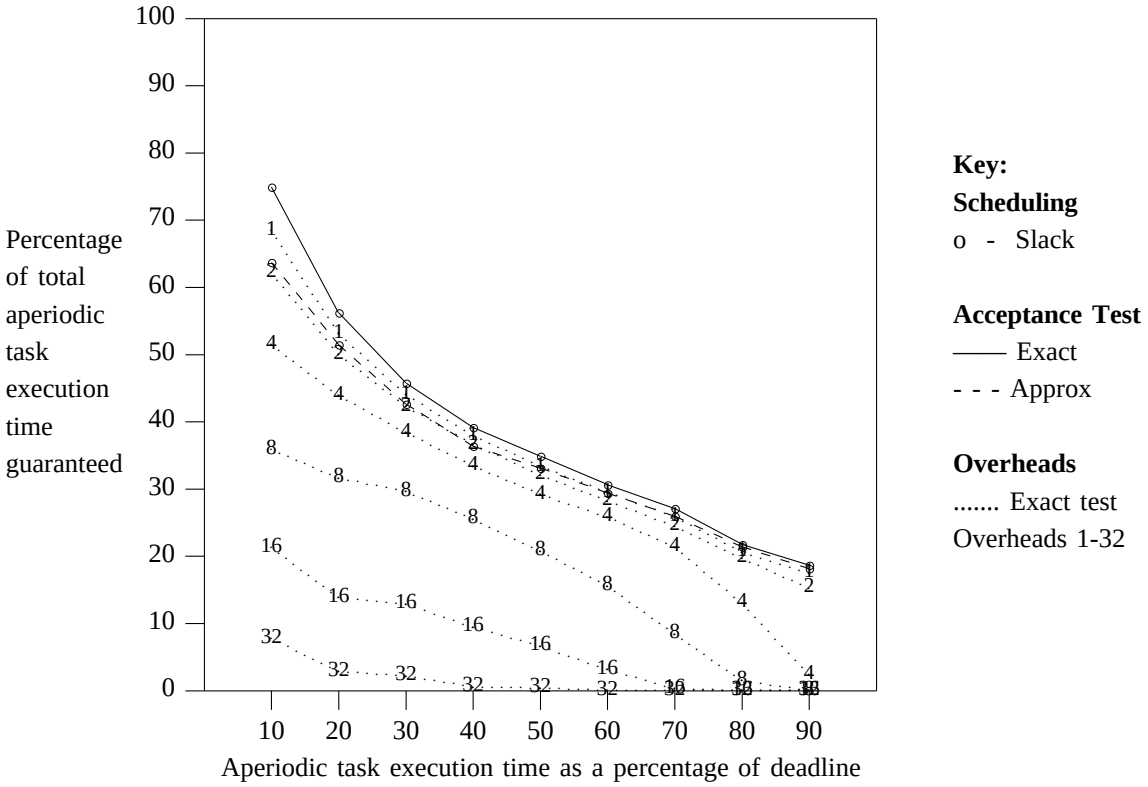
Expt 6.5: Periodic Tasks 30% utilisation, Aperiodics 70%



Expt 6.6: Periodic Tasks 50% utilisation, Aperiodics 50%



Expt 6.7: Periodic Tasks 70% utilisation, Aperiodics 30%



Expt 6.8: Periodic Tasks 90% utilisation, Aperiodics 10%

From the graphs, it is clear that increasing overheads rapidly results in degraded performance. Assuming that the task sets used are realistic with 1 tick = 1ms, an exact test requiring approximately an extra 1-2ms compared to the corresponding sufficient test, would result in inferior performance.

Typical worst case overheads for the sufficient acceptance tests, carried out on a 33 MHz i486, are given in the table below

Sufficient Acceptance Test Overheads	
$n+m$ tasks	Time (ms)
10	0.11
20	0.22
30	0.33
40	0.43
50	0.54

By comparison, each iteration of an exact test requires a similar amount of computation to the corresponding sufficient test. The number of iterations is however dependent upon the ratio of aperiodic task deadline to hard sporadic/periodic task minimum inter-arrival times. A rough approximation to the number of iterations required is given by:

$$\frac{D_A}{\min\limits_{\forall j \in hp(A)} (T_j)}$$

For the task sets studied, the mean number of iterations required was 3.5 whilst the worst case was 73. In the average case, an exact test may give superior performance, particularly for hard task sets of small cardinality. In the case of periodic / sporadic / aperiodic task sets with a similar distribution of deadlines to those used in our experiments, on average, using an exact test will result in better performance for task sets of cardinality <~ 50. (Assuming that 1.5ms additional overhead for an exact test gives performance similar to an equivalent sufficient test, on average an additional 2.5 iterations must be accommodated = 0.6ms per iteration,

which is possible for task sets of cardinality < ~ 50).

However, there are difficulties associated with implementing exact acceptance tests within an operating system kernel. Typically, such tests are executed non-pre-emptively within the scheduler. However, in the case of exact tests with effectively unbounded worst case execution times, this would lead to long non-pre-emptable kernel sections (of up to 5-35ms for 10-50 tasks with the distribution of deadlines considered): sufficient to cause many task sets to become unschedulable.

6.7 Summary

In this chapter, we introduced an optimal priority assignment policy for aperiodic tasks with arbitrary ready times and deadlines, scheduled along with a set of periodic / sporadic tasks. Our model assumed that the periodic / sporadic tasks were assigned unique fixed priorities according to some arbitrary policy and guaranteed to meet their deadlines via off-line feasibility analysis. In contrast, priority assignment and acceptance testing of aperiodic tasks was carried out on-line. Within this framework, we showed that assigning each aperiodic task the highest priority, such that every task of a lower priority has a later absolute deadline, maximises the computation time which can be made available before the aperiodic's deadline. Furthermore, this priority assignment policy is also optimal in terms of being able to guarantee subsequent aperiodic arrivals.

We derived exact and sufficient acceptance tests for guaranteeing firm or hard deadline aperiodic tasks. The time complexity of the sufficient tests is $O(n+m)$ where n is the number of hard periodic tasks (with off-line guarantees) and m is the number of previously guaranteed but as yet uncompleted hard aperiodic tasks. By comparison, the complexity of the exact tests is $O(Kn+m)$ where K is approximated by D_A/T^{MIN}, where D_A is the aperiodic deadline and T^{MIN} the minimum hard task inter-arrival time.

We showed that the scheduling strategy assumed by these acceptance tests fully integrates the scheduling of hard sporadic / periodic, firm aperiodic and soft tasks. Once guaranteed, firm aperiodic tasks are dealt with in exactly the same manner as hard tasks, whilst soft tasks may be executed at the highest priority under the control

of the Slack Stealing algorithm.

Similar acceptance tests were derived for Background and Dual Priority Scheduling approaches. We also extended this family of acceptance tests thus permitting aperiodic tasks to share resources with hard sporadic / periodic tasks.

We evaluated the effectiveness of the sufficient and exact acceptance tests using the percentage of total requested hard aperiodic task execution time as a performance metric. Our simulations showed that the sufficient tests provide performance which is 3-10% worst than that of their exact counterparts. However, investigation of overheads revealed that the theoretical performance advantage which exact tests have over their sufficient counterparts is negated by increased overheads equivalent to less than 5% of the mean aperiodic task deadline. For the task sets studied, this increase corresponds to an additional overhead of 1-2ms per acceptance test. Typical overheads for the sufficient tests were approximately 0.1ms for 10 tasks and 0.5ms for 50 tasks, assuming that the acceptance test was executed on a 33 MHz i486. Theoretically, using an exact test would result in better performance for task sets of cardinality $< \sim 50$, (assuming the distribution of deadlines given), however, the long (5-35ms for 10-50 tasks) non-pre-emptable sections required to accommodate such a test within the operating system kernel are generally unacceptable. For practical reasons, it is therefore preferable to use the sufficient acceptance tests derived in this chapter.

Chapter 7

Allocating Spare Capacity: Admission Policies

In chapter 1, two fundamental requirements for future real-time systems were presented: mission critical services must be guaranteed *a priori* to meet their deadlines, and system utility should be maximised.

In the mid 1970's, Jensen considered using *time-value* functions in task scheduling, with the objective of maximising system utility [68]. These time-value or utility functions, can be used to combine the notions of service precision, timeliness and importance. Thus utility functions provide a means of describing how the contribution made to the overall utility of a system, by a given service, depends upon the accuracy of the results produced by that service and the time at which they are delivered. Further, this concept extends to the idea of negative utility or *damage* caused when a safety critical service fails to meet its requirements [19].

Typical utility functions for a safety critical (hard) task and a task with a soft deadline are shown in figures 7.1 and 7.2 respectively (reproduced from [19]).

The contribution to overall system utility made by each task also depends upon the algebra of utility values. For independent services, system utility is typically the sum of contributions from individual services. However, when a single real-time service comprises a precedence related chain of tasks, the utility of the service is often only as good as its worst task. Hence service utility equates to the minimum utility of any of its constituent tasks [19].

With precedence constrained tasks, there is also a problem of "utility inversion", a high value task can be constrained to execute once a predecessor of low value has completed. Utility based scheduling policies may, however, delay execution of the low value task in order to accommodate other tasks with medium value at the expense of being unable to schedule the high value successor. In his thesis [114], Zlokapa addresses this problem by means of utility inheritance: predecessor tasks effectively inherit utility from their successors.

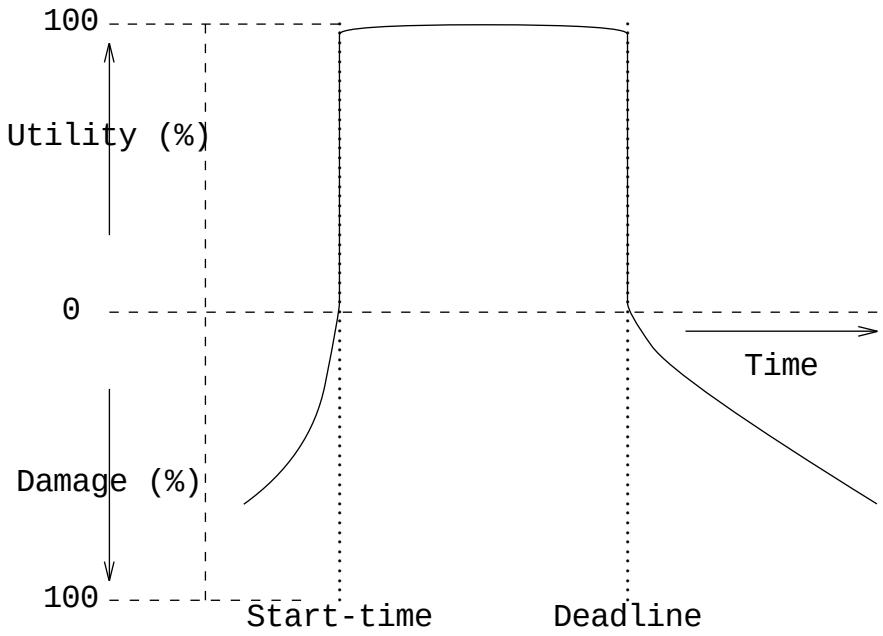


Figure 7.1: A Safety Critical (Hard Deadline) Task

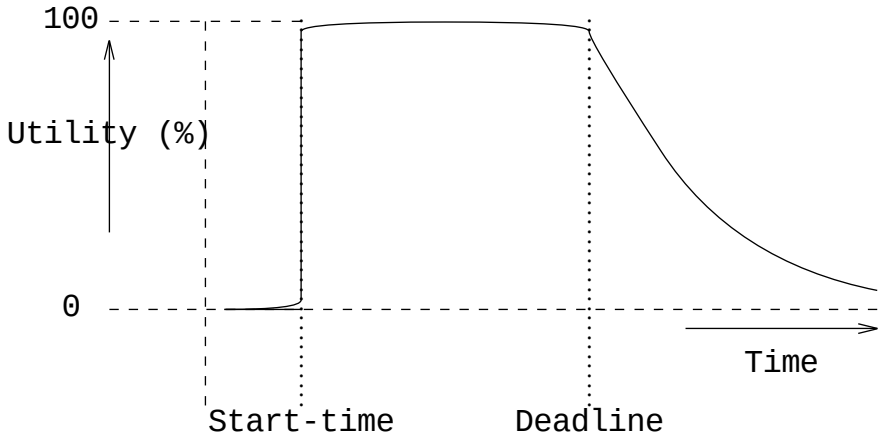


Figure 7.2: A Task with a Soft Deadline

With regard to scheduling tasks to meet both time constraints and to maximise system utility, fixed priority pre-emptive scheduling facilitates the *a priori* guarantee of hard tasks, however, in its purest form it is too inflexible to ensure a high level of system utility. Alternatively, Best-Effort scheduling [68] maximises system utility, but at best provides only probabilistic guarantees that time constraints will be met. In this chapter, we show how these two schemes can be integrated.

A new Adaptive Value-Density Threshold policy is introduced which approximates to the Best-Effort policy but incurs much lower overheads. This policy is used to admit optional utility enhancing tasks for on-line acceptance testing. Such tasks are assigned a fixed priority according to the optimal priority assignment policy described in section 6.3. If guaranteed by a sufficient on-line acceptance test (see section 6.4) then they are executed when no higher priority tasks are runnable. This approach combines the benefits of guaranteeing hard timing requirements with increased system utility.

We evaluate the performance of FCFS, Best-Effort and Adaptive Threshold policies when applied to Background, Dual Priority and Slack Stealing methods of scheduling optional components. The performance criteria used is the fraction of an upper bound on achievable system utility which is obtained by each combination of admission policy and scheduling method. The limitations of our simple model of utility functions are then discussed along with issues arising in the practical implementation of the above policies. Finally, we extend the Adaptive Value-Density threshold approach to Multiple Version scheduling.

First however, we review previous research into policies for maximising system utility. As part of this review, we build upon previous research by Baruah [18]: thus deriving an upper bound on the ratio of the maximum utility which can be guaranteed by an on-line admission policy to that which can be obtained by a clairvoyant algorithm, for the case of task sets comprising both mandatory and optional tasks.

7.1 Review of Scheduling Policies for Maximising Utility

A real-time system is said to be *overloaded* if it is not possible to meet the time constraints of all task invocations. Under overload, it is important that the system degrades gracefully, completing the most important tasks and discarding those of least utility (or value). In this section we review policies for task admittance / scheduling, which seek to maximise the total value of completed tasks.

Research in this area has mainly focused on a simple computational model: all tasks are assumed to be aperiodic, have arbitrary start times and deadlines, fixed execution times and a constant value which is obtained only if the task completes prior to its deadline.

Below, we review five admittance / scheduling policies which aim to maximise the total value of completed tasks.

1. Earliest Deadline First [41].

2. Maximum Value Density [68].

3. Best-Effort [68].

4. D^{over} [55].

5. Robust Earliest Deadline [27].

Examples of the operation of these algorithms are given using the task set detailed below. (Note, value-density is defined as the value which will be accrued at task completion divided by the task's remaining computation time).

Aperiodic tasks					
Name	Arrival	WCET	Deadline	Value	Value-Density
A	0	4	25	10	2.5
B	2	6	10	24	4
C	2	7	9	7	1
D	5	2	12	2	1
E	5	7	15	24.5	3.5
F	12	5	26	7.5	1.5
G	18	11	29	11	1
H	18	3	28	9	3

7.1.1 Earliest Deadline - FCFS

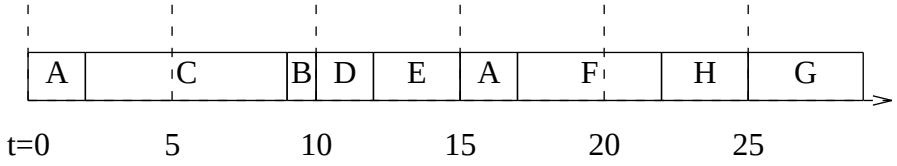
In 1974, Dertouzos [41] showed that for the underloaded case (i.e. < 100% utilisation), the Earliest Deadline First (EDF) algorithm achieves 100% of the maximum possible value. However, in the case of overload, its performance is very poor. This is due to a domino effect. EDF continually executes tasks with early deadlines which cannot be met. This delays the execution of tasks with later deadlines subsequently causing them to also miss their deadlines.

The performance of EDF under overload conditions may be improved by only scheduling those tasks which can be guaranteed to meet their deadlines. This requires an on-line acceptance test which determines if a newly arrived task is schedulable. Such tests have been given by Kim [54] and Buttazzo and Stankovic [27]. However, simply admitting tasks in First Come First Served (FCFS) order results in a low total value being accrued. In effect, tasks of low utility which arrive early are given preference over later arrivals with higher values. This effect is illustrated in figure 7.3, where the execution of task C with an early deadline and low value results in high value tasks B and E, both missing their deadlines.

Better performance can be obtained if task value (utility) is considered when arbitrating between conflicting demands for processor time.

7.1.2 Maximum Value-Density Scheduling

In his thesis [68], Locke gives two algorithms for scheduling tasks based on their utility functions: a forward looking *Best-Effort* algorithm and a greedy *Value-Density* algorithm. Both algorithms make use of the following result which was proved by Locke [68].

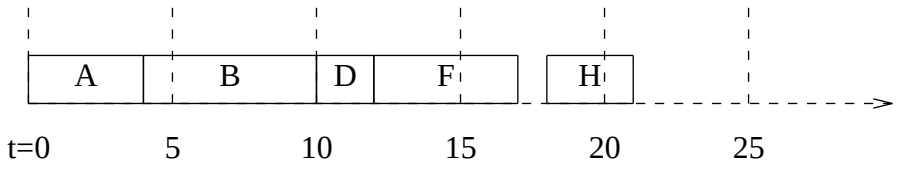


t=0, Task A arrives and commences execution.
t=2, Tasks B and C arrive. Task C has the earliest deadline and therefore executes.
t=5, Tasks D and E arrive, task C continues to execute.
t=9, Task C completes, accrueing a value of 7. Task B now has the earliest deadline and executes.
t=10, Task B misses its deadline and is abandoned. Task D executes.
t=12, Task D completes, accrueing a value of 2. Task F arrives. Task E executes as it has the earliest deadline.
t=15, Task E misses its deadine and is abandoned. Task A now has the earliest deadline and executes.
t=17, Task A completes, accrueing a value of 10. Task F executes.
t=18, Tasks G an H arrive. Task F continues to execute.
t=22, Task F completes, accrueing a value of 7.5. Task H executes.
t=25, Task H completes, accrueing a value of 9. Task G executes.
t=29, Task G misses its deadine and is abandoned.
Earliest Deadline First, total value accrued = 35.5.

Figure 7.3: Operation of the Earliest Deadline First Policy.

"Given a set of of tasks with precisely known constant values for completing them, it can be shown that a sequence of tasks in decreasing order by value density (V/C, in which V is its value and C is its processing time) will produce a total value at every task completion time at least as high as any other schedule."

The Value-Density scheduling algorithm, allocates the processor to the task with the highest value-density. In this case, value-density is defined as the expected value at task completion divided by the expected execution time. Note, tasks which are expected to complete after their deadlines have zero or very low expected values and are removed from the queue of ready tasks. Figure 7.4 illustrates the operation of this algorithm.



t=0, Task A arrives and commences execution.
t=2, Tasks B and C arrive. The value densities of tasks A, B and C are 5, 4 and 1 respectively. Task A has the highest value density and therefore continues to execute.
t=4, Task A completes, accrueing a value of 10. Task B now has the highest value density and therefore executes.
t=5, Tasks D and E arrive, with value densities of 1 and 3.5, compared to 4.8 and 1 for tasks B and C respectively. Task B continues.
t=10, Task B completes, accrueing a value of 24. Tasks C and E are abandoned as they cannot meet their deadlines. Task D executes.
t=12, Task D completes, accrueing a value of 2. Task F arrives and executes.
t=17, Task F completes with a value of 7.5.
t=18, Tasks G and H arrive, with value densities of 1 and 3 respectively. Task H therefore commences execution.
t=21, Task H completes with a value of 9. Task G is abandoned.
Value Density policy, total value accrued = 52.5.

Figure 7.4: Operation of the Value Density Policy.

In effect, the Value-Density algorithm assumes that there is always a high probability that a task with a high value-density and little slack will arrive soon. It therefore accrues utility early by executing the task with the highest value-density first. However, this is not always the best choice, as illustrated by the example given in figure 7.4. At time 2, the Value-Density algorithm executes task A, even though it has a later deadline than task B. This has the knock on effect of the algorithm being unable to schedule task E which has a high value.

7.1.3 Best-Effort Scheduling

In contrast with Value-Density scheduling, the Best-Effort algorithm assumes that the probability of a high value-density task arriving is low. It therefore attempts to obtain the highest possible value for executing those tasks which are currently runnable. To achieve this, the Best-Effort algorithm admits tasks for scheduling based upon their

value-densities and schedules them according to the Earliest Deadline First policy.

In summary, the Best-Effort algorithm operates as follows:

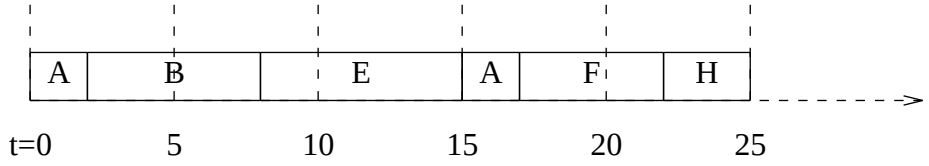
Locke's Best-Effort algorithm:

At each task arrival and completion:
Place any new task in the ready queue.
Transfer all tasks in the pending queue to the ready queue.
For each task in the ready queue (ordered by earliest deadline):
    Calculate the probability of overload using a heuristic based on expected computation times.
    While the probability of overload is greater than some threshold:
        Remove the task with the lowest value-density from the
        ready queue and put it back in the pending queue.
    Endwhile
Endfor
Execute the tasks in the ready queue in earliest deadline order.

The operation of the Best-Effort algorithm is illustrated in figure 7.5.

Through extensive simulations, Locke showed that the Best-Effort algorithm *"consistently achieved a high total value (utility) from the scheduling of processes (tasks) under time constraints"*, outperforming various simpler scheduling algorithms, such as Shortest Processing Time First, Earliest Deadline First, First Come First Served (FCFS) and Least Slack Time First. However, practical issues such as the complexity - $O(n^2)$ and potentially high overheads involved in Best-Effort scheduling were not addressed.

The overheads of Best-Effort scheduling were later investigated by Tokuda *et al* [109] and Wendorf [110]. Wendorf showed that executing the Best-Effort scheduling policy on the same processor as application tasks can result in very large overheads under overload conditions. In Wendorf's experiments, with a task set of cardinality 36, once the potential load reached 200%, up to 80% of the processors time was spent choosing which task to execute according to the Best-Effort policy, drastically reducing the total utility obtained. By comparison, off-loading the scheduling to a



t=0, Task A arrives and commences execution.
t=2, Tasks B and C arrive. Overload is detected. The value densities of the tasks are A - 5, B - 4, C - 1. Task C has the lowest value density and is therefore placed in the pending queue.
t=5, Tasks D and E arrive. Again, there is an overload. The value densities of the tasks are, A - 5, B - 8, C - 1, D - 1, E - 3.5. Tasks C and D are placed in the pending queue, removing the overload
t=8, Task B completes, accrueing a value of 24. Task C is abandoned as it can no longer meet its deadline. Task E commences execution.
t=12, Task F arrives. Task D is abandoned. Task E continues to execute.
t=15, Task E completes with a value of 24.5. Task A executes
t=17, Task A completes with a value of 10. Task F executes
t=18, Tasks G and H arrive. Overload is detected. Tasks F, G and H now have value densities of 1.875, 1 and 3 respectively. Task G is placed in the pending queue, relieving the overload. Task F has the earliest deadline and continues executing.
t=22, Task F completes with a value of 7.5. Task G is abandoned. Task H commences executing.
t=25, Task H completes with a value of 9.
Best-Effort policy, total value accrued = 75.

Figure 7.5: Operation of the Best-Effort Policy.

co-processor resulted in consistently high utility under overload conditions, as only 2% of the application processors capacity was required for scheduling.

7.1.4 D^{over} Algorithm

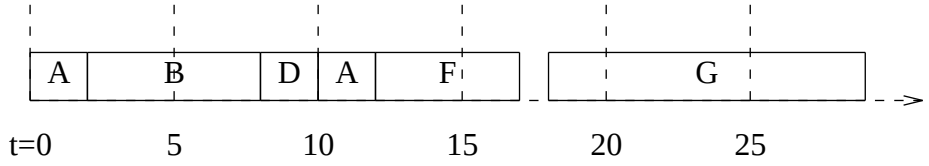
In 1992, Baruah *et al* showed via an adversary argument that the best an on-line algorithm can guarantee during overload, is $\frac{1}{(1+\sqrt{k})^2}$ of the value that can be achieved by a *clairvoyant* algorithm with knowledge of all future task arrivals [18]. Where, *k* is the *importance ratio* of the task set, defined as the ratio of the highest and lowest task value densities. This result is valid for the simple computational

model described earlier (section 7.1). Subsequently, Koren and Shasha [55] introduced the D^{over} algorithm which they proved achieves the above best possible guarantee, when task execution times are known upon arrival. They also showed that if task execution times are stochastic, such that $(1-\varepsilon)c_{\max} \leq c \leq c_{\max}$, where c is the actual execution time and $c_{\max}$ the worst case execution time, then the D^{over} algorithm achieves $\frac{1}{(1+\sqrt{k})^2+\varepsilon k(1+\sqrt{k})+1}$ of the maximum achievable value.

