An Investigation into Server Parameter Selection for Hierarchical Fixed Priority Pre-emptive Systems

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

Abstract

This paper investigates the problem of server parameter selection in hierarchical fixed priority pre-emptive systems. A set of algorithms are provided that determine the optimal values for a single server parameter (capacity, period, or priority) when the other two parameters are fixed. By contrast, the general problem of server parameter selection is shown to be a holistic one: typically the locally optimal solution for a single server does not form part of the globally optimal solution.

Empirical investigations show that improvements in remaining utilisation (spare capacity) can be achieved by choosing server periods that are exact divisors of their task periods; enabling tasks to be bound to the release of their server, enhancing task schedulability and reducing server capacity requirements.

1. Introduction

In automotive electronics, the advent of advanced high performance embedded microprocessors have made possible functionality such as adaptive cruise control, lane departure warning systems, integrated telematics and satellite navigation 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 following a similar trend in avionics.

Integrating a number of real-time applications onto a single microprocessor raises issues of resource allocation and partitioning.

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 ideally 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 ready 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 paper 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 based on 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 ready tasks should execute when the server is active.

1.1. Related work

In 1999, building upon the work of Deng and Liu [2], Kuo and Li [1] first introduced analysis of hierarchical fixed priority pre-emptive scheduling. They provided a simple utilisation based schedulability test, using the techniques of Liu and Layland [4].

In 2002, Saewong et al [5] provided response time analysis for hierarchical systems using Deferrable Servers or Sporadic Servers 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.

In 2003, Shin and Lee [6] provided analysis of fixed priority pre-emptive scheduling at the local level, given the bounded delay periodic resource model, introduced by Feng and Mok [3].

Also in 2003, Lipari and Bini [7] provided an alternative sufficient but not necessary response time formulation using an availability function to represent the time made available by a server from an arbitrary time origin. Lipari and Bini [7] investigated the problem of server parameter selection and considered the 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 built upon the work of Lipari and Bini, recognising that the server availability function depends on the *"maximum jitter that periods of server availability may suffer"*. This analysis is more accurate but can still be pessimistic.

In 2005, Davis and Burns [13] provided exact[1] (sufficient and necessary) response time analysis for independent hard real-time tasks scheduled under Periodic, Sporadic and Deferrable Servers.

In 2006, Davis and Burns [14] introduced and analysed the Hierarchical Stack Resource Policy (HSRP) based on the Stack Resource Policy of Baker [11]. Using the HSRP bounds the delays due to mutually exclusive access to resources shared between different applications.

In this paper, we investigate the problem of server parameter selection, choosing the period, capacity and priority of a server associated with each application. Unlike previous work by Lipari and Bini [7], we consider server parameter selection across multiple servers within a system. Our research builds upon previous work on schedulability analysis for hierarchical systems. In particular, we investigate how improvements in schedulability that can be achieved by binding tasks to a server of the appropriate period [13], impact on the problem of server parameter selection.

1.2. Organisation

Section 2 describes the terminology, notation and system model used in the rest of this paper. It also recapitulates the schedulability analysis for independent applications in hierarchical fixed priority pre-emptive systems given in [13]. Section 3 discusses server parameter selection. First considering how each of the three server parameters: capacity, period, and priority may be set if the other two are known; then discussing the general problem of selecting the optimal server capacity, period, and priority when all three parameters are effectively free variables. In Section 4, we outline an empirical investigation into the problem of server parameter selection, including the reductions in server utilisation that can be achieved by matching server periods to task periods, making possible the binding of tasks to the release of their server. Section 5 summarises the major contributions of the paper and suggests directions for future research.

[1] This analysis is exact if and only if, in the best case, the server can provide all of its capacity at the start of its period.

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.* Associated with each application is a *server*. The application tasks execute within the capacity of the associated server.

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 paper we analyse systems where the fixed priority pre-emptive scheduling policy is used for both global and local scheduling.

Application tasks may arrive 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, minimum inter-arrival time T_i, otherwise referred to as its *period*, and finally its *release jitter* J_i defined as the maximum time between the task arriving and it being ready to execute.

Application tasks are referred to as *bound* or *unbound* [15]. 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.

