Hierarchical Fixed Priority Pre-emptive Scheduling

Robert Davis and Alan Burns,
Real-Time Systems Research Group, Department of Computer Science,
University of York, YO10 5DD, York (UK)
rob.davis@cs.york.ac.uk, burns@cs.york.ac.uk

Abstract

This report focuses on the hierarchical scheduling of systems where a number of separate applications reside on a single processor. It addresses the particular case where fixed priority pre-emptive scheduling is used at both global and local levels, with periodic or deferrable servers associated with each application. Using response time analysis, an exact schedulability test is derived for application tasks. This test improves on previously published work. The analysis is extended to the specific case of harmonic tasks that can be bound to the release of their server. These tasks exhibit improved schedulability indicating that it is advantageous to choose server periods that enable some tasks to be bound to the release of their server. The use of Periodic, Sporadic and Deferrable Servers is considered with the conclusion that the simple Periodic Server dominates both Sporadic and Deferrable Servers when the metric is application task schedulability. As an adjunct some interesting results are presented on the optimal priority ordering of a set of Deferrable Servers, rate-monotonic priority ordering is not optimal in this case. The second part of the report investigates the problem of selecting the optimal set of server parameters. Analysis of this problem reveals a simple method for determining the optimal set of server capacities given fixed server periods and priorities. Despite this advance, empirical results show that server parameter selection exhibits dependencies between the set of servers and so the locally optimum selection of parameters for one server often does not result in an overall selection of parameters that is either schedulable or results in the maximum remaining system utilisation. The selection of server parameters is a holistic problem that may in practice only be soluble via search; a greedy approach to parameter selection is demonstrated to be ineffective.

Acknowledgements

This work was partially funded by the EU Information Society Technologies (IST) Program, Flexible Integrated Real-Time Systems Technology (FIRST) Project, IST-2001-34140.

1 Introduction

In automotive electronics, the advent of advanced high performance embedded microprocessors such as Freescale Semiconductor’s (Motorola’s) MPC5200 (PowerPC 603), Infineon’s TC1765 (TriCore) and NEC’s V850E/RS1 have made possible functionality such as adaptive cruise control, lane departure warning systems, integrated telematics and satellite navigation applications as well as advances in engine management, transmission control and body electronics. Where low-cost 8 and 16-bit microprocessors were previously used as the basis for separate Electronic Control Units (ECUs) each supporting a single hard real-time application, there is now a trend towards integrating functionality into a smaller number of more powerful microprocessors. The motivation for such integration comes mainly from cost reduction but also offers the opportunity of functionality enhancement. This trend in automotive electronics is mirrored by a similar trend in avionics.

Integrating a number of real-time applications onto a single microprocessor raises issues of resource allocation and partitioning. Disparate applications require access to processor and other resources in a manner that ensures they are able to complete the necessary computations within specified time constraints, whilst ensuring that they do not impinge upon the real-time behaviour of other applications.

When composing a system comprising a number of applications, it is typically a requirement to provide temporal isolation between the various applications. This enables the properties of previous system designs, where each application resided on a separate processor, to be retained. In particular if one application fails to meet its time constraints then there should be no knock on effects on other unrelated applications. There is currently considerable interest in hierarchical scheduling as a way of providing temporal isolation between applications executing on a single processor.

In a hierarchical system, a *global* scheduler is used to determine which application should be allocated the processor at any given time and a *local* scheduler is used to determine which of the chosen application's tasks should actually execute. A number of different scheduling schemes have been proposed for both global and local scheduling. These include cyclic or time slicing frameworks, dynamic priority based scheduling and fixed priority scheduling. In this report we focus on the use of fixed priority pre-emptive scheduling (FPPS) for both global and local scheduling.

Fixed priority pre-emptive scheduling offers advantages of flexibility over cyclic approaches whilst being sufficiently simple to implement, that it is possible to construct highly efficient embedded real-time operating systems that use this scheduling policy.

The basic framework for a system utilising hierarchical fixed priority pre-emptive scheduling is as follows. The system comprises a number of applications each of which is composed of a set of tasks. A separate *server* is allocated to each application. Each server has an execution capacity and a replenishment period, enabling the overall processor capacity to be divided up between the different applications. Each server has a unique priority that is used by the global scheduler to determine which of the servers with capacity remaining and tasks ready to execute should be allocated the processor. Further, each task has a unique priority within its application. The local scheduler, within each server, uses task priorities to determine which of an application's tasks should execute when the server is active. The basic model assumes that tasks and applications are independent, however the model can be extended to allow local resource sharing between tasks in the same application and global resource sharing between tasks in different applications.

1.1 Related Work

Kuo and Li [1] first introduced analysis of hierarchical fixed priority pre-emptive scheduling, building upon the work of Deng and Liu [2]. The analysis provided by Kuo and Li considered the use of Sporadic Servers [3] to execute applications. Using the techniques of Liu and Layland [4] they provided a simple utilisation based schedulability test. However for this utilisation based test to be applicable, severe restrictions were placed on the server parameters. In particular, each server period had to be the greatest common divisor (GCD) or a divisor of the GCD of all the tasks in the application.

Saewong et al [5] provided response time analysis for hierarchical systems using Deferrable Servers [6] or Sporadic Servers [3] to schedule a set of hard real-time applications. This analysis assumes that in the worst-case a server's capacity is made available at the end of its period. Whilst this is a safe assumption it is also pessimistic, for example, the highest priority server will typically have a response time that is much shorter than its period and so will always be able to make capacity available earlier than considered in [5]. The schedulability analysis given in [5] is sufficient but not necessary: there are some systems that it would deem unschedulable that are in fact schedulable.

Lipari and Bini [7] provide an alternative response time analysis formulation using an availability function to represent the time made available by a server from an arbitrary time origin. This formulation again makes the assumption that in the worst-case, server capacity is made available at the very end of the server's period. Lipari and Bini also investigate the problem of server parameter selection and consider choice of replenishment period and capacity for a single server in isolation, using a geometric approach based on an approximation of the server availability function.

In [8], Almeida builds upon the work of Lipari and Bini. The analysis given in [8] recognises that the server availability function depends on the *"maximum jitter that periods of server availability may suffer"*. A parameter (delta) is introduced into the analysis to represent the initial latency in server capacity becoming available. We note that setting this parameter to reflect the server's computed worst-case response time and hence its maximum jitter would result in more accurate analysis as demonstrated in section 3.1 of this report. Almeida also describes an approximate algorithm for selecting appropriate server parameters, for a single server, such that the server requires the minimum processor utilisation.

1.2 Organisation of this Report

This report is divided into two main sections addressing two key research topics:

1. Schedulability analysis for hierarchical systems.
2. Optimal server parameter selection.

Section 2 describes the terminology, notation and system model used in the rest of the report.

Section 3 presents schedulability analysis tests that compute the *exact* worst-case response time of tasks scheduled under a set of Deferrable Servers. This analysis is extended to accurately model *bound* tasks, the releases of which are synchronised with their server's period. The analysis is also extended to account for access to global shared resources. We evaluate the exact analysis presented in this report by comparing its effectiveness to that of previously published schedulability tests. Our empirical study investigates the effects of server context switch overheads and server algorithm selection on system schedulability.

Section 4 examines the problem of selecting server parameters. We provide methods that are able to derive optimum values for any one of the three server parameters (priority, period and capacity) if the other two are known. However we also demonstrate that the optimal selection of server periods and capacities has dependencies between the different servers and so cannot be solved by finding the local optima for a single server in isolation. Our empirical study investigates optimal server parameter selection and the utility of choosing server periods that are an exact divisor of one or more application task periods enabling those tasks to be bound to the release of the server.

Section 5 summarises the major contributions of the report and suggests directions for future research.

As an addendum to the main report, the appendix contains some interesting results on the optimal priority ordering of a set of Deferrable Servers.

2 Hierarchical Scheduling Model

2.1 Terminology and System Model

We are interested in the problem of scheduling multiple real-time *applications* on a single processor. Each application comprises a number of real-time *tasks*. *A*ssociated with each application is a *server*. The application tasks execute within the associated server, which affords them temporal isolation.

Scheduling takes place at two levels: *global* and *local*. The global scheduling policy determines which server has access to the processor at any given time, whilst the local scheduling policy determines which application task that server should execute. In this report we analyse systems where the fixed priority pre-emptive scheduling policy is used for both global and local scheduling.

Application tasks may arrive and become ready to execute either *periodically* at fixed intervals of time, or *sporadically* after some minimum inter-arrival time has elapsed. Each application task τ_i, has a unique priority *i* within its application and is characterised by its relative *deadline* D_i, *worst-case execution time* C_i, and minimum inter-arrival time T_i, otherwise referred to as its *period*. In addition, we will assume that each application contains one or more soft real-time tasks. These soft tasks are assumed to execute at lower priorities than the hard real-time tasks. The soft real-time tasks may however consume server capacity and hence affect the worst-case scenario for hard real-time task execution.

Each server has a unique priority *s*, within the set of servers and is characterised by its *capacity* C_S, *replenishment period* T_S, and *jitter* J_S. A server's capacity is the maximum amount of execution time that may be consumed by the server in a single invocation. The replenishment period is the minimum time before the server's capacity is available again. The server's jitter is the difference between the minimum and maximum time that can elapse between replenishment of the server's capacity and that capacity starting to be consumed given no higher priority interference.

Application tasks are referred to as *bound* or *unbound* [18]. Bound tasks have a period that is an exact multiple of their server's period and arrival times that coincide with replenishment of the server's capacity. Thus bound tasks are only ever released at the same time as their server. All other tasks are referred to as unbound.

A task's worst-case response time R_i, is the longest possible time from the task arriving to it completing execution. Similarly, a server's worst-case response time R_S, is the longest possible time from the server being replenished to its capacity being exhausted, given that there are tasks ready to use all of the server's capacity. A task is said to be *schedulable* if its worst-case response time does not exceed its deadline. A server is schedulable if its response time does not exceed its period.

The *critical instant* [4] for a task is defined as the pattern of execution of other tasks and servers that leads to the task's worst-case response time.

Initially we will assume that all applications and tasks are independent, however this restriction is lifted in section 3.4, permitting access to mutually exclusive local resources by tasks in the same application and access to mutually exclusive global resources by tasks in different applications.

2.2 Servers

In this report we consider the Deferrable Server (DS) [6] the Sporadic Server (SS) [3] and the Periodic or Polling Server (PS) [19].

The *Periodic Server* is invoked with a fixed period. If there are application tasks ready to use the server's capacity, then they are executed until the tasks either complete or the server's capacity is exhausted. If there are no tasks ready to use the server then its capacity is assumed to be idled away, as if there was a background task that is always ready to execute. Once the server's capacity is exhausted, the server suspends execution until its capacity is replenished at the start of its next period. If a task arrives before the server's capacity has been exhausted then it will be serviced. Execution of the server may be delayed and or pre-empted by the execution of other servers of a higher priority.

The *Deferrable Server* is also invoked with a fixed period. It differs from the Periodic Server in that if no tasks are ready to use the server then it may suspend its execution, preserving its capacity. The Deferrable Server's capacity may be preserved throughout its period. If an application task becomes ready late in the server's period it can be executed until either the server's capacity is exhausted or the end of the server's period is reached. At the end of the server's period any remaining server capacity is discarded and the server's capacity is then replenished. Again execution of the server may be delayed and or pre-empted by the execution of other servers of a higher priority. Schedulability analysis of the Deferrable Server needs to take account of the well-known phenomenon of *back-to-back* hits. By preserving its capacity until near the end of its period a high priority Deferrable Server can cause back-to-back interference of $2C_S$ on lower priority servers. Effectively a Deferrable Server has a jitter equal to $T_S - C_S$ [9].

The *Sporadic Server* differs from both the Periodic Server and the Deferrable Server in that its capacity is only replenished after it has been used. Assuming a Sporadic Server of capacity C_S, and replenishment period T_S; if some fraction f_S , of the server's capacity is used then capacity f_S is scheduled for replenishment at a time T_S after the start of the priority level *s* busy period which included consumption of that fraction of the server's capacity. This means that replenishment may take place less than T_S after capacity f_S started to be consumed provided that the processor was previously busy executing at a higher priority than *s*. In [3], Sprunt proved that in the worst-case the interference due to a Sporadic Server is equivalent to that of a simple Periodic Server. The implementation complexity and overheads of the Sporadic Server are however significantly greater than those of either the Periodic or Deferrable Server due to the requirement to keep track of a number of different replenishment times and capacities.

2.3 Busy Periods and Loads

The analysis presented in section 3 makes use of the concepts of *busy periods* and *loads*. For a particular application, a priority level *i* busy period is defined as an interval of time during which there is outstanding task execution at priority level *i* or above. Busy periods may be represented as a function of the outstanding execution time at and above a given priority level, thus $w_i(L)$ is used to represent a priority level *i* busy period equivalent to the longest time that the application's server can take to execute a given load *L*.

The load on a server is itself a function of the time interval considered. We use $L_i(w)$ to represent the total task executions of priority level *i* and above, released within a time window of length *w*.

3 Schedulability Analysis for Hierarchical Systems

In this section we present exact schedulability analysis for applications comprising bound and unbound hard real-time tasks executing under a set of Deferrable Servers in a hierarchical system scheduled at both global and local levels according to the fixed priority pre-emptive scheduling policy.

3.1 Exact Analysis

We derive the exact worst-case response time for a task τ_i ,executing under a server *S*, using the principles of Response Time Analysis [10] as follows:

1. Determine the *critical instant*: the pattern of server and task execution that leads to the worst-case response time of the task.
2. Derive a formula for $L_i(w)$, the load at priority level *i* and above, released in a window of length *w* starting at the critical instant.
3. Derive a formula for $w_i(L\ )$, the length of the priority level *i* busy period starting at the critical instant and finishing when the server has completed execution of the load *L*.
4. Combine the formulae for $L_i(w\ )$ and $w_i(L)$ into a recurrence relation that can be solved to find the worst-case response time of task τ_i.

3.1.1 Critical Instant

In [5], Saewong et al showed that the critical instant for a task scheduled under a Deferrable Server occurs when:

1. The server's capacity has been exhausted by lower priority tasks as early in its period as possible.
2. The task of interest and all higher priority tasks in the application arrive just after the server's capacity has been exhausted.
3. The server's capacity is replenished at the start of its next period, however further execution of the server is then delayed for as long as possible due to interference from other higher priority servers.

The critical instant is illustrated in Figure 1 below.

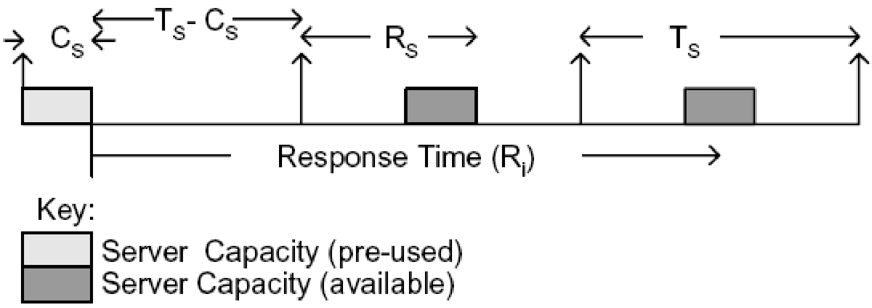

Figure 1 Critical Instant for tasks scheduled under a Deferrable server.

3.1.2 Response Time Analysis

The load due to application task execution released in any given time interval *w* is given by the equation:

$$L_i(w) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w}{T_j} \right\rceil C_j \tag{1}$$

Where *hp(i)* is the set of tasks with priorities higher than *i*. If the set of tasks had the processor to themselves then the time taken to execute this load would be the same as the load itself. As the RHS of the equation represents a monotonically non-decreasing function of *w* it could be solved using the following recurrence relation [10].

$$w_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{2}$$

Recurrence begins with $w_i^0 = C_i$ and ends when $w_i^n > D_i$ in which case the task is not schedulable or when $w_i^{n+1} = w_i^n$ in which case w_i^{n+1} gives the worst-case response time of the task.

However, we are interested in the case where tasks are executed under a server. In this case, the length of the busy period required to execute the load requires further consideration. Figure 2 illustrates the busy period in more detail.

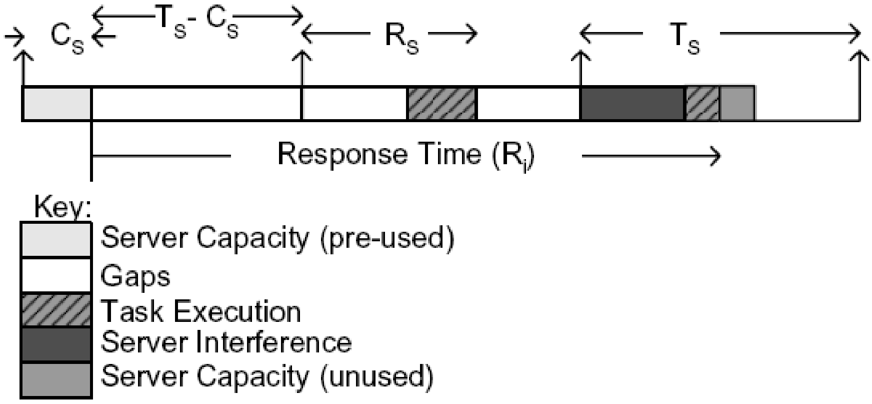

Figure 2 Busy Period

The extent of the busy period derives from three components:

1. The execution of tasks of priority *i* and higher released during the busy period. Equation (1) quantifies this load.
2. The gaps in any complete server periods.
3. Interference from higher priority servers in the final server period that completes execution of task τ_i.

The total length of the gaps in the complete server periods is given by:

$$\left\lceil \frac{L_i(w)}{C_S} \right\rceil (T_S - C_S) \tag{3}$$

The interference due to higher priority servers executing during the final server period that completes execution of task τ_i can be modeled in a number of ways.

1. The analysis given by Saewong et al in [5] assumes that each server's worst-case response time is equal to its period and effectively models this interference as $(T_S - C_S)$. This is a safe but pessimistic assumption. For the set of servers to be schedulable most if not all of the servers will have a response time that is shorter than their period[1]. In particular, the highest priority server will typically have a response time equal to its capacity.
2. The interference can be modeled as $(R_S - C_S)$. This removes much of the pessimism however it does not provide exact analysis. The analysis given by Almeida in [8] can be made to match this model if an appropriate *"initial latency"* is chosen. Although this is not explicitly stated in [8].
3. The exact worst-case interference in the final server period is dependent on the amount of task execution that the server needs to complete before the end of the busy period. This may be much less that the server's capacity and so the maximum interference may be considerable less than $(R_S - C_S)$. The exact interference can be calculated using information about server priorities, capacities and replenishment periods.