The D^{over} algorithm is implemented using three queues, a ready queue, a pending queue and a latest start time (LST) queue. The ready queue is ordered by earliest deadline and contains all tasks accepted for execution. The pending queue is also ordered by deadline and contains tasks which have arrived but have not as yet been accepted. Finally, the latest start time queue contains the same tasks as the pending queue, but ordered by absolute deadline less execution time. At any time, the task at the head of the ready queue is executed. Upon arrival, each task is first inserted into the pending and LST queues. When a task arrives or completes, the task at the head of the pending queue is subject to an acceptance test, if it can be accepted without causing overload, then it is removed from the pending and LST queues and inserted into the ready queue. When the latest start time of the task τ_{lst} at the head of the LST queue is reached, it is removed from the pending and LST queues and its value is compared with the total value of all the tasks in the ready queue. If its value is greater than $(1+\sqrt{k})$ times the total value of the tasks in the ready queue, then all the tasks in the ready queue are removed and inserted into the pending and LST queues and task τ_{lst} is placed in the ready queue and executed. Otherwise, task τ_{lst} is abandoned.

The complexity of the D^{over} algorithm is $O(logn)$ at each scheduling point, where n is the number of tasks present in the three queues.

Figure 7.6 illustrates the operation of the D^{over} algorithm.
Although the D^{over} algorithm guarantees to achieve at least $1/(1+\sqrt{k})^2$ of the total value achieved by a clairvoyant algorithm, for realistic importance ratios e.g. $k=10$ this bound is small (2.8%). However, in order to make this guarantee, in the case of extremely adverse arrival patterns the D^{over} algorithm is conservative in abandoning previously accepted tasks. For example, in figure 7.6, at time 8, the D^{over} algorithm



t=0, Task A arrives and commences execution.
t=2, Tasks B and C arrive. The system is overloaded. The LST interrupt for Task C is **t=2**. As the value of task C is less than $(1+\sqrt{k}) = 3$, times the total value of tasks A and B, it is abandoned. Task B executes as it has the earliest deadline.
t=5, Tasks D and E arrive. The system is again overloaded. The LST interrupt for task E is set to **t=8**. Task B continues to execute.
t=8, Task B completes, accrueing a value of 24. The LST interrupt for task E goes off. The value of task E is 24.5 which is less than $(1+\sqrt{k}) = 3$, times the total value of tasks A and D (12). Task E is therefore abandoned. Task D executes.
t=10, Task D completes, accrueing a value of 2. Task A executes.
t=12, Task A completes, accrueing a value of 10. Task F arrives and executes.
t=17, Task F completes with a value of 7.5.
t=18, Tasks G and H arrive. Overload is detected. The LST of task H is **t=25**. Task G executes.
t=25, The LST interrupt for task H goes off. As its value is less than 3 times the value of task G, task H is abandoned.
t=29, Task G completes with a value of 11.
D^{over} **policy, total value accrued = 54.5.**

Figure 7.6: Operation of the D^{over} Policy.

rejects task E (value 24.5) in favour of tasks A and D, even though executing task E only really conflicts with the execution of task D (value 2). This conservatism in rejecting previously accepted tasks results in average case performance which is generally worse than that of the Best-Effort scheduling algorithm.

7.1.5 RED Algorithm

In 1993, Buttazzo and Stankovic extended the Best-Effort approach with the Robust Earliest Deadline algorithm (RED) [27].

The RED algorithm is implemented using a ready queue and a pending queue. When a task arrives, it is inserted into the ready queue. The algorithm then determines if an overload exists in any of the intervals up to the deadline of each task in the ready queue. If no overload exists, the new task τ_{new} is accepted. Otherwise, the tasks in the ready queue are scanned in value order, least value first. The first task found which if removed results in the other tasks being schedulable, is rejected and placed in the pending queue. At each task completion, the task at the head of the pending queue is inspected, if it has negative laxity, it is rejected, otherwise the above acceptance test is repeated.

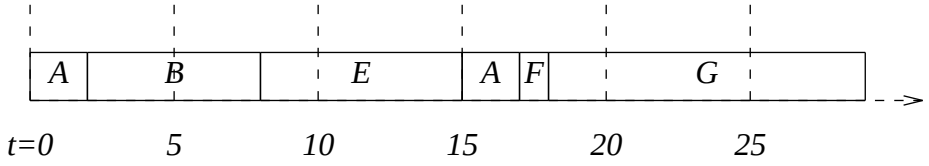
The operation of the RED algorithm is illustrated in figure 7.7.

The complexity of the RED algorithm is $O(n)$ at each scheduling point, where n is the total number of tasks in the ready queue and pending queue. This reduced complexity with respect to the Best-Effort approach is due to differences in the rejection policies used by the two methods. The RED algorithm rejects at most one task to make processing time available for a newly arrived task of high value. By comparison, the Best-Effort algorithm may reject any number of previously accepted tasks.

Unlike the Best-Effort approach which uses value density, the RED algorithm uses task value to determine which task to reject. This can however lead to a long task with high value (but low value density) being scheduled at the expense of a number of shorter tasks which individually have lower values, but in total, have a higher value. For example, in figure 7.7, at time 18, both tasks F and H are rejected in favour of task G which has a lower value and needs more execution time than both F and H combined.

7.1.6 Review Summary

In the underloaded case, the Earliest Deadline First algorithm achieves 100% of the possible value, assuming that task execution times are known upon arrival. In the presence of overload however, the EDF algorithm performs poorly.



t=0, Task A arrives and commences execution.
t=2, Tasks B and C arrive. The system is overloaded. The values of the tasks are, A - 10, B - 24, C - 7. Task C is the lowest value task which removes the overload. It is therefore placed in the pending queue. Task B executes.
t=5, Tasks D and E arrive. Again there is an overload. The values of the tasks in the run-queue are, A - 10, B - 24, D - 2, E - 24.5. Task D is the lowest value task which removes the overload. It is therefore placed in the pending queue and Task B continues to execute.
t=8, Task B completes, accrueing a value of 24. Task C is abandoned. Task D remains pending and Task E commences execution.
t=12, Task F arrives. Task D is abandoned as it can no longer meet its deadline. Task E continues to execute.
t=15, Task E completes with a value of 24.5. Task A executes
t=17, Task A completes with a value of 10. Task F executes
t=18, Tasks G and H arrive. Overload is detected. Tasks F, G and H have values of 7.5, 11 and 9 respectively. Both tasks F and H are placed in the pending queue, as they have lower values. Task G executes.
t=29, Task G completes with a value of 11. Tasks F and H are abandoned.
RED algorithm, total value accrued = 69.5.

Figure 7.7: Operation of the RED algorithm.

The Best-Effort, RED and D^{over} algorithms build upon the EDF approach. In the presence of overload, they admit or reject tasks based on their values or value-densities, thus relieving the overload. The EDF policy is then used to schedule the remaining tasks.

In the worst case, during overload, the best any on-line algorithm can guarantee to obtain is a small percentage of the total value obtained by a clairvoyant algorithm. (For task execution times which may be arbitrarily small and an importance ratio of 10, this bound is less than 1%).

In the average case however, the Best-Effort algorithm achieves a high total utility, outperforming other simpler techniques (provided that overheads are assumed to be negligible). In reality, the overheads of the Best-Effort algorithm may be large as its complexity is $O(n^2)$. The RED algorithm is somewhat less effective at obtaining maximum utility due to its policy of rejecting at most one task to accommodate a newly arrived task of high value. This however means that the RED algorithm has $O(n)$ complexity. Finally, the D^{over} algorithm has $O(logn)$ complexity and guarantees to obtain $\frac{1}{(1+\sqrt{k})^2}$ of the possible value obtained by a clairvoyant scheduler. In the average case however, this method is not particularly effective, due to the conservative rejection policy needed to provide the guarantee.

The table below summaries the performance of each algorithm, given the task set used in our examples.

Performance		
Policy	Total Value Obtained	Overheads
EDF	35.5	O(logn)
Value-Density	52.5	O(logn)
D^{over}	54.5	O(logn)
RED	69.5	O(n)
Best-Effort	75	$O(n^2)$

7.1.7 Maximum Guaranteed Utility

In this section, we build upon the research of Baruah [18]. Given a set of mandatory tasks which are guaranteed to meet their deadlines, (and therefore must be completed) and a set of optional tasks with constant values for their timely completion, we derive an upper bound on the ratio of the additional value which an on-line algorithm can guarantee to obtain from completing optional tasks, compared to that which can be obtained by a clairvoyant algorithm, with prior knowledge of optional task arrivals.

Theorem 7.1: The upper bound on the ratio of additional value guaranteed by an on-line algorithm, to that which can be obtained by a clairvoyant scheduler is zero; for a set of mandatory tasks which must be completed and a set of optional tasks with constant values for completion prior to their deadlines.

Proof: The proof follows from an adversorial argument. Let τ_1 be a hard task with computation time C_1 and deadline $D_1 = C_1 + \varepsilon$. Task τ_1 is a mandatory task and must therefore be completed, however, it contributes nothing to the additional value accrued for the timely execution of optional tasks. We assume that τ_1 is released at time 0.

At time 0, the adversary offers an optional task τ_A with computation time $C_A = \varepsilon$, deadline $D_A = \varepsilon$ and value $V_A = \varepsilon$.

Case 1: The on-line algorithm chooses to reject τ_A. The adversary offers no other optional tasks, thus the additional value obtained by the on-line algorithm is zero whilst a clairvoyant algorithm can achieve a value of ε by scheduling τ_A.

Case 2: The on-line algorithm chooses to execute τ_A. At time t, $\varepsilon < t < D_1 - \varepsilon$ the adversary offers optional task τ_B with $C_B = X$, absolute deadline $d_A = D_1 + X - \varepsilon$ and value Xk where k is the importance ratio. By executing τ_A, the on-line algorithm is unable to accommodate τ_B, whilst a clairvoyant scheduler can reject τ_A and achieve a value of kX by completing τ_B. The ratio of value obtained, for the completion of optional tasks, by the on-line algorithm and clairvoyant scheduler is therefore $\varepsilon : kX$. As $\varepsilon : X$ may be arbitrarily small, this bound is effectively zero.
□

Alternatively, if X is the ratio of the longest to shortest optional task execution time, then the bound becomes $1/kX$.

For realistic systems, this bound is very small, in the rest of this chapter, we therefore concentrate on defining admission policies which provide good average case performance but do not offer any guarantee of the value which will be obtained for completing optional tasks.

7.2 Admission Policies

In this section we integrate Best Effort and Fixed Priority approaches to task scheduling. We assume that the set of hard sporadic / periodic tasks complies with the computational model given in section 2.2.1. In addition, the following assumption applies: The hard deadline task set is assumed to be schedulable using fixed priority pre-emptive dispatching with a priority ordering determined by some means such as Deadline Monotonic priority assignment [63].

In addition to the set of hard tasks, we consider a set of optional tasks. Each optional task τ_A has a deadline D_A measured relative to its arrival and a worst-case execution time of C_A. Further, τ_A is assigned the optimal fixed priority level commensurate with its absolute deadline (i.e. the highest priority level such that every hard task and every previously guaranteed optional task of lower priority has a later absolute deadline). Each optional task is aperiodic and arrives at time ϕ_A (it arrives only once). Further, each optional task τ_A has a constant utility or value V_A, determined upon arrival, which will add to the total utility of the system, provided that τ_A completes by its deadline. There is no penalty and no utility in completing an optional task after its deadline.

Upon arrival, each optional task is subject to an admission policy and on-line acceptance test. Three admission policies are considered; a simple FCFS policy which admits optional tasks in arrival time order, a Best-Effort policy which takes account of the value of completing the various competing tasks, and a new Adaptive Value-Density Threshold policy which rejects optional tasks if their value-density is less than a variable cut-off, but otherwise admits tasks in order of arrival time. Each of these policies uses a sufficient on-line acceptance test (see section 6.3) to determine if each optional task is schedulable.

7.2.1 FCFS Policy

The simplest possible admission policy is FCFS. Under this policy, optional tasks are acceptance tested in order of arrival. Once accepted and given an on-line guarantee, an optional task will not have its guarantee rescinded. This means that a low value-density task arriving earlier than a high value-density task will prevent the latter from

executing, assuming that there is insufficient spare capacity to schedule both.

As the complexity of the sufficient acceptance tests given in chapter 6 is $O(n+m)$, where n is the number of hard tasks and m is the number of previously guaranteed but as yet uncompleted optional tasks, the overall complexity of the FCFS policy is also $O(n+m)$. Note, this assumes that rejected optional tasks are discarded: in effect each task is given only one chance of being accepted.

7.2.2 Best-Effort Policy

The Best-Effort approach to optional task admission is summarised below.

Best-Effort Policy:

```
Insert the new optional task in the (empty) pending-queue
For each optional task in the run-queue
        Calculate the task's current value-density
        If the task's value-density is less than that of the new
        optional task then
                rescind its on-line guarantee
                reclaim the task's remaining execution time
                remove it from the run-queue and
                insert it into the pending-queue
        endif
endfor
For each optional task in the pending-queue,
        (ordered by value-density)
        Remove the task from the pending-queue
        Check if it can be accepted
        If the optional task is accepted
                insert it into the run-queue
        endif
endfor
```

We note that when the on-line guarantee given to an optional task τ_A is rescinded, its remaining execution time, $c_A(t)$ is reclaimed, exactly as if it had completed early producing gain time $c_A(t)$ (see section 3.3.1).

In common with the FCFS policy, optional tasks have only one chance of being accepted under the Best-Effort policy. However, unlike FCFS, the Best-Effort policy arbitrates between competing optional tasks on the basis of their value-densities. Hence Best-Effort discards previously accepted optional tasks in favour of later

arrivals with higher value-densities. Despite this 'intelligent' determination of which optional task to admit, spare capacity may still be wasted if a low value-density task is partially executed and then abandoned.

The worst-case time complexity of the Best-Effort policy is $O(m(m+n))$.

7.2.3 Adaptive Threshold Policy

The Adaptive Value-Density Threshold policy (AVDT) approximates the operation of the Best-Effort policy but incurs a much lower overhead. In effect, each time the Best-Effort policy is executed, a value-density level cut-off may be observed; most optional tasks with a value-density below this level are not accepted for execution.

The AVDT policy filters out optional tasks whose value-densities are lower than some threshold level. Such tasks are rejected out of hand. Optional tasks which are not removed by this filter are admitted for acceptance testing in order of arrival. Further, the run-time guarantees afforded to optional tasks are never rescinded under the AVDT policy.

The value-density threshold used in the filter, is determined according to the equation below.

$$threshold = \frac{score(t)}{spare - time(t)} \tag{7.1}$$

Where $score(t)$ is the sum of the utilities of optional tasks completed up to time t and $spare - time(t)$ is the processor time not utilised by hard task computation up to time t. Stated otherwise, the threshold is the running average of system utility accrued per unit of processing time which was either idle or used to execute an optional task.

The AVDT policy is based upon the idea that executing low value-density optional components is counter productive, it is better to save spare capacity for high value-density tasks which might be expected to arrive soon. By setting the filtering threshold based on the utility or value achieved per unit of previous (potential) optional task execution, this policy adapts to the given level of optional components

present. In effect it predicts the need to save spare capacity for future high value-density tasks based upon its level of success in doing so in the past.

The overheads of the AVDT policy are similar to those of FCFS, however, additional accounting is required to maintain values for $score(t)$ and $spare-time(t)$.

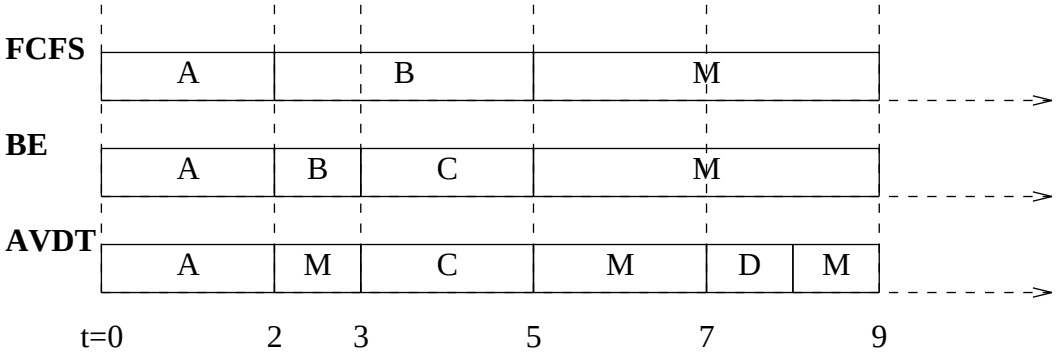
We note that when applying the AVDT policy in a highly dynamic system, it is necessary to determine the required filtering threshold using running values of $score(t)$ and $spare-time(t)$ which correspond to the utility achieved and optional task processing time available in, an interval of time prior to t. The appropriate length for this interval needs to reflect the dynamic nature of the system.

7.2.4 Policy Operation

We now give a simple example of the operation of the FCFS, Adaptive Threshold (AVDT) and Best-Effort (BE) policies. Our example is based on the task set detailed in the table below.

Task set					
Name	Type	Arrival time	Value	Execution time	Deadline
A	optional	0	6	2	2
B	optional	2	3	3	5
C	optional	3	10	2	5
D	optional	7	4	1	9
M	mandatory	0	-	4	9

(Note, deadline refers to the absolute deadline of the task). Initially, the slack on the mandatory task, M, is 5 units, (i.e. the mandatory task may be subject to an addition 5 units of interference without missing its deadline). Further, all the optional tasks have deadlines which are less than or equal that of the mandatory task and are therefore allocated higher priority levels.



FCFS:
t=0, Optional task A arrives. Mandatory task M has 5 units of slack hence, task A is accepted and commences executing.
t=2, task A completes with a value of 6. Optional task B arrives and is also accepted, reducing the slack on task M to zero.
t=3, Optional task C arrives and is rejected as there is no slack on either task B or M.
t=5, task B completes with a value of 3. Task M commences execution.
t=7, task D arrives and is rejected, as there is no slack on task M.

FCFS policy, total value accrued = 9.

Best-Effort:
t=0, Optional task A arrives, is accepted and commences executing.
t=2, task A completes with a value of 6. Optional task B arrives and is also accepted, reducing the slack on task M to zero.
t=3, Optional task C arrives. As task B has a lower value-density (1.5 v 5) its guarantee is rescinded in favour of task C.
t=5, task C completes with a value of 10. Task M commences execution.
t=7, task D arrives and is rejected, as there is no slack on task M.

Best-Effort policy, total value accrued = 16.

AVDT:
t=0, Optional task A arrives, as the value-density threshold is initially zero, task A is accepted and commences executing.
t=2, task A completes with a value of 6. Optional task B arrives and is rejected as the threshold is now 3 whilst the value-density of B is 1.
t=3, Optional task C arrives and is accepted as its value-density of 5 is greater than the threshold (3).
t=5, task C completes with a value of 10. Task M executes.
t=7, Optional task D arrives and is accepted as its value-density is the same as the threshold (4), and task M has 1 unit of slack remaining.
t=8, task D completes with a value of 4.

AVDT policy, total value accrued = 20.

Figure 7.8: Operation of the FCFS, Best-effort and AVDT Policies.

The example given in figure 7.8, serves to highlight the salient features of the policies studied: the FCFS policy accepts low value-density tasks at the expense of being unable to accept higher value-density tasks which arrive later. By comparison the Best-Effort policy initially accepts any schedulable low value-density task but later may rescind its guarantee (potentially wasting processor time), in order to accept a higher value density task. Finally, the Adaptive Threshold policy rejects low value-density tasks out of hand, preserving spare capacity for any subsequent arrivals of high value-density tasks.

7.3 Performance Evaluation

We now evaluate the performance of the FCFS, Best-Effort and Adaptive Threshold policies when combined with the sufficient acceptance tests given in chapter 6 for Background, Dual Priority and Slack Stealing methods of identifying spare capacity.

In our simulations, we assumed that the admission policy and scheduling overheads were zero. (Note, overheads are discussed further in section 7.4.2).

The criteria used to assess performance was the percentage of an upper bound on the achievable system utility which was obtained by each approach.

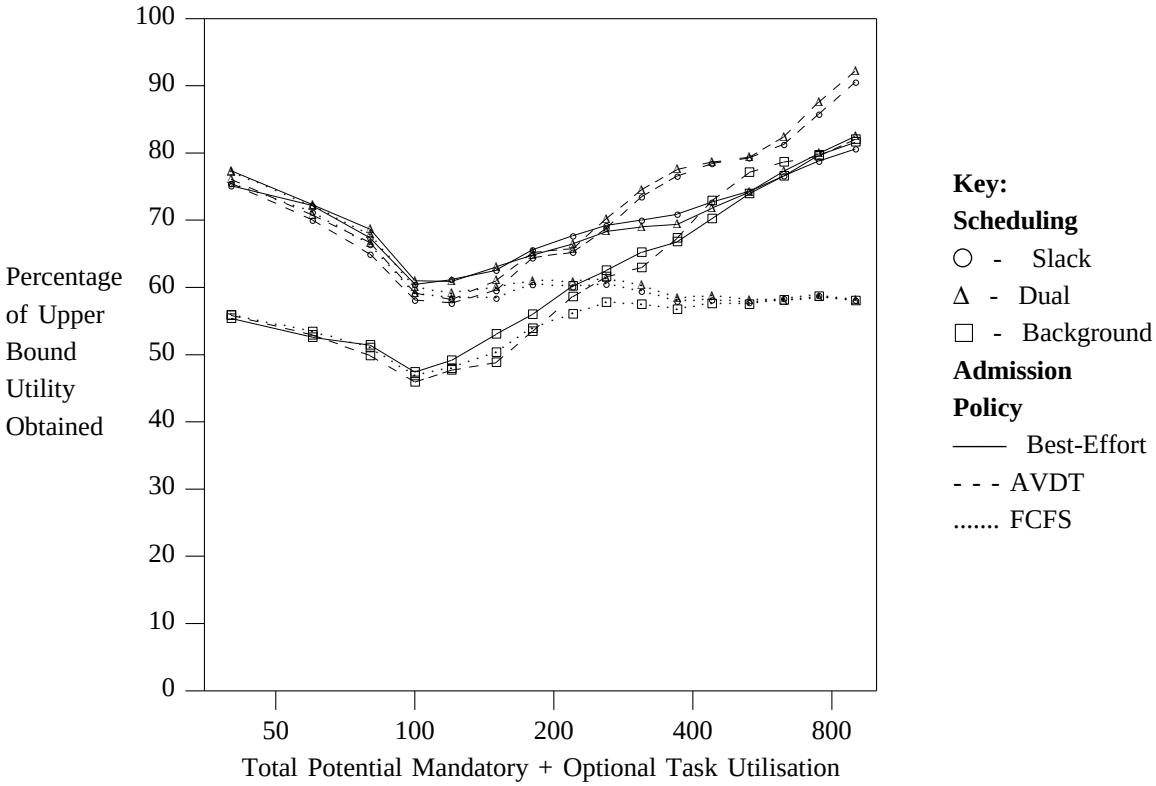
In our experiments, the hard task load was simulated using groups of task sets with utilisation levels of 30, 50, 70 and 90%. Each task set had an exponential distribution of task periods in the range 40 to 2560 ticks. Task deadlines and execution times were chosen at random such that $C \leq D \leq T$. The optional task load was simulated by a stream of aperiodic tasks with an exponential distribution of deadlines from 20 to 1280 ticks and execution times chosen at random to be 10,20,30...90% of the task's deadline. The mean optional task deadline was half the mean hard task deadline; a reasonable assumption given that optional tasks supporting mandatory counterparts typically have shorter deadlines. The arrival times of the optional tasks were given by a Poisson arrival process. Further, each optional task was given an initial value-density (from a uniform distribution in the range 1 to 10). This initial value-density was used to calculate the utility to be gained by completing the task before its deadline (utility equals initial value-density multiplied by

computation time). The number of optional tasks was varied to give combined potential utilisation levels (for hard + optional tasks) from 40 to 900%, plotted on the *x*-axis of the graphs.

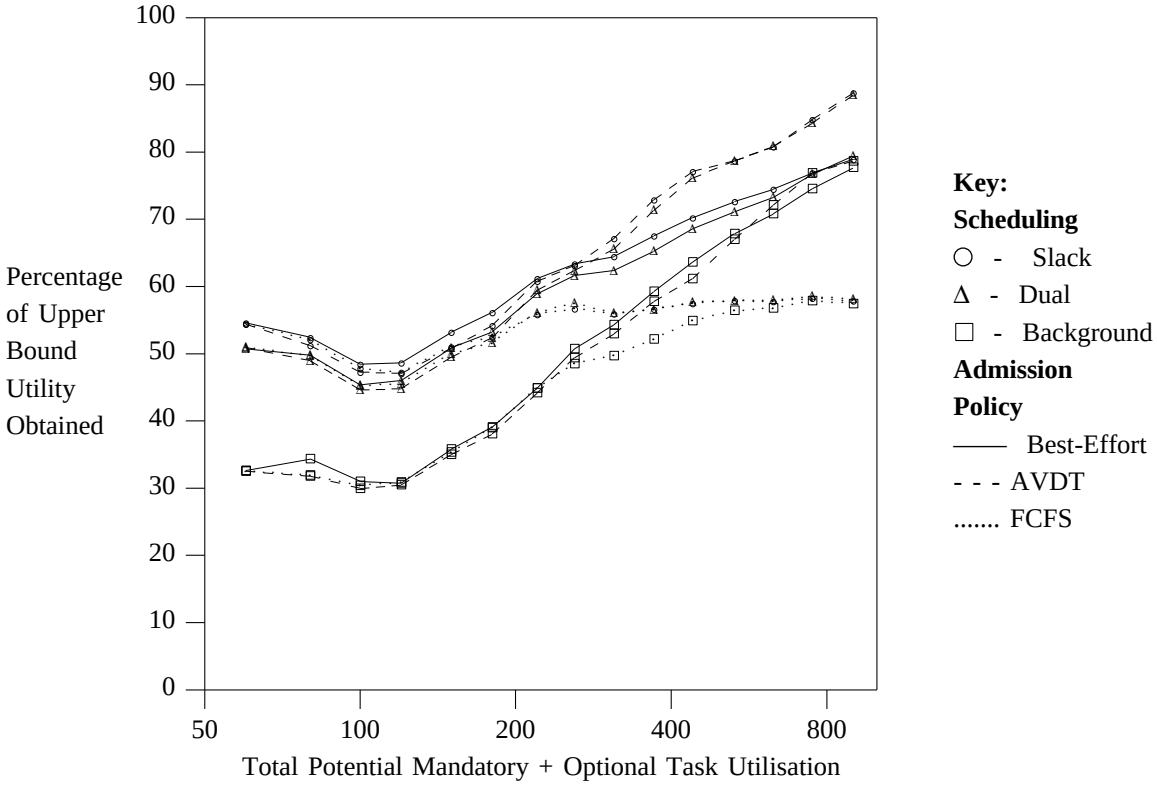
The queue of optional tasks was ordered by arrival time prior to commencing simulation. During each simulation run, once the arrival time of the optional task at the head of the queue had been reached, it was subject to the appropriate admission policy and acceptance test. The results presented in subsequent sections, represent the averages over a group of ten task sets.