Each server has a unique priority S, within the set of servers and is characterised by its *capacity* C_S, and *replenishment period* T_S. A server's capacity is the maximum amount of execution time that may normally be consumed by the server in a single invocation. The replenishment period is the maximum time before the server's capacity is available again.

A task's *worst-case response time* R_i, is the longest time from the task arriving to it completing execution. Similarly, a server's worst-case response time R_S, is the longest time from the server being replenished to its capacity being exhausted, given that there are tasks ready to use all of the server's available 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 worst-case response time does not exceed its period. The analysis used in this paper assumes that tasks have deadlines that are no greater than their periods, and that servers have deadlines that are equal to their periods.

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.

The schedulability analysis originally given in [13] and revisited in the remainder of this section assumes that all applications and tasks are independent. We note that this restriction can be lifted using the analysis given in [14].

In this paper we consider applications scheduled under a simple *Periodic Server*. The analysis can be extended to alternative server algorithms such as the Deferrable Server and the Sporadic Server, however due to space considerations these alternative server algorithms are not discussed further.

The Periodic Server is invoked with a fixed period and executes any ready tasks until its capacity is exhausted. Note each application is assumed to contain an idle task that continuously carries out built in tests, memory checks and so on, therefore the server's capacity is fully consumed during each period.

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 at a higher priority. The jitter of the Periodic Server is assumed to be zero and for the sake of simplicity, server jitter is therefore omitted from the schedulability analysis equations. The behaviour of the server does however add to the jitter of the tasks that it executes. The release jitter of the tasks is typically increased by $T_S - C_S$, corresponding to the maximum time that a task may have to wait from the server capacity being exhausted to it being replenished.

The analysis presented in the next section 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 higher.

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 (or 'window', hence *w*) equivalent to the 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, at priority level *i* and above, released by the application within a time window of length *w*.

2.2. Task schedulability analysis

In this section we revisit the schedulability analysis given in [13] for independent hard real-time applications.

Using the principles of Response Time Analysis [10], the worst-case response time for a task τ_i, executing under a server *S*, occurs following a critical instant where:

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

The worst-case response time of τ_i can be determined by computing the length of the priority level *i* busy period starting at the first release of the server that could execute the task (see Figure 1). This 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 execution of the task.

The task load at priority level *i* and higher, ready to be executed in the busy period w_i, is given by:

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

where *hp(i)* is the set of tasks that have priorities higher than task τ_i and J_j is the release jitter of the task, increased by $T_S - C_S$ in the case of unbound tasks, due to the operation of the server.

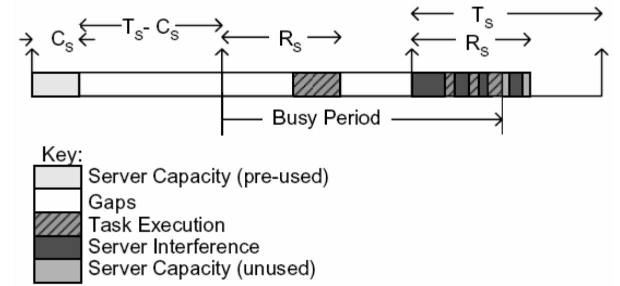

Figure 1 Busy period

The total length of gaps in complete server periods, not including the final server period, is given by:

$$\left(\left\lceil \frac{L_i(w_i)}{C_S} \right\rceil - 1 \right)(T_S - C_S) \tag{2}$$

The interference due to higher priority servers executing during the final server period that completes execution of task τ_i is dependent on the amount of task execution that the server needs to complete before the end of the busy period. The exact interference can be calculated using information about server priorities, capacities and replenishment periods.

Figure 1 illustrates the interference in the final server period. The extent to which the busy period w_i extends into the final server period is given by:

$$w_i - \left(\left\lceil \frac{L_i(w_i)}{C_S} \right\rceil - 1 \right) T_S \tag{3}$$

The full extent of the busy period, including interference from higher priority servers in the final server period, can be found using the recurrence relation

from [13], given by Equation (4):

$$w_i^{n+1} = L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right)}{T_X} \right\rceil C_X \quad (4)$$

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

Recurrence starts with a value of $w_i^0 = C_i + (\lceil C_i / C_S \rceil - 1)(T_S - C_S)$ and ends either when $w_i^{n+1} = w_i^n$ in which case $w_i^n + J_i$ gives the task's worst-case response time or when $w_i^{n+1} > D_i - J_i$ in which case the task is not schedulable.

