Resource Sharing in 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 focuses on resource sharing in hierarchical fixed priority pre-emptive systems where a number of separate applications, each with its own server, reside on a single processor. It defines the Hierarchical Stack Resource Policy, an appropriate global resource access policy that bounds priority inversion and also limits interference due to overruns during resource access.

The paper provides detailed response time analysis enabling the schedulability of application servers and tasks to be determined for systems with local and global resource access. This analysis is applicable to real-world systems where server-based applications need mutually exclusive access to shared resources such as communications buffers, peripheral devices, operating system calls and data structures shared with interrupt handlers.

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. In addition to the processor, tasks in disparate applications typically also need to share resources that must be accessed under mutual exclusion. For example, memory-mapped peripherals such as CAN controllers, FLASH memory used to store diagnostic information, and data structures shared with interrupt handlers. Tasks may also contain critical sections during which they cannot be pre-empted. These sections may correspond to operating system calls or intervals when interrupts are disabled to facilitate mutually exclusive access to data structures shared with interrupt handlers.

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.

This 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. Local resource sharing can be handled using protocols developed for monolithic, as opposed to hierarchical, fixed priority pre-emptive systems. Appropriate protocols, mechanism and schedulability analysis for global resource sharing are the focus of this paper.

1.1. Related work

1.1.1.Hierarchical fixed priority pre-emptive scheduling. 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. 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.

1.1.2 Resource sharing. In fixed priority pre-emptive systems, the need for mutually exclusive access to shared resources leads to *priority inversion* where a high priority task is blocked waiting for a low priority task to release a resource. In 1990, Sha et al. [18] showed that without an appropriate protocol for resource access, such priority inversion can be essentially unbounded; a low priority task accessing a resource can be pre-empted by tasks of medium priority further delaying execution of the blocked high priority task. A number of protocols based on *priority inheritance* and *priority ceilings* [18] have been developed to limit this priority inversion. The most important and widely used of these is the Stack Resource Policy (SRP) developed in 1991 by Baker [11], extending the Priority Ceiling Protocol of Sha et al. [18].

The SRP associates a priority ceiling with each resource. This ceiling priority is equal to the highest priority of any task that accesses the resource. At run-time, when a task accesses a resource, its priority is immediately increased to the ceiling priority of the resource. Thus SRP bounds the amount of blocking (or priority inversion) which a task is subject to, to the maximum time for which any lower priority task locks a resource that is shared with the task of interest or another task of higher priority. The SRP ensures that a task can only ever be blocked prior to actually starting to execute. This minimises the number of context switches, means that all the tasks can execute on a single stack, and prevents deadlock.

In 1999, Kuo and Li [1] showed that if tasks in hierarchical systems share resources locally (strictly within an application) according to the SRP 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 that is shared with the task of interest or a task of higher priority. Kuo and Li assumed that execution

[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.

of the task is suspended when the server's capacity is exhausted, even if the task currently has a resource locked. This scheme is highly effective for sharing local resources, however it is inappropriate for sharing global resources. Suspension of a task with a global resource locked would lead to unacceptably long delays for tasks in other applications wishing to access the resource.

To address this problem, 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 resource accesses in each task, whilst its period is the greatest common divisor of task periods. Accommodating such a server has a significant negative impact on system schedulability.

In 1997, in the context of hierarchical systems based on Earliest Deadline First (EDF) scheduling, Deng and Liu [2] suggested using non-pre-emptable global resource access. This bounds the maximum amount of blocking that a task may be subject to, to the maximum time that any other task locks any global resource. This scheme has the advantage of simplicity and hence low implementation overheads, however it has the disadvantage that all tasks are subject to blocking, even those in high priority servers that do not actually access any global resources themselves.

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 under EDF scheduling. Ghazalie and Baker recognised that if task execution was suspended during resource access, due to exhaustion of server capacity, excessive periods of blocking would ensue. They proposed that if a server's capacity is exhausted with a global resource locked then the server should be allowed to overrun until the resource is unlocked. 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.