7.3.1 Results

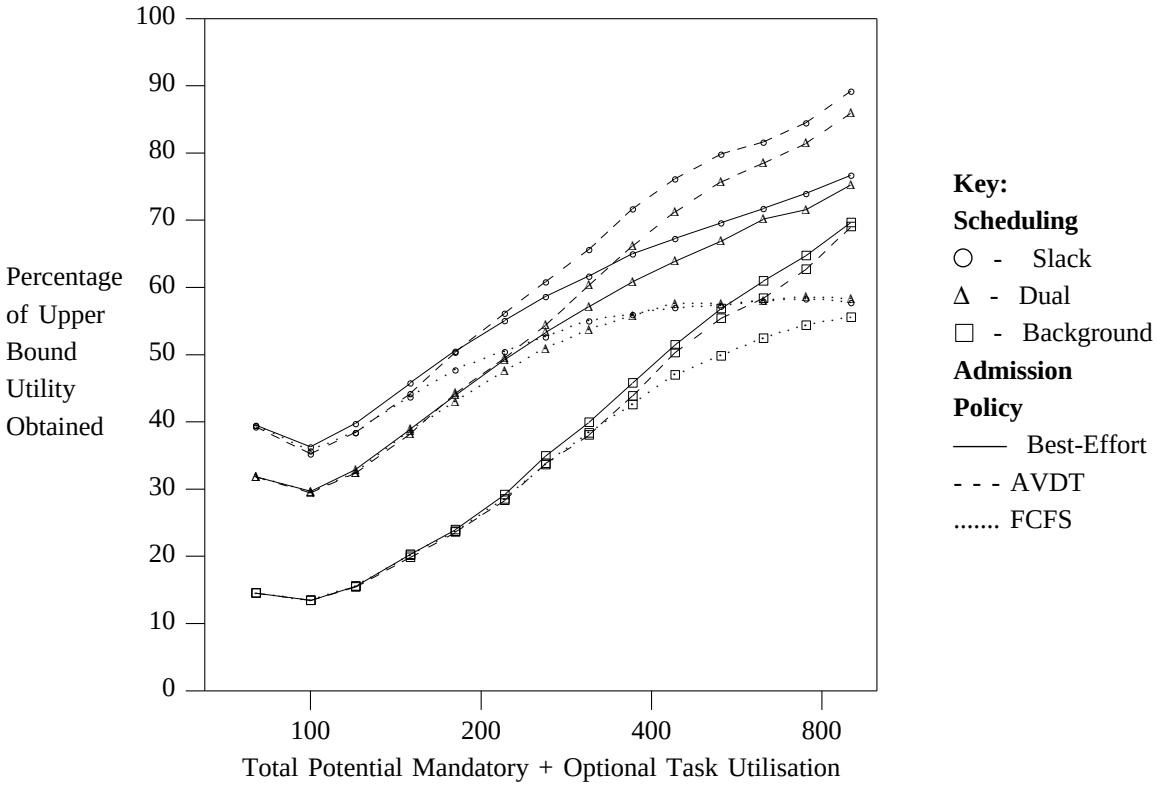
In experiments 7.1 to 7.4, we investigated the utility obtained by the three admission policies, FCFS, BE and AVDT when combined with Slack Stealing, Dual Priority and Background scheduling of optional tasks. The hard task sets used in experiments 7.1 to 7.4 had utilisations of 30, 50, 70, and 90% respectively.



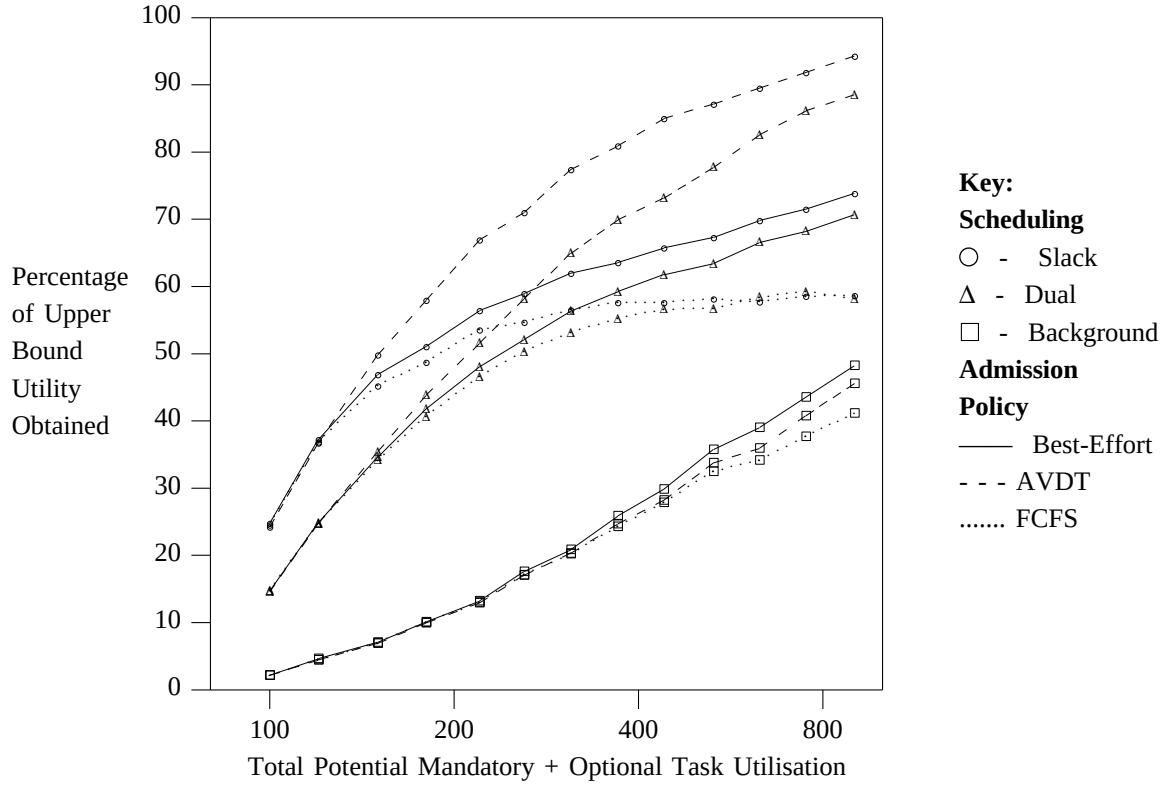
Expt. 7.1: Hard task set utilisation 30%.



Expt. 7.2: Hard task set utilisation 50%.



Expt. 7.3: Hard task set utilisation 70%.



Expt. 7.4: Hard task set utilisation 90%.

Each of the graphs shows the same basic behaviour for the nine approaches studied. At low optional task utilisation levels, the admission policy makes very little difference to the total utility obtained. This is because there is seldom more than one optional task pending at any given time. This optional task can either be accommodated by the scheduling method or not. Hence the utility obtained is highly dependent on scheduling method but insensitive to admission policy. As the utilisation of optional components is increased, up to a total utilisation level of 100%, all nine approaches show a decrease in the percentage of the upper bound obtained. This is because for utilisation levels of up to 100%, the upper bound corresponds to the sum of all optional tasks utilities, however as the total utilisation approaches 100%, an increasing proportion of the optional task load cannot be scheduled.

Above approximately 100% utilisation, the Best-Effort policy combined with each scheduling method provides increasing utility with increasing optional task utilisation. In effect, there are more schedulable high utility tasks to choose from, thus the Best-Effort policy is able to keep the processor busy with predominantly high value-density optional tasks. At very high utilisation levels, (e.g. 900%

equivalent to a 9 fold overload) the scheduling policy becomes less important as there are nearly always a number of high value-density optional tasks which could be scheduled. Thus the utility obtained becomes highly dependent on admission policy. At very high utilisation levels, the Best-Effort policy outperforms FCFS irrespective of the underlying scheduling method.

The Adaptive Threshold policy outperforms Best-Effort at high utility levels (>~150%). This can be explained by considering the following scenario. Assume that at a given time, there is a set of runnable optional tasks with medium or low value-densities. The Best-Effort policy admits a subset of these tasks, perhaps those with medium value-densities. By comparison, the AVDT policy rejects all these tasks and instead allows hard tasks to continue executing. Now at some later time, when a high value-density task arrives, the AVDT policy has preserved sufficient spare capacity to admit it. However, under the Best-Effort policy, any available spare capacity has been exhausted by executing lower value-density tasks, thus obtaining a lower total utility. This effect is most significant when the hard task set has a high utilisation (e.g. experiment 7.4) as there is a high probability that a high priority hard task can be executed, thus preserving the spare capacity.

This is a key difference between scheduling task sets comprising solely optional tasks and scheduling tasks sets containing a mixture of mandatory and optional tasks. In the latter case, it is often best not to schedule a low value optional task even though it is possible to do so, as hard tasks can execute, preserving spare capacity for later more valuable computation.

At very high utilisation levels, (e.g. 900%), the above effect becomes less marked as there are nearly always optional tasks with the highest value-density available for the Best-effort policy to admit. Further, when the AVDT policy is combined with Background scheduling, the above effect no longer occurs as optional tasks are only ever executed when there are no hard tasks runnable. Thus spare capacity is never preserved and the Best-Effort policy always outperforms AVDT.

Also, at very high utilisation levels (e.g. 300-900%), the FCFS policy tends towards attaining 55% of the upper bound utility. In fact, this is exactly as expected; with a very large number of optional tasks to choose from, the upper bound

corresponds to fully utilising the processor by executing optional tasks with the highest value-density (= 10), (as well as executing the mandatory tasks). By comparison, FCFS admission effectively makes a random choice, w.r.t. value-density. As the value-densities of optional tasks follow a uniform distribution in the range (1-10), the expected total utility for the FCFS policy is 55% of the upper bound.

7.4 Discussion

In this section, we discuss the limitations of our simple model of utility functions, the overheads of the various policies and issues relating to practical implementation.

7.4.1 Utility Model

Our simple model of utility functions assumes that the utility of an optional task is a constant value which is set upon arrival. Further, there is no utility in completing the task after its deadline. Although simplistic, this basic model has an appropriate form for use with optional components which improve the utility of mandatory tasks. Typically, such optional computation is of no value if it is not delivered prior to the deadline of the hard task which it supports. Further, setting the optional task's utility upon arrival, allows the necessary flexibility for optional components to have different utility levels depending upon the perceived state of the external environment and the mode of the system.

The simple model described above lacks some of the expressive power of the more general utility functions proposed by Locke [68]. It has however, the advantage of much lower overheads in terms of the evaluation of such functions.

7.4.2 Overheads

The worst-case overheads, in terms of the number of acceptance tests required per optional task arrival are: FCFS - $O(1)$, AVDT - $O(1)$ and Best-Effort $O(m)$. Where m is the number of optional tasks previously accepted, but as yet uncompleted. Recall that the complexity of the sufficient acceptance test used is $O(m+n)$, where n is the number of hard tasks.

The complexity of the AVDT policy is thus $O(m)$ less than that of Best-Effort. Given the results of the simulations reported in the previous section and the outcome of Wendorf's experiments [110] into the overheads of Best-Effort scheduling, it would appear that the Adaptive Threshold policy is a better choice for practical systems. However, the optional task load in our experiments was roughly consistent across each simulation duration. This allowed the Adaptive Threshold policy to effectively predict future optional task arrivals. In a more dynamic system, the threshold used by the AVDT policy needs to be based on a time interval which reflects the time period over which the optional task load may change dramatically. It remains an open issue; whether the performance of the AVDT policy would still approach that of Best-Effort under these more stressful circumstances.

7.4.3 System Dynamics

In hard real-time systems where the optional components are related to the (periodic) hard task set, the Adaptive Threshold policy represents a highly effective means of addressing one of the main issues in maximising system utility: to execute a low utility task now or to conserve spare capacity in the hope that a higher utility task will arrive soon. In this respect, the AVDT policy could be considered as providing a limited form of clairvoyance. If the system's average value-density is high then AVDT discards low value-density tasks as it assumes that ones of higher value-density will arrive soon. In a predominantly periodic system, with optional components related to the hard task set, this is the correct assumption to make and thus the policy performs well.

The policies described earlier assume that each optional task is either accepted or rejected out of hand. Alternatively, tasks failing the acceptance test could be kept on the pending queue awaiting further spare capacity. This increases the worst-case overheads and raises the issue of when to re-invoke the acceptance testing of such tasks. Typically, a pragmatic approach may be taken, whereby at each task completion, the optional task, if any, at the head of the pending queue is subject to admission / acceptance testing, as per the RED algorithm [27].

Finally, there is the orthogonal issue of whether or not the on-line guarantees given to optional tasks may be rescinded. If this is not permissible for any optional component, then the Best-Effort policy effectively reduces to simple FCFS. In general, allowing guarantees to be rescinded provides more flexibility: to, for example, schedule soft tasks at the highest priority level. However, some optional tasks may exhibit 1/0 constraints: once started, such tasks must be guaranteed to complete. In this case, it may still be permissible to rescind the on-line guarantee after acceptance testing but prior to the task commencing execution.

7.5 Multiple Versions Scheduling

In this section, we outline how the Adaptive Threshold approach can be extended to choosing which version or strategy to execute under the multiple versions paradigm [94].

The multiple versions paradigm assumes that there are a number of strategies available to implement a given service τ_i. We refer to these strategies as τ_i^1, τ_i^2, $\tau_i^3 ... \tau_i^k$. Each strategy τ_i^k is assumed to have a worst case execution time C_i^k and a value V_i^k which is determined upon arrival. Further, the service has a relative deadline D_i which is inherited by each strategy. A mandatory version of the service described by τ_i^{MAN}, C_i^{MAN}, V_i^{MAN} is guaranteed via off-line analysis.

Upon release of service τ_i, the maximum processing time, $A_i(t)$, which can be guaranteed to be made available before its deadline is calculated using the techniques given in chapter 6. (Note, this requires $O(n)$ time). In effect, $A_i(t)$ is the minimum of the processing time available at priority level i in the interval $[t, t+D_i)$ and the extra interference (termed slack) which any task of priority less than i may be subject to without causing its next deadline to be missed. Thus,

$$A_i(t) = C_i^{MAN} + \min\left(A_i(t, D_i), \min_{\forall j \in lp(i)} S_j(t) \right) \tag{7.2}$$

Where $A_i(t, D_i)$ is a lower bound on the additional execution time available at priority level i in the interval $[t, t+D_i)$ and $S_j(t)$ is the slack at priority level j at time t

Once the available computation time $A_i(t)$ has been calculated, then the set of schedulable strategies can be identified ($\tau_i^k : C_i^k \leq A_i(t)$).

To determine which of the schedulable strategies to choose for execution, we use the idea of effective system value-density, E^{SYS}, defined as the "running average" value accrued by the system per unit time. (Note, E^{SYS} is similar to the value-density threshold introduced earlier, however, as we now assume that all computation has a value, the average is taken over all processor time, not just that which is unused by mandatory tasks).

We assume that if a given strategy τ_i^k is executed, then it will require processing time C_i^k and accrue value V_i^k. Further, in any remaining processing time $(A_i(t) - C_i^k)$ it is assumed that the system will accrue value at a rate given by the current system value-density. Hence the effective value-density (E_i^k) of strategy τ_i^k is given by:

$$E_i^k = \frac{V_i^k + (A_i(t) - C_i^k) E^{SYS}}{A_i(t)} \tag{7.3}$$

Once the effective value-density of each schedulable strategy has been computed, then the one with the highest effective value-density is chosen for execution. (Note, as $A_i(t)$ is constant over all the strategies of τ_i then division by $A_i(t)$ is not required in determining which the strategy has the highest effective value-density). Finally, the slack at priority levels *i* and lower needs to be adjusted to reflect the strategy chosen for execution:

$$\forall j \in lp(i)\, S_j(t) := S_j(t) + (C_i^{MAN} - C_i^k) \tag{7.4}$$

Where the slack at each priority level is initially determined assuming that the guaranteed mandatory strategy for each service is always executed.

The complexity of the above approach is $O(n+k)$ where *n* is the number of services and *k* is the number of strategies for the service.

7.5.1 Example

We now give a simple illustration of the operation of the Adaptive Threshold policy, as applied to multiple versions scheduling. Our example assumes that there are 3 strategies, τ_i^1, τ_i^2 and τ_i^3 available to implement a given service, τ_i. Typically, τ_i^1 might be a "quick and dirty" approximation, τ_i^2 an efficient and relatively accurate approximation and τ_i^3 a precise but computationally expensive method. The computation times and values, for the particular invocation of τ_i examined, are as follows: $C_i^1 = 1$, $C_i^2 = 3$, $C_i^3 = 7$, $V_i^1 = 1$, $V_i^2 = 9$ and $V_i^3 = 14$, hence the value densities of τ_i^1, τ_i^2 and τ_i^3 are 1, 3 and 2 respectively.

Let us assume that the processing time available, $A_i(t)$, is 8, thus all three strategies are schedulable. We now consider 3 cases, corresponding to low, medium and high system value-densities.

Case 1: High system value-density:

Assuming that $E^{SYS} = 5$, then τ_i^1 has the highest effective value-density. Executing τ_i^1 for 1 unit of time accrues a value of 1, whilst over the 7 remaining time units, other tasks are expected to accrue a value of 35, giving a total of 36 and an effective value-density of 4.5. By comparison, executing τ_i^2 for 3 units of time and other tasks for 5, gives a total value of 34 and an effective value-density of 4.25. Finally, executing τ_i^3 for 7 units of time and other tasks for 1, gives a total value of 19 and an effective value-density of 2.375. In this case, the average system value-density is high compared to that of any of the strategies for τ_i, hence, it is better to execute a quick and dirty strategy, releasing spare capacity for other services with higher value-density methods.

Case 2: Medium system value-density:

Assuming that $E^{SYS} = 2$, then τ_i^2 has the highest effective value-density. The values expected to be accrued in the 8 time units available are 15, 19 and 16 for strategies τ_i^1, τ_i^2 and τ_i^3 respectively; giving effective value-densities of 1.875, 2.375 and 2. In this case, the average system value-density is similar to that of the strategies for τ_i, hence the method with the highest value-density (τ_i^2) also has the highest effective value-density.

Case 3: Low system value-density:

Assuming that $E^{SYS} = 0$, then τ_i^3 has the highest effective value-density. The values expected to be accrued in the 8 time units available are 1, 9 and 14 for strategies τ_i^1, τ_i^2 and τ_i^3 respectively; giving effective value-densities of 0.125, 1.125 and 1.75. In this case, the average system value-density is much less than that of the strategies for τ_i, hence, with little or no contribution from other services, the method with the highest value i.e. τ_i^3, (but not necessarily the highest value-density) has the highest effective value-density.

7.5.2 Local Optimality

Providing the assumptions that each strategy τ_i^k takes C_i^k to execute and that any spare time not used by τ_i^k will on average result in value being accrued at the rate of E^{SYS} hold, then the Adaptive Threshold approach makes locally optimal decisions regarding which strategy to execute, in order to achieve maximum system utility. Whether such a simple policy can be successfully employed in real systems where execution times are stochastic and the rate at which system utility is accrued varies widely, remains an open issue.

7.6 Summary

In this chapter, we showed that for the case of mixed task sets, comprising optional tasks with constant values for completion before their deadlines, and hard tasks which must meet their time constraints; the best an on-line algorithm can guarantee, for completing optional tasks, is $1/kX$ of the value which could be obtained by a clairvoyant scheduler. Where k is the importance ratio, equal to Max : Min optional task value-density and X is the ratio Max : Min optional task execution time. As this bound is very small for realistic task sets, our research therefore focussed on policies which have good average case behaviour.

We showed how Best-Effort and Fixed Priority scheduling can be integrated, combining the desirable features of both approaches: guaranteeing tasks with hard time constraints and maximising system utility.

Our approach partitioned the problem of scheduling optional components into two parts: first an admission policy determines the order in which optional tasks are presented for on-line acceptance testing. As part of the acceptance test, tasks are assigned a priority level commensurate with their deadlines, according to the optimal priority assignment policy given in chapter 6. Successful tasks are then scheduled on the basis of this priority.

We investigated the effectiveness of three admission policies combined with Background, Dual Priority and Slack Stealing methods of scheduling optional tasks. These policies were FCFS, Best-Effort and a new Adaptive Value-Density Threshold (AVDT) approach which uses a variable cut-off to filter out optional tasks on the basis of their value-densities. Our simulations showed that for an exponential distribution of mandatory and optional task deadlines, the AVDT policy can result in significantly higher system utility than the Best-Effort policy (assuming Slack Stealing or Dual priority as the underlying scheduling method). As the worst case overheads of the AVDT policy are $O(1)$ acceptance tests per optional task arrival, compared with $O(m)$ for the Best-Effort policy, it appears that the AVDT policy is the more appropriate for use in practical systems. Finally, we showed how the idea of a value-density threshold could be extended to scheduling tasks under the multiple versions paradigm.

Chapter 8

Proof of Concept: Implementation

Before we can be certain that any particular theoretical scheduling technique is truly practical and effective, it is necessary to tailor the theory to the performance characteristics of an actual real-time kernel, and to implement the new scheduling policy and determine the effects of overheads on its performance.

Previous chapters have investigated the performance of the Dual Priority and dynamic Slack Stealing algorithms via simulation. This chapter focuses on measuring the overheads of a proof of concept implementation and incorporating these overheads into the supporting feasibility analysis. Both dynamic Slack Stealing and Dual Priority algorithms were implemented within the DRTEE hard real-time kernel [16]. This kernel was chosen due to its simple analysable design, its policy mechanism separation which provides the interface necessary for integrating experimental scheduling policies, and most importantly, the accessibility of the source code.

First, the basic operations of the DRTEE kernel are described, including the timer, monitor and logging facilities which enable various performance metrics to be examined. We then provide analysis which tailors response time scheduling theory (section 2.1.4) to the characteristics of the DRTEE kernel, thus enabling the schedulability of tasks with hard deadlines to be determined, given the overheads due to the implementation of each scheduling policy. Furthermore, we explain how the non-integrated method of event handling, used in the kernel, means that the release of lower priority hard tasks and soft aperiodic tasks impinges upon the schedulability of high priority hard tasks.

Subsequently, the generation of the synthetic task sets used in performance testing, is described. We then report on a series of experiments used to determine the overheads involved in soft task scheduling using Background, Dual Priority and dynamic Slack Stealing methods, for a range of task set cardinalities. Finally, we

discuss the limitations of the proof of concept implementation and possible areas for future improvement within the kernel.

8.1 DRTEE Kernel

The DRTEE kernel version 1.0 [6] is a basic event driven kernel which runs on the Intel i486. Initially, the kernel supported only basic fixed priority scheduling with timer facilities to prevent budget or deadline overrun. As part of this research, policy modules were added which support dynamic Slack Stealing and Dual Priority scheduling. The monitoring facilities were also extended (see section 8.1.4).

8.1.1 Timer Facilities

The kernel interfaces to a custom designed timer board which provides a real-time clock (48 bits, 62.5ns resolution) and three count-down timers (32 bits, 1µs resolution). Note, only one of these count-down timers is actually used in version 1.0. The count-down timer is programmable: it can be set to any 32 bit value and raises an interrupt when the count reaches zero. The timer facilities enable accurate logging and monitoring of the time spent executing tasks and the time spent in the scheduler. It is also possible to determine the time spent in the Timer Interrupt Handler (TIH) by comparing logged real-time clock and counter values. We return to this point in section 8.3.

8.1.2 Kernel Operation

Task scheduling is implemented using an array of task control blocks (TCB) and an event queue. The *CHOOSE_TASK* policy module inspects the array of TCBs and determines which task to execute. *CHOOSE_TASK* policy modules were implemented for each of the three scheduling policies: Background, Dual Priority and dynamic Slack Stealing.

The DRTEE Kernel also makes use of an event queue. This is a doubly linked list of timer events ordered by event time: the real-time clock value at which the event notionally takes place. Each timer event data structure contains the event time, the task identifier corresponding to the event and an event type. Event types include:

DEADLINE, *BUDGET*, *PRIORITY_PROMOTION* and *RELEASE*.

The underlying operation of the scheduler loop is as follows:

1. On entering the scheduler, interrupts are disabled, the count-down timer is stopped and read (Note, it may have already stopped at zero if the scheduler has been entered via the TIH). The real-time clock is also read.

2. The policy module *ACCOUNTING* is called and updates variables such as the current task's remaining execution time and, in the case of the dynamic Slack Stealing policy, the slack at each priority level above that of the current task.

3. The scheduler determines if a timer event has occurred. If this is the case then the current timer event is removed from the event queue and processed according to its type, as follows:

RELEASE: The task identified by the event's task id is marked as runnable and the *RELEASE* procedure called. This policy module returns either a *DEADLINE* event, (or in the case of Dual Priority Scheduling a *PRIORITY_PROMOTION* event) which is inserted into the event queue.

PRIORITY_PROMOTION (Dual Priority Scheduling only): The task identified by the event's task id has its priority raised to the appropriate upper band priority level.

DEADLINE (missed): The task identified by the event's task id is marked as both killed and not runnable. This is a failure condition.