Note the use of *max*(0, …) in the 3rd term in Equation (4) 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.

3. 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.

$$1 - \sum_{\forall X \in servers} \frac{C_X}{T_X} \quad (5)$$

We use remaining processor utilisation as a metric as this provides a broad measure of the processing time that could potentially be made available to other applications that might be added to the system.

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 this paper.

3.1. 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 algorithm to derive the optimal set of server capacities for the set of server periods and priorities provided.

```
OPTIMAL SERVER CAPACITY ALLOCATION ALGORITHM
for each server, highest priority 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 algorithm 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 tasks.

Thus the set of minimum server capacities calculated in descending priority order are optimal for the *given* set of server priorities and periods.

3.2. Determining Server Priorities

For a set of Periodic Servers, rate-monotonic priority ordering (RMPO) [19] is the optimal priority ordering with respect to server schedulability. However, when task schedulability is considered, RMPO is no longer optimal, as shown by the examples in Sections 3.4 and 4.1. Similarly, deadline-monotonic priority assignment [20] 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].

The algorithm, given below, works 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 the tasks that they execute.

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

Note, as the server periods and capacities are fixed in this case, the remaining processor utilisation does not

change with different priority orderings, instead, the algorithm given below is optimal in the sense that it always finds a feasible priority ordering if such a priority ordering exists. The interested reader is directed to [11] for proof of why this method is optimal, despite the fact that it makes a greedy allocation, assigning the first unallocated yet schedulable server found to each priority level.

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

3.3. Determining Server Periods

If the server priorities and capacities are fixed, then a set of server periods can be systematically derived using the algorithm given below.