Figure 3 illustrates the exact interference present in the final server period

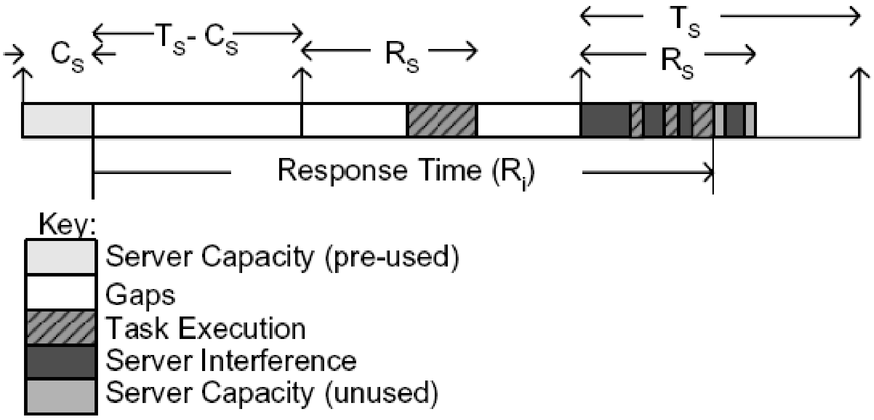

Figure 3 Interference in the final server period

The extent to which the busy period *w* extends into the final server period is given by:

$$w - \left((T_S - C_S) + \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil - 1 \right) T_S \right)$$

Which simplifies to:

$$w - \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil T_S - C_S \right)$$

[1] In considering this point it is important to distinguish between the response time of a Deferrable Server and the latest time it may execute due to the server's ability to suspend its execution if there are no tasks ready to execute.

Utilising the analysis of servers presented by Bernat and Burns [9], the interference due to higher priority servers in the above interval is given by:

$$I(w) = \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w - \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil T_S - C_S \right) + J_X}{T_X} \right\rceil C_X \qquad (4)$$

Where *hp(S)* is the set of servers with higher priority than server *S* and J_X is the release jitter of the higher priority server *X*. (For a Deferrable Server, $J_X = T_X - C_X$).

Combining the previous equations we have:

$$w = L_i(w) + \left\lceil \frac{L_i(w)}{C_S} \right\rceil (T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w - \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil T_S - C_S \right) + J_X}{T_X} \right\rceil C_X \qquad (5)$$

Note that the length of the busy period *w* appears on both sides of equation (5). This type of equation can be solved via a recurrence relation provided that the RHS is a monotonically non-decreasing function of *w*. It is not immediately obvious that this is the case here. However assuming that the servers are themselves schedulable, we observe that the interference in the server's final period, given by the summation term, is constrained to be between 0 and $(T_S - C_S)$. The summation term itself is a monotonically non-decreasing function of $L_i(w)$ except at values of $L_i(w) = nC_S$. At exactly these values the 2nd term increases by $(T_S - C_S)$, thus the 2nd and 3rd terms taken together form a monotonically non-decreasing function of the task load $L_i(w)$. The task load is itself a monotonically non-decreasing function of *w*, hence the RHS of the equation is a monotonically non-decreasing function of *w* and solution via a recurrence relation is possible although not entirely straightforward.

To solve equation (5) we need to modify the summation term to ensure correct convergence as intermediate values of *w* and $L_i(w)$ are calculated. This is as a direct consequence of the fact that the 3rd term alone is not a monotonically non-decreasing function of *w*. The modification simply ensures that the extent to which the busy period extends into the final server period is not considered to be an interval of negative length.

$$w_i^n = L_i(w_i^n) + \left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil (T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil T_S - C_S \right)\right) + J_X}{T_X} \right\rceil C_X \qquad (6)$$

Recurrence starts with a value of $w_i^0 = C_i + \lceil C_i / C_S \rceil (T_S - C_S)$ and ends either when $w^{n+1} = w^n$ in which case w^n gives the task's worst-case response time or when $w^{n+1} > D_i$ in which case the task is not schedulable. On each iteration of the recurrence relation, a new value of $L_i(w_i^n)$ is first calculated using equation (1).

Example

The following example shows how the recurrence relation iterates towards a solution. Consider a system comprising two Deferrable Servers with parameters given in the table below.

Server	C_S	T_S	J_S	R_S
HP	2	5	3	2
LP	8	20	12	16

In this example, we determine the worst-case response time of the two highest priority tasks in the application serviced by the lower priority server (LP). These tasks are characterised as follows.

Task	C_i	T_i	D_i
1	10	50	50
2	8	100	100

For the task at priority 1, we begin with $w_1^0 = C_i + \left\lceil \frac{C_i}{C_S} \right\rceil (T_S - C_S)$ and then iterate calculating alternate values of w_1^n and $L_1(w_1^n)$

$$w_1^0 = 10 + \left\lceil \frac{10}{8} \right\rceil (20 - 8) = 34 \qquad L_1(34) = 10$$

$$w_1^1 = 34 + \left\lceil \frac{\max\left(0,34 - \left(\left(\left\lceil \frac{10}{8} \right\rceil 20 - 8\right)\right)\right) + 3}{5} \right\rceil 2 = 36 \qquad L_1(36) = 10$$

$$w_1^2 = 34 + \left\lceil \frac{\max\left(0,36 - \left(\left(\left\lceil \frac{10}{8} \right\rceil 20 - 8\right)\right)\right) + 3}{5} \right\rceil 2 = 38 \qquad L_1(38) = 10$$

$$w_1^3 = 34 + \left\lceil \frac{\max\left(0,38 - \left(\left(\left\lceil \frac{10}{8} \right\rceil 20 - 8\right)\right)\right) + 3}{5} \right\rceil 2 = 38$$

As $w_1^3 = w_1^2$ the recurrence relation has converged on a solution. The worst-case response time of the task at priority 1 is 38.

For the task at priority level 2 we have:

$$w_2^0 = 8 + \left\lceil \frac{8}{8} \right\rceil (20-8) = 20 \qquad L_2(20) = 8 + \left\lceil \frac{20}{50} \right\rceil 10 = 18$$

$$w_2^1 = 18 + \left\lceil \frac{18}{8} \right\rceil (20-8) + \left\lceil \frac{\max\left(0,20-\left(\left\lceil \frac{18}{8} \right\rceil 20-8\right)\right)+3}{5} \right\rceil 2 = 56 \qquad L_2(56) = 8 + \left\lceil \frac{56}{50} \right\rceil 10 = 28$$

$$w_2^2 = 28 + \left\lceil \frac{28}{8} \right\rceil (20-8) + \left\lceil \frac{\max\left(0,56-\left(\left\lceil \frac{28}{8} \right\rceil 20-8\right)\right)+3}{5} \right\rceil 2 = 78 \qquad L_2(78) = 8 + \left\lceil \frac{78}{50} \right\rceil 10 = 28$$

$$w_2^3 = 28 + \left\lceil \frac{28}{8} \right\rceil (20-8) + \left\lceil \frac{\max\left(0,78-\left(\left\lceil \frac{28}{8} \right\rceil 20-8\right)\right)+3}{5} \right\rceil 2 = 80 \qquad L_2(80) = 8 + \left\lceil \frac{80}{50} \right\rceil 10 = 28$$

$$w_2^4 = 28 + \left\lceil \frac{28}{8} \right\rceil (20-8) + \left\lceil \frac{\max\left(0,80-\left(\left\lceil \frac{28}{8} \right\rceil 20-8\right)\right)+3}{5} \right\rceil 2 = 82 \qquad L_2(82) = 8 + \left\lceil \frac{82}{50} \right\rceil 10 = 28$$

$$w_2^5 = 28 + \left\lceil \frac{28}{8} \right\rceil (20-8) + \left\lceil \frac{\max\left(0,82-\left(\left\lceil \frac{28}{8} \right\rceil 20-8\right)\right)+3}{5} \right\rceil 2 = 82$$

As $w_2^5 = w_2^4$ the recurrence relation has converged on the solution. The worst-case response time of the task at priority 2 is 82.

The table below compares the response times of the tasks in the above example using, (1) the exact analysis introduced in this paper, (2) approximate analysis modeling interference in the final server period as $(R_S - C_S)$ and (3) the analysis given by Saewong et al in [5] treating interference in the final server period as $(T_S - C_S)$.

Task	***C_i***	***T_i***	***D_i***	**Response Times *R_i***		
				(1) Exact	**(2)**	**(3)**
1	10	50	50	38	42	46
2	8	100	100	82	84	88

This example clearly illustrates the improvements in task schedulability that can be obtained by using exact schedulability analysis.

3.1.3 Analysis of Bound tasks

The analysis derived in the previous sections considers only unbound tasks. Recall that unbound tasks may be released independently of their server. In this section, we provide analysis for the case of bound tasks that have periods that are strict multiples of their server's replenishment period and arrivals that are synchronised with the replenishment of the server's capacity.

The critical instant for a bound task differs from that of an unbound task in that the arrival of a bound task is synchronised with replenishment of the associated server. The critical instant for a bound task is illustrated in Figure 4 below.

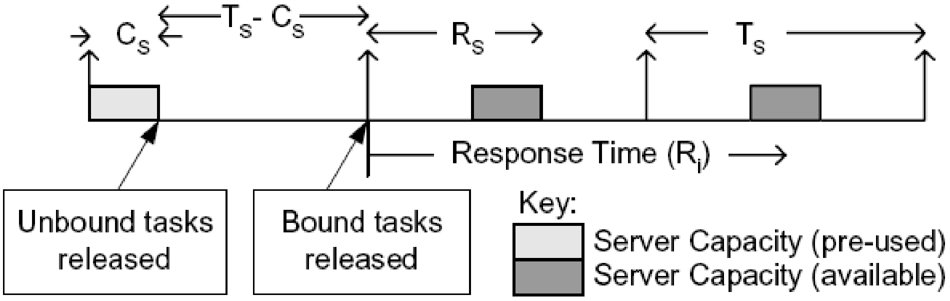

Figure 4 Critical Instant for Bound Tasks

In the case of a bound task, the critical instant is as follows:

1. The server's capacity has been exhausted by lower priority tasks as early as possible in its period.
2. All higher priority *unbound* tasks in the application arrive just after the server's capacity has been exhausted.
3. The *bound* task of interest and all higher priority *bound* tasks in the application arrive at the start of the server's next period.
4. The server's capacity is replenished at the start of its next period, however further execution of the server is delayed for as long as possible due to interference from other higher priority servers.

The worst-case response time of a bound task τ_i, can be found by determining the extent of the priority level *i* busy period deriving from the critical instant described above. As before the busy period can be viewed as being made up of three components:

1. The execution of task τ_i and tasks of higher priority released during the busy period.
2. The gaps in any complete periods of the server.
3. Interference from higher priority servers in the final server period that completes the execution of task τ_i.

The task load at priority level *i* and higher, to be executed in a busy period *w*, starting at the release of the bound task of interest is given by:

$$L_i(w) = C_i + \sum_{\substack{\forall j \in hp(i) \\ bound}} \left\lceil \frac{w}{T_j} \right\rceil C_j + \sum_{\substack{\forall j \in hp(i) \\ unbound}} \left\lceil \frac{w + (T_S - C_S)}{T_j} \right\rceil C_j \qquad (7)$$

Where *hp(i)bound* and *hp(i)unbound* are the sets of bound and unbound tasks within the application, that have priorities higher than *i*. Note as the unbound tasks cannot be executed by the server until its next replenishment the release of the unbound tasks is effectively jittered by $(T_S - C_S)$ from their arrival.

The gaps in complete server periods, not including the final server period, are given by:

$$\left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil - 1 \right)(T_S - C_S) \qquad (8)$$

The extent to which the busy period *w* extends into the final server period is given by:

$$w - \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil - 1 \right) T_S$$

Hence the full extent of the busy period is given by:

$$w = L_i(w) + \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w - \left(\left\lceil \frac{L_i(w)}{C_S} \right\rceil - 1 \right) T_S + J_X}{T_X} \right\rceil C_X \qquad (9)$$

Again it can be shown that the RHS of equation (9) is a monotonically non-decreasing function of the task load and can therefore be solved via a recurrence relation.

$$w_i^n = L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right) + J_X}{T_X} \right\rceil C_X$$

(10)

Recurrence starts with a value of $w_i^0 = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right)(T_S - C_S)$ and ends either when $w^{n+1} = w^n$ in which case w^n gives the task's worst-case response time or when $w^{n+1} > D_i$ in which case the task is not schedulable. On each iteration of the recurrence relation, a new value of $L_i(w_i^n)$ is first calculated using equation (7).

Example

We illustrate the operation of the recurrence relation given by equation (7) and equation (10) using the example described in section 3.1.2. This also highlights the improvement in schedulability that can be made by binding tasks to a server. Recall that the example system comprises two Deferrable Servers.

Server	C_S	T_S	J_S	R_S
HP	2	5	3	2
LP	8	20	12	16

This time, we will calculate the worst-case response time of the 2nd highest priority task in the application serviced by the lower priority server (LP). The highest priority task cannot be bound to the server, as its period is not a multiple of the server period.

As before, the tasks are characterised as follows.

Task	C_i	T_i	D_i
1	10	50	50
2	8	100	100

For the task at priority 2, we begin with $w_2^0 = C_2 + \left(\left\lceil \frac{C_2}{C_S} \right\rceil - 1\right)(T_S - C_S)$ and then iterate calculating alternate values of w_2^n and $L_2(w_2^n)$

$$w_2^0 = 8 + \left(\left\lceil \frac{8}{8} \right\rceil - 1\right)(20 - 8) = 8 \qquad L_2(8) = 8 + \left\lceil \frac{20}{50} \right\rceil 10 = 18$$

$$w_2^1 = 18 + \left(\left\lceil \frac{18}{8} \right\rceil - 1\right)(20 - 8) + \left\lceil \frac{\max\left(0{,}18 - \left(\left\lceil \frac{18}{8} \right\rceil - 1\right)20\right) + 3}{5} \right\rceil 2 = 44 \quad L_2(44) = 8 + \left\lceil \frac{56}{50} \right\rceil 10 = 28$$

$$w_2^2 = 28 + \left(\left\lceil \frac{28}{8} \right\rceil - 1\right)(20 - 8) + \left\lceil \frac{\max\left(0{,}44 - \left(\left\lceil \frac{28}{8} \right\rceil - 1\right)20\right) + 3}{5} \right\rceil 2 = 66 \quad L_2(66) = 8 + \left\lceil \frac{78}{50} \right\rceil 10 = 28$$

$$w_2^3 = 28 + \left(\left\lceil \frac{28}{8} \right\rceil - 1\right)(20 - 8) + \left\lceil \frac{\max\left(0{,}66 - \left(\left\lceil \frac{28}{8} \right\rceil - 1\right)20\right) + 3}{5} \right\rceil 2 = 68 \quad L_2(68) = 8 + \left\lceil \frac{80}{50} \right\rceil 10 = 28$$

$$w_2^4 = 28 + \left(\left\lceil \frac{28}{8} \right\rceil - 1\right)(20-8) + \left\lceil \frac{\max\left(0{,}68 - \left(\left\lceil \frac{28}{8} \right\rceil - 1\right)20\right) + 3}{5} \right\rceil 2 = 70 \; L_2(70) = 8 + \left\lceil \frac{82}{50} \right\rceil 10 = 28$$

$$w_2^5 = 70$$

As $w_2^5 = w_2^4$ the recurrence relation has converged on the solution. The worst-case response time of the task at priority 2 is 70. This compares favourably with a response time of 82 when the task is unbound.

3.1.4 Analysis of Unbound tasks

Finally, we extend the analysis presented in section 3.1.2 to include the effect of higher priority bound tasks on the response time of an unbound task. The critical instant for an unbound task is illustrated in Figure 5.

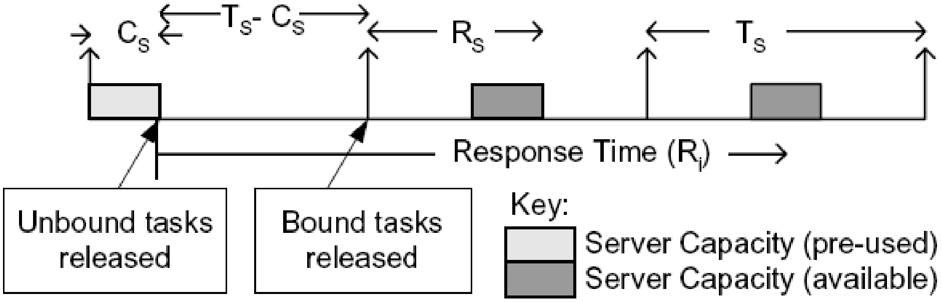

Figure 5 Critical Instant for an unbound task

In the general case of an unbound task the critical instant occurs as follows:

1. The server's capacity has been exhausted by lower priority tasks as early in its period as possible.
2. The unbound task of interest and all higher priority *unbound* tasks in the application arrive just after the server's capacity has been exhausted.
3. All higher priority *bound* tasks in the application arrive at the start of the server's next period.
4. The server's capacity is replenished at the start of its next period, however further execution of the server is then delayed for as long as possible due to interference from other higher priority servers.

It is not immediately obvious that the critical instant should involve release of higher priority bound tasks after an interval $(T_S - C_S)$ from the start of the busy period. However we can see that this must be the case as follows: Let τ_j be a high priority bound task and *t* the time of the first release of the server in the busy period of the unbound task τ_i, that is under analysis. There are two possibilities to consider:

1. There is no priority level *j* computation outstanding at the start of the priority level *i* busy period. In this case, the maximum interference due to task τ_j occurs when it is released simultaneously with the first replenishment of the server at time *t*.
2. There is priority level *j* computation outstanding at the start of the priority level *i* busy

period. In this case all the time from the previous release of τ_j constitutes a priority level *j* and hence also a priority level *i* busy period. Thus we may move the release of τ_i back to $(T_S - C_S)$ prior to the release of τ_j without altering the time at which τ_i completes execution, thus extending the length of the busy period for τ_i. However, note that we now have the situation described in 1.

Repeating the above argument for each high priority bound task shows the definition of the critical instant given above to be correct.

Using this definition we can extend equation (1) to account for the load due to higher priority bound tasks.

$$L_i(w) = C_i + \sum_{\substack{\forall j \in hp(i) \\ bound}} \left\lceil \frac{w - (T_S - C_S)}{T_j} \right\rceil C_j + \sum_{\substack{\forall j \in hp(i) \\ unbound}} \left\lceil \frac{w}{T_j} \right\rceil C_j \tag{11}$$

This equation is then combined with the recurrence relation below, from equation (6), to determine the response time of an unbound task that is subject to interference from higher priority bound and unbound tasks.

$$w_i^n = L_i(w_i^n) + \left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil (T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil T_S - C_S\right)\right) + J_X}{T_X} \right\rceil C_X \tag{12}$$

3.2 Bound v Unbound

In this section, we show that for any task where we have the choice of synchronizing its release with the replenishment of the server (making the task *bound*) or not doing so (making the task *unbound*), then for reasons of schedulability it is always preferable to bind the task to the release of its server.

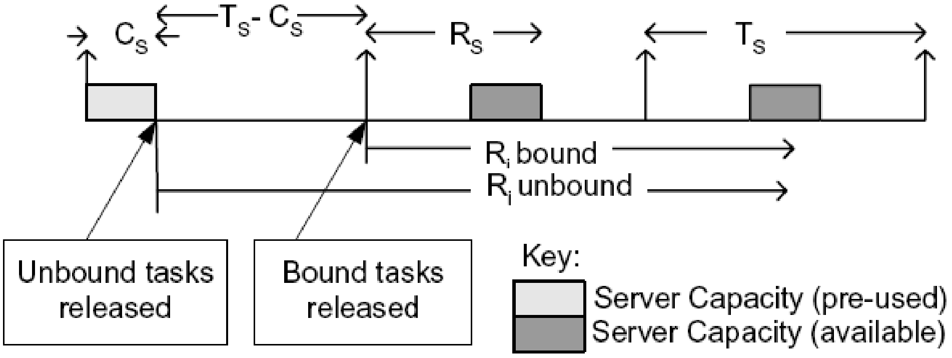

Figure 6 Critical Instant for a task (bound or unbound)

Figure 6 illustrates the worst-case response times for a task depending on whether it is bound or unbound. As no server capacity is available in the first gap of length $(T_S - C_S)$ between the release of the bound and unbound tasks, the busy period ends at exactly the same point irrespective of whether the task is bound or unbound. Hence when the task is bound, its worst-case response time is exactly $(T_S - C_S)$ less than if it was unbound, (assuming that the task is schedulable in both cases).