BUDGET (overrun): If the current task is a guaranteed task, then it is marked as both killed and not runnable. Again this is a failure condition. If the current task is a soft task, then the budget event is used to force a return to the scheduler once the soft task's allotted execution time has been consumed. The task is therefore left as runnable. Note, this is the mechanism by which soft tasks are executed in slack time under the dynamic Slack Stealing policy.

4. If the scheduler has been entered via the task completion call gate, then there is no timer event to process, however, the scheduler now has the completion of the current task to deal with. On task completion, the *COMPLETION* procedure is called. This policy module removes the corresponding *PRIORITY_PROMOTION* or *DEADLINE* event from the event queue and in the case of periodic tasks, inserts a new *RELEASE* event into the event queue. Finally, the current task is marked as not runnable.

5. A runnable task is chosen for execution by the *CHOOSE_TASK* policy module. *CHOOSE_TASK* also determines the execution time budget for the task. Note, the idle task is always runnable and is therefore chosen if no application task wishes to execute.

6. Immediately prior to task dispatch, the real-time clock is read and the count down timer set to a value corresponding to the elapsed time to the next event. Note, if the time of the next event is at or before the current time, then the counter is set to 1 (as the interrupt is generated on the transition from 1 to 0). This ensures that the timer interrupt will be recognised immediately that interrupts are enabled.

7. The task chosen for execution is dispatched. (As part of dispatch, interrupts are enabled).

8. The task executes. (Note, it may not actually get chance to do any processing if there is a timer interrupt pending).

Either:

9a. The task completes, in which case, the scheduler is re-entered via the task completion call gate (which disables interrupts).

or

9b. The count down timer reaches zero and raises the timer interrupt, in which case, the scheduler is re-entered via the TIH, which also disables interrupts.

8.1.3 Discussion

There are two important underlying features of the operation of the DRTEE kernel.

First, the kernel provides a non-integrated method of setting timer interrupts. This simplifies both the kernel implementation and its analysis, however, the fact that the event which is set is simply the one with the earliest notional arrival time, means that the release events of low priority tasks interfere with the execution of higher priority tasks, increasing their response times. Note, that this is not the case with low priority completions, as these events can only occur when no higher priority task is runnable. We return to this issue in section 8.4.

Second, the scheduler is "one pass": on any invocation of the scheduler, only one logical event (i.e. task release, deadline, priority promotion or budget overrun) is dealt with. Again, this simplifies the implementation of the kernel and the analysis of its overheads. If subsequent events, still in the event queue, have notionally expired, then they are dealt with by setting the count-down timer to 1 and exiting the scheduler. The count-down timer immediately expires, raising the timer interrupt, which causes the scheduler to be re-entered via the TIH.

It would appear that only dealing with one event per pass through the scheduler is inefficient, due to the additional overheads of dispatch and the TIH, compared to looping within the scheduler until all notionally expired events have been dealt with. Whilst this is undoubted true in the average case, in the worst case, we must allow for the fact that event times may be such that they each occur a vanishingly small time after task dispatch. In these worst case circumstances, the scheduler will only ever deal with one event at a time, resulting in similar worst case overheads.

8.1.4 Logging and Monitoring

Within the DRTEE kernel, a comprehensive set of monitoring and logging facilities are provided.

Monitor Information	
Field	Contents
00	Number of hard task releases
01	Number of hard task completions
02	Number of hard task budget overruns
03	Number of hard task deadlines missed
04	Number of soft task releases
05	Number of soft task completions
06	Number of soft task budget overruns
07	Number of soft task deadlines missed
08	Total response time (soft tasks)
09	Number of acceptance tests
0a	Number of optional tasks accepted
0b	Total value accrued
0c	Max. task execution time
0d	Total soft / optional execution time
0e	Max. scheduler time for hard task release
0f	Max. scheduler time for soft task release
10	Max. scheduler time for hard task completion
11	Max. scheduler time for soft task completion
12	Max. scheduler time for priority promotion
13	Max. scheduler time for budget event
14	Total time for simulation run
15	Total time spent executing tasks (inc. idle)

The Monitor facilities allow data such as the number of task completions, deadline misses, total soft task response time and maximum scheduler execution time to be inspected at the end of an experimental run. The table above gives a summary of the information monitored. In addition, the monitor also records the maximum response time of each of the first 20 hard tasks.

The data logging facilities enable a trace of scheduler decisions and task execution to be constructed. On each pass through the scheduler, a log entry is made into a circular buffer. At the end of a run, the log entries present in the buffer can be inspected. Each log entry records; the real-time clock and count-down timer values on entering the scheduler, the event type (e.g. task release, deadline, budget, completion or priority promotion), any interrupts which are raised, the id of the task chosen for execution and its execution time budget, the real-time clock value on exiting the scheduler, and finally, the count-down timer value set.

8.2 Analysis of Kernel Overheads

As mentioned previously, the DRTEE kernel operates a non-integrated scheduling and timer interrupt policy. This means that the execution of a high priority hard task is interrupted in order to deal with the release of any lower priority tasks. Although simplifying the implementation of the kernel, this leads to undesirable priority inversion.

8.2.1 Computational Model and Notation

The operation of the kernel can be modelled as a set of fictitious tasks of different priorities, with timing attributes inherited from their actual counterparts. The following variables represent the worst case scheduler execution time (WCST) for dealing with the various events. (Note, the values of these variables are typically dependent upon the number of tasks present in the application and the scheduling policy used, see section 8.3).

K^{MREL} - The WCST for a hard (mandatory) task release event.

K^{OREL} - The WCST for a soft (optional) task release event.

K^{MCOMP} - The WCST for a hard (mandatory) task completion.

K^{OCOMP} - The WCST for a soft (optional) task completion.

K^{BUDGET} - The WCST for a budget event.

K^{PROMO} - The WCST for a priority promotion event.

(Note, the initial version of the kernel used in our experiments supports only periodic hard tasks).

In determining the worst case response time of hard task τ_i, we must account for an initial period of blocking due to the scheduler dealing with the release or completion of either lower priority hard or soft tasks, interference due to the release, execution and completion of higher priority hard tasks and interference due to the release of low priority hard and soft tasks. We note that this requires that the timing attributes (i.e. minimum inter-arrival time) of soft tasks are known *a priori*, as this has an impact on the schedulability of hard tasks. This is a major drawback of the non-integrated scheduling and event handling policy. We return to this issue in section 8.4.

In the following analysis, the overheads of scheduling events are modelled as a set of high priority ficticious tasks with, for example an execution time K^{MREL} and a period inherited from the task which the scheduling event corresponds to.

8.2.2 Analysis of Background Scheduling

The schedulability of a hard task τ_i executing under the DRTEE Background scheduling policy, is determined by the following analysis:

$$B_i = max(K^{MREL}, K^{OREL}, K^{MCOMP}, K^{OCOMP}) \tag{8.1}$$

$$r_i^{m+1} = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil (C_j + K^{MREL} + K^{MCOMP}) +$$

$$\sum_{\forall j \in lp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil K^{MREL} + \sum_{\forall o \in soft} \left\lceil \frac{r_i^m}{T_o} \right\rceil K^{OREL} \tag{8.2}$$

Where *soft* is the set of soft / optional tasks. Note, the completion overhead for task τ_i is not included in the task's response time, as we are interested in the response time to the last observable operation of task τ_i, this does not include the final context switch away from τ_i [22]. Iteration begins with $r_i = 0$ and ends when $r_i^{m+1} = r_i^m$ or $r_i^m > D_i$ in which case, τ_i is unschedulable. If task τ_i is schedulable, then its worst case response time is given by the final value of r_i.

8.2.3 Analysis of Dual Priority Scheduling

Under Dual Priority Scheduling, we are interested in both determining the schedulability of each hard task τ_i and the corresponding maximum permissible priority promotion delay Y_i. Following the analysis given in chapter 5, we assume that task τ_i will not be able to execute until its priority is promoted to its upper band level. Once its priority has been promoted, then τ_i will be subject to interference from hard tasks with a higher promoted priority than *i*, including their release, priority promotion and completion overheads. With the non-integrated scheduling model, it will also be subject to interference due to the release and priority promotion events of lower priority hard tasks and the release events of optional tasks. Hence we have:

$$B_i = max(K^{MREL}, K^{OREL}, K^{MCOMP}, K^{OCOMP}, K^{PROMO}) \tag{8.3}$$

$$r_i^{m+1} = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil (C_j + K^{MREL} + K^{MCOMP} + K^{PROMO}) +$$

$$\sum_{\forall j \in lp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil (K^{MREL} + K^{PROMO}) + \sum_{\forall o \in soft} \left\lceil \frac{r_i^m}{T_o} \right\rceil K^{OREL} \tag{8.4}$$

Given the worst case response time of task τ_i as calculated above, its maximum priority promotion delay is given by:

$$Y_i = D_i - r_i \tag{8.5}$$

8.2.4 Analysis of Dynamic Slack Stealing

There is a fundamental difference between Slack Stealing and Dual Priority or Background approaches to soft task scheduling. With both of the latter techniques, soft task execution is pre-empted when a hard task becomes runnable at a higher priority level (i.e. at an upper band priority level in the case of Dual Priority scheduling). Under the dynamic Slack Stealing approach, a hard task τ_j may be runnable at effectively the same priority as the soft task which is executing. In this case, task τ_j needs to be resumed once the available slack has been exhausted, i.e. once:

$$\min_{\forall i \in lp(j)} S_i(t) = 0 \tag{8.6}$$

Thus a budget event must be set to terminate the execution of a soft task in slack time. Further, the budget set, needs to account for the fact that before hard task τ_j can be resumed, an additional pass through the scheduler may occur, dealing with either the completion of the soft task or the expiry of its execution time budget. (Note, the next event might be a release, in which case no extra scheduling event is observed). To ensure that the hard tasks remain schedulable, any additional pass through the scheduler must be accounted for in terms of decrementing the available slack. Hence at time *t*, when τ_j is the highest priority runnable hard task, and there is a soft task pending, then the soft task may execute provided that:

$$\min_{\forall i \in lp(j)} S_i(t) > \max(K^{OCOMP}, K^{BUDGET}) \tag{8.7}$$

In which case, its execution time budget is set to:

$$\min_{\forall i \in lp(j)} S_i(t) - \max(K^{OCOMP}, K^{BUDGET}) \tag{8.8}$$

Further, in dealing with soft task completion or budget expiry, the scheduler calls the *ACCOUNTING* policy module to decrement the available slack by the appropriate overhead i.e. K^{OCOMP} or K^{BUDGET}.

Under the dynamic Slack Stealing approach, the feasibility of hard tasks is determined according to the equations given below.

$$B_i = max(K^{MREL}, K^{OREL}, K^{MCOMP}, K^{OCOMP}, K^{BUDGET}) \tag{8.9}$$

$$r_i^{m+1} = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil (C_j + K^{MREL} + K^{MCOMP}) +$$

$$\sum_{\forall j \in lp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil K^{MREL} + \sum_{\forall o \in soft} \left\lceil \frac{r_i^m}{T_o} \right\rceil K^{OREL} \tag{8.10}$$

Similarly, the calculation of available slack, using the dynamic approximation detailed in section 4.1.3, is modified as follows: the interference in the interval $[t, d_i(t))$, due to each higher priority task τ_j and its associated scheduling overheads is given by:

$$I_j(t, d_i(t)) = \begin{pmatrix} 0 & \textbf{if } c_j(t) = 0 \\ c_j(t) + K^{MCOMP} & \textbf{otherwise} \end{pmatrix} + f_j(t, d_i(t))(K^{MREL} + C_j + K^{MCOMP}) +$$

$$\min\left((K^{MREL} + C_j + K^{MCOMP}), (d_i(t) - x_i(t) - f_j(t, d_i(t))\, T_j)_0 \right) \tag{8.11}$$

Where:

$$f_j(t, d_i(t)) = \left\lfloor \frac{d_i(t) - x_i(t)}{T_j} \right\rfloor_0 \tag{8.12}$$

Further, the kernel overheads due to the releases of lower priority hard tasks and soft tasks are given by:

$$K_j(t, d_i(t)) = f_j(t, d_i(t))\, K^{MREL} + \min\left(K^{MREL}, (d_i(t) - x_j(t) - f_j(t, d_i(t))\, T_j)_0 \right) \tag{8.13}$$

and

$$K_o(t, d_i(t)) = f_o(t, d_i(t))\, K^{OREL}$$

$$+\min\left(K^{OREL},(d_i(t)-x_o(t)-f_o(t,d_i(t))\,T_o)_0\right) \tag{8.14}$$

respectively.

Note, the time t used in determining, for example, the time $x_j(t)$ to the next release of task τ_j is assumed to be the exit time from the current scheduler invocation. This is found by adding the worst case scheduler overhead for an invocation of the particular type (e.g. K^{MCOMP}) to the real-time clock value read on entry to the scheduler. This ensures that the slack calculated is a lower bound on that which is actually available at dispatch time. This lower bound is given by:

$$S_i(t)=d_i(t)-c_i^*(t)-\sum_{\forall j\in hp(i)} I_j(t,d_i(t))-\sum_{\forall j\in lp(i)} K_j(t,d_i(t))-$$

$$\sum_{\forall o\in soft} K_o(t,d_i(t)) \tag{8.15}$$

Where $c_i^*(t)=c_i(t)$ if task τ_i is runnable and C_i otherwise.

8.3 Performance Measurement and Evaluation

In this section, we describe a series of experiments used to determine the average and worst case overheads of Background, Dual Priority and dynamic Slack Stealing scheduling policies as implemented in the DRTEE kernel. Previous simulations (see chapters 4 and 5), investigated the average case performance of the different techniques over many task sets. In this chapter, we focus on the effect of overheads and thus make use of tasks sets which theoretical (i.e. assuming zero scheduling overheads) present the same load, but in practice lead to different levels of overhead due to the variation in task set cardinality.

8.3.1 Task Set Generation

The task sets used to evaluate the DRTEE kernel and experimental scheduling policies resembled those used in previous simulations.

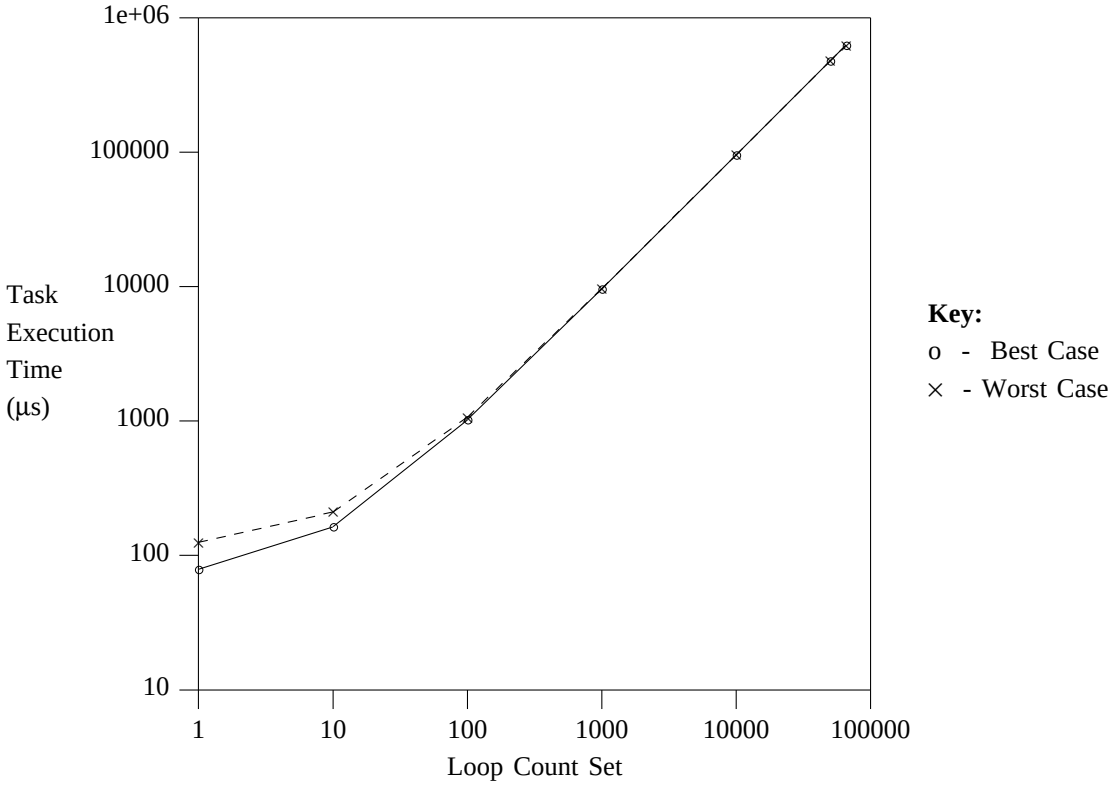
The synthetic application comprised two sets of tasks, *mandatory* and *optional*. The optional task set was used to represent soft tasks. These tasks had an exponential distribution of periods in the range 80-2560ms (and very long deadlines c. 30 minutes). The utilisation of the optional task set was varied from 10 to 30%. The cardinality of the optional task set was 5.

The mandatory task sets comprised 5, 10, 15, 20, 25, 30 or 35 tasks, again with an exponential distribution of periods in the range 80-2560ms. The utilisation of each mandatory task set was 50%. Each mandatory task set with greater than 5 tasks was effectively replicated from a similar task set of cardinality 5. Hence a mandatory task set of cardinality 20, comprised 4 replicas of each of the 5 tasks in the base set (cardinality 5). Each of these replicas had one quarter of the execution time requirement of the task it was derived from and shared the same timing requirements. Thus the higher cardinality task sets presented exactly the same processor load in terms of task execution, however, the scheduling requirements were markedly increased. This approach was taken to factor out variations due to random differences in the timing characteristic of task sets generated with differing cardinalities.

As part of task generation, each mandatory task set was shown to be feasible under each of the scheduling policies (Background, Dual Priority and Slack Stealing), assuming the worst case scheduling overheads measured on the DRTEE test-bed and the timing characteristics of the corresponding optional task set.

Once the timing characteristics of a task set had been determined, the task set generation tool was used to produce an application definition file This was down-loaded to the DRTEE kernel to set up Task Control Block data such as; task periods, wcets, deadlines etc. Further, the task set generation tool also produced a Task#.c file for each application task by copying a template file and replacing tokens corresponding to execution time. The template file contained the C code for two nested *for* loops. In effect, one of the loop counts was given a value such that the worst case execution time of the task corresponded to the desired quantity. Graph 8.1 shows how the measured worst and best case execution times of a task varied according to the loop count. (Note, the minimum and maximum permitted task execution times produced by the task set generation tool were, 80µs and 500ms respectively).

From experiment 8.1, it is clear that the synthetic tasks exhibited very little variation in execution time, thus providing a severe test of the scheduling theory and kernel implementation: even a small degree of optimism in the calculation of slack or priority promotion times would lead to missed deadlines.



Expt 8.1: Synthetic Task Execution Time v. Loop Count

8.3.2 Measurements

Before any proper experimentation could take place, it was first necessary to determine the worst case scheduling overheads given a particular policy and number of tasks. This was achieved by generating random task sets of the desired cardinality and recording the worst case time spent in the scheduler for various events. These worst case times were then fed into the task generation tool and further task sets produced. This process was repeated a number of times until stable worst case values were obtained. The final values were used in the generation of the task sets used in subsequent experiments.

We note that this method is not entirely satisfactory, as we cannot be sure that the worst case scheduler overheads for each event type and task set cardinality have been obtained. However, in each experimental run, the scheduler overheads were monitored and the worst case compared with the values previously found. If a larger value was produced, then the series of experiments was abandoned, a new worst case value set and task generation and subsequent experiments repeated.

Ideally, worst case execution time analysis could be applied to the kernel, however, it is unreasonable to do this by hand due to the complexity and size of the kernel and unfortunately no analysis tools exist as yet (the development of such a tool is part of on-going research).

Anomalies

Initial experimentation with one or two tasks showed up two anomalies which had to be corrected for. First, the monitored value of the time spent in the scheduler, recorded only the time between reading the real-time clock at the start of the scheduler code and a similar reading immediately prior to task dispatch. However, this did not include the overheads of the TIH or task completion call gate prior to entering the scheduler, nor the time spent dispatching an application task before it was actually executed by the processor. Measured worst case values were obtained for these quantities by reference to logged and monitored data.

The ‘TIH + Dispatch’ time was found by subtracting the value set in the count-down timer from the time elapsed between setting the count-down timer at the end of one scheduler invocation and reading the real-time clock at the start of the next scheduler invocation, following a count-down timer interrupt. The dispatch time was found via measuring the difference in the apparent execution time of a task with a long computation time when it is allowed to execute without pre-emption and when it is interrupted by a series of budget events. The latter is longer due to the inclusion of multiple dispatch calls. Finally, the overhead of the task completion call gate was found by setting up a race condition between budget expiry and task completion. Immediately the task completion call gate is entered, interrupts are disabled. Thus by setting progressively smaller budgets for a given task, it is possible to determine when task completion is just before the timer interrupt and hence the actual task

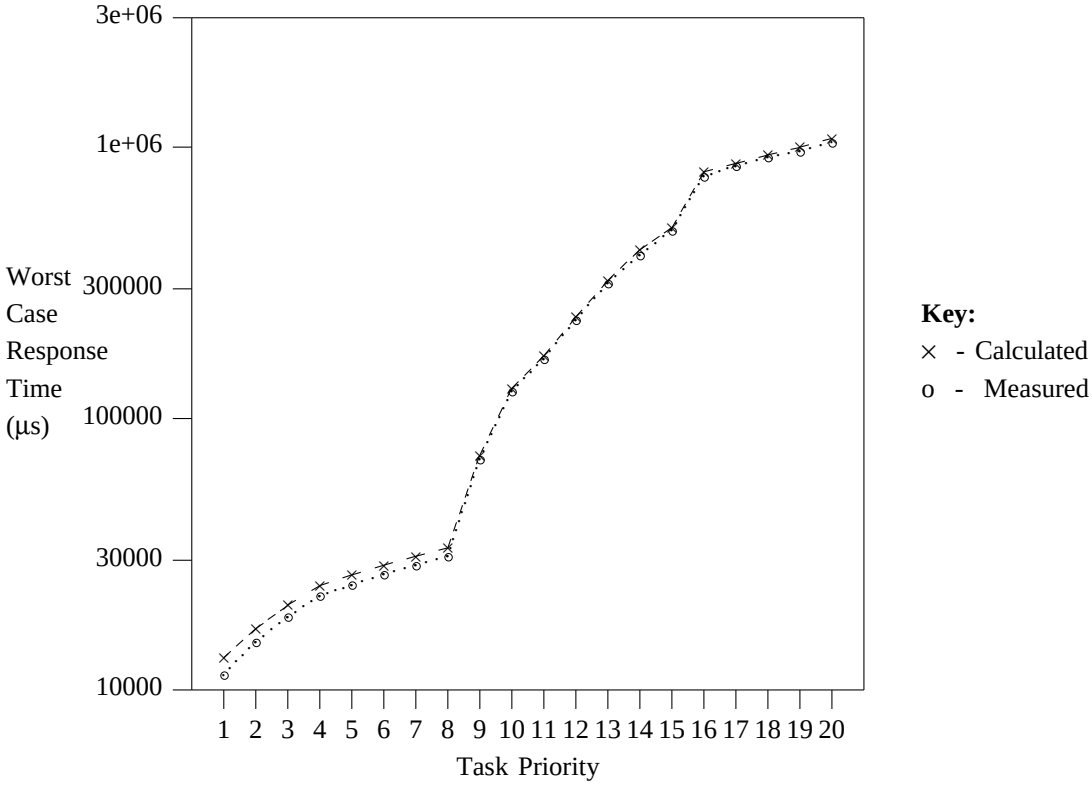
execution time (excluding task completion call) and thus the task completion call gate overhead.

Missing Time	
TIH	162µs
Task Completion	120µs
Dispatch	17µs

We now present the results of a series of experiments which investigated scheduling overheads. First, we compare measured and calculated worst case task response times under the various scheduling policies. We then examine how the worst case and average case overheads of each scheduling policy vary with the cardinality of the mandatory task set. Finally, the mean response time of soft tasks was used as a performance metric to illustrate how scheduling policy effectiveness is degraded by increasing overheads.

8.3.3 Measured v. Calculated Worst Case Response Times

In experiment 8.2, we compared the calculated and measured worst case response times of mandatory tasks scheduled under fixed priority pre-emptive scheduling (i.e. the Background policy for soft tasks). In this experiment, the task set comprised 20 mandatory tasks with a total utilisation of 80% and 5 optional tasks with a utilisation of 10%. The worst case Scheduler overheads amounted to 6.3% of the total processing time, whilst the average case overheads were 5.5%. The small differences between measured and calculated worst case response times for the tasks were due to a combination of the small variations in both scheduler and task execution times.



Expt 8.2: Worst Case Response Times

Worst Case Response Times (µs)			
Priority	Calculated	Measured	% Difference
1	13130	11358	13.5
2	16820	14970	11.0
3	20510	18550	9.6
4	24200	22135	8.5
5	26490	24328	8.2
6	28780	26523	7.8
7	31070	28718	7.6
8	33360	30916	7.3
9	73310	70561	3.7
10	129440	125685	2.9

Priority	Calculated	Measured	% Difference
11	169780	165342	2.6
12	236490	230485	2.5
13	322240	314254	2.5
14	418180	398056	4.8
15	503540	491858	2.3
16	810410	776479	4.2
17	870010	850295	2.3
18	935210	914021	2.3
19	1000410	962301	3.8
20	1076580	1036154	3.8

8.3.4 Overheads v. Number of Tasks

In this section, we investigate how the worst case and average case scheduling overheads of Background, Dual Priority and Slack Stealing approaches vary with task set cardinality. In these experiments, the mandatory task sets used had a utilisation of 50% (cardinality of 5-35) whilst the optional task set had a utilisation of 30% and a cardinality of 5. (Note, a maximum of 40 tasks are permitted by the DRTEE kernel).