```
OPTIMAL SERVER PERIOD ALLOCATION ALGORITHM
for each server, highest priority first
{
    binary search for the maximum server
    period Z that results in the server and
    its tasks being schedulable.
    if no schedulable period found
    {
        exit system not schedulable
    }
    else
    {
        set the period of the server to Z
    }
}
```

This algorithm works 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.

The optimal server period allocation algorithm given above 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.

3.4. 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 general problem of server parameter selection.

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:

Table 1

Task	C_i	T_i	D_i	Server
τ_1	10	20	20	S_A
τ_2	4	24	24	S_B

Further, assume that server context switch overheads[2] 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.

[2] Such overheads are incorporated in our analysis, by assuming that each server must utilise part of its capacity for the context switch to it, prior to executing any of its tasks.

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.

3.5. 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 3.1. 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.

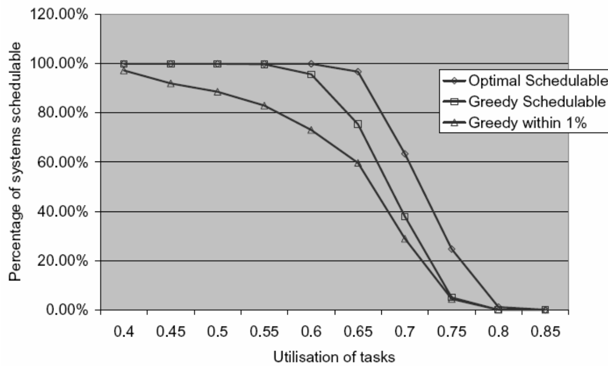

Figure 2

Figure 2 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, in the general case, server parameter selection does not appear to have an analytical solution. The best that we can 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. 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 does not typically lead to a globally optimal solution.

4. Empirical Investigation

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

The aim of these investigations is to highlight interesting properties of the server parameter selection problem, which may be useful in devising solutions that are appropriate for real systems. The algorithms and techniques that we use in our investigation, such as exhaustive search, are not intended as solutions to the general problem, they are merely tools with which to improve our understanding. We therefore give no analysis of the complexity or execution time of these methods. Similarly, the tasksets used in the experiments detailed here are not meant to reflect the sets of tasks found in real applications; instead, simple tasksets were used so that we could reason about the properties that they have in common with real systems, such as harmonic / non-harmonic task periods, and periods that are multiples of the server period.

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 algorithm presented in section 3.1, 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 on a printed copy, the figures are clearer when displayed in colour. It is therefore suggested that readers view this paper online; the figures will then appear in colour.

4.1. Experiment 1

In this experiment, we used a simple taskset comprising the three tasks given in Table 2 below.

Table 2

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

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

Figure 3 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.

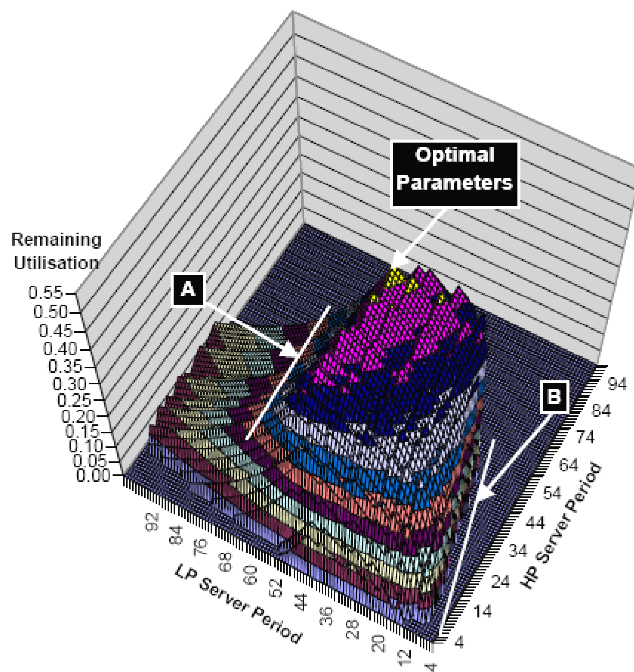

Figure 3

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 low priority 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 high priority server periods and the resultant large capacity of that server result in the low priority server being unschedulable with relatively short periods.

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 the case when task schedulability is also a factor.

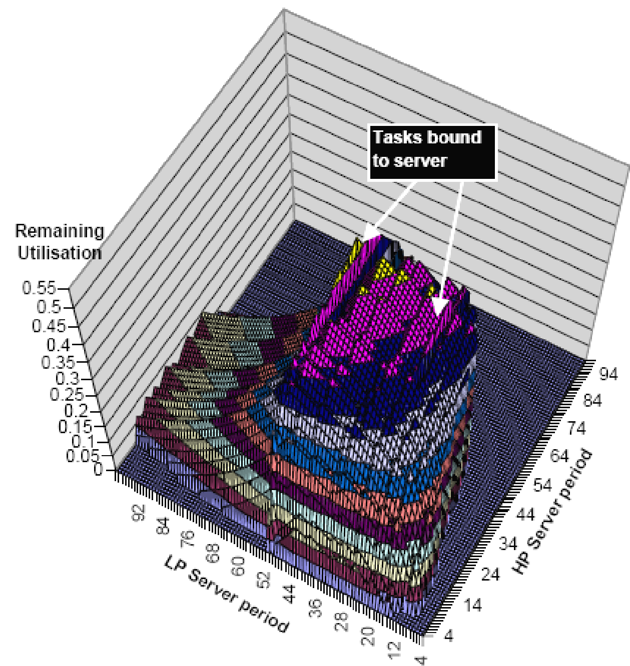

Figure 4