In [14], Niz et al provided a brief description of the *multi-reserve PCP* scheme. This scheme enforces execution time budgets on task execution and on resource access via the use of reserves (servers). The multi-reserve PCP scheme uses task priorities to determine multi-reserve priorities in a way that emulates the SRP. However, its applicability appears to be limited to systems with only one task per server.

1.2. Organisation

Section 2 describes the terminology, notation and system model used in the rest of this paper. It also re-visits the schedulability analysis for independent applications in hierarchical fixed priority pre-emptive systems given in [13]. Section 3 defines the Hierarchical Stack Resource Policy (HSRP) for global resource access. This policy combines server and task ceiling priorities with a server overrun and payback mechanism to limit both priority inversion and interference. Section 4 provides response time analysis for applications sharing both local resources using the SRP and global resources using the HSRP. Section 5 provides an example examining server and task response times under the HSRP. Section 6 makes a number of recommendations about using the HSRP in real-world systems. Finally, section 7 summarises the major contributions of this paper and suggests directions for future research.

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.

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 minimum 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 given in this paper assumes that both tasks and servers have deadlines that are no greater than 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. This restriction is lifted in section 4 and the analysis extended to take account of blocking effects due to tasks accessing resources that are shared locally within a single application or globally between tasks in multiple applications.

In this paper we consider applications scheduled under a simple *Periodic Server*. The analysis presented is extensible 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).

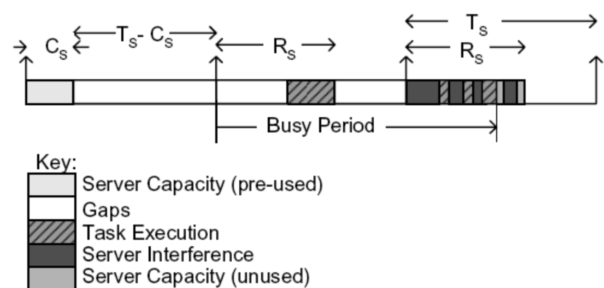

Figure 1 Busy period

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:

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

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 $T_S - C_S$ due to the operation of the server.

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

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

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:

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

The full extent of the busy period, including interference from higher priority servers in the final server period, can be found using the following recurrence relation, presented in [13]:

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

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. Resource access policies

In this section, we describe resource access policies, mechanisms and schedulability analysis for both local and global resource access.

3.1. Local resources

3.1.1 Stack Resource Policy. Following Kuo and Li [1], we assume that access to local resources is according to the SRP:

1. Associated with each local resource is a ceiling priority. This local ceiling priority is equal to the highest priority of any task that accesses the resource.
2. Whilst a task accesses a local resource, its priority is increased to the local ceiling priority of the resource.
3. If the capacity of the task's server is exhausted whilst the task is accessing a local resource, then the server simply suspends execution.

3.2. Global resources

We assume that there is a set of globally shared resources *G*. Each application 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 $b_{r,i} < C_i$ and $b_{r,i} < C_S$. We note that for a well-constrained application $b_{r,i}$ will typically be much smaller than these values.

3.1.1 Hierarchical Stack Resource Policy. We assume that access to global shared resources is according to the Hierarchical Stack Resource Policy (HSRP) defined below. The HSRP is based on the SRP, extended to hierarchical systems, and utilising the overrun and payback mechanism described in [15]:

1. Associated with each global resource is a ceiling priority. This is a global (i.e. server) priority level that is equal to the highest priority of any server that executes a task that also accesses the resource.
2. Whilst a task accesses a global shared resource the priority of its server is increased to the global ceiling priority associated with the resource.
3. Whilst a task accesses a global shared resource, the priority of the task itself is increased to the highest local priority level within its application.
4. If the server's capacity is exhausted whilst a task

has a global resource locked, then the server continues to execute the task until the resource access is completed.
5. (Optionally) 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.