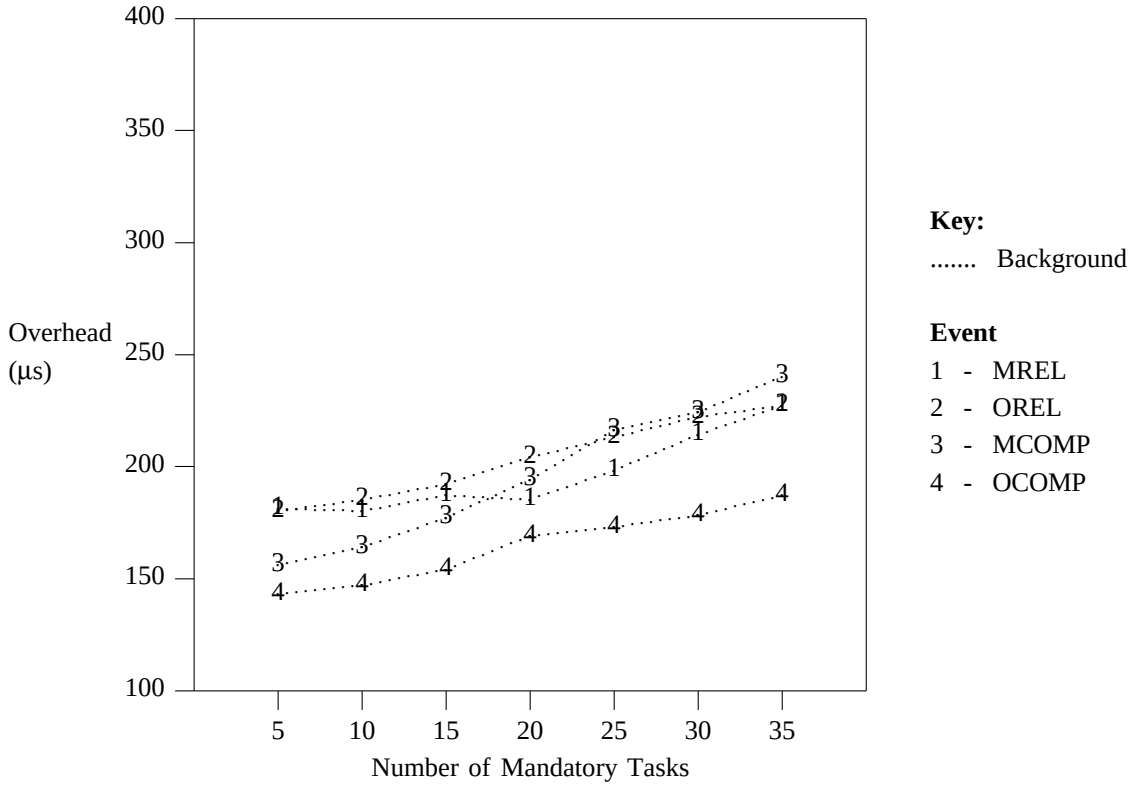
The graphs for experiments 8.3, 8.4 and 8.5 illustrate how the overheads of a single pass through the scheduler vary with task set cardinality for each scheduling approach. Each of these graphs shows the different overheads for the various types of scheduling event (e.g. MREL - mandatory task release, OREL - optional task release, MCOMP - mandatory task completion, OCOMP - optional task completion, PROMO - priority promotion and BUDGET - soft task budget expiry).

The overheads of Background and Dual Priority scheduling are well contained (<250µs for Background and < 300µs for Dual Priority) and show a linear growth with increasing task set cardinality. Similarly, graph 8.5 shows that the overheads of all scheduling events except mandatory task completions are less than 400µs for the

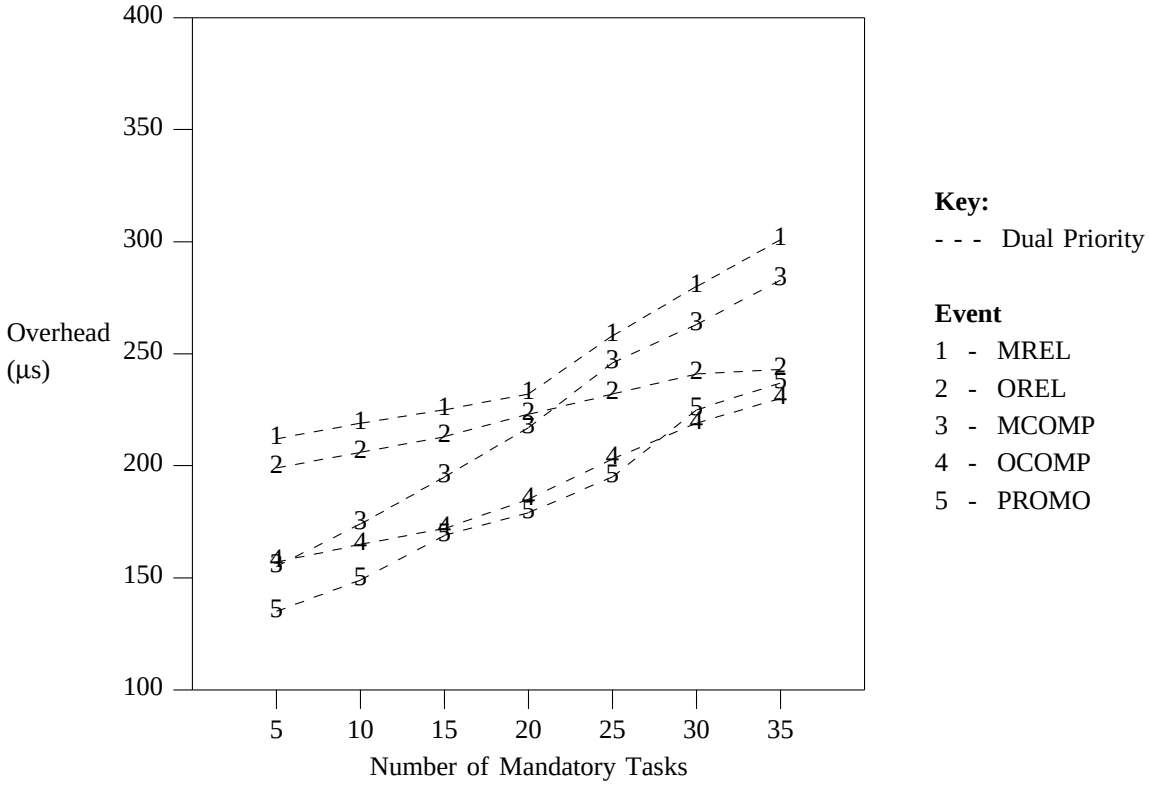
dynamic Slack Stealing approach. However, the calculation of slack, performed at the completion of each mandatory task, although also growing linearly, represents a large overhead: >1500μs for 35 tasks. This trend is reflected in graph 8.6 which illustrates the total scheduling overheads of each approach as a percentage of the total processing time.

These results show that although all three approaches incur overheads which are $O(n)$ at each scheduling point, it is necessary to also take into account the constant of proportionality when assessing the impact of these overheads. This constant is much larger for the dynamic Slack Stealing approach and its performance suffers as a consequence.

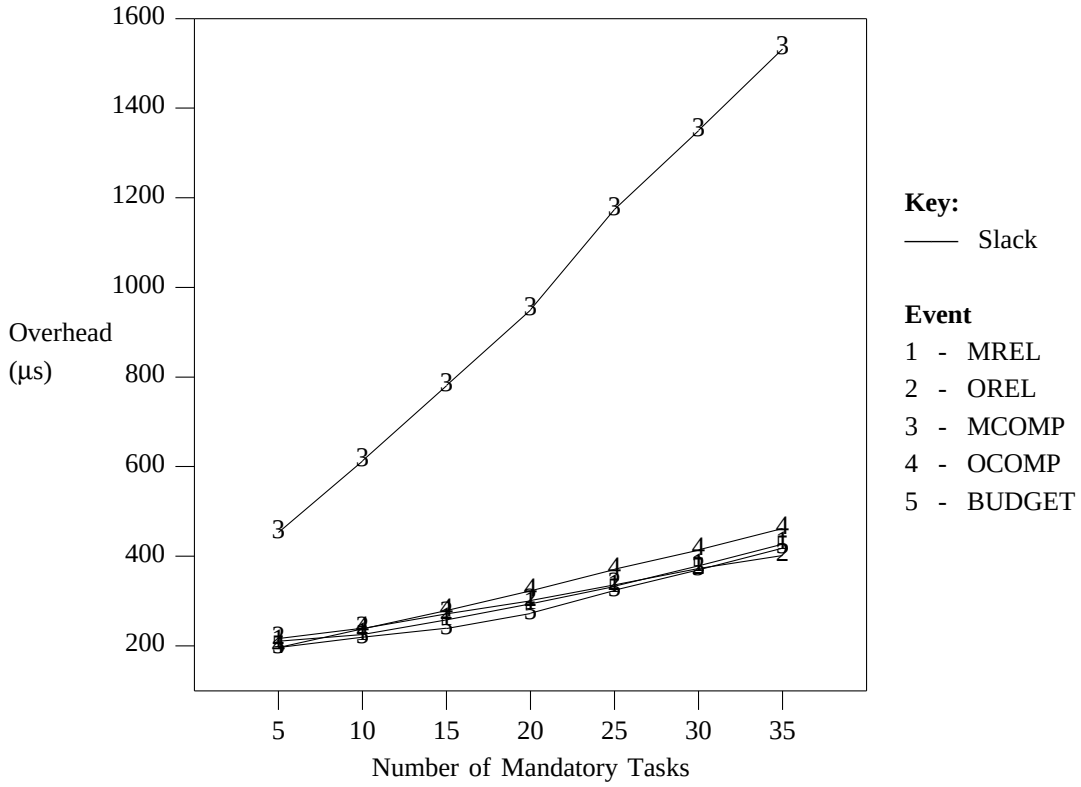
Experiment 8.7 shows the effect which increasing overheads (due to higher task set cardinality) have on the response time of soft tasks. In this experiment, once the task set cardinality reaches 25, the response time of soft tasks scheduled under the Slack Stealing approach rapidly degrades to that of Background scheduling.



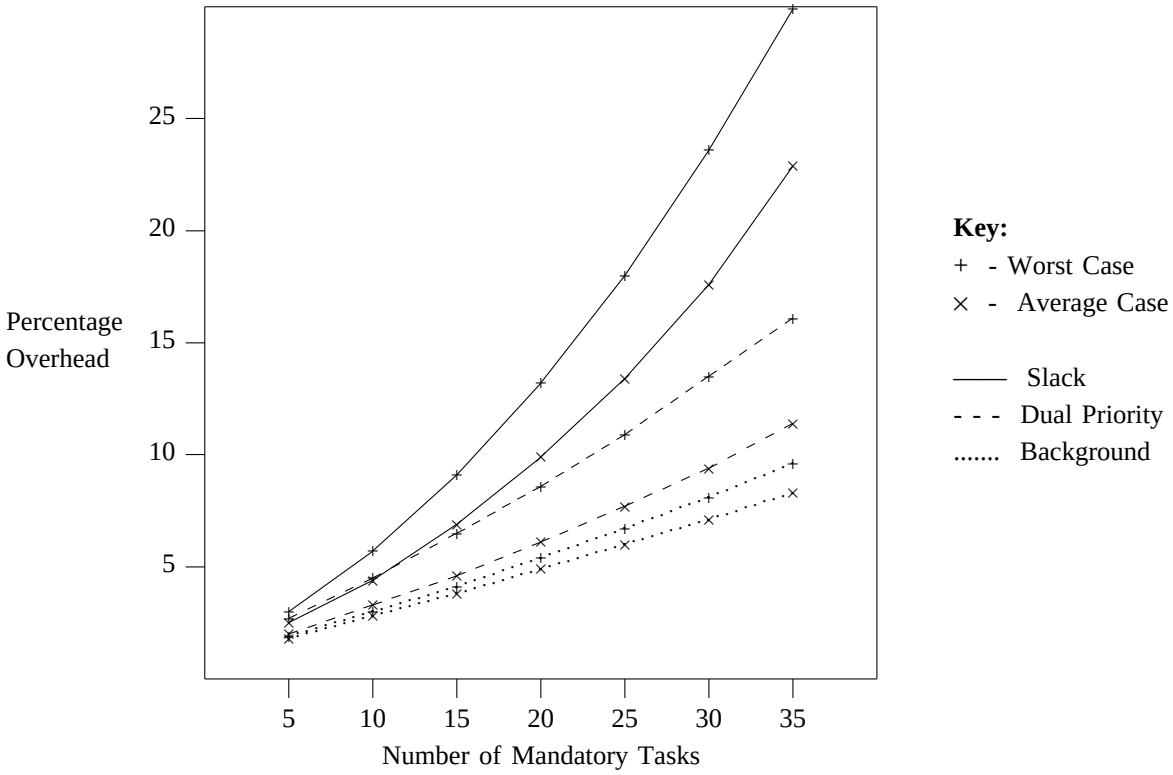
Expt 8.3: Background: Scheduling Overheads



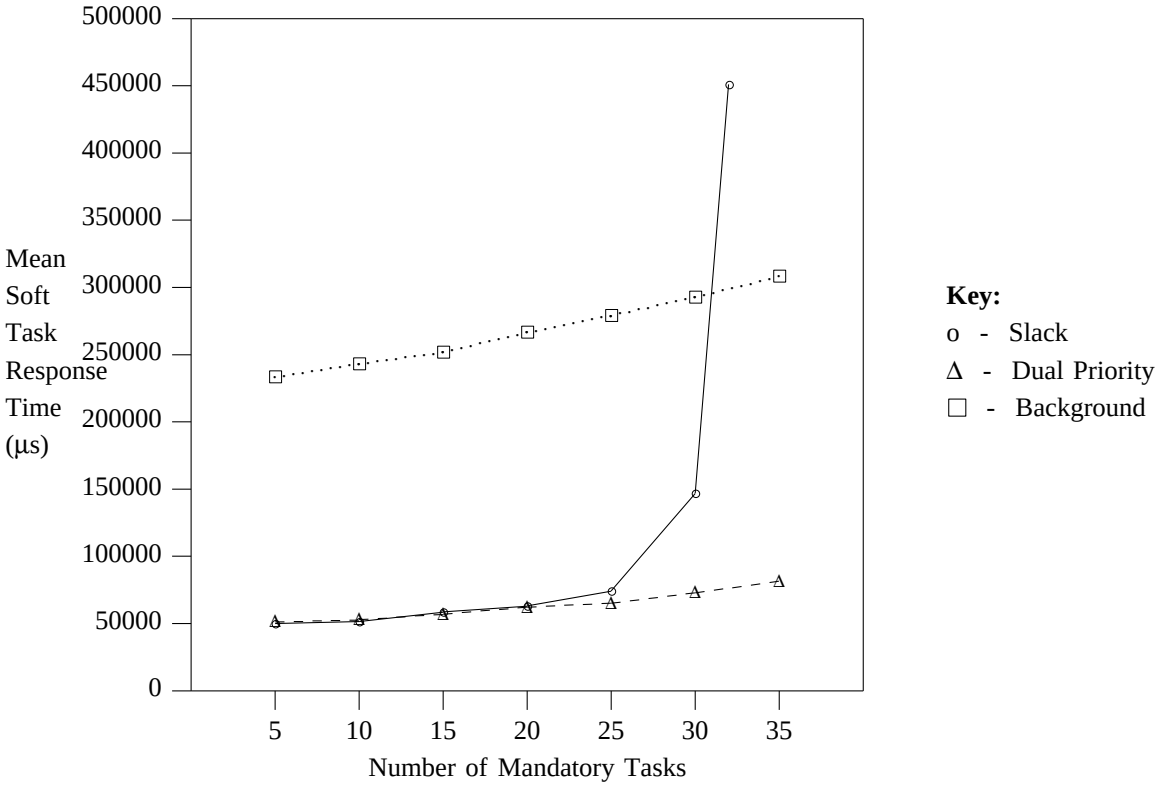
Expt 8.4: Dual Priority: Scheduling Overheads



Expt 8.5: Slack Stealing: Scheduling Overheads



Expt 8.6: Total Overheads as a Percentage of Processing Time



Expt 8.7: Soft Task Response Time

8.4 Limitations and Possible Improvements

During development and experimentation with the DRTEE kernel, two areas where the implementation could be improved were identified.

8.4.1 16 v 32 Bit Machine Code

The kernel was compiled using the Zortech C compiler which produces 16 bit machine code instructions. However, most of the variables manipulated by the kernel, indeed all the times (e.g. periods, release times, deadlines, budgets, priority promotion times and slack) were 32 bit quantities. This resulted in an implementation which was considerably less efficient than a corresponding 32 bit version. (Note, the initial version was produced in 16 bit form due to the unavailability of a suitable 32 bit compiler). This explains why the difference between the scheduling time for mandatory task completion, including the overhead of calculating slack (1300µs) is approximately 1000µs greater than that for mandatory task release (for 30 tasks scheduled under the slack stealing algorithm - see experiment 8.5); whereas this difference was expected (see section 4.3.4) to be approx 330µs assuming a 32 bit compiler.

8.4.2 Integrated Event Handling and Scheduling

The second drawback of the initial kernel implementation is the non-integrated method of handling events. As noted previously, this means that the release of low priority tasks impinges upon the schedulability of those with higher priorities.

It is intended that subsequent versions of the kernel will address this problem by only setting events which have a higher priority than the task which has been chosen for execution. Integrated event handling and scheduling can be achieved via the model and analysis given in appendix C.

Background Scheduling

The feasibility of hard tasks scheduled using the integrated event handling and scheduling approach is found according to the equations below:

$$B_i = max(K^{MREL}, K^{OREL}, K^{MCOMP}, K^{OCOMP}) \tag{8.16}$$

$$r_i^{m+1} = B_i + K^{MREL} + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil (C_j + K^{MREL} + K^{MCOMP}) \tag{8.17}$$

By comparison with equations (8.1) and (8.2) it is clear that the integrated approach has the significant advantage that the timing attributes of soft / optional tasks have no impact on the feasibility of hard tasks. Only the blocking factor due to the scheduling overheads of a single soft task release or completion need be considered.

Dual Priority Scheduling

Under the integrated Dual priority approach, although the initial priority of a hard task is actual in the low priority band, all release events are assumed to have the corresponding upper band priority. If this were not the case, then release and subsequent priority promotion events could be subject to unbounded blocking due to soft tasks executing at middle band priority levels. Equations (8.3) and (8.4) are modified accordingly:

$$B_i = max(K^{MREL}, K^{OREL}, K^{MCOMP}, K^{OCOMP}, K^{PROMO}) \tag{8.18}$$

$$r_i^{m+1} = B_i + K^{MREL} + K^{PROMO} + C_i +$$

$$\sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^m}{T_j} \right\rceil (C_j + K^{MREL} + K^{MCOMP} + K^{PROMO)} + \tag{8.19}$$

Given the worst case response time of task τ_i as calculated above, its maximum priority promotion delay is again given by:

$$Y_i = D_i - r_i \tag{8.20}$$

Dynamic Slack Stealing

Using an integrated approach to event handling and scheduling, the feasibility of hard tasks under the dynamic Slack Stealing algorithm may be determined according to equations (8.16) and (8.17) above, with the values of K^{MREL}, K^{MCOMP} etc reflecting the scheduling overheads of Slack Stealing rather than Background scheduling.

The handling of soft task release events is however somewhat more complex under the dynamic Slack Stealing algorithm. Here, we wish to deal with soft task releases as soon as possible, assuming that there is sufficient slack available. Hence if:

$$\min_{\forall i \in lp(j)} S_i(t) > \max(K^{OREL}) \tag{8.21}$$

Then soft task release events are considered to be of higher priority than the highest priority runnable hard task. The timer event set prior to dispatch may therefore correspond to a soft task release, even though there are hard tasks runnable. The scheduling overhead of dealing with soft task releases therefore reduces the slack at each priority level and must be accounted for when handling the event. (Note, this is different from the non-integrated approach where such overheads interfered with hard task execution and were therefore accounted for when the slack at each priority level was calculated, rather than when they actually occurred).

The calculation of slack at priority level *i* also needs to be modified to reflect the reduced interference on hard tasks.

$$S_i(t) = \left(d_i(t) - c_i(t) - \sum_{\forall j \in hp(i)} I_j(t, d_i(t)) \right)_0 \tag{8.22}$$

Where $I_j(t, d_i(t))$ is defined by equation (8.12).

8.5 Summary

The experiments reported in this chapter showed that it is important to take into account the actual overheads of a given scheduling policy. Although all three policies examined (Background, Dual Priority and Dynamic Slack Stealing) have $O(n)$ overheads per scheduling event, the constants of proportionality are very different. Thus the overheads of dynamic Slack Stealing rapidly increase the response time of soft tasks and decrease the feasibility of hard tasks. In our experiments, the performance of the dynamic Slack Stealing algorithm degraded to worst than that of background scheduling, once the cardinality of the task set reached 30. The Dual Priority approach proved to be more effective due to its significantly lower overheads.

We showed that scheduling theory can be tailored to the characteristics of a given real-time kernel implementation and that the worst case response times of tasks can be bounded to within a few percent of the measured values.

Finally, we suggested a means of improving the schedulability of high priority hard tasks by using scheduling mechanisms which avoid priority inversion. This is a particularly important issue if the timing characteristics of soft tasks are not known *a priori*. In this case, an integrated event handling and scheduling policy is essential if hard task timing constraints are to be guaranteed.

Chapter 9

Conclusions and Future Work

9.1 Review of Objectives

In chapter 1, we stated that the aim of the research described in this thesis is to provide the mechanisms and theory required to enable the utility of real-time systems to be improved whilst also ensuring that the *a priori* guarantees made regarding hard time constraints are honoured at run-time.

The central proposition of the thesis is that:

On-line techniques, which exploit knowledge of the run-time state of the system, can be used to integrate adaptive scheduling policies into the framework provided by fixed priority pre-emptive scheduling. Thus combining the benefits of guaranteed predictability with the flexibility necessary to attain improved system utility.

In the previous seven chapters, we charted the development of the theory and mechanisms required to meet this goal. In chapter 2, we concluded that fixed priority pre-emptive scheduling provides a suitable basis for research into the techniques needed to facilitate the dynamic, adaptive and intelligent behaviour required in the next generation of real-time systems. We noted that much of this adaptive behaviour can be integrated into real-time systems through the use of techniques aimed at improving the utility of hard real-time services.

To achieve this integration we developed a three tier approach, comprising:

1. Mechanisms which identify spare capacity.

2. Acceptance tests which provide on-line guarantees for utility-enhancing tasks.

3. Admission policies which arbitrate between competing optional components.

Chapters 3, 4 and 5 recorded the development of theoretical and practical methods of identifying spare capacity. Chapter 6 built upon this work, providing an optimal priority assignment algorithm for aperiodic tasks with firm deadlines and a family of exact and sufficient acceptance tests which enable on-line guarantees to be given to aperiodic tasks with firm or hard deadlines. The scheduling strategy assumed by these acceptance tests fully integrates the scheduling of hard sporadic / periodic, firm aperiodic and soft tasks.

Chapter 7 provided the third tier of our approach; presenting a method of integrating Best-Effort and fixed priority scheduling via an Adaptive Threshold admission policy. This policy is used to arbitrate between optional components, competing for execution time on the basis of their contribution to system utility.

Finally, chapter 8 examined the issues involved in implementing the underlying algorithms (dynamic Slack Stealing and Dual Priority scheduling). This highlighted the need to tailor scheduling theory to the characteristics of a particular kernel implementation and the effects which basic kernel mechanisms have on hard task feasibility; particularly in systems which also contain soft / optional components. Analysis showed that there are inherent problems in using a non-integrated event handling / scheduling approach in this case.

9.2 Meeting the Objectives

In chapter 1, we cited various criteria for judging whether the techniques developed in this thesis are useful in meeting the objectives given above. These criteria were; coverage, performance / overheads and simplicity.

9.2.1 Coverage

The dynamic Slack Stealing and Dual Priority methods of identifying spare capacity are both applicable to task sets with a wide variety of timing characteristics including periodic, sporadic and adaptive release patterns, release jitter and bounded blocking. In addition, and unlike previous techniques, both of these methods are able to take advantage of spare capacity which comes about due to sporadic / adaptive tasks not

arriving at their maximum rate.

The on-line acceptance tests developed in chapter 5 cover a similar spectrum of task timing behaviours. Such coverage is essential if these techniques are to be of practical use in the next generation of hard real-time systems.

9.2.2 Performance / Overheads

The performance of the dynamic Slack Stealing, Dual Priority and Extended Priority Exchange algorithms were compared in terms of their effectiveness at identifying spare capacity, as reflected in their ability to responsive schedule a stream of soft tasks. Our simulation studies showed that both new techniques outperform the EPE algorithm, particularly when much of the spare capacity is due to sporadic / adaptive tasks not arriving at their maximum rate - a likely scenario in next generation real-time systems. Measurements of the overheads and performance of the Slack Stealing and Dual Priority algorithms as implemented in an experimental real-time kernel, showed that although $O(n)$, the overheads of the dynamic Slack Stealing algorithm become prohibitive once the cardinality of the task set reaches 25-30. The overheads of the Dual Priority approach are the same order as those of simple fixed priority pre-emptive scheduling. With hardware co-processor support mitigating the higher scheduling overheads of dynamic Slack Stealing, it provides an optimal method of scheduling soft tasks. When scheduling is carried out by the processor which also executes application tasks, the Dual Priority approach can be recommended as the more practical method.

In chapter 2, two existing on-line acceptance tests were shown to be flawed: each accepted aperiodic tasks for execution when in fact their deadlines would later be missed. Chapter 6, presented families of exact and sufficient on-line acceptance tests based upon the dynamic Slack Stealing and Dual Priority approaches. The exact tests provided a baseline against which the performance of approximate tests could be judged. The sufficient acceptance tests provided a practical and effective method of giving on-line guarantees to optional components with firm deadlines. Using these approximate methods rather than the exact tests has the advantage of significantly lower worst case overheads. Typically, these overheads are approximately 0.5ms, (for 50 tasks on an i486 running at 33MHz) enabling the longest critical section in the

kernel to be bounded.