Figure 4 shows a very similar graph to Figure 3; 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 low priority server periods of 25 and 50.

Comparing Figure 3 and Figure 4 shows that allowing tasks to be bound to the release of their server results in a change in the optimal server parameters. With bound tasks, 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 high priority server period, despite the fact that this server executes an identical taskset. 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 high priority server in this case.

4.2. Experiment 2

In this experiment we used a simple taskset comprising the four tasks given in Table 3 below. Each server was required to execute a copy of this taskset, making the server priority ordering irrelevant. Again a server context switch overhead of 2 time units was assumed.

Table 3

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

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 taskset is a reasonable one: there are many real world systems that have such harmonic relationships between their task periods.

Figure 5 illustrates the remaining processor utilisation for various server periods for the taskset in Table 3. 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%.

By comparison, Figure 6 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 6 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 easy to see the advantage of binding tasks to the servers.

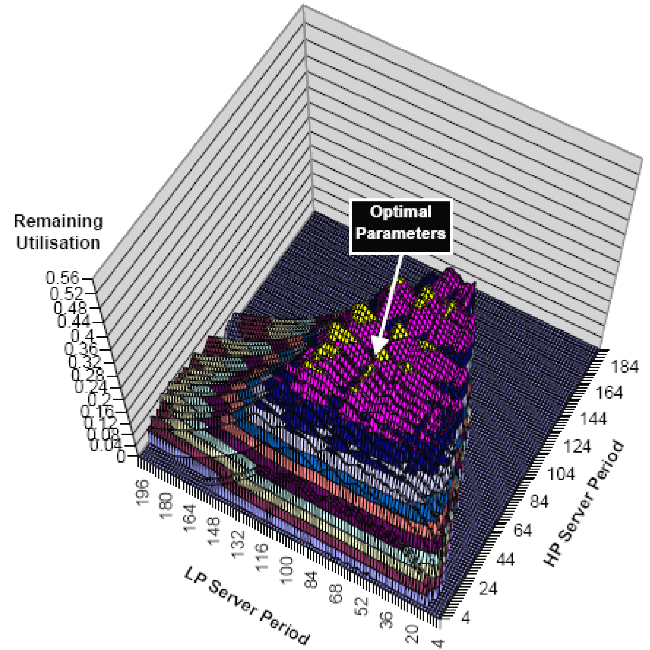

Figure 5

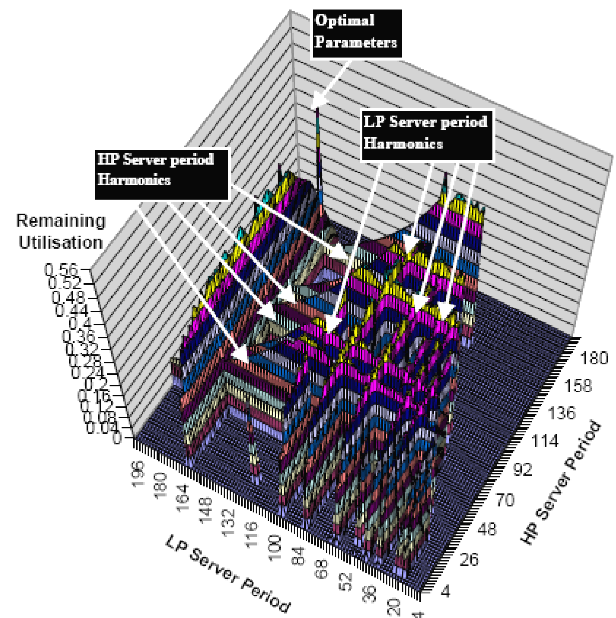

Figure 6

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 low priority server and 48, 60, 96, 120 and 160 for the high priority 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%).

Recall that task deadlines were less than periods for the taskset 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.

Permitting tasks to be bound to their server results in solutions that are very different, in terms of server utilisation, from those that are available when all the tasks are 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.

4.3. Additional Experiments

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

- 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.
- 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 a significant increase in remaining utilisation can be achieved.
- 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 longer server periods lead to lower overheads.

5. Summary and conclusions

In this paper we investigated the problem of selecting appropriate server parameters for a single processor system, running multiple applications using hierarchical fixed priority pre-emptive scheduling. 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.

5.1. Contribution

The major contributions of this work are as follows:

- Providing a set of algorithms that determine the optimal value for one server parameter (capacity, period, or priority) when the other two parameters are fixed.
- Showing that in general 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. Deriving local optima (for each server) does not lead to the globally optimal solution.
- Showing that whilst Rate Monotonic Priority Assignment (RMPO) is the optimal priority assignment policy for Periodic Servers when only server schedulability is considered, this is no longer the case when the schedulability of tasks executed by the server is also taken into account.
- Illustrating the increase in remaining utilisation (spare capacity) that can be achieved by choosing server periods that are exact divisors of their task periods. This enables tasks to be bound to the release of their server, greatly enhancing task schedulability.

5.2. Future work