It might seem that point 3 above unnecessarily increases the task's priority in the case where the task belongs to the highest priority server involved in access to the resource. Under the SRP we might expect such a task's priority to be increased so it is equal to the highest priority of any task in the application that accesses the resource, but not necessarily equal to the highest priority of *any* task in the application. However there is a good reason for making global resource accesses effectively non-pre-emptive with respect to other tasks in the same application. As the server is permitted to overrun whilst a global resource is locked, the maximum overrun is determined by the overall time for which the resource is held. To avoid a detrimental effect on schedulability, the HSRP must ensure that this time cannot be extended via pre-emption by other tasks in the same application.

3.3. Budget enforcement

To ensure that erroneous behaviour of one application cannot cause tasks in another application to miss their deadlines, a run-time mechanism is required that ensures that any task accessing a global resource *r* cannot exceed its budgeted time $b_{r,i}$ in that resource. We assume that if a task exceeds this budget, then its execution is abandoned and its server's priority reset. This prevents further failures in other tasks and servers. It is beyond the scope of this paper, which is concerned with schedulability analysis, to consider the consequences of this abandonment, but typically the task could be aborted and the resource returned to a stable state via a roll-back mechanism.

3.4. Notation

B_{SO} is defined as the longest time for which any task in server *S* may access a global resource. B_{SO} is effectively the *server overrun time,* equal to the longest time that server *S* may overrun.

B_S is defined as the longest time for which a task, in a server of lower priority than *S*, can access a resource that has a ceiling priority equal to or higher than *S*. B_S corresponds to the longest time that an invocation of server *S* can be blocked from executing by a server of lower priority.

B_i is defined as the longest time for which a task in the same application and of lower priority than task τ_i, accesses either: (i) a global shared resource or (ii) a local shared resource with a ceiling priority greater than or equal to the priority of τ_i. B_i corresponds to the longest time that a task in the same application and of lower priority than task τ_i can execute at priority *i* or higher during a priority level *i* busy period.

4. Schedulability with shared resources

4.1. Server schedulability

The SRP, used for local resource access, does not alter server priorities. As task execution is suspended when server capacity is exhausted during local resource access, local resource access cannot cause server overruns. Hence local resource access has no effect on server schedulability.

The HSRP, used for global resource access, does alter server priorities and can result in server overruns. The worst-case effects on the schedulability of a server *S* due to global resource access under HSRP 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*. Resource *r* has the longest access time B_S, of any global resource shared by a task in a server of lower priority than *S* and another task in server *S* or a server of higher priority than *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 accessing global resources. With the overrun and payback mechanism, the first invocation of each higher priority server in the busy period of *S*, has an execution time of $C_X + B_{XO}$, whilst subsequent invocations have an execution time of C_X as their capacity is reduced by B_{XO} but they may also overrun by B_{XO}.

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

$$w_S^{n+1} = C_S + B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} B_{XO} + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w_S^n}{T_X} \right\rceil C_X \qquad (5)$$

The recurrence relation given by Equation (5) starts with $w_S^0 = 0$ and ends when either $w_S^{n+1} = w_S^n$ in which case w_S^{n+1} gives the worst-case response time of the server or when $w_S^{n+1} > T_S$ in which case the server is unschedulable.

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 overrun by server *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, interference due to any overrun plus the replenished capacity of the server cannot exceed C_S. Again this must be completed by the end of the server period. There may of course be a further global resource access causing an overrun at the end of this server period, however the same argument applies. Hence for the server to be schedulable only C_S needs to be accommodated in each server period.

An alternative formulation is possible if we relax the rule that any server overruns are deducted from the subsequent replenishment capacity. In this case, the schedulability analysis is formulated as if each higher priority server *X* had a capacity of $C_X + B_{XO}$.

$$w_S^{n+1} = C_S + B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w_S^n}{T_X} \right\rceil (C_X + B_{XO}) \quad (6)$$

Note that whilst server *S* completes execution of its normal capacity in the response time given by Equation (6), we must also ensure that any overrun by *S* cannot impact its own next invocation. Hence we must ensure that the busy period calculated by Equation (7) is also less than the server's period.

$$w_S^{n+1} = C_S + B_{SO} + B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{w_S^n}{T_X} \right\rceil (C_X + B_{XO}) \quad (7)$$

Although Equation (6) leads to longer response times than Equation (5); it may be preferable, for systems with short global resource access times, to simply allow for server overruns without the additional overheads of monitoring the overrun and adjusting the subsequent capacity replenishment.