In chapter 7, we showed that in the case of mixed mandatory and optional task sets, the maximum value which any on-line algorithm can guarantee to obtain through scheduling optional tasks is very small compared to that which a clairvoyant algorithm may obtain (typically 1% for a realistic range of value-densities and execution times). We therefore focussed on policies with good average case behaviour.

The Adaptive Threshold (AVDT) admission policy was shown to provide similar or better performance than that of the Best-Effort policy (given either Dual Priority or Slack Stealing as the underlying scheduling algorithm). Further, the overheads of the AVDT policy are one acceptance test per optional task arrival, whilst in the worst case, the Best-Effort policy results in *m* such acceptance tests, (where *m* is the number of optional tasks previously given an on-line guarantee but which have not as yet completed).

9.2.3 Simplicity

The Dual Priority scheduling algorithm is simple to implement, requiring only additional kernel support for changing the priority of tasks a given time after their release. The burden of complexity is restricted to the off-line analysis, which is in fact no more complex than that for simple fixed priority pre-emptive scheduling.

The implementation requirements of the Adaptive Threshold admission policy are also very simple. The value and expected execution time of an optional task must be determined upon arrival and compared with the running average of system value-density.

The dynamic Slack Stealing algorithm and the approximate on-line acceptance tests are somewhat more complex, making more use of run-time information pertaining to the guaranteed tasks. However, the computation time of each of these algorithms is bounded, enabling them to be implemented within a real-time kernel or scheduling co-processor.

9.3 Contribution

The major contribution of this thesis lies in providing a scheduling strategy which enables system utility to be improved via the timely execution of optional components. These components may be used to improve upon the precision, reliability, frequency and confidence in, the results produced by their mandatory counterparts. The strategy developed supports optional components which have aperiodic arrival patterns, share resources with the mandatory tasks and have soft or firm deadlines.

The dynamic Slack Stealing and Dual Priority scheduling algorithms represent effective methods of identifying spare capacity in real-time systems with complex timing characteristics (e.g. sporadic / adaptive arrival patterns, release jitter, arbitrary deadlines etc). They enable optional tasks with soft deadlines to be scheduled responsively, thus maximising their value to the system. Building upon these methods, the corresponding approximate acceptance tests enable optional components with firm deadlines or indeed alternatives with hard deadlines to be guaranteed at run-time. Finally, the Adaptive Threshold admission policy provides a simple yet effective means of determining which optional components to consider for execution in order to achieve the highest possible system utility.

In addition, the research comprising this thesis has lead to an improved understanding of the techniques used in examining task feasibility and the criteria influencing the design of real-time kernels.

9.4 Further Work

In this section, we outline further areas of research which have opened up due to the progress recorded in this thesis.

9.4.1 Sensitivity Analysis

In the initial design of a real-time system, the application is typically decomposed into a set of tasks and resources which are assigned execution time budgets. During subsequent development, progressively more accurate estimates, measurements and analysis of wcets become available. However, it is often the case that wcets exceed

the initial budgets, leading to an unschedulable system. Similarly, with an operational system, it may be necessary to add enhancements which cause the wcets of certain tasks to increase or new tasks to be added. In both cases, sensitivity analysis allows the system developer to determine if optimisation is needed and to focus effort on those tasks / resources where it will have the most benefit in terms of obtaining a schedulable system.

Despite the wealth of research into feasibility tests (reviewed in chapter 2), these tests often provide little or no indication of the changes in task timing characteristics required to achieve a feasible system, nor any indication of the extent to which the worst case execution times of tasks may be increased without causing deadlines to be missed (in the case of a feasible system). In practice, however, it is useful to know how sensitive system feasibility is to changes in task timing characteristics. Sensitivity analysis provides such information.

The algorithm for determining slack, given in chapter 3, can be used to calculate the amount by which the execution time of each task may be increased in order to obtain a system which is just schedulable. This information provides system designers with the necessary sensitivity data and indicators as to which task(s) to optimise [32]. A prototype sensitivity analysis tool has recently been developed to automate this process.

9.4.2 Feasibility Tests for SRPT Scheduling

The Shortest Remaining Processing Time first (SRPT) scheduling policy has the desirable property that it minimises the mean response time of tasks. Similarly, the Maximum Value Density first (MVD) policy maximises the value accrued per unit time. The feasibility of tasks scheduled under both these algorithms can be determined via feasibility tests derived from the algorithm for calculating slack given in chapter 3.

9.4.3 Dual Priority Scheduling in Distributed Systems

The Dual Priority scheduling policy can be used in distributed real-time systems, for example, to control the order in which messages are broadcast on a Controller Area Network (CAN) bus [104]. In this scenario, the Dual Priority approach can be used to improve the response time of messages with soft deadlines.This approach has a significant advantage over "server" based methods. With a server based method, global information about the remaining capacity of the server must be maintained by each node. This is not possible unless each node receives all soft messages, however, this imposes a large overhead due to handling interrupts for what are essentially unwanted messages. (Note, CAN controllers provide an effective means of filtering out unwanted messages, reducing unnecessary interrupts to a minimum).
Alternatively, nodes may be separately allocated a small slice of available bandwidth, however, this approach is ineffective when the number of soft messages sent by each node varies at run-time: the bandwidth is not available where it is needed.

Tindell has also shown that the Dual Priority approach can be used to prevent the jitter on message arrival, which is inherited by tasks waiting on that message, from impinging upon the feasibility of lower priority tasks [106].

9.4.4 Avoiding Priority Inversion in Real-Time Kernels

The research reported in chapter 8 highlighted the need to avoid priority inversion in the underlying kernel mechanisms used to choose which task to execute and which timer interrupt event to set. For systems containing soft tasks with arbitrary timing behaviours, it is essential that such priority inversion is avoided if hard timing constraints are to be guaranteed. Further work is needed in this area.

9.4.5 Fault Tolerant Real-Time Systems

The results presented in this thesis form a basis for further investigation into adaptive strategies, scheduling policies and feasibility tests for fault-tolerant real-time systems [80]. Such systems can take advantage of diverse implementations under the multiple versions paradigm, use spare capacity to improve reliability / error checking and on-line acceptance tests to guarantee that sufficient execution time is available for re-

trys.

9.5 In Conclusion

The research described in this thesis provides a basis for implementing flexible and adaptive real-time systems. Such systems are expected to do the best they can to obtain results of the highest possible utility, whilst in the worst case, still guaranteeing a minimum acceptable level of performance.

We note that in order for such adaptable real-time systems to be developed, a shift in emphasis is needed in the requirements specification and design phases. The traditional approach of attempting to make the behaviour of a system entirely deterministic will no longer be sufficient. The development of suitable specification methods and acceptance metrics remains one of the greatest challenges in the deployment of next generation real-time systems.

Appendix A: Acceptance Tests

In this appendix, we review the acceptance tests given by Thuel and Lehoczky, giving examples where the tests guarantee hard aperiodic tasks which subsequently miss their deadlines. Further, we identify the faulty assumptions at the root of the problem in each case. First, to aid discussion, we outline the computational model used [83].

A.1 Computational Model

This computational model assumes that there are *n* hard periodic tasks. Each periodic task is assigned a unique priority from 1 to *n*. Further, each task τ_i has a period, worst case execution time, deadline and initiation time denoted by T_i, C_i, D_i and ϕ_i respectively. These parameters are assumed to be known deterministic quantities. It is also assumed that the deadlines of tasks are less than or equal to their periods. At run-time, each periodic task τ_i gives rise to an infinite sequence of invocations. The *j*th invocation ($j = 1,2,3...$) of task τ_i, denoted by τ_{ij}, is released at time $\phi_i + (j-1)T_i$ and requires at most C_i units of execution to be completed by its deadline $d_{ij} = \phi_i + (j-1)T_i + D_i$. An arbitrary set of hard aperiodic tasks is also considered. Each hard aperiodic task τ_F is released at some arbitrary time ϕ_F and has a relative deadline of D_F. In the subsequent analysis, it is also assumed that tasks are independent, do not voluntarily suspend themselves and that any task can be instantly pre-empted by another task of higher priority.

A.2 Static Slack Stealing Algorithm

Before describing the acceptance test, we first introduce the data structures used, by giving an outline of the analysis of the Slack Stealing algorithm [59]. The operation of the Slack Stealing algorithm relies on a table of values, recording the cumulative slack, A_{ij} for each invocation τ_{ij} of a hard periodic task τ_i. These values are used to define a cumulative slack function $A_i(0,t)$, which gives the largest amount of aperiodic processing possible at priority level *i* or higher in the interval $[0,t)$ such that all invocations of task τ_i meet their deadlines. In turn, the cumulative slack function $A_i(0,t)$ is used to calculate the aperiodic processing time available $A_i^*(t)$ at

time t whilst guaranteeing that task τ_i will meet its next deadline. Finally, to ensure that the deadlines of all hard periodic tasks are met, the Slack Stealing algorithm uses the functions $A_i^*(t)$ to determine the actual processing time $A^*(t)$, which it will make available to aperiodic tasks. In [59], the cumulative slack, A_{ij} is defined as:

> *The largest amount of aperiodic processing possible at level i or higher during* $[0, C'_{ij}]$, *such that* $C'_{ij} \leq d_{ij}$, *where* C'_{ij} *refers to the completion time of* τ_{ij}.

Further, the cumulative slack function $A_i(0,t)$ used to ensure the schedulability of all invocations of task τ_i is defined as follows:

$$\forall t \quad : \quad C'_{ij-1} \leq t < C'_{ij,} j \geq 1 \quad : \quad A_i(0,t) = A_{ij} \tag{A.1}$$

It is also asserted that:

> $A_i(0,t)$ *gives the largest amount of aperiodic processing possible at priority level i or higher during* $[0,t]$ *such that all invocations of task* τ_i *meet their deadlines.*

Strictly, the above assertion and the definition of $A_i(0,t)$ are not entirely consistent. This discrepancy can be seen by considering the first invocation of a single periodic task with the following timing attributes: $C_1 = 1$, $T_1 = 10$, $D_1 = 10$, $\phi_1 = 0$, $d_{11} = 10$. Now the latest completion time of the first invocation is $C'_{11} = 10$ and hence the cumulative slack, $A_{11} = 9$. However, for say $t = 2$, the largest amount of aperiodic processing possible in the interval $[0,2)$ (i.e. $A_1(0,2)$) is clearly 2 rather than 9.

Effectively, $A_i(0,t)$ represents the cumulative priority level i aperiodic processing possible in the interval $[0,t)$ *plus* that level i aperiodic processing which is permitted from time t onwards before the current invocation of task τ_i must be resumed to avoid missing its deadline.

The analysis given in [59] continues:

$$A_i^*(t) = A_i(0,t) - A'_i(t) - I_i(t) \tag{A.2}$$

Where $A'_i(t)$ denotes the cumulative aperiodic processing consumed at priority level i or higher during the interval $[0,t)$ and $I_i(t)$ denotes the level i inactivity during

$[0,t)$.

The Slack Stealing algorithm thus enables aperiodic task execution whilst

$$A^*(t) = \min_{(1 \le i \le n)} A_i^*(t) > 0 \qquad \text{(A.3)}$$

Despite the minor discrepancy noted earlier the above inequality is correct for scheduling soft aperiodic tasks. This is because $A^*(t)$ actually represents the aperiodic processing which is permitted from time *t* onwards until one of the hard deadline tasks must be resumed. In the case of scheduling soft aperiodic tasks, this is exactly the quantity required.

A.3 Acceptance Test #1

In the case of scheduling hard aperiodic tasks, the effects of the discrepancy noted previously are far more significant. Consider a hard aperiodic task which is released at time t_a and has a deadline at time t_b. It is necessary to find the aperiodic processing time available in the interval $[t_a, t_b)$ in order to determine if the task can be guaranteed.

Thuel and Lehoczky [83] calculate the aperiodic processing time available as follows: first, the invocation *j* of task τ_i which constrains the cumulative level *i* slack at time t_b is found. This is then used to find the maximum aperiodic processing time available in the interval $[t_a, t_b)$ without violating any deadlines of task τ_i. Note, C'_{ij} is the latest possible completion time for invocation *j* of task τ_i.

$$A_i(t_a, t_b) = \min\left(A_{ij}, A_{ij-1} + (t_b - C'_{ij-1})\right) - A'_1(t_a) - I_i(t_a)$$
$$A_i^*(t_a, t_b) = \min\left((t_b - t_a), A_i(t_a, t_b)\right) \qquad \text{(A.4)}$$

Here, in effect, an attempt was made to correct the minor discrepancy noted previously. The analysis given in [83] states that:

At time C'_{ij-1}, there is a jump in the cumulative slack function $A_i(0,t)$ of size $\Delta s = (A_{ij} - A_{ij-1})$. This means that at time C'_{ij-1}, Δs additional units of aperiodic processing time become available. If $t_b < (C'_{ij-1} + \Delta s)$, some of the additional processing time Δs cannot be used until after the aperiodic deadline. Thus the amount of cumulative aperiodic processing available in $[0, t_b)$, as constrained by task τ_i is given by:
$\min\left(A_{ij}, A_{ij-1} + (t_b - C'_{ij-1})\right)$

However, this simple correction is not sufficient, as it does not account for the fact that the Δs additional units may not be available until after time t_b due to the execution requirements of higher priority periodic tasks. This point is illustrated by the example given in figure A.1.

The analysis of Thuel and Lehoczky proceeds from the above equations by asserting that:

As in the soft aperiodic case, once the maximum cumulative aperiodic processing times are computed for each task, the aperiodic processing time available at the highest priority level is the minimum of these values as given by: $A^*(t_a, t_b) = \min_{(1 \le i \le n)} A_i^*(t_a, t_b)$

Unfortunately, this assertion is not valid. In the soft aperiodic case, the interval over which slack is stolen extends from time t to time $t + A^*(t)$. During this interval, no hard tasks execute and therefore none can complete. Hence, the slack available in the interval is limited by a single periodic task. Equation (A.3) therefore gives an accurate value for the available slack. However, in the hard aperiodic case, the interval $[t_a, t_b)$ may constitute many sub-intervals of aperiodic execution followed by hard periodic task execution. In each sub-interval, the slack available may be limited by a different periodic task. Simply taking the minimum slack available over the entire interval is not sufficient: equation (A.4) can give much larger values than the actual slack available. Again, this point is illustrated by the example in figure A.1. In this example, the acceptance test outlined above guarantees a hard aperiodic task. However, the aperiodic processing time made available by the Slack Stealing algorithm between the release and the deadline of this task is substantially less than its execution requirement, causing its deadline to be missed.

The example is based on the task set detailed below.

Hard Periodic tasks				
Priority	Period	Offset	Deadline	WCET
1	36	13	5	4
2	36	7	6	5
3	36	0	7	6

For this task set, the cumulative slack functions $A_i(0,t)$ are as follows:

$$A_1(0,t) = \begin{cases} 14, & 0 \le t \le 18 \\ 46, & 18 < t \le 54 \end{cases}$$

$$A_2(0,t) = \begin{cases} 8, & 0 \le t \le 13 \\ 35, & 13 < t \le 49 \end{cases}$$

$$A_3(0,t) = \begin{cases} 1, & 0 \le t \le 7 \\ 22, & 7 < t \le 43 \end{cases}$$

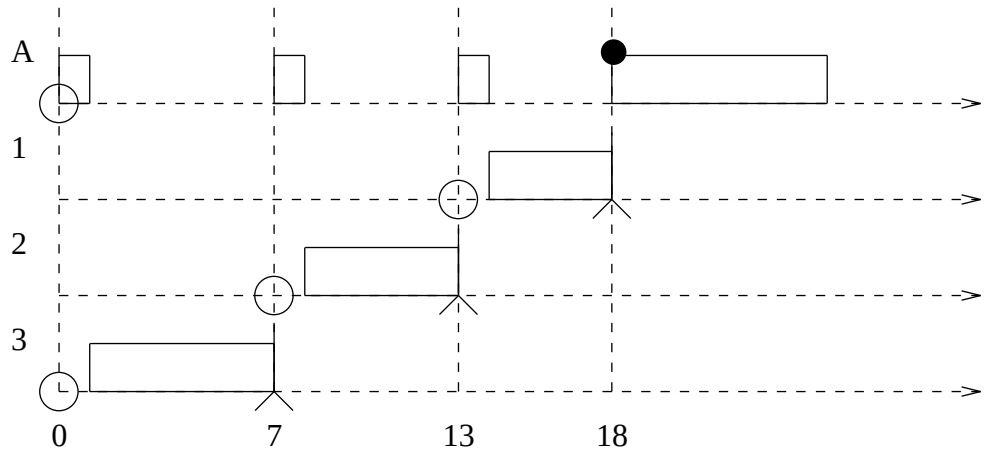
In addition to the hard periodic tasks given above, we also consider a single aperiodic task which is released at time $t = 0$ with a deadline of 18 and a processing requirement of 10 units.

According to the analysis given by Thuel and Lehoczky, the maximum aperiodic processing time in the interval $[0,18)$ is calculated as follows. Note, the aperiodic processing time and idle time prior to the interval are zero.

$$A_1(0,18) = \min(46, 14+(18-18)) = 14$$
$$A_2(0,18) = \min(35, 8+(18-13)) = 13$$
$$A_3(0,18) = \min(22, 1+(18-7)) = 12$$

The maximum aperiodic processing permitted in the interval $[0,18)$

$$A^*(0,18) = \min(14,13,12) = 12$$



Hard aperiodic task execution is shown on timeline A. Similarly, the execution of periodic tasks is illustrated on timelines 1, 2 and 3. Note, task execution is as scheduled by the Slack Stealing algorithm.

Figure A.1: Insufficient Acceptance Test #1.

Hence the aperiodic task is given a guarantee. The execution of the tasks, as scheduled by the Slack Stealing algorithm, is shown in figure A.1. By inspection, it is clear that only 3 units of aperiodic processing may be permitted in the interval without causing a periodic task to miss its hard deadline. Indeed, the Slack stealing algorithm affords the aperiodic task 3 units of execution time prior to its deadline being missed at time $t = 18$.

A.4 Acceptance Test #2

The revised acceptance test presented by Thuel at the 1993 Real-Time Systems Symposium [84], uses a different approach to calculate the aperiodic processing time available in a given interval.

Off-line, a list of triplets of the form $\{t_h, i', s(t_h)\}$ is defined. Where t_h is the h^{th} latest completion time of a periodic task, i' is the priority of that task and $s(t_h)$ is the amount by which the slack at priority level i' increases at that completion of task $\tau_{i'}$. The list is ordered by increasing values of t_h.

At run-time, to determine the aperiodic processing time available in a given interval, $[t_a, t_b)$, requires iteration over all the triplets with latest completion times in that interval. A set of *n* scratch variables $S_i(t)$ are used to store the slack available at priority level *i* at any arbitrary time *t*. A counter $S^*(t)$ is used to store the aperiodic processing time available from time *t* onwards at the highest priority level.

$$S^*(t) = \min_{\forall i} S_i(t) \tag{A.5}$$

Further, a cumulative slack counter *Q* is used to total the available aperiodic processing time found at each iteration.

Assuming a hard aperiodic task arrives at time t_a, with a deadline at t_b, the acceptance test proceeds as follows: first, the scratch variables $S_i(t_a)$ are set to $A_i^*(t_a)$ and the aperiodic processing time available immediately ($S^*(t_a)$) is found.

$$Q = \min(S^*(t_a), t_b - t_a) \tag{A.6}$$

Iteration then proceeds over all the triplets with completion times in the interval. On each iteration, the scratch variables are updated according to the data stored in the triplet. (Note, on the first iteration, $t_h = t_a$).

$$S_k(t_{h+1}) = \begin{cases} S_k(t_h) - S^*(t_h) - C^*, & \text{if} k < i' \\ S_k(t_h) - S^*(t_h) + s(t_{h+1}), & \text{if} k = i' \\ S_k(t_h) - S^*(t_h), & \text{if} k > i' \end{cases} \tag{A.7}$$

Where C^* is the execution time remaining at time t_a for the invocation which completes at time t_{h+1}. A new value of $S^*(t_{h+1})$ is determined and *Q* updated accordingly.

$$Q = Q + \min(S^*(t_{h+1}), t_b - t_{h+1}) \tag{A.8}$$

Once all the triplets with completion times in the interval have been processed, *Q* gives the available aperiodic processing time.

Unfortunately, there is a subtle problem with this approach: the latest completion times and indeed the order of task completions may change at run-time. This is illustrated by the following example based on the task set defined below.

Hard periodic tasks				
Priority	Period	Deadline	Offset	WCET
1	8	1	7	1
2	6	6	4	1
3	7	7	0	1

The slack functions for these tasks are:

$A_{11} = 7,\ A_{12} = 14$
$A_{21} = 8,\ A_{22} = 12$
$A_{31} = 5,\ A_{32} = 9$

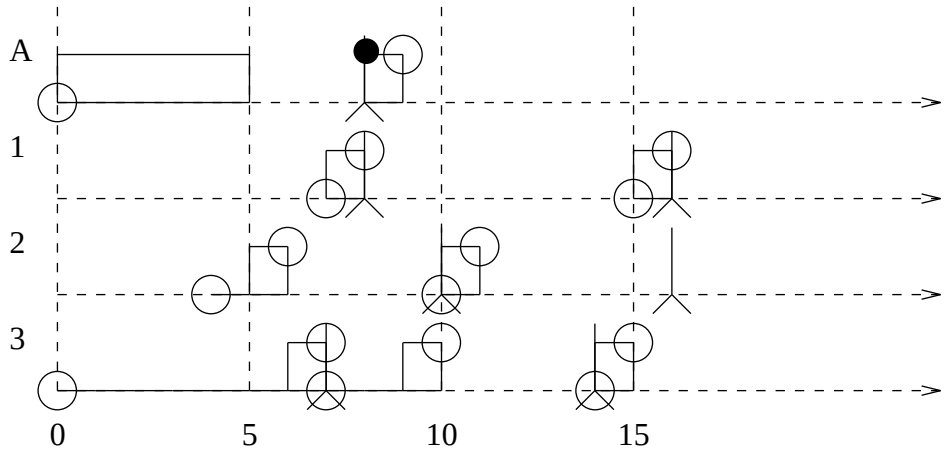
and the latest completion times are:

$C'_{11} = 8,\ C'_{12} = 16$
$C'_{21} = 10,\ C'_{22} = 15$
$C'_{31} = 7,\ C'_{32} = 14$

At time 0, we assume that an aperiodic task arrives and requests 6 units of computation by its deadline at time 8. Using the approach given by Thuel and Lehoczky [84], the completion time triplets {latest completion time, priority, slack increment} of interest are {7,3,4}, {8,1,7} and {10,2,5}. Thus the aperiodic processing time available in the interval [0,8), is calculated as follows:

$S^*(0) = 5$
$S_1(7) = 7 - 5 - 1 = 1$
$S_2(7) = 8 - 5 - 1 = 2$
$S_3(7) = 5 - 5 + 4 = 4$
$S^*(7) = 1$

Hence $Q = 6$ and the aperiodic task is given a guarantee. However, τ_{21} (the first invocation of task 2) is forced to complete at time 6 in order for τ_{31} to meet its deadline at time 7. Hence the slack available in the interval is in fact only 5 units, as illustrated by figure A.2. The aperiodic task therefore misses its deadline at time 8.



Hard aperiodic task execution is shown on timeline A. Similarly, the execution of periodic tasks is illustrated on timelines 1, 2 and 3. Note, task execution is as scheduled by the Slack Stealing algorithm.

Figure A.2: Insufficient Acceptance Test #2.

It is interesting to also consider the case where τ_{31} executes and completes at time 1, whereupon an aperiodic task arrives and requests 6 units of execution time by its deadline at time 8. Now, the Slack Stealing algorithm can delay the execution of τ_{21} until time 9, without causing any deadlines to be missed, thus making 6 units of computation available to the aperiodic task. In this case, the order of task completion is different, τ_{31} completes before τ_{21}.

The above example highlights the problem with using latest completion time triplets to determine the slack available in an interval: the latest completion time of a task may change at run-time. To formulate an exact acceptance test using this approach would require run-time maintenance of the latest completion times, associated triplets and their position in the list.

Appendix B: Calculating Slack for Tasks with $D > T$

In this appendix, we describe an iterative computation which determines a monotonically increasing lower bound on the slack at priority level *i*, given that $D_i > T_i$. For task sets with utilisation < 100%, this lower bound is guaranteed to converge to the exact slack available. Further, we provide a criteria for determining convergence using a monotonically decreasing upper bound.

B.1 Analysis

In the following analysis, we assume that all invocations of task τ_i have the same base priority *i* and that earlier releases take precedence over later releases. That is invocation q cannot commence execution until invocation $q-1$ has completed. $S_i^T(t,q)$ is notation for the slack on the qth invocation of task τ_i, which is present prior to the release of the $(q+1)$th invocation. Further, $x_i^*(t,q)$ denotes the earliest possible time at which the qth invocation of task τ_i may be released, ($x_i^*(t,q) = (d_i(t) - D_i) + qT_i$, for a periodic task). Hence $S_i^T(t,q)$ corresponds to the level *i* idle time in the interval $[t, t + x_i^*(t,q+1))$. Similarly, $S_i^D(t,q)$ is the slack found on the qth invocation of task τ_i prior to its deadline (denoted by $d_i(t,q)$, where $d_i(t,q) = d_i(t) + qT_i$). Hence $S_i^D(t,q)$ is equivalent to the level *i* idle time in the interval $[t, t + d_i(t,q))$ *excluding* computation corresponding to invocations q+1, q+2, q+3... of task τ_i.

We now derive relationships between the slack present on different invocations of task τ_i. These relationships are then used to provide upper and lower bounds on the exact slack available.