Further the interference that the task can have on lower priority tasks can be no greater if the task is bound rather than unbound. The interference can however be less due to its release later in the busy period. (Compare the interference terms for bound and unbound tasks in both equation (7) and equation (11)).

Overall, 'bound' can be said to *dominate* 'unbound'. If a task is schedulable when it is unbound, then it is guaranteed to also be schedulable if it is bound to the release of its server, further its response time will be reduced by exactly $(T_S - C_S)$. It may also reduce the response times of lower priority tasks.

3.2.1 Release Jitter Model

Comparing the busy periods for bound and unbound tasks, we see that in terms of schedulability, an unbound task is identical to a bound task that has a *release jitter* of $(T_S - C_S)$. We can therefore use the equations presented for bound tasks to determine the response times of both bound and unbound tasks.

The load at priority level *i* and above that becomes ready to execute in an interval *w* is given by:

$$L_i(w) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w + J_j}{T_j} \right\rceil C_j \tag{13}$$

where J_j is the maximum release jitter of task τ_j due to the operation of the server. This is zero for a bound task and $(T_S - C_S)$ for an unbound task.

The length of the priority level *i* busy period required for the server to execute this load is given by:

$$w_i^n = L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right) + J_X}{T_X} \right\rceil C_X \tag{14}$$

Recurrence starts with a value of $w_i^0 = C_i + \left(\left\lceil \frac{C_i}{C_S} \right\rceil - 1 \right)(T_S - C_S)$ and ends either when $w^{n+1} = w^n$ in which case $w^n + J_i$ gives the task's worst-case response time or when $w^{n+1} > D_i - J_i$ in which case the task is not schedulable. On each iteration of the recurrence relation, a new value of $L_i(w_i^n)$ is first calculated using equation (13).

3.3 Task Priority Ordering

In this section, we consider the priority ordering of tasks *within* each application. As the processing time made available to the tasks is a monotonic non-decreasing function of time, then the optimal priority assignment policies for applications scheduled on a single processor apply. Hence:

1. For a set of independent tasks that are: bound to the server, may be released simultaneously, have $C_i \le D_i = T_i$ and do not voluntarily give up the processor, then rate monotonic priority assignment is optimal [4].
2. Relaxing the above assumptions to allow tasks to have deadlines less than their periods; deadline monotonic priority assignment is optimal [23].
3. Further relaxing the restrictions to allow unbound as well as bound tasks then *deadline minus jitter* monotonic priority assignment is optimal. Here the task τ_i with the smallest value of $D_i - J_i$ is given the highest priority, where $J_i = 0$ for a bound task and $J_i = T_S - C_S$ for an unbound task. (Note we assume that $D_i \le T_i$ for every task τ_i).
4. For bound and unbound tasks with more complex requirements such as arbitrary deadlines, offsets and deadlines prior to completion then the optimal priority assignment algorithm [11] developed by Audsley may be used to determine the optimal priority ordering. It should be noted that the analysis given in this report is not applicable to such systems and would need to be extended.

Example

The following simple example illustrates the use of ‘deadline minus jitter’ monotonic priority ordering. Consider a single server HP, with a capacity of 5, a period of 20 and a response time of 5. This server is used to execute a task set, with the two highest priority tasks given in the table below.

Task	***Type***	C_i	T_i	D_i	J_i	D_i -J_i
τ_A	bound	5	40	25	0	25
τ_B	unbound	5	50	35	15	20

If task τ_A is assigned a higher priority than τ_B, then the response times of the two tasks are 5 and 40 respectively, with τ_B missing its deadline. The alternative ‘deadline minus jitter’ monotonic ordering gives response times of 20 for τ_B and 25 for τ_A, with both tasks meeting their deadlines.

3.4 Blocking

The analysis we have derived so far considers only independent tasks. In this section, we consider the effects of tasks accessing shared resources. Resources may be shared either:

1. Locally: between the tasks of a single application.
2. Globally: between tasks in different applications.

3.4.1 Locally shared resources

In [1] Kuo and Li showed that if tasks share resources that are utilised strictly within a single application according to the Stack Resource Policy (SRP) [12] then task schedulability analysis may be simply extended to account for blocking equal to the maximum time that any lower priority task within the application locks a resource shared with the task of interest or a task of higher priority. Here it is assumed that the server would simply suspend execution of a task when its capacity is exhausted even if that task currently has a resource locked. This scheme is however inappropriate for sharing mutually exclusive resources between applications as suspension of a task with a global resource locked would lead to unacceptability long periods of priority inversion.

3.4.2 Globally Shared Resources

In 1995 Ghazalie and Baker [15] analysed the effect of access to mutually exclusive globally shared resources on schedulability for the case of a single server. Kuo and Li [1] and Niz *et al* [14] later addressed the problem of sharing global resources within the context of hierarchical fixed priority pre-emptive scheduling.

In [1] Kuo and Li introduced a common server for all globally shared resource accesses. This has the disadvantage that as more tasks are added that share global resources so the common server's capacity must be made larger: its capacity is effectively the sum of the lengths of the critical sections in each task, whilst its period is the GCD of task periods. Accommodating such a server has a significant impact on system schedulability.

In [14], Niz et al presented the *multi-reserve PCP* scheme. This scheme permits resources to be shared between tasks executed by different servers. With the multi-reserve PCP scheme, a special 'ceiling priority' reserve is created for each application task that requires access to a specific shared resource. This reserve effectively has a period equal to that of the task and a capacity of C_i^R, where C_i^R is the maximum time that the task spends accessing the shared resource. The priority of each special reserve is strictly higher than that of any application tasks that access the resource. A common server is used to permit access to the special reserves associated with a single resource, with admission control ensuring that only one task can access a reserve through the common server at any one time. This effectively limits priority inversion in the same way as the Stack Resource Policy.

Although [14] extends the Stack Resource Policy to hierarchical systems, the schedulability of such systems is not fully addressed. In simple fixed priority pre-emptive systems (e.g. a single application) using the Stack Resource Policy to control resource access minimizes priority inversion whilst leaving interference due to higher priority tasks unchanged. In hierarchical systems, priority inversion is again minimized using the Stack Resource Policy however there is also an effect of increased interference.

Example of increased interference:

The following example illustrates how resource sharing in a hierarchical system results in increased interference on lower priority servers and tasks as well as blocking effects on higher priority applications.

The example system comprises two Deferrable Servers, HP and LP with capacities and periods given in the table below.

Server	C_S	T_S	J_S	R_S
HP	3	8	5	3
LP	4	12	8	10

Let us consider the worst-case response time of the highest priority unbound task in each application. These tasks are shown in the table below:

Task	***Server***	C_i	T_i	R_i
A	HP	6	50	16
B	LP	4	100	18

The response time of these tasks, with no resource accesses, is illustrated by the timelines in Figure 7 below.

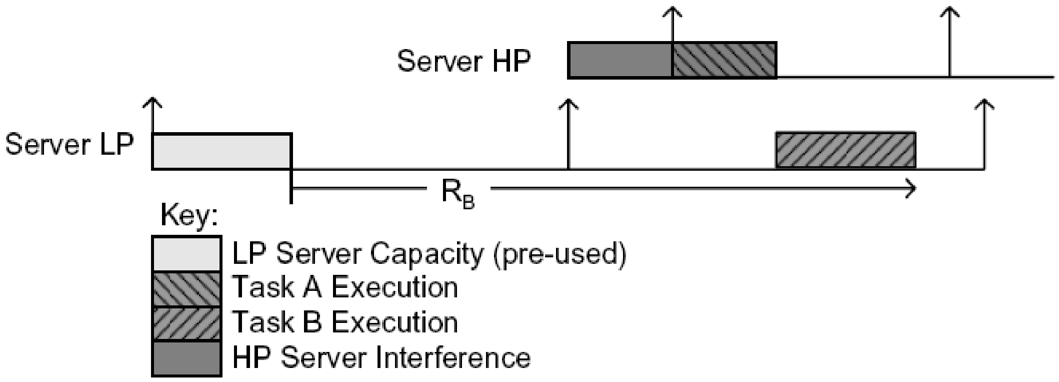

Figure 7

Now consider the situation where a global resource is shared between tasks A and B. Assuming that the resource is locked for 2 units of time by each task; one possible scenario is shown in Figure 8. Before the HP server's capacity is exhausted, task A locks the shared resource. Now either the resource access can be completed increasing the response time of server LP and task B, or alternatively task A could be suspended with the resource locked, in which case task B would end up blocked until after the next invocation of server HP. Either way, the response time of tasks handled by the *lower priority* server are increased as a result of the shared resource. Note that invoking a separate server to deal with the resource access results in exactly the same problem; the response time of the lower priority task is still increased.

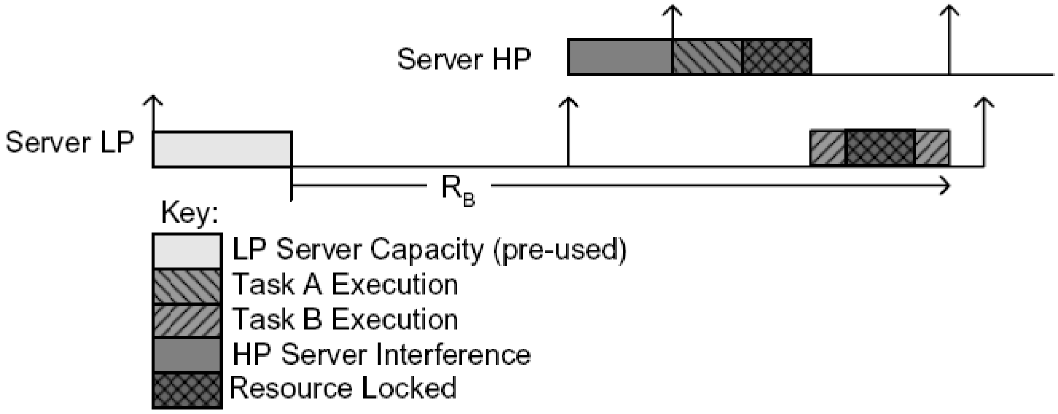

Figure 8

This problem was recognised by Ghazalie and Baker in their analysis of single server systems [15]. They proposed that if a server's capacity is exhausted within a critical section then the server should be allowed to overrun until the critical section ends. This overrun is limited to the maximum resource access time. To avoid a cumulative effect in subsequent server periods any overrun is then deducted from the capacity allocated at the start of the next server period.

3.4.3 Analysis of Blocking and Server Overruns

In this section, we provide schedulability analysis for global resource access under the following assumptions:

1. We assume that there is a set of globally shared resources *G*. Each task τ_i may access a global shared resource *r*, for at most an execution time $b_{r,i}$. This critical section is assumed to be less than the task's worst-case execution time and also less than the associated server's capacity, so $b_{r,i} < C_S$ and $b_{r,i} < C_i$. For a well-constrained application, $b_{r,i}$ will typically be much smaller than these values.
2. Whilst a task accesses a global shared resource the priority of its server is increased to a ceiling priority that is strictly higher than that of any server that executes a task that also accesses the same resource.
3. Whilst a task accesses a global shared resource, the priority of the task itself is also increased. If any task in a higher priority server accesses the same resource, then the priority of the task is increased to a level that is strictly higher than that of all the other tasks within its server. If there are no tasks within higher priority servers that access the same resource, then the priority of the task is increased to a ceiling priority strictly higher than that of any task within its server that accesses the same resource.
4. If a server's capacity is exhausted whilst executing a critical section then it continues to execute at the relevant ceiling priority until the critical section is completed.
5. If a server overruns then the capacity allocated to it at the start of the next server period is reduced by the amount of the overrun.

The longest critical section within an application executed by server *S* is given by:

$$B_S^{APP} = \max_{\forall r \in G}(b_{r,i} \mid i \in tasks(S))$$

where *tasks(S)* is the set of tasks handled by server *S*. The value of B_S^{APP} corresponds to the longest time that server *S* may overrun.

The longest time that server *S* can be blocked due to lower priority servers executing at a priority higher than *S* due to operation of the synchronization protocol is given by:

$$B_S = \max_{\forall Y \in lp(S)} (b_{r,i} \mid i \in tasks(Y), r \in global(S,Y))$$

where *lp(S)* is the set of servers with lower priority than server *S*, *tasks(Y)* is the set of tasks executed by server *Y* and *global(S,Y)* is the set of global resources shared between tasks executed by server *S* or higher priority servers and a task executed by server *Y*.

Server Schedulability

The worst-case effects on the schedulability of a server *S* due to global resource access occur as follows:

1. When server *S* is released, a lower priority server is running and the task that it is executing has just started accessing a global shared resource *r*. This critical section has a ceiling priority higher than that of server *S* and a length of B_S.

2. Once server *S* is released, all subsequent releases of servers of higher priority than *S* overrun by their maximum amount due to tasks entering critical sections. This means that the first invocation of each higher priority server, in the busy period of *S*, has an execution time of $C_X + B_X^{APP}$, whilst subsequent invocations have an execution time of C_X as their capacity is reduced by B_X^{APP} and they also overrun by B_X^{APP}. The additional interference due to this behaviour of the higher priority servers is given by:

   $$\sum_{\forall X \in hp(S)} B_X^{APP}$$

   where *hp(S)* is the set of servers with higher priority than server *S*.

Note, when considering the schedulability of server *S*, we only require that the server's normal capacity C_S be completed within its period. We do not need to include any overrun by server *S* in the analysis of *S* itself. This is because any critical section and hence overrun by *S* in one period leads to a reduction in the capacity available in the next period by exactly the amount of the overrun. Hence in the next period, any interference due to the overrun plus the capacity of the server is limited to a total of C_S. Again this must be completed by the end of the server period. There may of course be a further critical section and overrun at the end of this server period, however the same argument continues to apply.

Server schedulability can be determined by incorporating the blocking and interference factors into the appropriate recurrence relation:

$$w_i^{n+1} = C_S + B_S + \sum_{\forall X \in hp(S)} B_X^{APP} + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w_i^n + J_X}{T_X} \right\rceil C_X \qquad (15)$$

where J_X is the server's jitter, equal to $T_X - C_X$ in the case of a Deferrable Server. The recurrence relation given by equation (15) starts with $w_i^0 = C_S$ and ends when either $w_i^{n+1} = w_i^n$ in which case w_i^{n+1} gives the worst-case response time of the server or when $w_i^{n+1} > T_S$ in which case the server is unschedulable.

An alternative formulation is possible if we relax the rule that any server overruns are deducted from the subsequent replenishment capacity. In this case, each server invocation may overrun by

B_X^{APP}. Effectively the schedulability analysis is formulated as if each server had a capacity of $C_X + B_X^{APP}$ although only C_X can be guaranteed for task execution.

$$w_i^{n+1} = C_S + B_S^{APP} + B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w_i^n + J_X}{T_X} \right\rceil (C_X + B_X^{APP}) \tag{16}$$

Although equation (16) leads to longer response times than equation (15), it may be preferable for systems with very short critical sections, to simply allow for server overruns without the additional overheads of monitoring the overrun and adjusting the subsequent capacity replenishment.

Task Schedulability

Task schedulability is dependent on both server blocking due to global resource access and also task blocking due to local resource access.

The worst-case blocking experienced by a task τ_i due to local resource access under the operation of the Stack Resource Policy is given by B_i, where B_i is the longest time for which a task of lower priority than *i* executes with a local resource locked that is shared with task τ_i or a task of higher priority than *i*.

The worst-case effects on the schedulability of a task τ_i within the application executed by server *S* due to both local and global resource access occurs as follows:

1. Immediately prior to the final period of server *S* that will complete execution of task τ_i, a lower priority server is running and the task that it is executing has just started accessing a global shared resource *r*. This critical section has a ceiling priority higher than that of server *S* and a length of B_S.

2. In the final period of server *S* that will complete execution of task τ_i, all releases of servers of higher priority than *S* overrun by their maximum amount due to tasks accessing globally shared resources. Again this means that the first release of each server has an execution time of $C_X + B_X^{APP}$, whilst subsequent releases have an execution time of C_X as their capacity is reduced by B_X^{APP} and they also overrun by B_X^{APP}. The additional interference due to this behaviour of the higher priority servers is given by:

$$\sum_{\forall X \in hp(S)} B_X^{APP}$$

3. Either:

   The release of server *S* immediately prior to the release of task τ_i overruns by B_S^{APP} due to global resource access. This means that the following replenishment capacity will be reduced by B_S^{APP}.

   Or:

   The release of server *S* immediately prior to the release of task τ_i ends just after a lower priority task τ_j, within server *S*, has locked a locally shared resource that is also accessed by task τ_i or a task of higher priority. The length of this local resource access is B_i, the longest time for which any lower priority task executes with a locally shared resource locked that is also accessed by task τ_i or a task of higher priority.

These blocking and interference factors can be added into equation (13) and equation (14) to calculate the worst-case response time of bound and unbound tasks. The blocking and interference in the final server period that completes execution of the task is increased by the factors described in points 1 and 2 above. Further, the load that the server must execute to complete the task is also effectively increased by the maximum of the two factors described in point 3.

Incorporating these factors, the load at priority level *i* and above that becomes ready to execute in an interval *w* is given by:

$$L_i(w) = \max(B_i, B_S^{APP}) + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w + J_j}{T_j} \right\rceil C_j \qquad (17)$$

where J_j is the maximum release jitter of task τ_j due to the operation of the server. This is zero for a bound task and $(T_S - C_S)$ for an unbound task.

The length of the priority level *i* busy period required for the server to execute this load is given by:

$$\begin{aligned} w_i^n &= L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + \\ & B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} B_X^{APP} + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right) + J_X}{T_X} \right\rceil C_X \end{aligned} \qquad (18)$$

Recurrence starts with a value of $w_i^0 = 0$ and ends either when $w^{n+1} = w^n$ in which case $w^n + J_i$ gives the task's worst-case response time or when $w^{n+1} > D_i - J_i$ in which case the task is not schedulable. On each iteration of the recurrence relation, a new value of $L_i(w_i^n)$ is first calculated using equation (17).

Again an alternative formulation is possible if we relax the rule that any server overruns are deducted from the subsequent replenishment capacity. Formulation of the appropriate equations is left as an exercise to the reader.

It is evident from the above analysis that the use of long critical sections in hierarchical fixed priority pre-emptive systems can have a large cumulative impact on the schedulability of both servers and application tasks.

Development of effective protocols for global resource sharing in hierarchical fixed priority pre-emptive systems remains an open area for research.

We note that alternative approaches have been developed for hierarchical systems with a somewhat different set of assumptions scheduled using dynamic priorities [16][17]. These approaches avoid server overrun either by revising server parameters prior to entering critical sections [16] or by executing critical sections using the bandwidth of blocked servers [17]. It remains an open question whether an approach based on avoiding server overruns would be successful in hierarchical fixed priority systems.

3.5 Evaluation of Schedulability Analysis

The exact analysis derived in section 3.1.2 enables us to determine the response time and hence schedulability of application tasks given a known set of server parameters. Alternatively it can also be used as the basis of a simple binary search to determine the minimum server capacity required to schedule application tasks given a known server period.

In this section we report the results of experiments used to examine the relationship between minimum server capacity, server replenishment period and overheads. We also compared different schedulability analysis methods in terms of the minimum server capacity they deem necessary to schedule an application.