4.2. Task schedulability

In a hierarchical system, task schedulability depends upon two factors, both of which are increased by resource access.

1. The worst-case load that must be executed during the busy period of the task.
2. The worst-case time that the server takes to execute this load.

4.2.1. Task load. Accesses to local and global resources have an effect on the amount of task load that has to be executed before task τ_i can be completed. The SRP serialises local resource access such that only one task of a lower priority than task τ_i, can access a shared resource with a ceiling priority higher than or equal to priority *i* at any given time. Similarly, the HSRP serialises access to global resources such that only one task in server *S* can access a global shared resource at any given time. These protocols are compatible and proper nesting of local and global resource accesses is permitted.

Taking global and local resource access into account, the function $L_i(w)$, determining the maximum task load at priority *i* and above ready to execute in a window of length *w* is given by:

$$L_i(w_i) = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \quad (8)$$

where B_i is the blocking factor due to local and global resource access defined in section 3.4 and J_j is the release jitter of task τ_j. Note J_j is increased due to the operation of the server by $T_S - (C_S - B_{SO})$ with the payback mechanism, or by $T_S - C_S$ without the payback mechanism.

4.2.2 Server execution. Accesses to globally shared resources have three effects, each of which increases the time that the server can take to execute a given load:

1. The server's capacity is exhausted earlier in its period. This occurs when the previous invocation of the server overran by B_{SO} due to a task accessing a global shared resource. This overrun must be paid back, decreasing the server's replenishment capacity to $C_S - B_{SO}$ and thus increasing the maximum time between exhaustion of this capacity and the next replenishment to $T_S - C_S + B_{SO}$ as shown in the second server period in Figure 2.
2. 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. This resource access has a ceiling priority equal to or higher than that of server *S* and a length B_S.
3. 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. Assuming that the payback mechanism is used, this means that the first release of each server has an execution time of $C_X + B_{XO}$, whilst subsequent releases have an execution time of C_X as their capacity is reduced by B_{XO} and they overrun by B_{XO}.

4.2.3 Worst-case scenario. Figure 2 illustrates the critical instant leading to the worst-case response time of task τ_i.

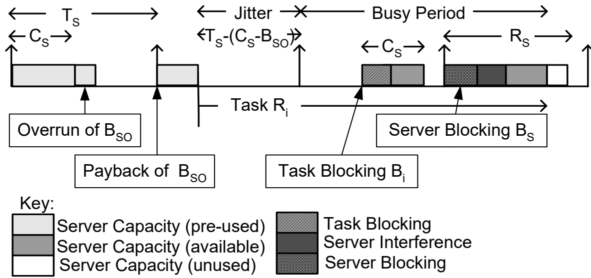

Figure 2: Critical instant

We note that when it is a global resource access that defines the longest blocking period B_i, then the scenario shown in Figure 2 cannot actually occur. This is because global resource access immediately prior to the server capacity being exhausted leads to an overrun as illustrated in Figure 3.

It is easy to show that the scenario in Figure 3 cannot result in a longer response time than that in Figure 2. (For task loads that exceed C_S, the response time is the same in both cases. For task loads $\leq C_S$, the response time is less for the scenario in Figure 3 as the task completes execution using a smaller amount of server capacity in the 3rd server period illustrated).