Consider the two intervals $[t, t + x_i^*(t,q+1))$ and $[t, t + x_i^*(t,q+2))$, over which $S_i^T(t,q)$ and $S_i^T(t,q+1)$ are calculated. The level *i* idle time in the interval $[t, t + x_i^*(t,q+1))$ is the same when computing both $S_i^T(t,q)$ and $S_i^T(t,q+1)$. As further idle time may be found in the interval $[t + x_i^*(t,q+1), t + x_i^*(t,q+2))$, we have:

$$\forall q = 0,1,2,3... \qquad S_i^T(t,q+1) \geq S_i^T(t,q) \tag{B.1}$$

Similarly, assuming that $f(q)$ is the largest integer such that $d_i(t,q) \geq x_i^*(t,(q+1)+f(q))$. (In the case of a periodic task, $f(q)$ is a constant: $f(q) = \left\lfloor \frac{D_i}{T_i} \right\rfloor - 1$). The interval over which $S_i^D(t,q)$ is calculated includes the interval over which $S_i^T(t,q+f(q))$ is calculated. Further, in calculating $S_i^D(t,q)$, only computation due to invocations 0,1..q of τ_i are included, whilst, invocations $q+1$, $q+2$... $q+f(q)$ are are also included when $S_i^T(t,q+f(q))$ is calculated. Hence:

$$\forall q = 0,1,2,3... \qquad S_i^D(t,q) \geq S_i^T(t,q+f(q)) \tag{B.2}$$

(Note, in general a similar relationship between $S_i^D(t,q+1)$ and $S_i^D(t,q)$ cannot be established. This is because, the idle time in the interval $[t,t+d_i(t,q))$ differs between calculations of $S_i^D(t,q+1)$ and $S_i^D(t,q)$. Computation time due to invocation q+1 of task τ_i is included in the former but not in the latter).

The exact slack is given by:

$$\min_{\forall q=0,1,2,3...} \left(S_i^D(t,q) \right) \tag{B.3}$$

Using inequalities (B.1) and (B.2), we may construct a series of increasing lower bounds on the exact priority i slack. As

$$S_i^T(t,k+f(k)) \leq \min_{\forall q=k,k+1,k+2...} \left(S_i^D(t,q) \right) \tag{B.4}$$

The kth lower bound is given by:

$$\min \left(S_i^T(t,k+f(k)), \min_{\forall q=0,1,2...k-1} \left(S_i^D(t,q) \right) \right) \tag{B.5}$$

Iteration proceeds by calculating values of $S_i^T(t,q)$ and $S_i^D(t,q)$ in the order required to form the lower bounds given by k = 0,1,2,3.... For example, with $f(q)=1$, the order is $S_i^T(t,1)$, $S_i^D(t,0)$, $S_i^T(t,2)$, $S_i^D(t,1)$, $S_i^T(t,3)$, forming lower bounds $S_i^T(t,1)$, $\min(S_i^T(t,2), S_i^D(t,0))$, $\min(S_i^T(t,3), S_i^D(t,0), S_i^D(t,1))$. At any point, iteration may

be terminated and the largest lower bound found so far returned as the slack available at priority level *i*. Further, we may define the *k*th upper bound on the level *i* slack for *k* = 1,2,3... as:

$$\min_{\forall q=0,1,2,\ldots k-1}\left(S_i^D(t,q)\right) \tag{B.6}$$

When the upper and lower bounds converge, the exact level *i* slack has been determined. We note that convergence is guaranteed for task sets with utilisation < 100%. In pathological cases, as utilisation tends towards 100%, convergence may require an arbitrarily large number of invocations to be examined. However, the invocation of task τ_i with the least slack must be one of the first h_i examined, (where h_i is the number of invocations of task τ_i in LCM_i, the least common multiple of the subset of tasks with priority *i* or higher). This can be seen by considering the slack on the *q*th and $(q+h_i)$th invocations of task τ_i. All the level *i* idle time in LCM_i lies between these two invocations, hence the slack on the $(q+h_i)$th invocation exceeds that on the *q*th by the level *i* idle time in the level *i* LCM. Hence the invocation with the least slack must be one of the first h_i.

In practice, the upper and lower bounds generally converge rapidly. Further, complexity may be reduced by limiting *k* (and therefore *q*) to some constant at the expense of providing a lower bound rather than the exact slack available.

B.1.1 Calculating slack on the *q*th invocation

In calculating the available slack, we make use of the following information, typically derived from data stored in the task control block):

$c_i(t)$ - The remaining execution time budget for the current (i.e. oldest ready) invocation of τ_i.

$e_i(t)$ - The number of released but as yet uncompleted invocations of τ_i at time *t*.

$x_i(t)$ - The earliest next release of an invocation of τ_i (measured relative to t).

Further, we denote computation due to task τ_i, but excluding invocations $q+1$, $q+2$..., which is ready in the interval $[t, t+w_i^m(t))$ by $c_i(t,q,w_i^m(t))$.

$c_i(t,q,w_i^m(t)) =$

$$c_i(t) + \min\left[\left(e_i(t)-1\right)_o, q\right] C_i + \min\left[\left(q-e_i(t)+1\right)_0, \left\lceil \frac{w_i^m(t)-x_i(t)}{T_i} \right\rceil_0\right] C_i \qquad \text{(B.7)}$$

The following algorithm calculates the values of $S_i^D(t,q)$ and $S_i^T(t,q)$ for invocation q of task τ_i. Note, we use *interval* to denote the length of the interval over which slack is determined. Thus when calculating $S_i^T(t,q)$ and $S_i^D(t,q)$, *interval* is set to $x_i^*(t,q+1)$ and $d_i(t,q)$ respectively.

Algorithm for calculating slack on the *q*th invocation of task τ_i.

$S_i(t,q) = 0$
$w_i^{m+1}(t) = 0$
do while $w_i^{m+1}(t) \le interval$
  $w_i^m(t) = w_i^{m+1}(t)$
  $w_i^{m+1}(t) = S_i(t,q) + c_i(t,q,w_i^m(t)) +$

$$\sum_{\forall j \in hp(i)} \left[c_j(t) + \left(\left(e_j(t)-1\right)_0 + \left\lceil \frac{w_i^m(t)-x_j(t)}{T_j} \right\rceil_0 \right) C_j \right]$$

  if $w_i^m(t) = w_i^{m+1}(t)$
  then $gap = v_i(t,q,w_i^m(t))$
    $S_i(t,q) = S_i(t,q) + gap$
    $w_i^{m+1}(t) = w_i^{m+1}(t) + gap + \varepsilon$
  end if
enddo
return $S_i(t,q)$ (B.8)

The function $v_i(t,q,w_i(t))$ returns the length of the level i idle period from time $t+w_i(t)$ excluding invocations q+1,q+2, q+3 etc of task τ_i:

$$v_i(t,q,w_i(t)) = \min \left(\begin{array}{l} \left(interval - w_i(t) \right)_0, \\ \min\limits_{\forall j \in hp(i)} \left(\left\lceil \frac{w_i(t)-x_j(t)}{T_j} \right\rceil_0 T_j + x_j(t) - w_i(t) \right), \\ n_i(t,q,w_i(t)) \end{array} \right) \quad \text{(B.9)}$$

The function $n_i(t,q,w_i(t))$ returns the next release of an invocation of task τ_i up to and including the qth invocation.

$$n_i(t,q,w_i(t)) =$$

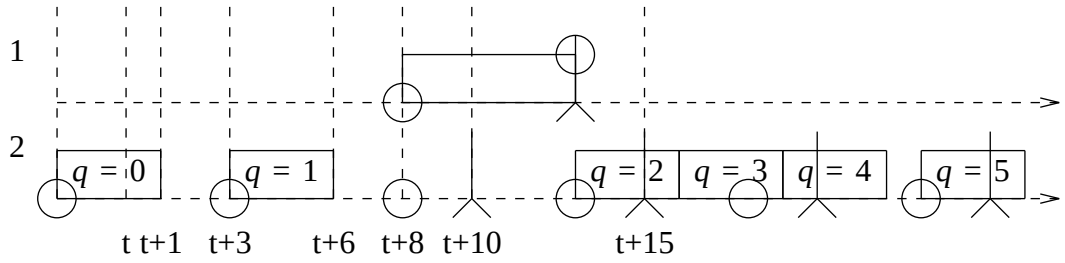
$$\begin{cases} \left\lceil \frac{w_i(t)-x_i(t)}{T_i} \right\rceil_0 T_i + x_i(t) - w_i(t) & \textbf{if} \quad \left\lceil \frac{w_i(t)-x_i(t)}{T_i} \right\rceil_0 \le \left(q - e_i(t) + 1 \right)_0 \\ \infty & \textbf{otherwise} \end{cases} \quad \text{(B.10)}$$

We note that iteration is required for tasks with arbitrary deadlines. For tasks with $D \le T$, convergence is guaranteed with k = 1, as $S_i^T(t,-1) \le S_i^D(t,0) \le S_i^T(t,0)$. In practice of course, only $S_i^D(t,0)$ need be calculated for tasks with $D \le T$.

B.1.2 Example slack calculation

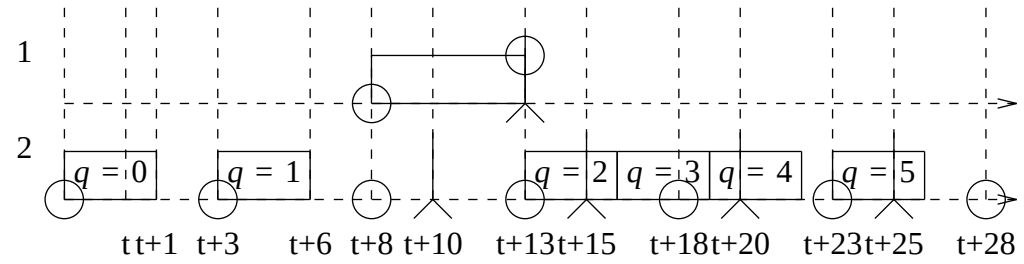
The example given in figure B.1 illustrates the calculation of slack on a single invocation of a task with $D > T$. The timing attributes of the tasks used in the example are given in the table below. Using this task set, figure B.2 illustrates how converging upper and lower bounds on the available slack can be calculated.

Hard tasks				
Priority	Period	Deadline	offset	wcet
1	100	5	10	5
2	5	12	0	3



Calculation of the slack on invocation $q=1$ of task τ_2 at time t
At time t, we have outstanding computation: $c_1(t)=0$ and $c_2(t)=1$ ($e_2(t)=1$). The next releases of tasks τ_1 and τ_2 (relative to t) are at: $x_1(t)=8$ and $x_2(t)=3$. The end of the interval under examination is at $t+d_2(t,1)=t+15$.
Applying the algorithm for calculating slack:
Iteration 1: $w_2^1(t)=c_2(t)=1$
Iteration 2: $w_2^2(t)=1$. Identifying the end of a level 2 busy period at $t+1$. $v_2(t,1,2)=x_2(t)-w_2^2(t)=2$, corresponding to the idle period $[t+1,t+3)$. Converting this idle period to slack, we have $S_2(t,1)=2$ and $w_2^2(t)=3+\varepsilon$.
Iteration 3: $w_2^3(t)=S_2(t,1)+c_2(t)+C_2=6$
Iteration 4: $w_2^4(t)=6$, again identifying the end of a busy period at $t+6$. $v_2(t,1,6)=x_1(t)-w_2^4(t)=2$, corresponding to an idle period of length 2 prior to the release of task τ_1 at $t+8$. $S_2(t,1)=4$ and $w_2^4(t)=8+\varepsilon$
Iteration 5: $w_2^5(t)=S_2(t,1)+c_2(t)+C_2+C_1=13$.
Note, computation time due to the q = 2 invocation of τ_2 has not been included as it is effectively of lower priority than the q = 1 invocation.
Iteration 6: $w_2^6(t)=13$, indicating the end of a busy period at $t+13$, (excluding invocations of task τ_2 with q > 1).
$v_2(t,1,13)=d_2(t,1)-w_2^6(t)=2$ corresponding to the 2 idle ticks prior to the end of the interval at $t+15$. $S_2(t,1)=6$ and $w_2^6(t)=15+\varepsilon>d_2(t,1)$, hence iteration is complete.
The slack on invocation q = 1 of task τ_2 at time t is 6.

Figure B.1: Calculation of Slack on a Single Invocation of a Task with $D > T$.



Calculation of upper and lower bounds on the priority 2 slack at time t

Default: The lower bound on priority 2 slack is 0 and the upper bound is ∞.
Iteration k=0: The lower bound is given by $S_2^T(t,1)$, i.e. the slack present in the interval $[t,t+8)$. $S_2^T(t,1)=4$ comprising the idle periods $[t+1,t+3)$ and $[t+6,t+8)$.
Hence the lower bound is 4 and the upper bound is ∞.
Iteration k=1: The lower bound is given by $\min(S_2^T(t,2),S_2^D(t,0))$.
$S_2^T(t,2)$ is the slack in the interval $[t,t+13)$ = 4.
$S_2^D(t,0)$ is the slack in the interval $[t,t+10)$ *ignoring* computation due to invocations of task τ_2 with $q > 0$. $S_2^D(t,0)$ = 7, comprising the idle period $[t+1,t+8)$.
Hence the lower bound is 4 and the upper bound is 7.
Iteration k=2: The lower bound is given by $\min(S_2^T(t,3),S_2^D(t,0),S_2^D(t,1))$.
$S_2^T(t,3)$ is the slack in the interval $[t,t+18)$ = 4.
$S_2^D(t,1)$ is the slack in the interval $[t,t+15)$ *ignoring* computation due to invocations of task τ_2 with $q > 1$. $S_2^D(t,1)$ = 6, comprising the idle periods $[t+1,t+3)$, $[t+6,t+8)$ and $[t+13,t+15)$.
Hence the lower bound is 4 and the upper bound is 6.
Iteration k=3: The lower bound is given by $\min(S_2^T(t,4),S_2^D(t,0),S_2^D(t,1),$ $S_2^D(t,2))$, $S_2^T(t,4)$ is the slack in the interval $[t,t+23)$ = 5.
$S_2^D(t,2)$ is the slack in the interval $[t,t+20)$, *ignoring* computation due to invocations of task τ_2 with $q > 2$. $S_2^D(t,2)$ = 8, comprising the idle periods $[t+1,t+3)$, $[t+6,t+8)$ and $[t+16,t+20)$.
Hence the lower bound is 5 and the upper bound is 6.
Iteration k=4: The lower bound is $\min(S_2^T(t,5),S_2^D(t,0),S_2^D(t,1),S_2^D(t,2),$ $S_2^D(t,3))$. $S_2^T(t,5)$ is the slack in the interval $[t,t+28)$ = 7.
$S_2^D(t,3)$ is the slack in the interval $[t,t+25)$, *ignoring* computation due to invocations of task τ_2 with $q > 3$. $S_2^D(t,3)$ = 10, comprising the idle periods $[t+1,t+3)$, $[t+6,t+8)$ and $[t+19,t+25)$.
Hence both the lower and upper bounds are 6.
As the upper and lower bounds have converged, iteration is terminated.
The exact slack at priority level 2 at time t is 6.

Figure B.2: Iterative Computation of Slack for Tasks with Arbitrary Deadlines.

The complexity of calculating the exact slack for a task with $D > T$ is $O(kmn^{2})$. Where, k is the number of iterations before convergence, m is the number of iterations of the 'do while' loop and n is the number of higher priority tasks.

Appendix C: Avoiding Priority Inversion in the Kernel

C.1 Integrated Event Handling and Scheduling

The initial DRTEE kernel implementation has the drawback that the release of a low priority task impinges upon the schedulability of those with higher priorities. This problem can be avoided and the priority inversion caused by the kernel substantially reduced using the mechanisms described below. Note, the pseudo code given applies only to Background Scheduling with no semaphore access, however a similar approach can be applied when semaphores are locked according to the Ceiling Semaphore Protocol or when hard task priorities are promoted due to the operation of the Dual Priority Scheduling algorithm.

The integrated scheduling model uses a `run_table` which is implemented as an array of pointers to Task Control Blocks (TCB), indexed by priority. If a task is runnable at a given priority level, then the entry in the `run_table` corresponding to that priority level is a pointer to the task's control block. Array entries corresponding to priority levels at which no task is currently runnable are set to NULL. In addition, we use a `task_table` which is an array indexed by base priority. Each slot in this array contains a pointer to the TCB of the task with the corresponding base priority. This array is static, the slot contents are unchanged at run-time.

As there can only be at most one timer event outstanding per task, we assume that this event is stored in the task's control block.

On entering the scheduler, let `cur_task` be a pointer to the TCB of the task which was executing, `cur_pri` its active priority and `cur_event` a pointer to the next timer event of priority higher than or equal to `cur_pri.` There are two different ways in which the scheduler can be entered; via the task completion call gate or via the timer interrupt handler.

Task Completion: At the completion of the current task, there can be no higher priority tasks runnable. Hence the scheduler need only examine priority levels lower than `cur_pri` to determine which is the highest priority runnable task. Similarly, the scheduler need only inspect these same priority levels to determine if there is an

earlier event (i.e. earlier than `cur_event`) which must be set. The pseudo code for determining which task to dispatch and which event to set is given below.

Pseudo Code for Task Completion

```
run_table[cur_pri] = NULL:  /* task completed */
old_pri = cur_pri;
exit = FALSE
/* find the highest priority runnable task */
for(pri = cur_pri; exit != TRUE; pri = next_lower(pri)) {
       task = run_table[pri];
       if(task!= NULL) {
              cur_task = task;
              cur_pri = pri;
              exit = TRUE;
       }
}
/* find the next event to set */
for(pri = old_pri; higher_or_equal(pri, cur_pri);
                                    next_lower(pri)){
       task = task_table[pri];
        if(task!= NULL &&
           task->next_event->time < cur_event->time) {
              cur_event = task->next_event;
       }
}
set_timer_event(cur_event);
dispatch(cur_task);
```

Note, the first `for` loop always terminates as the idle task has the lowest priority and is always runnable. (The function `next_lower(pri)` returns the priority level immediately below `pri.` The function `higher_or_equal(pri,cur_pri)` returns `TRUE` whilst `pri` is an equal or higher priority level than `cur_pri`).

Timer Event: At the expiry of a timer event, of type *RELEASE*, the scheduler needs to make the task corresponding to that event runnable at the appropriate priority level and set up the next timer event of an equal or higher priority.

Pseudo Code for Timer Expiry

```
task = cur_event->task
run_table[task->base_priority] = task;
cur_pri = task->base_priority;
cur_task = task;
set_next_event(task);
cur_event = cur_task->next_event;
exit = FALSE
/* find the next event to set */
for(pri = high_pri; higher(pri, cur_pri); next_lower(pri)) {
       task = task_table[pri];
        if(task!= NULL &&
           task->next_event->time < cur_event->time) {
              cur_event = task->next_event;
        }
}
set_timer_event(cur_event);
dispatch(cur_task);
```

(Note, `high_pri` is the highest base priority of a task, the function `higher(pri,cur_pri)` returns `TRUE` while `pri` is a higher priority level than `cur_pri` and `set_next_event(task)` sets up the task's next timer event).

The above pseudo-code ensures that whenever, the scheduler is entered, the only processing which takes place is on behalf of tasks with a priority higher than or equal to that of the highest priority runnable task. As the TCB's are examined in priority order, this continues to be the case, as tasks are made runnable.

By comparison with the more conventional run-queue / delay-queue model, the above approach has the advantage that the execution of high priority tasks is not interrupted by the release of lower priority tasks. Such tasks would not in any case be executed until the higher priority task has completed.

A scheduler using the traditional run-queue / delay-queue model typically has worst case overheads of $O(i)$, where i is the priority of the task, at each task release, due to insertion of the released task into the run-queue and $O(n)$ overheads at each task completion, due to inserting the completed task into the delay-queue. Further, the overheads of releasing low priority tasks impinges upon the schedulability of higher priority tasks. This is particularly undesirable when the lower priority tasks are optional / soft tasks whose timing characteristics may be unknown *a priori*.

By comparison, the integrated scheduling / event handling approach advocated above also has $O(i)$ overheads at each task release, due to finding the next timer event to set, and $O(j)$ overheads at task completion, where j is the priority of the next runnable task. However, this approach ensures that the release of lower priority tasks is postponed until any higher priority tasks have finished executing. This removes the priority inversion and improves the feasibility of high priority hard tasks. Stated otherwise, the interference which high priority tasks are subject to is significantly reduced under the integrated approach.

References

1. *Intel i486 Microprocessor*, April 1989.

2. Ada 9X Mapping/Revision Team, Intermetrics, ‘‘Ada 9X Reference Manual, Draft Version 4.0’’, Ada 9X Project Report (September 1993).

3. R. Arnold, F. Mueller, D. B. Whalley and M. Harmon, ‘‘Bounding Worst-Case Instruction Cache Performance’’, *Proceedings 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, pp. 172-181 (7-9 December 1994).

4. N. Audsley, A. Burns, R. Davis, K. Tindell and A.J. Wellings, ‘‘Fixed Priority Pre-emptive Scheduling: An Historical Perspective’’, *Real_Time Systems* **8**(3), pp. 173-198 (1995).

5. N. C. Audsley, ‘‘Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times’’, YCS 164, Dept. Computer Science, University of York (December 1991).

6. N. C. Audsley, ‘‘Outline of DRTEE Kernel Functionality’’, Technical Note, Department of Computer Science, University of York (January 1993).

7. N. C. Audsley, ‘‘Flexible Scheduling in Hard Real-Time Systems’’, D.Phil. Thesis, Department of Computer Science, University of York, UK (August 1993).

8. N. C. Audsley, A. Burns, R. I. Davis and A. J. Wellings, ‘‘Appropriate Mechanisms for the Support of Optional Processing in Hard Real-Time Systems’’, *Proceedings IEEE Real-Time Operating Systems and Software*, Seattle, USA, pp. 23-27 (May 18-19 1994).

9. N. C. Audsley, A. Burns, R. I. Davis and A. J. Wellings, ‘‘Integrating Best-Effort and Fixed Priority Scheduling’’, *Proceedings IFIP Workshop on Real-Time Programming* , Lake Konstanz, Germany (June 1994).

10. N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell and A. J. Wellings, ‘‘Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling’’, *Software Engineering Journal* **8**(5), pp. 284-292 (September 1993).

11. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘Hard Real-Time Scheduling: The Deadline Monotonic Approach’’, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA (15-17 May 1991).

12. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘Incorporating Unbounded Algorithms Into Predictable Real-Time Systems’’, YCS 171, Department of Computer Science, University of York (March 1992).

13. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ''Incorporating Unbounded Algorithms Into Predictable Real-Time Systems'', *Computer Systems Science and Engineering* **8**(3), pp. 80-89 (April 1993).

14. N. C. Audsley, A. Burns and A. J. Wellings, ''Deadline Monotonic Scheduling Theory and Application'', *Control Engineering Practice* **1**(1), pp. 71-78 (1993).

15. N. C. Audsley, R. I. Davis and A. Burns, ''Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems'', *Proceedings 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, pp. 12-21 (7-9 December 1994).

16. N. C. Audsley and M. F. Richardson, ''DRTEE Kernel Rationale and Interface Specification'', Technical Note, Department of Computer Science, University of York (Sept. 1992).

17. C. M. Bailey, E. Fyfe, T. Vardanega and A. J. Wellings, ''The Use of Preemptive Priority-Based Scheduling in Space Applications'', *Proceedings Real Time Systems Symposium, IEEE Computer Society*, North Carolina, pp. 253-257 (December 1993).

18. S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha and F. Wang, ''On the Competitiveness of Online Real-Time Task Scheduling'', *Real-Time Systems* **4**(2), pp. 124-144 (1992).

19. A. Burns, ''The Application of Utility Measurements in Real-Time Systems Design'', RTRG/91/105, Real-Time Research Group, Department of Computer Science, University of York.

20. A. Burns, A. M. Lister and A. J. Wellings, ''A Review of Ada Tasking'', in *Leture Notes in Computer Science vol. 262*, Springer-Verlag (1987).

21. A. Burns and J. A. McDermid, ''Real-Time Safety-Critical Systems: Analysis and Synthesis'', *Software Engineering Journal*, pp. 267-281 (November 1994).

22. A. Burns, K. W. Tindell and. A. J. Wellings, ''Fixed Priority Scheduling with Deadlines Prior to Completion'', *Proceedings of Sixth Euromicro Workshop on Real-Time Systems*, pp. 138-142, Department of Computer Science, University of York, UK (June 1994).

23. A. Burns and A. J. Wellings, ''Criticality and Utility in the Next Generation'', *Real-Time Systems* **3**(4), pp. 351-354 (1991).

24. A. Burns and A. J. Wellings, ''Safety Kernels and the Ada Programming Language'', *Proceedings 1992 Ada UK Conference (13-15 October 1992)*, London, UK, pp. 56-70, IOS Press (1992).

25. A. Burns and A.J. Wellings, ''Dual Priority Assignment: A Practical Method for Increasing Processor Utilization'', pp. 48-55 in *Proceedings of 5th Euromicro Workshop on Real-Time Systems, Oulu*, IEEE Computer Soc. Press

(1993).

26. A. Burns, A. J. Wellings and A. D. Hutcheon, ‘‘The Impact of an Ada Run-time System’s Performance Characteristics on Scheduling Models’’, pp. 240-248 in *Ada sans frontieres Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science 688*, Springer-Verlag (1993).

27. G. C. Buttazzo and J. A. Stankovic, ‘‘RED: Robust Earliest Deadline Scheduling’’, CMPSCI Technical Report 93-25, Dept. of Computer Science, University of Massachusetts at Amherst (March 1993).

28. S. Chen, J.A. Stankovic, J.F. Kurose and D. Towsley, ‘‘Performance evaluation of two new disk scheduling algorithms for real-time systems’’, *Real Time Systems* **3**(3), pp. 306-336 (1991).