Today it is possible using the analysis techniques described in this paper 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 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 [17] as the policy of choice at both global

and local scheduling levels.

6. Acknowledgements

This work was partially funded by the EU funded FRESCOR project.

7. References

[1] T-W. Kuo, C-H. Li. "A Fixed Priority Driven Open Environment for Real-Time Applications". In *proceedings of IEEE Real-Time Systems Symposium*, pp. 256-267, IEEE Computer Society Press, December 1999.
[2] Z. Deng, J.W-S. Liu. "Scheduling Real-Time Applications in an Open Environment". In *proceedings of the IEEE Real-Time Systems Symposium*. pp. 308-319, IEEE Computer Society Press, December 1997.
[3] X. Feng, A. Mok. "A Model of Hierarchical Real-Time Virtual Resources". *In proceedings of IEEE Real-Time Systems Symposium*. pp. 26-35, IEEE Computer Society Press, December 2002.
[4] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment" *JACM*, 20 (1) 46-61, January 1973.
[5] S. Saewong, R. Rajkumar, J. Lehoczky, M. Klein. "Analysis of Hierarchical Fixed priority Scheduling". In *proceedings of the ECRTS*, pp. 173-181, 2002.
[6] I. Shin, I. Lee. "Periodic Resource Model for Compositional Real-Time Guarantees". *In proceedings of the IEEE Real-Time Systems Symposium*. pp. 2-13, IEEE Computer Society Press, December 2003.
[7] G. Lipari, E. Bini. "Resource Partitioning among Real-Time Applications". In *proceedings of the ECRTS*, pp July 2003.
[8] L. Almeida. "Response Time Analysis and Server Design for Hierarchical Scheduling". In *proceedings of the IEEE Real-Time Systems Symposium Work-in-Progress*, December 2003.
[9] G. Bernat, A. Burns. "New Results on Fixed Priority Aperiodic Servers". In *proceedings of the IEEE Real-Time Systems Symposium*, pp. 68-78, IEEE Computer Society Press, December 1999.
[10] N.C. Audsley, A. Burns, M. Richardson, A.J.Wellings. "Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling". *Software Engineering Journal*, 8(5) pp. 284-292, 1993.
[11] T.P. Baker. "Stack-based Scheduling of Real-Time Processes." *Real-Time Systems Journal* (3)1, pp. 67-100, 1991.
[12] L. Sha, J.P. Lehoczky, R. Rajkumar. "Solutions for some Practical Problems in Prioritised Preemptive Scheduling" In *proceedings of the IEEE Real-Time Systems Symposium*, pp. 181-191, IEEE Computer Society Press, December 1986.
[13] R.I. Davis, A. Burns "Hierarchical Fixed Priority Pre-emptive Scheduling" In *proceedings of the IEEE Real-Time Systems Symposium,* pp. 389-398, IEEE Computer Society Press, December 2005.
[14] R.I. Davis, A. Burns "Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems" In *proceedings of the IEEE Real-Time Systems Symposium,* pp. 257-268, IEEE Computer Society Press, December 2006.
[15] EU Information Society Technologies (IST) Program, Flexible Integrated Real-Time Systems Technology (FIRST) Project, IST-2001-34140.
[16] M. Caccamo, L. Sha. "Aperiodic Servers with Resource Constraints" In p*roceedings of the IEEE Real-Time Systems Symposium*. pp. 161-170, IEEE Computer Society Press, December 2001.
[17] G. Lamastra, G. Lipari, L. Abeni. "A Bandwidth Inheritance Algorithm for Real-Time Task Synchronisation in Open Systems" In *proceedings of the IEEE Real-Time Systems Symposium*, pp. 151-160, IEEE Computer Society Press, December 2001.
[18] L. Sha, R. Rajkumar, J.P. Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". *IEEE Transactions on Computers*, 39(9): 1175-1185, 1990.
[19] C.L. Liu, J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment", *Journal of the ACM*, 20(1): 46-61, January 1973.
[20] J.Y.-T. Leung, J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-time Tasks". *Performance Evaluation*, 2(4): 237-250, December 1982.