The application task set used for our initial empirical investigation comprises three unbound tasks listed in the table below.

Priority	**Execution Time**	**Period**	**Deadline**
1	5	50	50
2	7	125	125
3	6	300	300

Table 1

Simulations were performed for a simple system of two Deferrable Servers. The higher priority server (HP) had a fixed capacity of 4 and period of 10 time units. The lower priority server (LP) was responsible for executing the tasks listed in Table 1 above. The period of the lower priority server was varied during the simulation and the minimum capacity commensurate with a schedulable system was calculated for each period.

3.5.1 Effect of overheads

There are two reasons why it is important to consider the effects of overheads when examining the choice of server periods and capacities. Firstly, the server implementation in any real system is likely to incur significant overheads. Secondly, from a theoretical standpoint, ignoring overheads leads to the conclusion that the optimal selection of server parameters involves selecting infinitesimally small values for the servers’ periods and capacities.

The effects of server context switch overheads can be modeled by considering the server’s capacity to be consumed first by context switch overheads and then by task execution. This is a safe if potentially slightly pessimistic approach to modeling overheads.

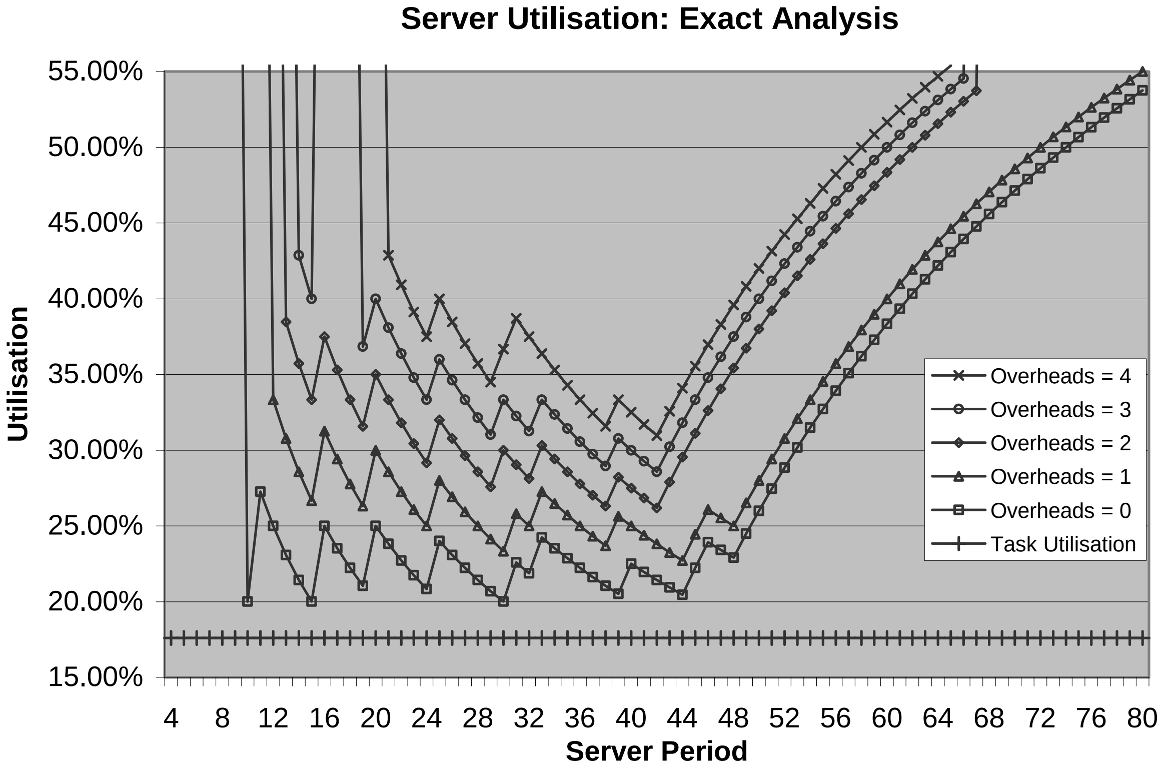

Figure 9 Overheads: Exact Analysis

Figure 9 illustrates the effect of server context switch overheads. The graph plots the minimum utilisation of the LP server necessary to achieve a schedulable system against the server's period for a variety of levels of overheads. The total utilisation of the application tasks is 17.6% represented by the horizontal line immediately below the jagged lines depicting server utilisation.

From the graph, it is clear that overheads markedly increase the required server utilisation at short server periods. Hence, whilst the optimum server period is 10,15 or 30 without taking account of overheads, it is 42 or 44 when the effects of overheads are included.

As server utilisation is simply C_S / T_S, moving along a line on the graph from left to right, the server utilisation decreases with increasing period, until an increase in server capacity is required at which point it increases sharply, giving the characteristic saw-tooth shape.

It is interesting to note that the system remains schedulable for server replenishment periods that exceed the deadline of the highest priority task. This is somewhat counter-intuitive, however it is nevertheless correct. The server's relatively large capacity and short response time mean that it can schedule a task that has a shorter period than that of the server itself, as illustrated in Figure 10.

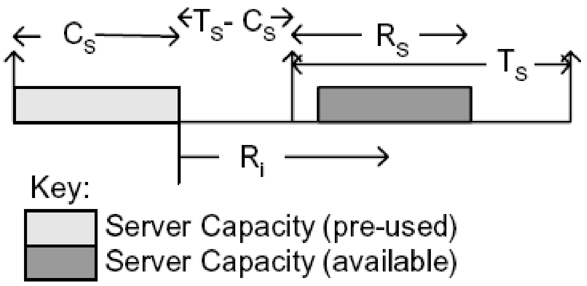

Figure 10

Once the server's period exceeds that of the highest priority task, each increase in server period requires a corresponding increase in server capacity to keep the interval from task release to the start of the task being serviced constant and hence the task schedulable. As the server period increases so its capacity is forced to increase with the server utilisation tending towards 100%.

3.5.2 Comparison of Analysis Methods

Figure 11 below, illustrates the minimum utilisation of the low priority server that was deemed necessary to schedule the task set from Table 1 using (1) the exact analysis presented in this report and (2) the analysis of Saewong et al [5] which models interference in the final server period as $(T_S - C_S)$.

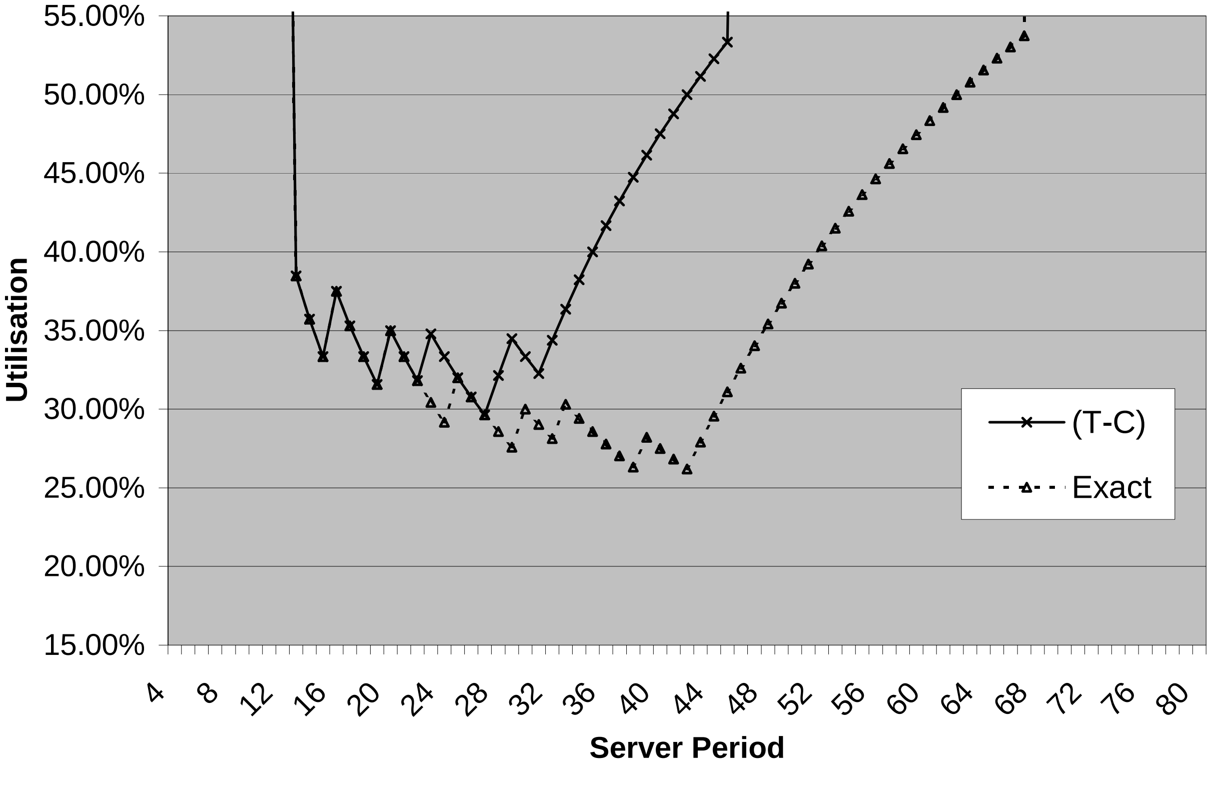

Figure 11 Comparison of Analysis Methods

The graphs assume server context switch overheads of 2 time units. Hence the line for the exact analysis is the same as the 'overheads = 2' line in Figure 9

It can be seen from the graph that the exact analysis presented in this report achieves the lowest server utilisation of 26.19% (period = 42, capacity = 11). By comparison, the analysis of Saewong et al [5] achieves a server utilisation of 29.63% (period = 27, capacity = 8). The overall utilisation of the task set is 17.6% and the overheads are 4.76% (period = 46) or 7.41% (period = 27).

This simulation provides a clear example of the difference that more precise analysis can make to the feasibility of a system. Using the exact analysis presented in this report means that a server can be used which effectively requires 3.44% less processor utilisation, equivalent to 19.5% of the actual task load. Further the ability to use a longer server period reduces the time wasted due to server context switch overheads from 7.41% down to 4.76%.

3.6 Choice of Server Algorithm

The critical instant described in section 3.1.1 and the exact schedulability analysis given in this report is applicable to Periodic, Sporadic and Deferrable Servers.

For all three server algorithms, the critical instant occurs when the server's capacity is exhausted as early as possible in its period, then there is a delay of $(T_S - C_S)$ before the server's capacity is replenished with subsequent capacity replenishments taking place after a period of T_S.

The only differences in analysis are as follows:

- When calculating interference from higher priority servers, Periodic Servers and Sporadic Servers have a jitter of zero whilst Deferrable Servers are treated as having a jitter equal to $(T_S - C_S)$ [10].
- Tasks cannot be bound to a Sporadic Server due to its non-periodic behaviour in anything other than the worst-case scenario.

3.6.1 Periodic v Deferrable Server

Inspection of the exact analysis (equation (10) and equation (12)) shows that Periodic Servers *dominate* Deferrable Servers. That is there are no systems comprising a set of hard real-time application task sets that can be scheduled using a set of Deferrable Servers that cannot also be scheduled using an equivalent set of Periodic Servers with the same periods and capacities. There are however many sets of applications that can be scheduled using Periodic Servers that cannot be scheduled using Deferrable Servers. This is because the Deferrable Server has a drawback compared to a Periodic or Sporadic Server when used to service hard real-time tasks; the effect of *back-to-back* hits referred to earlier. Using a set of Deferrable Servers results in the lower priority servers receiving back-to-back interference from those of higher priority, increasing their response times and hence degrading the systems ability to schedule hard real-time applications.

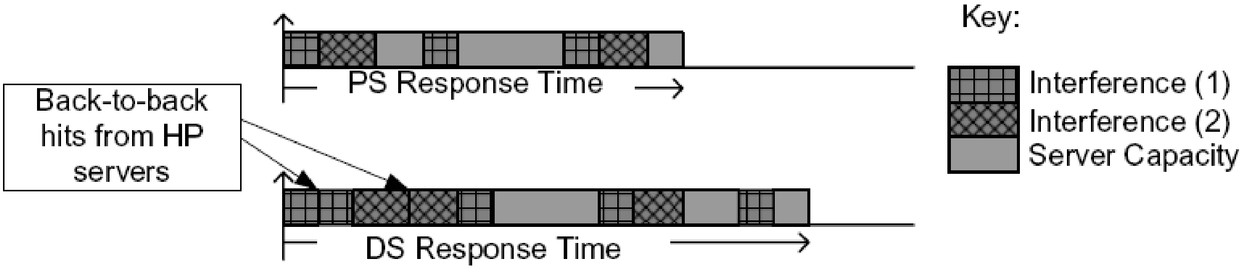

Figure 12 Longer response times due to Deferrable Servers

Figure 12 illustrates the longer worst-case response times of a system of Deferrable Servers due to back-to-back hits.

We repeated the simulations described at the start of section 3.5, this time for a system of two Periodic Servers. Again the higher priority server (HP) had a fixed capacity of 4 and period of 10 time units. The lower priority server (LP) was responsible for executing the tasks listed in Table 1. The period of the lower priority server was varied during the simulation and the minimum capacity commensurate with a schedulable system was calculated for each period.

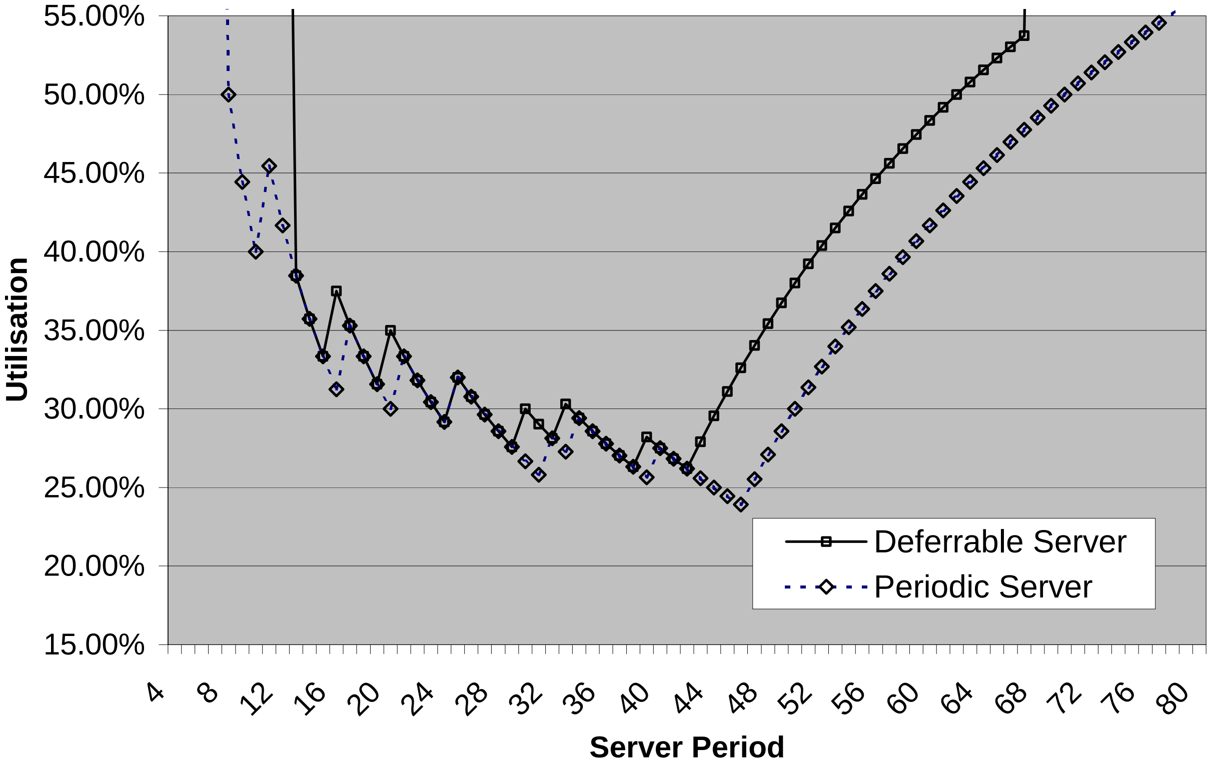

Figure 13 Comparison of Server Algorithms

Figure 13 compares the utilisation of server LP, required to schedule the task set when both server are (1) Deferrable (2) Periodic. As expected, the Periodic Server approach dominates the Deferrable Server algorithm. For short server periods (of 8-12 time units), using Periodic Servers results in a schedulable system whereas using Deferrable Servers does not. This is a direct consequence of the back-to-back hit phenomenon. The minimum required server utilisation is 23.91% for the Periodic Server approach (period = 46, capacity = 11) compared to 26.19% for the Deferrable Server approach (period = 42, capacity = 11).

We performed some additional simple experiments to further illustrate the effect that choice of server method has on system schedulability. These experiments simulated simple systems comprising two applications and hence two servers with 3 tasks each. The systems had overall utilisation values that varied from 40% to 95%. For each utilisation level, 100 task sets were generated. For each task set, an exhaustive search of server period combinations was conducted with the aim of determining whether Deferrable or Periodic Servers could schedule the task set. The results are presented in Figure 14 below.

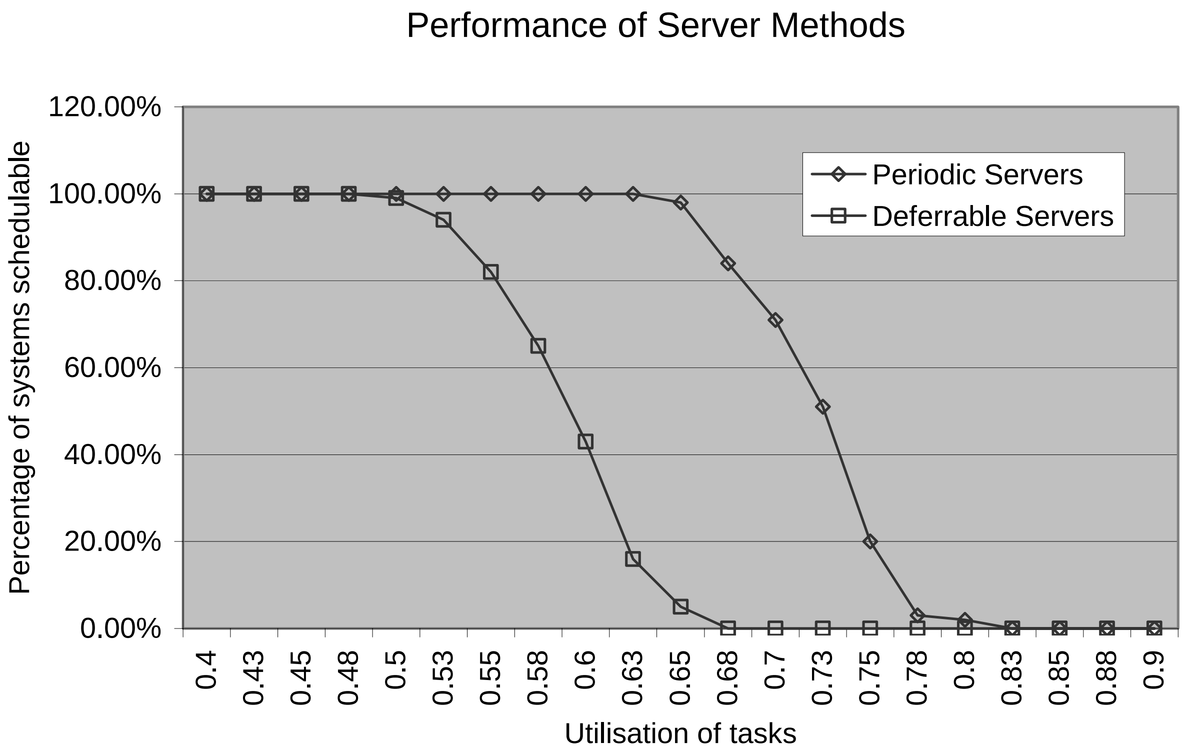

Figure 14 Periodic Servers versus Deferrable Servers

Figure 14 plots the percentage of task sets that are schedulable at each utilisation level. It is evident from the graph that using Periodic Servers is preferable to using Deferrable Servers in terms of being able to schedule hard real-time tasks.