29. H. Chetto and M. Chetto, ‘‘Some Results of the Earliest Deadline Scheduling Algorithm’’, *IEEE Transactions Software Engineering* **15**(10), pp. 1261-1269 (October 1989).

30. R. I. Davis, ‘‘On Improving the Utility of Hard Real-Time Services in Fixed Priority Pre-emptive Systems’’, MPhil/DPhil Qualifying Dissertation, Dept. Computer Science, University of York (1993).

31. R. I. Davis, ‘‘Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems’’, YCS217, Dept. Computer Science, University of York (1993).

32. R. I. Davis, ‘‘Sensitivity Analysis for Fixed Priority Pre-emptive Systems’’, Technical Note, Department of Computer Science, University of York (October 1994).

33. R. I. Davis, ‘‘Integrating Best-Effort Policies into Hard Real-Time Systems based on Fixed Priority Pre-emptive Scheduling. ’’, YCS240, Dept. Computer Science, University of York (1994).

34. R. I. Davis, ‘‘Dual Priority scheduling: A Means of Providing Flexibility in Hard Real-Time Systems’’, YCS230, Dept. Computer Science, University of York (1994).

35. R. I. Davis, ‘‘Guaranteeing X in Y: On-line Acceptance Tests for Hard Aperiodic tasks Scheduled by the Slack Stealing Algorithm’’, YCS231, Dept. Computer Science, University of York (1994).

36. R. I. Davis and A. Burns, ‘‘Review of an On-line Acceptance Test for Hard Aperiodic Tasks Scheduled by the Slack Stealing Algorithm’’, Letter to J. Lehoczky and S. Thuel (November 1993).

37. R. I. Davis and A. Burns, ‘‘Optimal Priority Assignment for Aperiodic Tasks with Firm Deadlines in Fixed Priority Pre-emptive Systems’’, *Information Processing Letters* **53**(5), pp. 249-254 (10th March 1995).

38. R. I. Davis, S. Punnekkat, N. C. Audsley and A. Burns, ‘‘Flexible Scheduling for Adaptable Real-Time Systems’’, *Proceedings IEEE Real-Time Technology and Applications Symposium*, Chicago, USA (15-17th May 1995).

39. R. I. Davis, K. W. Tindell and A. Burns, ‘‘Scheduling Slack Time in Fixed Prioroity Pre-emptive Systems’’, *Proceedings IEEE Real-Time Systems Syposium*, pp. 222-231 (December 1993).

40. K. S. Decker, V. R. Lesser and R. C. Whitehair, ‘‘Extending a Blackboard Architecture for Approximate Processing’’, *Real-Time Systems* **2**(1/2), pp. 47-80 (May 1990).

41. M. Dertouzos, ‘‘Control Robotics: The Procedural Control of Phtsical Processes’’, *Proceedings of IFIP congress*, pp. 807-813 (1974).

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

43. A. Dix, R.F. Stone and H. Zedan, ‘‘Design Issues for Reliable Time-Critical Systems’’, pp. 305-322 in *Proc. 1989 Real-Time Systems Symposium: Theory and Applications*, ed. H. Zedan, North Holland (1990).

44. M. S. Fineberg and O. Serlin, ‘‘Multiprogramming for Hybird Computation’’, *Proceedings AFIPS Fall Joint Computing Conference*, pp. 1-13 (1967).

45. A. Garvey and V. Lesser, ‘‘Scheduling Satisfising Tasks with a Focus on Design-to-time Scheduling.’’, *Proceedings of IEEE Workshop on Imprecise and Approximate Computation.*, pp. 25-29 (December 1992).

46. R. Gerber and S. Hong, ‘‘Semantic-Based Compiler Transformations for Enhanced Schedulability’’, *Proceedings IEEE Real-Time Systems Symposium*, Raleigh-Durham, North Carolina, pp. 232-242 (December 1993).

47. R. Gerber, S. Hong and M. Saksena, ‘‘Guaranteeing End-to-end Timing Constraints by Calibrating Intermediate Processes’’, *Proceedings 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, pp. 192-203 (7-9 December 1994).

48. M. G. Harbour, M. H. Klein and J. P. Lehoczky, ‘‘Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority’’, *Proceedings 12th IEEE Real-Time Systems Symposium*, San Antonio, TX, USA, pp. 116-128 (3-6 December 1991).

49. P. K. Harter, ‘‘Response Times in Level-Structured Systems’’, *ACM Transactions on Computer Systems* **5**(3), pp. 232-248 (August 1987).

50. A. E. Howe, D. M. Hart and P. R. Cohen, ‘‘Addressing Real-Time Constraints in the Design of Autonomous Agents’’, *Real-Time Systems* **2**(1/2), pp. 81-98 (May 1990).

51. E. D. Jensen, C. D. Locke and H. Tokuda, ''A Time-Driven Scheduling Model for Real-Time Operating Systems'', *Proceedings 6th IEEE Real-Time Systems Symposium*, pp. 112-122 (3-6 December 1985).

52. M. Joseph and P. Pandya, ''Finding Response Times in a Real-Time System'', *The Computer Journal (British Computer Society)* **29**(5), pp. 390-395, Cambridge University Press (October 1986).

53. D. I. Katcher, H. Arakawa and J. K. Strosnider, ''Engineering and Analysis of Fixed Priority Schedulers'', *IEEE Transactions on Software Engineering* **19**(9), pp. 920-934 (September 1993).

54. Y. S. Kim, ''An Optimal Scheduling Algorithm for Pre-emptable Real-Time Tasks'', *Information Processing Letters* **50**(1), pp. 43-48 (1994).

55. G. Koren and D. Shasha, ''D^{over}: An Optimal Online scheduling Algorithm for Overloaded Real-Time Systems'', *Proceedings IEEE Real-Time Systems Symposium*, Raleigh-Durham, North Carolina, pp. 290-299 (1993).

56. B. W. Lampson and D. D. Redell, ''Experience with Processes and Monitors in Mesa'', *CACM* **23**(2), pp. 105-117 (February 1980).

57. J. S. Lark, L. E. Erman, S. Forrest, K. P. Gostelow, F. Hayes-Roth and D. M. Smith, ''Concepts, Methods and Languages for Building Timely Intelligent Systems.'', *Real-Time Systems* **2**(1/2), pp. 127-148 (May 1990).

58. J. P. Lehoczky, ''Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines'', *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, pp. 201-209 (5-7 December 1990).

59. J. P. Lehoczky and S. Ramos-Thuel, ''An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks Fixed-Priority Pre-emptive systems '', *Proceedings IEEE Real-Time Systems Symposium*, pp. 110-123 (December 1992).

60. J. P. Lehoczky, L. Sha and J. K. Strosnider, ''Enhanced Aperiodic Responsiveness in Hard Real-Time Environments'', *Proceedings IEEE Real-Time System Symposium*, San Jose, California, pp. 261-270 (1987).

61. J. Lehoczky, L. Sha and Y. Ding, ''The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour'', *Proceedings IEEE Real-Time Systems Symposium*, Santa Monica, California, pp. 166-171, IEEE Computer Society Press (5-7 December 1989).

62. J. Y. T. Leung and M. L. Merrill, ''A Note on Preemptive Scheduling of Periodic, Real-Time Tasks'', *Information Processing Letters* **11**(3) (November 1980).

63. J. Y. T. Leung and J. Whitehead, ''On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks'', *Performance Evaluation (Netherlands)* **2**(4), pp. 237-250 (December 1982).

64. K. J. Lin, S. Natarajan and J. W. S. Liu, ‘‘Imprecise Results: Utilizing Partial Computations in Real-Time Systems’’, *Proceedings 8th IEEE Real-Time Systems Symposium* , Fairmont Hotel, San Jose, California, pp. 210-217 (1-3 December 1987).

65. C. L. Liu and J. W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment’’, *Journal of the ACM* **20**(1), pp. 40-61 (1973).

66. J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. S. Yu, J. Y. Chung and W. Zhao, ‘‘Algorithms for Scheduling Imprecise Computations’’, *IEEE Computer*, pp. 58-68 (May 1991).

67. C. S. Lizza, S. B. Banks and M. A. Whelan, ‘‘Pilot’s Associate: Evolution of a Functional Prototype’’, *AGARD Conference Proceedings 499 (Machine Intelligence for Aerospace Electronic Systems)*, Lisbon, Portugal, pp. 16.1-16.12 (1991).

68. C. D. Locke, ‘‘Best-Effort Decision Making for Real-Time Scheduling’’, CMU-CS-86-134 (PhD Thesis), Computer Science Department, CMU (May 10, 1986).

69. C. D. Locke, ‘‘Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives’’, *Real-Time Systems* **4**(1), pp. 37-53 (March 1992).

70. C. D. Locke, D. R. Vogel and T. J. Mesler, ‘‘Building a Predictable Avionics Platform in Ada: A Case Study’’, *Proceedings of the IEEE Real Time Systems Symposium*, pp. 181-189 (December 1991).

71. E. J. Lovesey and R. I. Davis, *Integrating Machine Intelligence into the Cockpit to Aid the Pilot*, Proceedings AGARD conference on Machine Intelligence for Aerospace Electronic Systems, 1991.

72. A. K. L. Mok, ‘‘Fundamental Design Problems of Distributed Systems For The Hard Real-Time Environment’’, MIT/LCS/TR-297, Laboratory of Computer Science, Massachsetts Institute of Technology (1983).

73. J. D. Northcutt, *Mechanisms for Reliable Distributed Real-tine Operating Systems: The Alpha Kernel*, Academic Press Inc. (1987).

74. Y. Oh and S. H. Son, ‘‘Pre-emptive Scheduling of Periodic Tasks on Multiprocessor: Dynamic Algorithms and their Performance’’, TR-CS-93-26, University of Virginia (May 1993).

75. Y. Oh and S. H. Son, ‘‘Enhancing Fault tolerance in Rate-Monotonic Scheduling’’, *Real-Time Systems* **7**(3), pp. 315-329 (November 1994).

76. C.A. O’Reilly and A.S. Cromarty, *Fast is not real-time. Designing effective real-time AI systems*, Applications of Artificial Intelligence, 1985.

77. D. W. Payton and T. E. Bihari, ‘‘Intelligent Real-Time Control of Robotic Vehicles’’, *CACM* **34**(8), pp. 48-63 (August 1991).

78. H. Peng, W. Zhang, A. Arai, Y. Lin, T. Hessburg, P . Devlin, M. Tomizuka and S. Shladover, ‘‘Experimental Automatic Lateral Control Systems for an Automobile’’, Technical Report: PATH research report UCB-ITS-PRR-92-11.

79. P. Pleinevaux, ‘‘An Improved Hard Real-Time Scheduling for the IEEE 802.5’’, *Real-Time Systems* **4**(2), pp. 99-112 (June 1992).

80. S. Punnekkat, ‘‘Static Analysis and Dynamic Decisions for Scheduling Fault-Tolerant Real-Time Task Sets’’, DPhil Thesis Proposal, Department of Computer Science, University of York (June 1995).

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

82. R. Rajkumar, L. Sha and J. P. Lehoczky, ‘‘An Experimental Investigation of Synchronisation Protocols’’, *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 11-17 (May 1989).

83. S. Ramos-Thuel and J. P. Lehoczky, ‘‘On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed Priority Systems’’, *Proceedings Real-Time Systems Symposium* (December 1993).

84. S. Ramos-Thuel and J. P. Lehoczky, ‘‘A correction note to: On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed Priority Systems’’, *Handout at Real-Time Systems Symposium* (December 1993).

85. K. Schwan and H. Zhou, ‘‘Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads’’, *IEEE Transactions on Software Engineering* **18**(8), pp. 736-748 (August 1992).

86. O. Serlin, ‘‘Scheduling of Time Critical Processes’’, *Proceedings AFIPS Spring Computing Conference*, pp. 925-932 (1972).

87. L. Sha, J. P. Lehoczky and R. Rajkumar, ‘‘Solutions For Some Practical Problems in Prioritised Preemptive Scheduling’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 181-191 (1986).

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

89. L. Sha, R. Rajkumar, J. Lehoczky and K. Ramamritham, ‘‘Mode Change Protocols for Priority-Driven Premptive Scheduling’’, *Real-Time Systems* **1**(3), pp. 244-264 (1989).

90. L. Sha, B. Sprunt and J. P. Lehoczky, ‘‘Aperiodic Task Scheduling for Hard Real-Time Systems’’, *Real-Time Systems* **1**(1), pp. 27-69 (1989).

91. M. Silly, H. Chetto and N. Elyounsi, ‘‘An Optimal Algorithm for Guaranteeing Sporadic Tasks in Hard Real-time Systems’’, *2nd IEEE Symposium on Parallel and Distributed Systems*, pp. 578-585 (Dec. 1990).

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

93. J. A. Stankovic and K. Ramamritham, ‘‘The Spring Kernel: A New Paradigm for Real-Time Operating Systems’’, COINS Technical Report 88-97, Department of Computer and Information Science, University of Massachusetts at Amherst (Novenber 7, 1988).

94. J.A. Stankovic and K. Ramamritham, ‘‘What is Predictability for Real-Time Systems?’’, *Real-Time Systems* **2**(4), pp. 247-254 (1990).

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

96. S. R. Thuel and J. P. Lehoczky, ‘‘Algorithms for Scheduling Hard Aperiodic Tasks in Fixed Priority Systems using Slack Stealing’’, *Proceedings 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, pp. 22-33 (7-9 December 1994).

97. T. Tia, J. W. S. Liu and M. Shankar, ‘‘Algorithms and Optimality of Scheduling Aperiodic Requests in Fixed-Priority Pre-emptive Systems’’, Technical Report, Dept. of Computer Science, University of Illinois at Urbana-Champaign (April 1994).

98. K. Tindell and A. Burns, ‘‘Scheduling Hard Real-Time Multi-Media Disk Traffic’’, YCS204, Department of Computer Science, University of York (July 1993).

99. K. Tindell, A. Burns and A. J. Wellings, ‘‘Allocating Real-Time Tasks (An NP-Hard Problem Made Easy)’’, *Real Time Systems* **4**(2), pp. 145-165 (1992).

100. K. Tindell and J. Clark, ‘‘Holistic Schedulability Analysis for Distributed Hard Real-Time Systems’’, *Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)* (November-December 1993).

101. K. W. Tindell, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, YCS 189, Department of Computer Science, University of York, UK (December 1992).

102. K. W. Tindell, ‘‘Fixed Priority Scheduling of Hard Real-Time systems’’, D.Phil. Thesis 94/03, Department of Computer Science, University of York, UK (1994).

103. K. W. Tindell, A. Burns and A. J. Wellings, ‘‘Mode Changes in Priority Pre-emptive Scheduled Systems’’, *Proceedings IEEE Real Time Systems Symposium*, pp. 100-109 (December 1992).

104. K. W. Tindell, A. Burns and A. J. Wellings, ‘‘Calculating Controller Area Network (CAN) Message Response Times’’, pp. 35-40 in *Proceedings of IFAC DCCS‘94, Toledo, Spain* (1994).

105. K. W. Tindell, A. Burns and A. J. Wellings, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, *Real-Time Systems* **6**, pp. 133-151 (1994).

106. K. W. Tindell and H. Hansson, ‘‘CAN, Dual Priorities and Release Jitter’’, in *Proceedings 2nd International CAN Conference* (October 1995).

107. K. W. Tindell, H. Hansson and A. J. Wellings, ‘‘Analysing Real-Time Communications: Controller Area Network (CAN)’’, *Proceedings 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, pp. 259-265 (7-9 December 1994).

108. H. Tokuda, C. W. Mercer, Y. Ishikawa and T. E. Marchok, ‘‘Priority Inversions in Real-Time Communication’’, *Proceedings 10th IEEE Real-Time Systems Symposium*, Santa Monica, California, pp. 348-359, IEEE Computer Society Press (5-7 December 1989).

109. H. Tokuda, J. W. Wendorf and H. Y. Wang, ‘‘Implementation of a Time-Driven Scheduler for Real-Time Operating Systems’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 271-280 (December 1987).

110. J. W. Wendorf, ‘‘Implementation and Evaluation of a Time-Driven Scheduling Processor’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 172-180 (1988).

111. Technical Committee of Operating Systems WG15, ‘‘Real-time Extensions for Portable Operating Systems’’, P1003.4/D11 Unapproved Draft (October 1991).

112. C. Y.Park and A. C. Shaw, ‘‘Experiments with a Program Timing Tool Based on Source-Level Timing Schema’’, *IEEE Computer*, pp. 48-57 (May 1991).

113. N. Zhang, A. Burns and M. Nicholson, ‘‘Pipelined Processors and Worst-Case Execution Times’’, *Real-Time Systems* **5**(4), pp. 319-343 (1993).

114. G. Zlokapa, ‘‘Real-Time Systems: Well-Timed Scheduling and Scheduling with Precedence Constraints’’, CMPSCI Technical Report 93-51, Department of Computer Science, University of Massachusetts at Amherst (February 1993).

Contents

Acknowledgements **3**

Declaration **5**

Abstract **7**

1 Introduction **9**
1.1 Background 12
1.2 Motivation 14
1.2.1 Next Generation Real-Time Systems 15
1.2.2 Techniques for Improving Utility 17
1.3 Objectives and Approach 19
1.3.1 Methodology 20
1.4 Research Scope 22
1.5 Thesis Organisation 23

2 Fixed Priority Pre-emptive Scheduling: A Review **27**
2.1 Advances in Fixed Priority Pre-emptive Scheduling Theory 28
2.1.1 Computational Model, Definitions and Notation 28
2.1.2 Early Work 29
2.1.3 Optimal Priority Assignment Policies 30
2.1.4 Schedulability Tests 31
2.1.5 Task Interdependency 33
2.1.6 Practical Considerations 34
2.2 Algorithms for Providing Spare Capacity at High Priority Levels 38
2.2.1 Background Processing 41
2.2.2 Polling Server 42
2.2.3 Deferrable Server 44
2.2.4 Priority Exchange Algorithm 46
2.2.5 Sporadic Server 47
2.2.6 Extended Priority Exchange Algorithm 50
2.2.7 Slack Stealing Algorithm 52
2.2.8 Critique 54
2.3 Acceptance Tests: Guaranteeing Spare Capacity 58
2.4 Summary 59

3 Identifying Spare Capacity: An Exact Approach 61
3.1 Computational Model and Assumptions 62
3.2 Schedulability Analysis 63
3.2.1 Dynamic Slack Stealing Algorithm 67
3.2.2 Optimality of the Dynamic Algorithm 69
3.2.3 Non-Optimality w.r.t. Minimising Soft Task Response Times ... 70
3.2.4 Feasibility of the Dynamic Algorithm 71
3.3 Stochastic Timing Attributes 73
3.3.1 Reclaiming Gain-Time 74
3.3.2 Release Jitter 75
3.4 Synchronisation 75
3.4.1 Resource Sharing between Hard Tasks 75
3.4.2 Reclaiming Unused Blocking Time 76
3.4.3 Resource Sharing between Hard and Soft Tasks 79
3.5 Context Switches 81
3.5.1 Incorporating Context Switch Overheads 82
3.5.2 Reclaiming Unused Context Switch Time 83
3.5.3 Reducing the Number of Context Switches 84
3.6 Arbitrary Deadlines 85
3.7 Summary 86

4 Identifying Spare Capacity: Approximate Slack Stealing 87
4.1 Approximate Slack Stealing Algorithms 88
4.1.1 Generic Slack Stealing Algorithm Formulation 88
4.1.2 Static Approximation 90
4.1.3 Dynamic Approximation 92
4.1.4 SASS Algorithm 94
4.1.5 DASS Algorithm 94
4.1.6 PASS Algorithm 94
4.2 Performance Evaluation 96
4.2.1 Simulation 96
4.2.2 Periodic Tasks: Experiments 4.1-4.4 98
4.2.3 Gain Time: Experiments 4.5-4.8 100
4.2.4 Sporadic Tasks: Experiments 4.9 - 4.12 103
4.2.5 Adaptive Tasks: Experiments 4.13 - 4.16 106
4.2.6 Mixed Task Attributes: Experiments 4.17 - 4.18 109
4.3 Overheads and Implementation Issues 110
4.3.1 Data Consistency 110
4.3.2 Slack Stealing: Static Approximation 111
4.3.3 Slack Stealing: Dynamic Approximation 112
4.3.4 DASS Algorithm Overheads 113
4.3.5 PASS Algorithm Overheads 117
4.3.6 HASS Algorithm Overheads 118
4.4 Summary 122

5 Identifying Spare Capacity: Dual Priority Scheduling 123
5.1 Dual Priority Scheduling 124
5.1.1 Computational Model, Assumptions and Notation 124
5.1.2 Basic Analysis 125
5.1.3 Synchronisation 126
5.1.4 Release Jitter 128
5.1.5 Arbitrary Deadlines 129
5.1.6 Choosing Priority Promotion Times 130
5.1.7 Run-Time Operation of Dual Priority Scheduling 131
5.1.8 Example 132
5.2 Performance Evaluation 133
5.2.1 Periodic Tasks: Experiments 5.1-5.4 133
5.2.2 Gain Time: Experiments 5.5-5.8 135
5.2.3 Sporadic Tasks: Experiments 5.9 - 5.12 135
5.2.4 Adaptive Tasks: Experiments 5.13 - 5.16 135
5.2.5 Mixed Task Attributes: Experiments 5.17 - 5.18 136
5.3 Overheads and Implementation Issues 145
5.4 Summary 146

6 Guaranteeing Spare Capacity: Acceptance Tests 147
6.1 Overview of Previous Research 148
6.1.1 Example 150
6.2 Priority Assignment 152
6.2.1 Computational Model and Notation 152
6.2.2 Optimal Priority Assignment 153
6.2.3 Optimality w.r.t Scheduling Subsequent Aperiodic Arrivals 156
6.2.4 When to Perform an Acceptance Test? 159
6.3 Acceptance Tests for the Slack Stealing Algorithm 160
6.3.1 Exact Test: Aperiodics with Specific Deadlines 160
6.3.2 Generic Acceptance Test Procedure and Computational Model 161
6.3.3 Exact Test: Aperiodics with Arbitrary Deadlines 162
6.3.4 Sufficient Test: Aperiodics with Arbitrary Deadlines 164
6.3.5 Slack Stealing 166
6.3.6 Incorporating Blocking 167
6.3.7 Resource Sharing between Aperiodics and Hard Tasks 168
6.4 Acceptance Tests for Background Scheduling 169
6.5 Acceptance Tests for Dual Priority Scheduling: 170
6.5.1 Exact Acceptance Test: Aperiodics with Arbitrary Deadlines ... 171
6.5.2 Sufficient Acceptance Test: Aperiodics with Arbitrary Deadlines 173
6.5.3 Resource sharing 174
6.5.4 Soft Tasks 175
6.5.5 Scheduling Policies 175
6.6 Performance Evaluation 175
6.6.1 Results 176
6.6.2 Practical Considerations 180
6.6.3 Overheads: Simulation 180
6.7 Summary 184

7 Allocating Spare Capacity: Admission Policies 187
7.1 Review of Scheduling Policies for Maximising Utility 189
7.1.1 Earliest Deadline - FCFS 191
7.1.2 Maximum Value-Density Scheduling 191
7.1.3 Best-Effort Scheduling 193
7.1.4 D^{over} Algorithm 195
7.1.5 RED Algorithm 197
7.1.6 Review Summary 198
7.1.7 Maximum Guaranteed Utility 200
7.2 Admission Policies 202
7.2.1 FCFS Policy 202
7.2.2 Best-Effort Policy 203
7.2.3 Adaptive Threshold Policy 204
7.2.4 Policy Operation 205
7.3 Performance Evaluation 207
7.3.1 Results 208
7.4 Discussion 212
7.4.1 Utility Model 212
7.4.2 Overheads 212
7.4.3 System Dynamics 213
7.5 Multiple Versions Scheduling 214
7.5.1 Example 216
7.5.2 Local Optimality 217
7.6 Summary 217

8 Proof of Concept: Implementation 219
8.1 DRTEE Kernel 220
8.1.1 Timer Facilities 220
8.1.2 Kernel Operation 220
8.1.3 Discussion 223
8.1.4 Logging and Monitoring 223
8.2 Analysis of Kernel Overheads 225
8.2.1 Computational Model and Notation 225
8.2.2 Analysis of Background Scheduling 226
8.2.3 Analysis of Dual Priority Scheduling 227
8.2.4 Analysis of Dynamic Slack Stealing 228
8.3 Performance Measurement and Evaluation 230
8.3.1 Task Set Generation 230
8.3.2 Measurements 232
8.3.3 Measured v. Calculated Worst Case Response Times 234
8.3.4 Overheads v. Number of Tasks 236
8.4 Limitations and Possible Improvements 240
8.4.1 16 v 32 Bit Machine Code 240
8.4.2 Integrated Event Handling and Scheduling 240
8.5 Summary 243

9 Conclusions and Future Work 245
9.1 Review of Objectives 245
9.2 Meeting the Objectives 246
9.2.1 Coverage 246
9.2.2 Performance / Overheads 247
9.2.3 Simplicity 248
9.3 Contribution 249
9.4 Further Work 249
9.4.1 Sensitivity Analysis 249
9.4.2 Feasibility Tests for SRPT Scheduling 250
9.4.3 Dual Priority Scheduling in Distributed Systems 251
9.4.4 Avoiding Priority Inversion in Real-Time Kernels 251
9.4.5 Fault Tolerant Real-Time Systems 251
9.5 In Conclusion 252

Appendix A: Acceptance Tests 253
A.1 Computational Model 253
A.2 Static Slack Stealing Algorithm 253
A.3 Acceptance Test #1 255
A.4 Acceptance Test #2 258

Appendix B: Calculating Slack for Tasks with $D > T$ 263
B.1 Analysis 263
B.1.1 Calculating slack on the qth invocation 265
B.1.2 Example slack calculation 267

Appendix C: Avoiding Priority Inversion in the Kernel 271
C.1 Integrated Event Handling and Scheduling 271

References 275