3.6.2 Periodic Server v Sporadic Server

The same critical instant and exact schedulability analysis applies to systems comprising Sporadic Servers as it does to systems of Periodic Servers, with one key difference: tasks cannot be bound to Sporadic Servers and must therefore always be treated as unbound.

This means that Periodic Servers *dominate* Sporadic Servers. That is there are no systems comprising a set of hard real-time application task sets that can be scheduled using a set of Sporadic Servers that cannot also be scheduled using an equivalent set of Periodic Servers with the same periods and capacities.

Further, the Sporadic Server is far more complex to implement than the Periodic Server and so in practice the performance of a system based on Sporadic Servers would be inferior to that of a Periodic Server based system due to increased overheads.

Binding tasks to their server can improve system schedulability, effectively reducing the server utilisation required to schedule the task set. To illustrate the effect of making tasks ‘bound’ rather than ‘unbound’ we performed simulations of a simple system of two Periodic Servers. The higher priority server (HP) had a fixed capacity of 10 and a period of 32 time units. The lower priority server (LP) was responsible for executing the tasks listed in Table 2 below. The period of the lower priority server was varied during the simulation and the minimum capacity commensurate with a schedulable system was calculated for each period.

Priority	Exec. Time	Period	Deadline
1	8	160	100
2	12	240	200
3	16	320	300
4	24	480	400

Table 2

In this case, the task periods and deadlines were chosen to emphasize the effect of having tasks bound to the release of the server. The task periods were chosen such they would be harmonics of a number of different server periods.

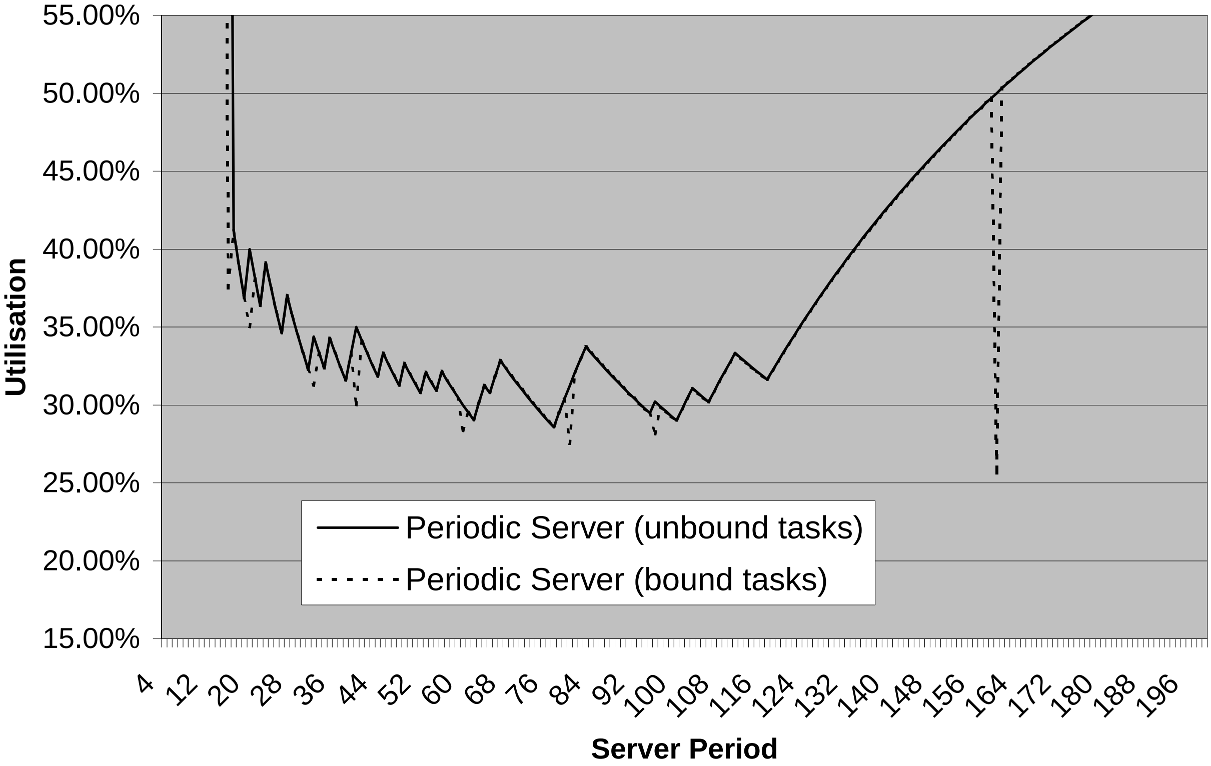

Figure 15 Periodic Servers: bound and unbound tasks

Figure 15 shows the different server utilisations required to schedule the task set for a range of server periods. The two lines on the graph are both for Periodic Servers, however the dashed line illustrates the effect of binding tasks to the server whenever a task's period is an exact multiple of that of the server. This results in improvements in task response times and hence a system which is schedulable for lower server capacities. This is apparent from the graph for server periods of 16, 20, 32, 40, 60, 80, 96 and 160.

We note that although 120 is a harmonic of two of the task periods, there is no improvement in schedulability or reduction in server capacity when this value is selected for the server period. This is because the highest priority task (period 160) remains unbound for a server period of 120.

Further, this task has a deadline of 100, so with an LP server period of 120, and a HP server capacity of 10, the LP server is forced to have a capacity of at least 38 to schedule the highest priority task alone. As this server capacity is sufficient to schedule all the other tasks as well, there is no difference in server utilisation (31.6%) between the bound and unbound cases.

Treating all tasks as unbound results in a minimum server utilisation of 28.57% for a server period of 77. Permitting tasks to be bound to the server reduces this minimum utilisation to 25.63% for a server period of 160.

We note from the shape of the graph that the problem of selecting the optimal server period does not lend itself to being easily solved via generic search techniques. The optimal server period, 160 in this case, is a single excellent solution surrounded by neighbouring solutions that are very poor.

3.6.3 Periodic Server Behaviour

We note that the analysis of Periodic Servers in the previous sections assumes that the Periodic Servers can service tasks that arrive after the start of the server's period. Effectively server capacity of at least $(C_S - t)$ is assumed to remain at time $t \le C_S$ from the start of the server period. This is a sensible model for many hierarchical systems, as each of the applications running on the system will typically contain an idle task that executes at a background priority level when all the application's other tasks are inactive. This idle task is often used to implement built-in-tests of the application and its memory areas and some types of watchdog functionality.

An alternative behavior for a Periodic Server is for the server's capacity to be discarded at the start of its period if no tasks are ready to use it. We refer to these servers as *Discarding Periodic Servers*. Discarding capacity in this way reduces the server's ability to guarantee hard real-time applications. With this server behaviour, a critical instant occurs when at the start of the server's period its capacity is discarded and then the task of interest is released along with all other tasks of higher priority in the application. Effectively the busy period extends C_S further than when the server consumes its capacity, for example via an idle task.

Figure 16 shows how the busy period extends from the start of the server period. (Compare with Figure 1).

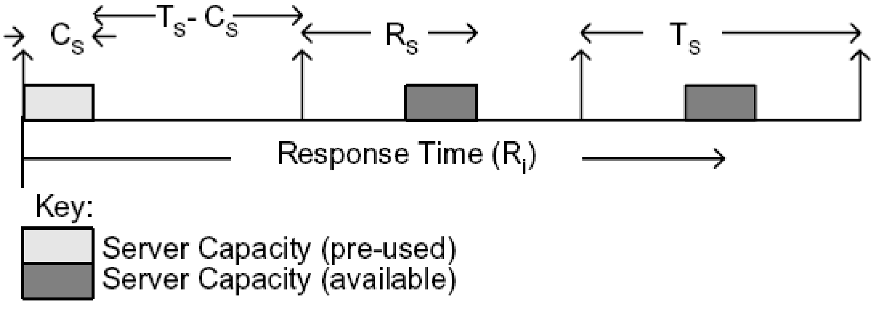

Figure 16 Critical Instant when Capacity Discarded

The amount, by which the busy period for a task is extended, with respect to the busy period under a Periodic Server that does not immediately discard its capacity, is as follows:

Discarding Periodic Servers: At least C_S : the associated server's capacity.

Deferrable Servers: At least $\sum_{\forall X \in hp(S)} C_X$: the sum of the capacities of all higher priority servers (see section 7.1)

This means that when we consider simple systems of just two servers with roughly the same capacities, we would expect the performance of the Deferrable and Discarding Periodic Server approaches to be similar. This is borne out in experiments on two-server system that show using Discarding Periodic Servers results in performance that is broadly similar to the Deferrable Servers. Note however that there are systems that are schedulable using Discarding Periodic Servers that are not schedulable using Deferrable Servers and vice-versa. Neither method dominates the other.

We note however that as the number of applications in the system increases, the performance of the Deferrable Server approach degrades. The table below gives an indication of this degradation in performance. It shows the response time of a hard real-time task requiring 5 units of computation time executing under each of a set of 6 servers. Each server is assumed to have a period of 100 and capacity of 10.

	Task worst-case response time		
Server Priority	**Periodic Servers**	**Discarding Periodic Servers**	**Deferrable Servers**
1	95	105	95
2	105	115	115
3	115	125	135
4	125	135	155
5	135	145	175
6	145	155	Not Schedulable

Note that the system of six Deferrable Servers is not schedulable as in the worst-case; the lowest priority server receives back-to-back hits from each of the other five higher priority servers, filling its entire period with interference.

3.6.4 Recommended Server Algorithm

Both Deferrable and Sporadic Server algorithms were designed to provide responsive scheduling for soft aperiodic tasks in single application systems, whilst in the worst-case appearing to be similar to a periodic hard real-time task in terms of their effects on system schedulability. It is perhaps therefore no surprise that these mechanisms are no better than the much simpler Periodic Server approach when it comes to dividing up processor capacity between a number of hard real-time applications. It is a very different problem from the one for which they were designed.

Although we can recommend the use of Periodic Servers when the sole criteria is guaranteeing the deadlines of hard real-time application tasks, this does not mean that there is no place for Deferrable or Sporadic Servers in hierarchical systems. When quality of service (QoS) is also an issue, it may be appropriate to use a different approach. This is discussed in section 3.7.

3.7 Flexible Scheduling and Gain Time

So far we have only considered simple systems where each application comprises a set of hard-real-time tasks and some lower priority soft real-time tasks, possibly including an idle task. The only criterion of interest has been guaranteeing that hard real-time tasks will complete prior to their deadlines.

In more complex systems, as well as guaranteeing hard real-time task deadlines, quality of service (QoS) issues are important. In general, improved quality of service may be provided by:

1. Responsive execution of soft real-time tasks.
2. Executing optional components or alternative versions of hard real-time tasks that improve the quality of the results produced.
3. Sharing execution time between tasks to ameliorate the effects of an execution time overrun.

To provide enhanced quality of service, flexible scheduling techniques are required that identify spare processor capacity and make it available in a timely manner. There is a considerable body of research into techniques and mechanisms for flexible scheduling in fixed priority preemptive systems that could potentially be applied to hierarchical systems.

Examples include:

1. Identifying and exploiting *gain time* [22], which becomes available when tasks execute for less than their worst-case execution times.
2. Using server based approaches within each application to responsively schedule soft real-time tasks.
3. Capacity sharing [20] and History Rewriting [21] between tasks in the same application to ameliorate the effects of overruns and exploit spare capacity to execute optional components.
4. Capacity sharing between applications.
5. Dual-Priority scheduling [13].

Flexible scheduling in hierarchical systems is a broad and interesting area for future research. It is however beyond the scope of this report.

3.8 Open and Closed Systems and Online analysis

Hierarchical systems may be either *closed* where all the hard real-time application tasks are known prior to the system executing or *open* where new applications can be added at run-time. Examples of closed systems include most automotive electronics and avionics whilst open systems include personal digital assistants (PDAs) and telecomms.

In closed systems all schedulability analysis is performed offline and exact schedulability analysis is appropriate. In open systems schedulability analysis may be performed online as an acceptance test prior to guaranteeing the execution of an application. Here the computational overhead of the schedulability tests may be of some concern. However unless there is a very short deadline on accepting an application this is still unlikely to mean that methods other than exact analysis are appropriate.

For comparison purposes the schedulability tests required to admit a new application are as follows:

Schedulability Tests required for:	New and all lower priority Servers	All tasks in new server	All tasks in all lower priority servers
Exact Analysis	Yes	Yes	Yes
$(R_S - C_S)$ **Method**	Yes	Yes	Yes
$(T_S - C_S)$ **Method**	Yes	Yes	No

All the schedulability tests are pseudo-polynomial in their computation time, with a dependency on the task and server periods. The exact analysis can be expected to take more computation time than the other methods due to iteration over the higher priority servers in the interference term and the requirement to check schedulability of all tasks executed by lower priority servers.

Modeling interference from higher priority servers as $(R_S - C_S)$ removes the need to iterate over the higher priority servers when calculating task response times. However it is still necessary to check schedulability of all the tasks in lower priority applications as an increase in the response time of a server has an impact on the response time of its associated tasks.

Finally, the $(T_S - C_S)$ method will typically require the least computation time as the pessimistic assumptions made about interference mean that the only tasks that need to be checked for schedulability are those in the new application. Provided that all the existing servers remain schedulable then their associated tasks will be schedulable also.

The approximate analysis methods may of course reject applications that are in fact schedulable.

We note that the above discussion assumes that any new applications are independent of, and so do not share resources with existing applications and also that the parameters of existing servers are not changed when a new application is added.

4 Server Parameter Selection

In this section, we consider the problem of server parameter selection.

The overall problem may be stated as follows: Given a set of applications to be scheduled, with each application allocated a single server, what is the *optimum* set of server parameters (priority, period and capacity) that leads to a schedulable system whilst preserving the maximum remaining processor utilisation.

$$1-\sum_{\forall X \in servers} \frac{C_X}{T_X} \tag{19}$$

There are two sets of schedulability constraints on any given system.

1. The servers must have worst-case response times that do not exceed their periods. (Each server *S* must guarantee to provide capacity C_S in each of its periods T_S).

2. The tasks executed by the servers must have worst-case response times that do not exceed their deadlines.

The problem of server parameter selection can be generalized further by permitting more than one server to be used to handle each application (i.e. statically allocating the tasks from a single application to more than one server) this is however beyond the scope of our work.

4.1 Determining Server Capacities

In this section, we consider the sub-problem of determining server capacities *given* a known set of server priorities and periods. Given these parameters, we can use the following simple algorithm to derive the optimal set of server capacities for the set of server periods and priorities provided.

```
For each server in priority order highest first
{
     Binary search between 0 and the server period for
          the minimum capacity Z that results in the
          server and its tasks being schedulable.
     If no schedulable capacity found
     {
          exit system not schedulable
     }
     else
     {
          set the capacity of the server to Z
     }
}
```

This method works because:

1. The capacities of lower priority servers are not required when determining the schedulability of a higher priority server or the tasks that it services.
2. Any increase in the capacity of a higher priority server cannot decrease the response time of a lower priority server or the tasks it schedules. Hence increasing the capacity of a higher priority server beyond that determined by the above algorithm cannot lead to a lower priority server requiring less capacity to schedule its application tasks.

Hence the set of minimum server capacities calculated in descending priority order are optimal for the set of server priorities and periods selected.

This method of determining server capacities simplifies the overall problem a little. However the difficulty remains: how to select server priorities and periods?

We note that using a binary search to determine server capacities is possible using the exact schedulability analysis presented in section 3, however the binary search method fails to function correctly when the approximate ($R_S - C_S$) analysis is used. This happens because the worst-case response times of tasks *appear* as if they are not monotonically non-decreasing with respect to decreasing server capacity. This effect is due to the dependence of the interference term ($R_S - C_S$) on the server's response time and the fact that the interference term can increase more than the associated increase in the server's capacity. To appreciate this problem, consider the following two Deferrable Servers:

Server	$\boldsymbol{C_S}$	$\boldsymbol{T_S}$	$\boldsymbol{J_S}$	$\boldsymbol{R_S}$
HP	2	5	3	2
LP	7	20	13	15

According to the approximate ($R_S - C_S$) analysis, server LP can provide 2 units of task execution time in a worst-case response time of $(T_S - C_S) + (R_S - C_S) + 2 = 23$. However, if we reduce the capacity of server LP to 6, then its response time becomes 12 giving a worst-case response time of 22 for the same 2 units of task execution time. Both task response times are upper bounds on the actual values, however the non-monotonic behaviour prevents the use of binary search techniques. When calculated using the exact analysis, task response times are monotonically non-decreasing with decreasing server capacity, allowing binary search techniques to be used. The exact response times for this example are 19 when the server capacity is 7 and 20 when the server capacity is reduced to 6.

4.2 Determining Server Priorities

Section 7.2 in the Appendix discusses priority assignment policies based on considerations of server schedulability. In the case of Periodic Servers, rate-monotonic priority assignment (RMPA) is optimal whilst for a set of Deferrable Servers under certain conditions "*period plus capacity*" monotonic priority assignment is shown to be optimal. However, the analysis presented in the appendix only considers the schedulability of the servers themselves. Adding the constraint that the tasks executed by the servers must also be schedulable has a significant effect on server priority assignment. In particular when task schedulability is considered, empirical investigations show that RMPA is no longer optimal for Periodic Servers. Similarly, deadline-monotonic priority assignment based on the deadline of the shortest deadline task in each application is not optimal either.

For the sub-problem where server periods and capacities are known then a feasible priority ordering can be determined, if one exists, using a variation on the Optimal Priority Assignment Algorithm devised by Audsley [11]. This is possible because:

1. The specific priority ordering of higher priority servers has no effect on the schedulability of a lower priority server or the tasks that it executes.
2. The parameters selected for a low priority server have no bearing on the schedulability of the higher priority servers or tasks that they execute.

```
for each priority level, lowest first
{
     for each unallocated server
     {
          if the server and its tasks are schedulable at
          this priority level
          {
               allocate the server to this priority
               break (continue with outer loop)
          }
     }
     return unschedulable
}
return schedulable
```

Optimal Priority Assignment Algorithm

This algorithm requires *n(n+1)/2* tests of server and associated task schedulability compared to the *n!* potential server priority orderings.

Although the optimal priority assignment algorithm is guaranteed to find a schedulable priority ordering if one exists, it will not necessarily determine the priority ordering that results in the highest remaining processor utilisation.

4.3 Determining Server Periods

If the server priorities and capacities are fixed, then a set of server periods can be systematically derived. This is possible because:

1. The parameters selected for a low priority server have no bearing on the schedulability of the higher priority servers or the tasks that they execute.
2. The interference on lower priority servers and the tasks they execute is monotonically non-increasing with respect to increases in the period of each higher priority server.

This means we can perform a simple binary search to determine the maximum possible period for the highest priority server, commensurate with the server and its tasks remaining schedulable. Once this maximum period has been selected, this process can be repeated for the next highest priority server and so on, until a set of maximum server periods have been derived.

This set of maximum periods is optimal for the given set of server priorities and capacities, in the sense that the servers will have the minimum total utilisation and the system will be schedulable with this set of server periods if it is schedulable for any selection of server periods.

4.4 Overall parameter selection

Although it is possible to systematically derive one of the server parameters (priority, period or capacity) if the other two are fixed, this still leaves the problem of selecting the other two parameters.

Our experiments have shown that even if the problem is simplified by fixing server priorities, it is still difficult to find the combination of server periods and capacities required to achieve the minimum total utilisation. This is because the set of period and capacity values for each server that result in the minimum total utilisation (global optima), do not necessarily correspond to those values that result in the minimum utilisation for any of the servers taken individually (local optima). This can be seen in systems of just two servers. Typically the period-capacity pair that results in the minimum utilisation for the higher priority server has a long period and large capacity. However, as a consequence of the large capacity of the higher priority server, the period of the lower priority server has to be reduced, increasing its overall utilisation. In fact the lower priority server and its tasks may simply be unschedulable due to the large amount of interference from the higher priority server. Halving the period of the higher priority server increases its utilisation, as a result of overheads, but also typically allows the lower priority server to have a much longer period for the same capacity. Although a shorter period for the higher priority server results in a larger utilisation for that server this can be more than compensated for by a reduction in utilisation of the lower priority server.

Example:

Consider two Periodic Servers S_A and S_B. Each server has a single (unbound) hard real-time task to accommodate. The task parameters are given in the table below:

Task	$\boldsymbol{C_i}$	$\boldsymbol{T_i}$	$\boldsymbol{D_i}$	**Server**
τ_1	10	20	20	S_A
τ_2	4	24	24	S_B

Further, assume that server context switch overheads are 1 time unit and that the processor needs to provide each invocation of a server with this context switch time before it can execute its tasks.

Now consider the choice of server periods, assuming that S_A has the higher priority. The lowest utilisation for S_A occurs for a period of 20 and a capacity of 11 (55% utilisation). However, with these parameters for S_A there are no parameters for S_B that result in a schedulable system. To accomodate task τ_2, the period of S_B is constrained according to:

$$(T_B - C_B) + nT_B + C_A + C_B \le D_2$$

and so $(n+1)T_B \le 13$ where n+1 is the number of invocations of S_B that are used to service task τ_2. Also for S_B to be schedulable $T_B \ge C_A + C_B$. Thus possible periods for S_B are constrained to lie in the range 11 to 13 with a maximum possible capacity for S_B of 2. None of these parameter selections result in task τ_2 being schedulable.

However if we choose $T_A = 10$, $C_A = 6$ then S_A has a utilisation of 60% which is 5% greater than before. However, S_B is now just schedulable with $T_B = 9$, $C_B = 3$ (33.3% utilisation). The overall server utilisation is 93.3%. Note that the servers are in the reverse of rate-monotonic priority order.

4.5 Greedy Algorithms

In this section, we compare the performance of a greedy method of server parameter selection with that of optimal parameter selection.

The greedy algorithm proceeds as follows. For each server, highest priority first: scan through the range of potential server periods. For each possible server period, use a binary search to determine the minimum possible server capacity. Select the pair of server parameters (period and capacity) that provide the minimum utilisation for the server (local optima). This process is then repeated for each lower priority server in turn.

For comparison purposes an exhaustive search of possible server period combinations was used to determine the optimal selection of periods and capacities. This was possible for simple systems comprising just two applications. For each combination of server periods, the optimal server capacities were computed using the algorithm described in section 4.1. This method yields the global optima.

Our experimental investigation involved generating simulated systems comprising two applications of 3 unbound tasks each with overall task utilisation levels of 40 to 85%. 100 systems were generated for each utilisation level. For each system we then used the greedy and exhaustive (optimal) algorithms to select server periods and capacities. Note that the server priority ordering was fixed and the use of Periodic Servers was assumed.

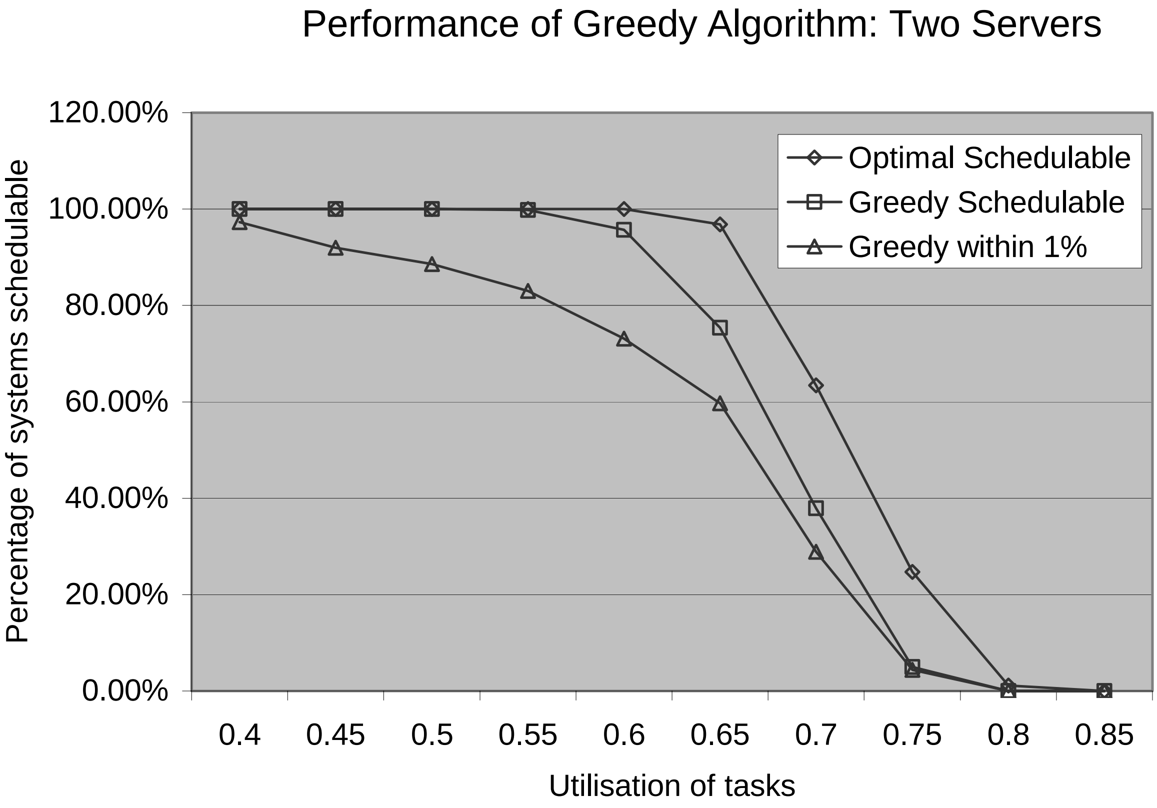

Figure 17 Greedy Algorithm for Server Parameter Selection

Figure 17 shows the performance of the greedy algorithm in terms of the number of systems it was able to schedule compared to the optimal algorithm. At low system utilisations, the greedy approach is able to find a schedulable set of server parameters however its performance drops off significantly before that of the optimal algorithm. We also compared the number of solutions that the greedy algorithm produced that were within 1% of the optimal server utilisation levels. It is apparent from the graph that even at relatively low utilisation levels, the greedy approach results in a large number of sub-optimal solutions.

We would expect that the performance of the greedy approach to deteriorate with an increasing number of servers. As performance is relatively poor even for two server systems, this approach has little to recommend it.

To summarise, server parameter selection does not appear to have an analytical solution. The best that we can currently achieve is to select server priorities and periods according to some search algorithm (potentially exhaustive search in the case of simple systems) and to derive the optimal set of server capacities via a binary search using the analysis presented in section 3.

It is clear that any approach to server parameter selection based on determining the best parameters for a single server in isolation is flawed. Dependencies between the parameters chosen for one server influence the choice of feasible parameters for others servers in such a way that choosing solutions that are locally optimal do not necessarily lead to a globally optimal solution.

4.6 Empirical Investigation

In this section we present the results of empirical investigations into the selection of server parameters for simple systems.

With systems comprising just two Periodic Servers, it is possible to exhaustively evaluate all possible combinations of server periods. In this experimental investigation, we used a binary search and the analysis presented in section 3 to determine the minimum capacity for each Periodic Server, for every possible combination of server periods.

The results of the experiments are presented as 3-D graphs of the remaining processor utilisation:

$$1 - \sum_{\forall X \in servers} \frac{C_X}{T_X}$$

The remaining utilisation (z-axis) is plotted against the period of the lower priority server (x-axis) and the period of the higher priority server (y-axis). The remaining utilisation surface is colour coded according to its value. Peaks in the surface represent the best choices of server periods. Although it is possible to understand and interpret the figures in this section when they are viewed in black and white, the figures are clearer when displayed in colour. It is therefore suggested that readers who only have access to a black and white printout of this report view the figures online.

4.6.1 Experiment 1

In this experiment, we used a task set comprising the three tasks given in the table below.

Priority	Execution Time	Period	Deadline
1	5	50	50
2	7	125	125
3	6	300	300

Table 3

Each server was required to execute a copy of this task set, thus making the server priority ordering irrelevant. A representative server context switch overhead of 2 time units was assumed.

Figure 18 illustrates the remaining processor utilisation for all combinations of low priority (LP) and high priority (HP) server periods in the range 4-100. In this case, all the tasks were considered to be unbound, irrespective of whether their periods were a multiple of the server's period.

The graph shows a jagged landscape of remaining utilisation, dependent on the relationship between the server periods and those of the tasks. The peaks in remaining utilisation are closer together at shorter server periods. This is because the peaks relate to values of the server periods that are fractions of the task periods. For example: 1/6, 1/5, 1/4, 1/3, 1/2.

A number of interesting features are visible in the graph. In the region indicated by label "A", the LP server's period exceeds that of the highest priority task it must execute, this results in the server's capacity increasing with each increase in its period, leading to a significant tail off in the remaining processor utilisation. In the region indicated by label "B", long HP server periods and the resultant large capacity of the HP server result in the LP server being unschedulable with relatively short periods.

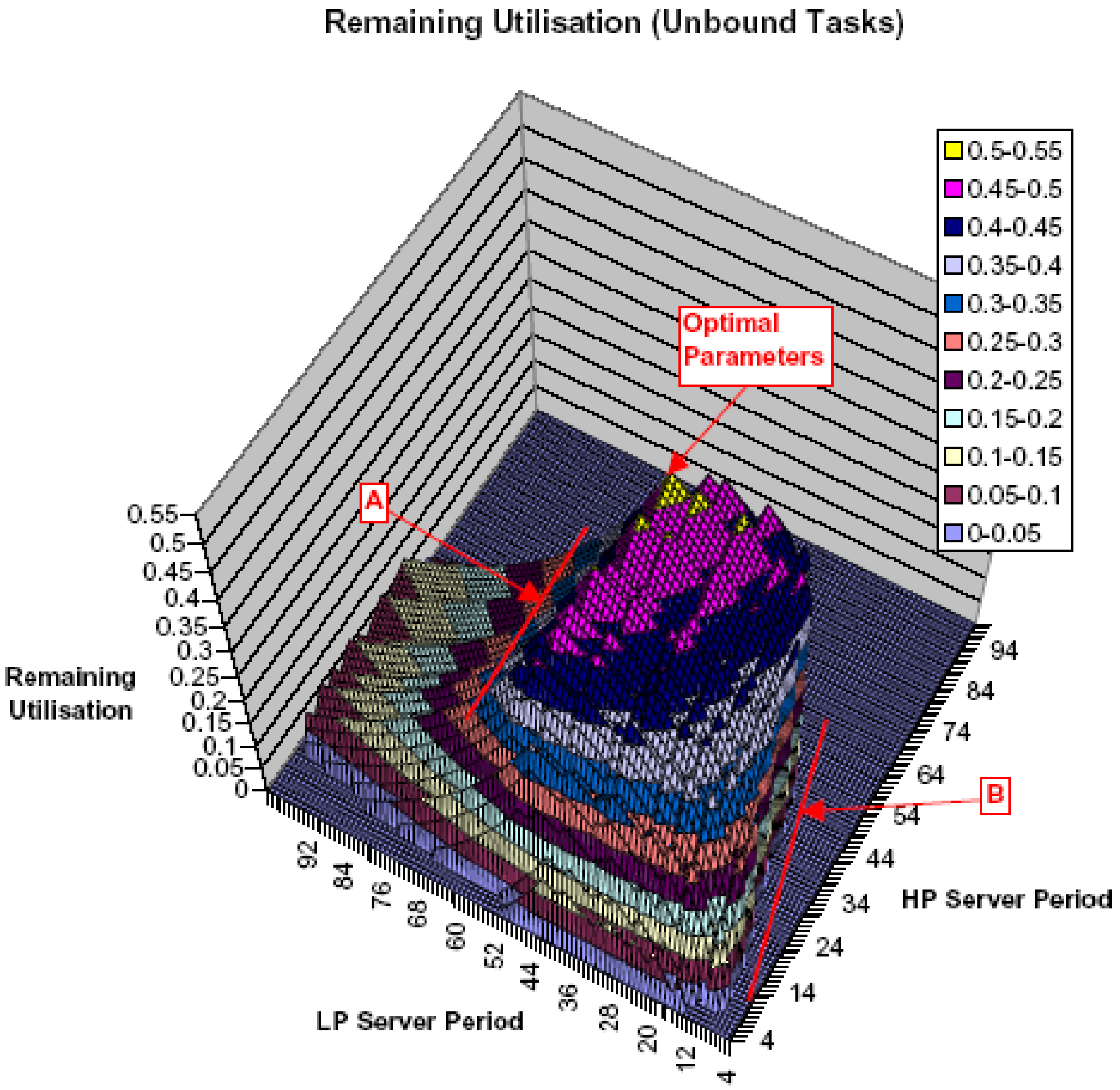

Figure 18

The optimal selection of server periods ($T_{HP} = 50$ and $T_{LP} = 43$) gives a maximum remaining utilisation of 52.4%. Note this optimum selection of parameters has the servers in the opposite of rate-monotonic priority ordering. This is a clear example of the fact that although the optimum priority ordering for Periodic Servers is rate-monotonic when only server schedulability is considered, this is not necessarily the case when task schedulability is also a factor.

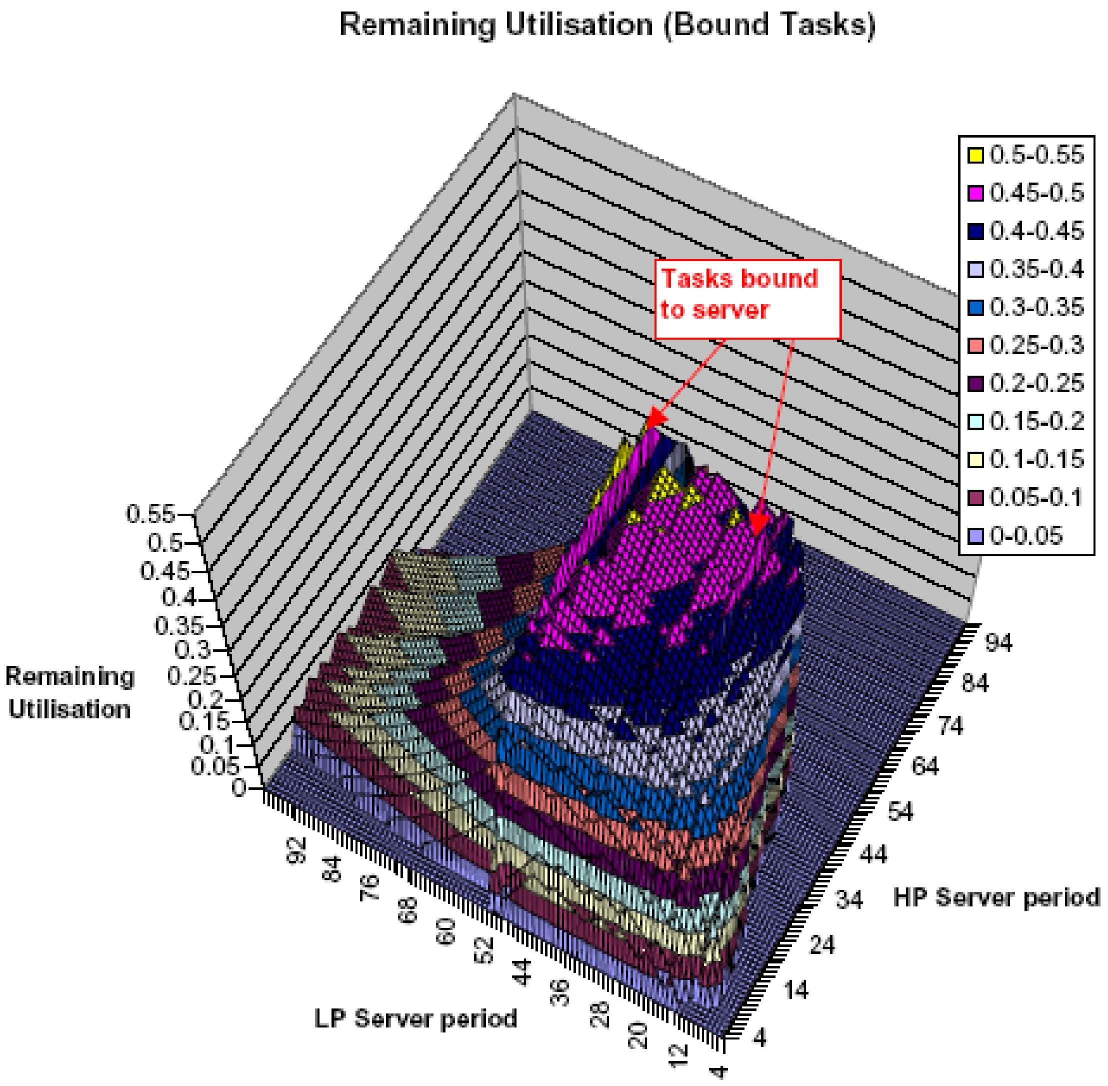

Figure 19

Figure 19 shows a very similar graph to Figure 18, however this time whenever a server's period is an exact divisor of the period of a task, that task is bound to the server. This results in two increased 'ridges' with respect to treating the tasks as always being unbound. These ridges occur for LP server periods of 25 and 50.

Allowing tasks to be bound to the release of their server results in a change in the optimal server parameters. The maximum remaining utilisation of 54% occurs when both servers have a period of 50, which is a harmonic of two of the task periods (50 and 300).

It is interesting to note that there are no additional ridges corresponding to particular values of the HP server period, despite the fact that this server executes an identical task set. The reason for this is that in the case of the highest priority server only, if a task's deadline is equal to its period and is also an exact multiple of the server's period, then the amount of execution time that a server of a given capacity can make available to the task is the same irrespective of whether the task is bound to the release of the server or not. As the task's period is an exact multiple *n* of the server's period, then in both bound and unbound cases, the server can make exactly *n* times its capacity available by the task's deadline (which is also equal to *n* times the server's period), hence there is no observable advantage in tasks being bound to the HP server in this case.

4.6.2 Experiment 2

In this experiment we used a task set comprising the four tasks given in the table below.

Priority	Exec. Time	Period	Deadline
1	8	160	100
2	12	240	200
3	16	320	300
4	24	480	400

Table 4

Each server was required to execute a copy of this task set, making the server priority ordering irrelevant. Again a representative server context switch overhead of 2 time units was assumed.

In this case, the task periods and deadlines were chosen to emphasize the effect of having tasks bound to the release of the server. The task periods were chosen such that they would be harmonics of a number of different server periods. Further, the task deadlines were chosen to be strictly less than the corresponding task periods as this also enhances the difference between the server capacity required if tasks are treated as bound versus unbound. It should however be noted that this task set is a reasonable one: there are many real world systems that have such harmonic relationships between their task periods.

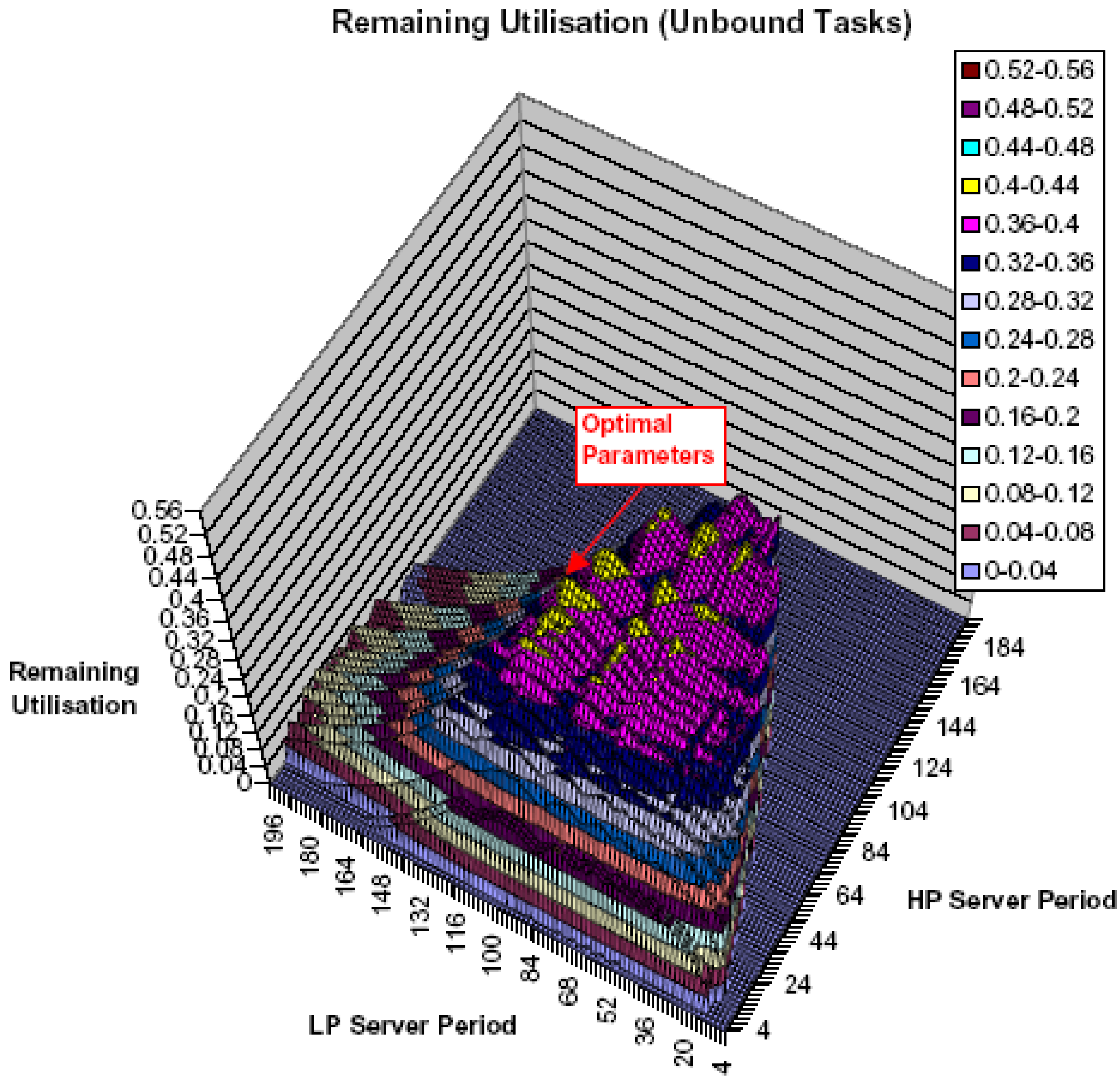

Figure 20

Figure 20 illustrates the remaining processor utilisation for various server periods for the task set in Table 4. In this case, all the tasks were assumed to be unbound. The optimal selection of server periods ($T_{HP} = 64$ and $T_{LP} = 100$) gives a maximum remaining utilisation of 42.875%.

Figure 21 illustrates the remaining utilisation for various server periods when tasks can potentially be bound to the servers. A task is treated as being bound to its server if the task's period is an exact multiple of the server's period. Note that Figure 21 shows only data for those server periods that result in one or more bound tasks and where the resultant remaining utilisation is higher than it would otherwise be if all the tasks were treated as being unbound. This makes it very easy to see the advantage of binding tasks to the servers.

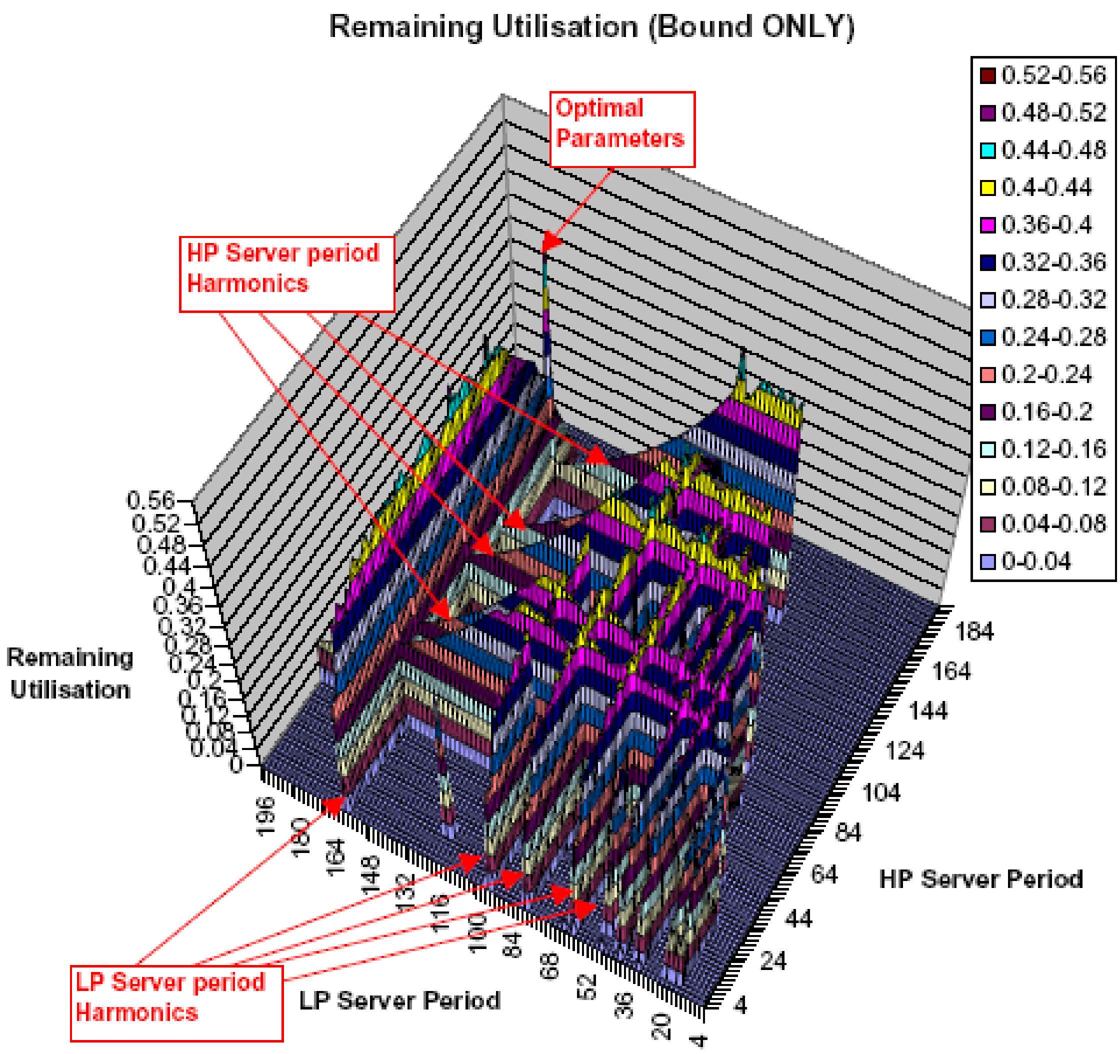

Figure 21

There are a large number of possible server periods that the task periods are harmonics of. The harmonic periods that provide an advantage in terms of reduced server capacity are 16, 20, 32, 40, 48, 60, 64, 80, 96 and 160 in the case of the LP server and 48, 60, 96, 120 and 160 for the HP server. The optimal selection of server periods ($T_{HP} = 160$ and $T_{LP} = 160$) gives a maximum remaining utilisation of 51.25%. This is a significant increase in remaining utilisation compared with treating all the tasks as unbound (42.875%).

The deadlines were less than the task periods for the task set used in this experiment. This highlights the difference between the analysis of bound and unbound tasks. If a task is unbound, then for server periods greater than the task's deadline, the server's capacity has to increase significantly to ensure that the task is schedulable, resulting in a marked reduction in remaining utilisation. This is not the case when a task is bound to the server, with both task and server sharing the same period, a short deadline task may be schedulable for a small server capacity. Thus permitting tasks to be bound to a server results in solutions that are very different in terms of server utilisation from those that are available when all the tasks are unbound.

For comparison purposes, Figure 22 shows the remaining utilisation for all combinations of server periods. Here tasks are treated as bound if their period is an exact multiple of the server's replenishment period; otherwise they are treated as unbound.

It is interesting to note that the optimum selection of server periods occurs as a spike in the remaining utilisation surface. This has implications for search techniques aimed at determining the optimal selection of server parameters. Given such a discontinuous landscape, a general-purpose search technique such as simulated annealing or genetic algorithms may not be effective without the use of heuristics to locate potential good solutions based on harmonics.

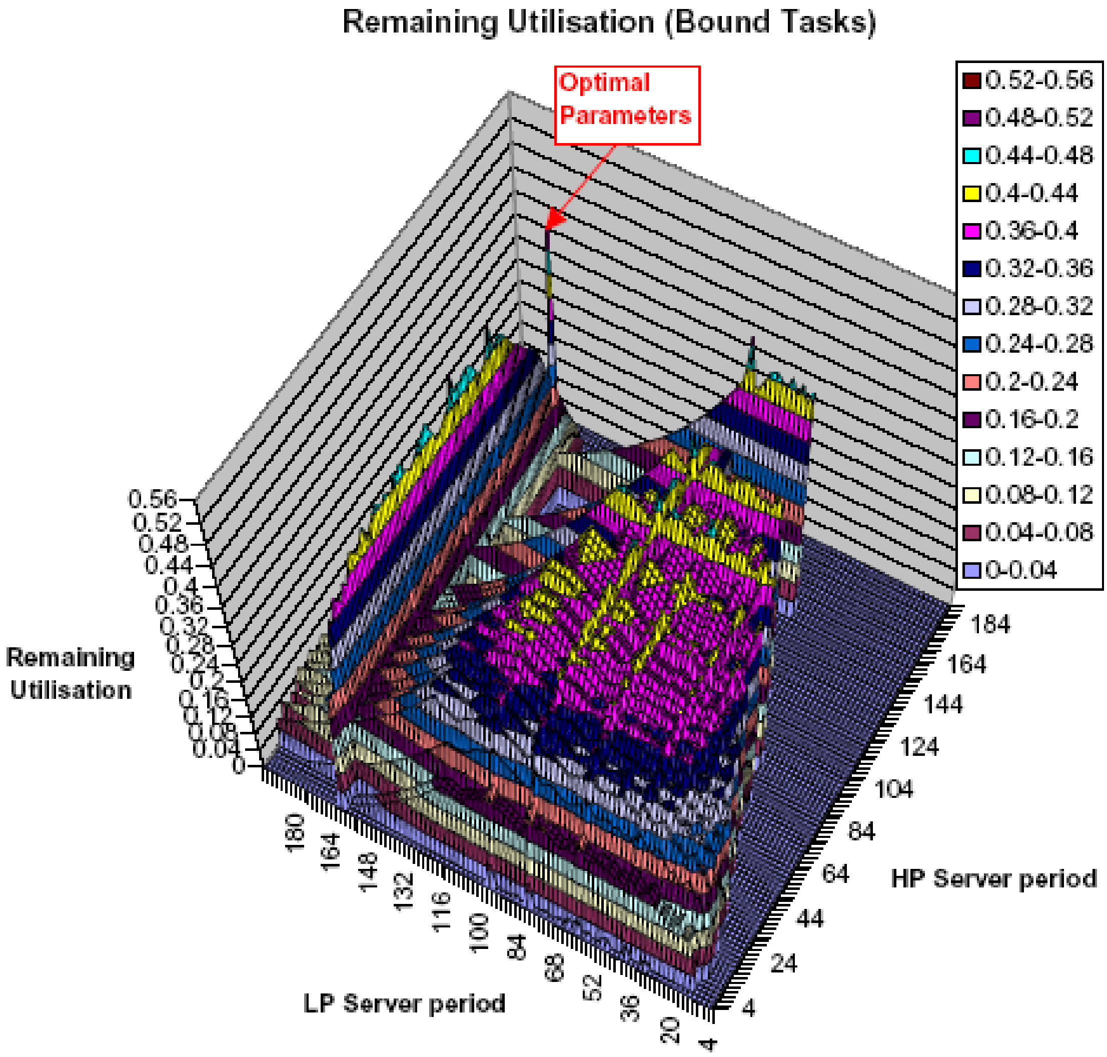

Figure 22

4.6.3 Additional Experiments

We performed a number of additional experiments similar to those described in the previous two sections. The basic trends visible in these experiments were as follow:

1. **Number of tasks**: Increasing the number of tasks in an application (or more correctly increasing the number of distinct task periods) results in a change in the topology of the remaining utilisation landscape. More tasks imply an increased number of valleys each with less depth. With 10 or more tasks with co-prime periods precise choice of server period becomes less important. In this case there is a region of values that give similar levels of remaining utilisation.

2. **Harmonic task periods**: If a period can be chosen for the server that exactly divides a number of task periods and those tasks can be bound to the server then an increase in

remaining utilisation can often be achieved.

3. **Deadline less than period**: Binding tasks to a server appears to have the biggest impact when the shortest deadline task is bound to the server. This is because the range of values possible for the server's period is effectively constrained to less than the shortest task deadline in the case of unbound tasks and to less than the shortest task period in the case of bound tasks. Permitting a greater useful range of server periods typically results in better solutions as long server periods lead to lower overheads.

Figure 23 illustrates the remaining utilisation surface for a two-server system where each server schedules an application comprising 10 tasks with co-prime periods. Figure 24 provides a comparable set of results when the tasks may be bound to the server. In this case the advantage of binding tasks to the server is limited and most effective when the server period is equal to that of the shortest period (and deadline) task.

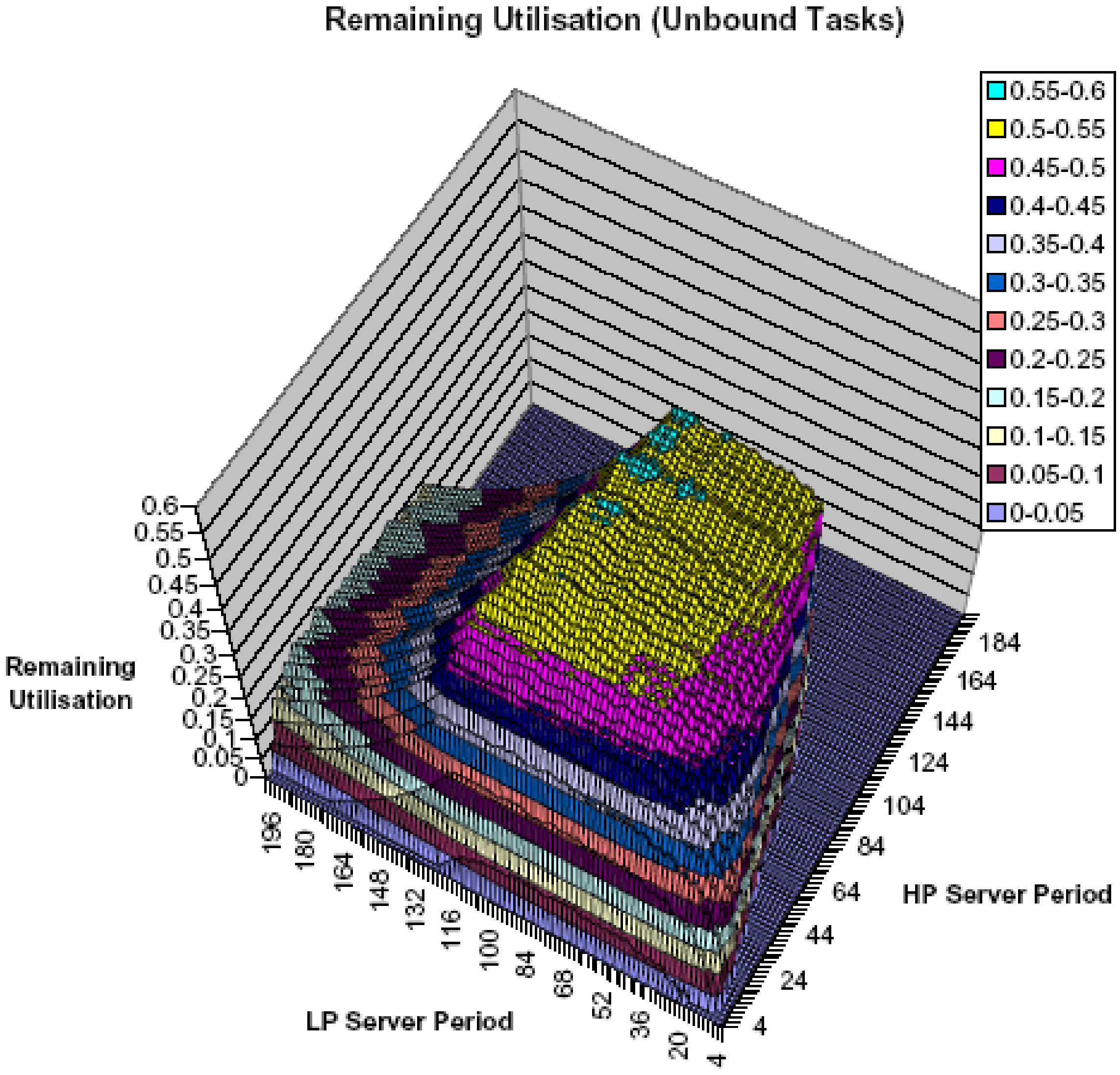

Figure 23

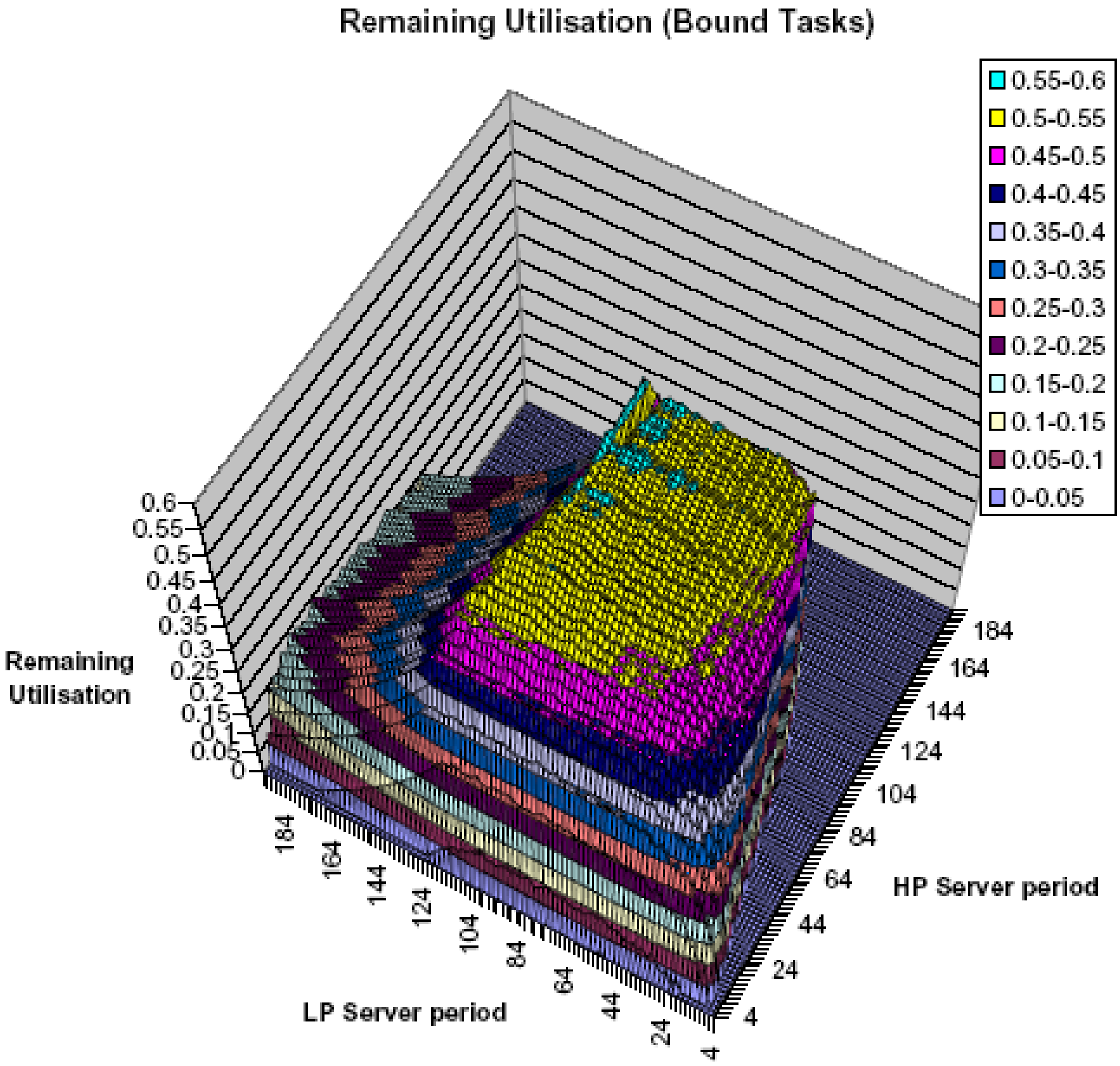
Remaining Utilisation (Bound Tasks)
0.55-0.6
0.5-0.55
0.45-0.5
0.4-0.45
0.35-0.4
0.3-0.35
0.25-0.3
0.2-0.25
0.15-0.2
0.1-0.15
0.05-0.1
0-0.05
Remaining Utilisation
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
LP Server period
184
164
144
124
104
84
64
44
24
4
HP Server period
184
164
144
124
104
84
64
44
24
4

Figure 24

5 Summary and Conclusions

In this report we addressed the problem of scheduling a number of applications on a single processor using a set of servers. The motivation for this work comes from the automotive, Avionics and other industries where the advent of high performance microprocessors is now making it both possible and cost effective to implement multiple applications on a single platform.

Our research has focused on systems that are scheduled using fixed priority pre-emptive scheduling at both local and global scheduling levels.

5.1 Contribution

The major contributions of this work are:

- Exact response time analysis for hard real-time tasks scheduled under Periodic, Sporadic and Deferrable Servers. This analysis provides a reduction in the calculated worst-case response times of tasks when compared to previous published work. A similar improvement is also apparent in the server capacity and replenishment periods deemed necessary to schedule a given task set.
- Extension of the analysis to tasks that are bound to the release of their server. We showed that permitting tasks to be bound to a server with the appropriate replenishment period always enhances task schedulability and can reduce the server capacity required.
- Comparison of Periodic, Sporadic and Deferrable Servers in terms of their ability to guarantee the deadlines of hard real-time tasks. The Periodic Server was shown to completely dominate the other server algorithms on this metric.
- Extension of the schedulability analysis to hierarchical systems where tasks in disparate applications are permitted to access mutually exclusive global shared resources.
- Evidence that server parameter selection is a holistic problem. It is not sufficient to determine the optimal set of server parameters for each server in isolation as these parameters have an effect on the choice of possible values for other servers.

In addition to these analytical results, we conducted an empirical investigation into the selection of server parameters. Our empirical results clearly illustrated the advantages of choosing server periods that are exact divisors of the task periods, thus enabling tasks to be bound to the release of the server.

5.2 Future Work

Today it is possible using the analysis techniques described in this report to determine the optimal set of server parameters via an exhaustive search of possible periods and priorities for simple systems comprising 3 or 4 applications. Further work is required to provide an effective algorithm capable of choosing an optimal or close to optimal set of server parameters given systems comprising perhaps ten or more applications.

A global optimisation technique such as simulated annealing or genetic algorithms could possibly be used as the high-level search method, with selection of locally optimal server capacities via a binary search. It should be noted however that the spiky topography of the remaining utilisation surface makes effective search difficult. As an alternative approach, the use of heuristics, such as checking all possible combinations of harmonics, may be effective in some cases.

Another interesting area of future research involves incorporating Quality of Service (QoS) requirements into hierarchical fixed priority pre-emptive systems. Here additional servers could be deployed at both levels in the hierarchy to make spare capacity available responsively. An interesting alternative would be to use Dual Priority Scheduling [13] as the policy of choice at both global and local scheduling levels.

6 References

[1] T-W. Kuo, C-H. Li. “A Fixed Priority Driven Open Environment for Real-Time Applications”. *In proceedings of the IEEE Real-Time Systems Symposium*. Madrid, Spain, December 1998.

[2] Z. Deng, J.W-S. Liu. “Scheduling Real-Time Applications in an Open Environment”. *In proceedings of the IEEE Real-Time Systems Symposium*. December 1997.

[3] B. Sprunt. “Aperiodic Task Scheduling for Real-Time Systems”. *Ph.D. Dissertation*, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[4] C.L.Liu, J.W.Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment” *JACM*, Vol. 20, No. 1 January 1973, p46-61.

[5] S. Saewong, R. Rajkumar, J. Lehoczky, M. Klein. “Analysis of Hierarchical Fixed priority Scheduling”. *Proceedings of the ECRTS,* pages 173-181, 2002.

[6] J.K. Strosnider, J.P. Lehoczky, L. Sha. “The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”. *IEEE Transactions on Computers,* 44(1) January 1995.

[7] G. Lipari, E. Bini. “Resource Partitioning among Real-Time Applications”. *Proceedings of the ECRTS,* Porto, Portugal, July 2003.

[8] L. Almeida. “Response Time Analysis and Server Design for Hierarchical Scheduling”. *Proceedings Real-Time Systems Symposium Work-in-Progress 2003.*

[9] G. Bernat, A. Burns. “New Results on Fixed Priority Aperiodic Servers”. *Proceedings of the IEEE Real-Time Systems Symposium*. Phoenix, Arizona, December 1999.

[10] N. Audsley, A. Burns, M. Richardson, A.J.Wellings. “Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling”. *Software Engineering Journal,* 8(5) pages284-292, 1993.

[11] N. Audsley. “On Priority Assignment in Fixed Priority Scheduling” *Information Processing Letters*. Vol. 79 No. 1 pages 39-44. 2001.

[12] T.P.Baker. “Stack-based Scheduling of Real-Time Processes.” *Real-Time Systems Journal* (3)1, pages 67-100. 1991.

[13] R.I.Davis, A.J.Wellings. “Dual Priority Scheduling” *Proceedings of the 16th IEEE Real-Time Systems Symposium*. Pages 100-109, December 1995.

[14] D.Niz, L.Abeni, S.Saewong, R.Rajkumar. “Resource Sharing in Reservation-Based Systems” *Proceedings IEEE Real-Time Systems Symposium*. Pages 171-180. London, UK, December 2001.

[15] T.M.Ghazalie, T.P.Baker. “Aperiodic Servers in a Deadline Scheduling Environment” *Real-Time Systems.* 9(1) July 1995.

[16] M. Caccamo and L. Sha. “Aperiodic Servers with Resource Constraints” *Proceedings IEEE Real-Time Systems Symposium*. Pages 161-170. London, UK, December 2001.

[17] G. Lamastra, G. Lipari, L. Abeni. “A Bandwidth Inheritance Algorithm for Real-Time Task Synchronisation in Open Systems” *Proceedings IEEE Real-Time Systems Symposium*. Pages 151-160. London, UK, December 2001.

[18] EU Information Society Technologies (IST) Program, Flexible Integrated Real-Time Systems Technology (FIRST) Project, IST-2001-34140.

[19] L. Sha, J.P.Lehoczky and R. Rajkumar. “Solutions for some Parctical Problems in Prioritised Preemptive Scheduling” *Proceedings IEEE Real-Time Systems Symposium*. Pages 181-191. 1986.

[20] M. Caccamo, G.Buttazzo, L.Sha. “Capacity Sharing for Overrun Control.” *Proceedings IEEE Real-Time Systems Symposium*. Pages 295-304. 2000.

[21] G.Bernat, I.Broster, A.Burns. “Rewriting History to Exploit Gain Time.” *Proceedings IEEE Real-Time Systems Symposium*. Pages 328-335. 2004.

[22] N.C.Audsley, A. Burns, R.I.Davis, A.J.Wellings. “Integrating Best Effort and Fixed Priority Scheduling” *In Proceedings of IFAC/IFIP Workshop on Real-Time Programming.* Lake Constance, Germany. 1994.

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

7 Appendix

7.1 Comparison of Interference due to Deferrable and Periodic Servers

In this section, we compare the interference due to a set of Deferrable Servers with that of a set of Periodic Servers with exactly the same parameters (periods and capacities).

Consider the response time of a low priority server S where all higher priority servers are Periodic Servers:

$$w_S^{n+1} = C_S + \sum_{\forall X \in hp(S)} \left\lceil \frac{w_S^n}{T_X} \right\rceil C_X \qquad (20)$$

The solution to this equation and hence the response time of the low priority server is constrained to lie in one of the intervals $(nT_X + C_X, (n+1)T_X]$ where n is an integer. (These intervals are closed-open in the sense that the end of the interval may be a solution, but the start of the interval may not). The fact that the solution is constrained to lie in one of these intervals can be seen by considering the alternative. If the length of the busy period increased such that it was in the interval $(nT_X, nT_X + C_X]$ then interference from another invocation of server X would need to be accounted for, thus increasing the length of the busy period by at least C_X and hence ensuring that any solution is in one of the intervals $(nT_X + C_X, (n+1)T_X]$.

Now consider the response time of a low priority server when all the higher priority servers are Deferrable Servers. In this case the response time may be found according to [9]:

$$w_S^{n+1} = C_S + C_X + \sum_{\forall X \in hp(S)} \left\lceil \frac{w_S^n - C_X}{T_X} \right\rceil C_X \qquad (21)$$

As $\forall X : C_X < T_X$, the length of the busy period which provides the solution to equation (21) must be at least as long as the busy period which forms a solution to equation (20), given the same set of servers. As a starting point in solving equation (21) we are free to choose any value that can be guaranteed not to exceed the solution. We therefore choose the value R_A that is the solution to equation (20). Now we know that for every higher priority server, R_A must fall into an interval $(n_X T_X + C_X, (n_X + 1)T_X]$ where the total interference from that server is given by $n_X C_X$. However if R_A is in the interval $(n_X T_X + C_X, (n_X + 1)T_X]$ then the interference due to server X according to equation (21) is $(n_X + 1)C_X$. As this is the case for all higher priority servers, the response time of the low priority server assuming interference from Deferrable Servers is at least $\sum_{\forall X \in hp(S)} C_X$ greater than it is for an equivalent set of Periodic Servers.

7.2 Priority Ordering of Servers

In this section, we consider the optimal priority ordering of servers with respect to their own schedulability.

A server is schedulable if its response time is no greater than its period. A system of servers is schedulable if all the servers in the system are schedulable. Servers are assigned priorities according to some priority assignment policy. A priority ordering is said to be feasible with respect to a given system if it results in the system being schedulable.

A priority assignment policy is *optimal* if and only if that policy results in a feasible system, whenever there exists a feasible priority ordering for the system.

For a system comprising only Periodic Servers, the rate-monotonic priority assignment (RMPA) policy is optimal as the servers behave in the same way as simple periodic tasks [4].

For a system comprising a set of Deferrable Servers, it is easy to construct examples to show that RMPA is not optimal. (Such an example appears at the end of section 7.2.2)

In [9] Bernat and Burns showed that the analysis of Deferrable Servers resembles that of periodic tasks with jitter equivalent to $T_S - C_S$.

$$w_i^n = C_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w_i^n + (T_X - C_X)}{T_X} \right\rceil C_X$$

We note from the above recurrence relation used to determine server response times that:

1. Lower priority servers have no effect on the response time of a higher priority server.
2. The priority ordering between higher priority servers has no effect on the response time of the server of interest.

This means that the Optimal Priority Assignment Algorithm developed by Audsley [11] can be used to select the optimal priority assignment for a set of Deferrable Servers.

```
for each priority level, lowest first
{
     for each unallocated server
     {
          if the server is schedulable at this priority
          {
               allocate the server to this priority
               break (continue with outer loop)
          }
     }
     return unschedulable
}
return schedulable
```

Optimal Priority Assignment Algorithm

The algorithm works because we can determine the schedulability of a server at a given priority level without reference to the priority ordering of the set of higher priority servers.

This algorithm requires *n(n+1)/2* schedulability tests compared to the *n!* potential priority orderings.

Let us assume that there is a feasible priority ordering. Now consider the operation of the optimal priority assignment algorithm. It selects a server that is schedulable at the lowest priority level, note there must be at least one if there exists a feasible priority ordering. Let us assume that this server was at priority level *j* in the feasible ordering. The response times of all servers with higher priority than *j* are unaffected by moving this server to the lowest priority level. Further all servers of lower priority than *j* are subject to less interference and so their response times cannot increase, hence they too all remain schedulable. Finally, the server that was placed at the lowest priority is also schedulable so the system remains schedulable. This argument can be repeated at successively higher priority levels thus proving that if a feasible priority ordering exists then the optimal priority assignment algorithm will find a feasible ordering.

7.2.1 Partial Ordering

Consider a set of Deferrable Servers that have capacities and periods such that for any pair of servers, the servers can be labeled *A* and *B* such that both of the following conditions hold: $T_A + C_A \geq T_B + C_B$ and $C_A \geq C_B$.

For such a set of servers, *period plus capacity* monotonic priority assignment is optimal. That is optimal priority assignment can be achieved by assigning the server with the smallest value of $T_S + C_S$ the highest priority, the server with the next smallest value of $T_S + C_S$ the second highest priority and so on.

Proof:

Let us assume that we have a system of Deferrable Servers which is schedulable and has two servers labeled S_A and S_B where S_A is at a higher priority level and the two conditions $T_A + C_A \geq T_B + C_B$ and $C_A \geq C_B$ hold. To prove that 'period plus capacity' monotonic priority assignment is optimal, it is sufficient to prove that we may exchange the priorities of the two servers S_A and $S_B$and that each will remain schedulable. By repeatedly applying such an exchange of priorities it is possible to convert any feasible priority ordering into a 'period plus capacity' monotonic ordering that is also feasible.

Consider the response time of S_B when it is at the lower priority level *x*:

$$R_B = C_B + \left\lceil \frac{R_B + (T_A - C_A)}{T_A} \right\rceil C_A + I_x(R_B) \tag{22}$$

Where $I_x(R_B)$ is the interference from all servers of higher priority than *x* with the exception of S_A which is considered separately.

Equation (22) simplifies to:

$$R_B = C_B + C_A + \left\lceil \frac{R_B - C_A}{T_A} \right\rceil C_A + I_x(R_B) \tag{23}$$

As S_B is schedulable at this priority level, $R_B \leq T_B$. As $T_A \geq T_B - C_A$ the ceiling function in equation (23) must evaluate to 1 and therefore:

$$R_B = C_B + 2C_A + I_x(R_B) \tag{24}$$

Now consider the response time of S_A when the server priorities are reversed and server S_A is at priority level *x*:

$$R_A = C_A + C_B + \left\lceil \frac{R_A - C_B}{T_B} \right\rceil C_B + I_x(R_A) \qquad (25)$$

We know that equation (23) converges on a value of $R_B = C_B + 2C_A + I_x(R_B) \leq T_B$ starting with an initial value of $C_B + 2C_A$. As $C_A \geq C_B$ the initial value for equation (25), $2C_B + C_A$ is no greater than $C_B + 2C_A$ and the interference function for other servers is the same. Therefore equation (25) must converge on a value that is no greater than R_B and hence no greater than T_B so:

$$R_A = C_A + 2C_B + I_x(R_A) \qquad (26)$$

The interference is monotonically non-decreasing in the length of the time interval, so as $R_A \leq R_B$ it follows that $I_x(R_A) \leq I_x(R_B)$. Comparing equation (24) and equation (26) we have $R_A + C_A \leq R_B + C_B$. As $R_B + C_B \leq T_B + C_B$ and one of our pre-conditions is that $T_A + C_A \geq T_B + C_B$ then it follows that $R_A \leq T_A$ and so S_A is schedulable at the lower priority level, thus proving the theorem.

As well as providing an optimal ordering of server priorities in cases where a full ordering is possible based on the criteria $T_A + C_A \geq T_B + C_B$ and $C_A \geq C_B$, this result is also useful when the set of servers only has a partial ordering based on these criteria. In this case the criteria can be used in conjunction with the Optimal Priority Assignment Algorithm to reduce the number of schedulability tests required. On each inner loop of the algorithm it is unnecessary to perform a schedulability test on a server S_B if a server S_A has already been found to be unschedulable and $T_A + C_A \geq T_B + C_B$ and $C_A \geq C_B$. In this case S_B will also be unschedulable.

7.2.2 No Partial Ordering

This section provides a simple example showing that if $T_A + C_A \geq T_B + C_B$ but $C_A < C_B$ then 'period plus capacity' monotonic priority ordering is not optimal.

Server	**Priority**	**Capacity**	**Period**	**Period + capacity**	**Response Time**
A	1	2	12	14	2
B	2	1	16	17	5
C	3	5	11	16	11

In the priority order presented in the above table, all the Deferrable Servers are schedulable. However placing them in 'period plus capacity' order results in server *B* becoming unschedulable. Note that placing the servers in rate-monotonic priority order also results in server *B* having the lowest priority and hence being unschedulable.