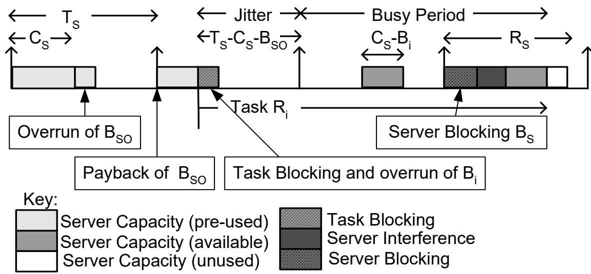

Figure 3

Whilst global resource access might result in a potentially shorter response time, it is also the case that such resource access could be completed immediately prior to exhaustion of the server's normal capacity in the second server period shown in Figure 3. This has the potential to cause 'push through' blocking where at most B_i units of additional high priority task execution are pushed into the busy period of task τ_i, effectively producing the same worst-case load and execution scenario as depicted in Figure 2.

To ensure that our analysis computes a sufficient worst-case response time in all circumstances, we assume the behaviour shown in Figure 2, even when a global resource is responsible for the maximum task blocking time B_i. We recognise that this may result in pessimism in the analysis of tasks where all of the following hold:

1. The blocking factor B_i is due to global resource access by lower priority tasks in the same application rather than local resource access.
2. There are no higher priority tasks that can provide additional interference in the form of 'push through' blocking.
3. The task's worst-case response time is less than $2T_S - C_S$.

4.2.4 Response times. Incorporating the server blocking factors into the response time analysis equations, and assuming that the overrun and payback mechanism is used, the length of the priority level *i* busy period required for the server to execute the task load is given by:

$$w_i^{n+1} = L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} B_{XO} + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right)}{T_X} \right\rceil C_X \quad (9)$$

Where $L_i(w_i)$ is given by Equation (8) and is recalculated on each iteration of the recurrence relation. 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. (Note that the task jitter is increased by $T_S - (C_S - B_{SO})$ when the overrun and payback mechanism is used).

$$w_i^{n+1} = L_i(w_i^n) + \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right)(T_S - C_S) + B_S + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{\max\left(0, w_i^n - \left(\left\lceil \frac{L_i(w_i^n)}{C_S} \right\rceil - 1 \right) T_S \right)}{T_X} \right\rceil (C_X + B_{XO}) \quad (10)$$

An alternative formulation is possible if we relax the rule that any server overruns are deducted from the subsequent replenishment capacity. In this case, the length of the priority level *i* busy period required for the server to execute the task load is given by Equation (10). The task load is again given by Equation (8). In

this case the task jitter is increased by $T_S - C_S$ due to the operation of the server.

4.3. Non-pre-emptive global resource access

We observe that a simple scheme utilising non-pre-emptive execution for global resource access, as suggested in [2], is effectively a special case of the HSRP (without the payback mechanism), where the global ceiling priority of each resource is set to the highest priority of *any* server, as opposed to the highest priority of those servers with tasks that access the resource. As such, non-pre-emptive execution of global resources can be analysed using the schedulability analysis for the HSRP, i.e. Equations (6), (7) and (10), provided that an appropriately revised blocking factors (B_S) are used.

A simple non-pre-emptive scheme for global resource access gives rise to blocking factors that are always at least as large as those for the HSRP and so the HSRP dominates the non-pre-emptive approach in terms of both server and task schedulability.

With non-pre-emptive execution, just one long global resource access in a low priority application is enough to result in a large blocking factor for all higher priority applications (tasks and servers) irrespective of whether or not they share that resource. For this reason, non-pre-emptive global resource access is inappropriate for many hierarchical systems. However, for systems where *all* global resource access times are very short, the benefits of simplicity and lower implementation overheads may be enough to warrant considering a simple non-pre-emptive mechanism despite the potential for increased blocking times.

5. Example

In this section, we use a simple example system to illustrate the effects of local and global resource access on server and task worst-case response times using the HSRP with and without the payback mechanism.

The example system comprises three applications, each scheduled under a Periodic Server defined by the parameters given in Table 1. (The times are in microseconds – µS).

Table 1: Server parameters

Server	Period	Capacity	T-C	U
S_A	2000	500	1500	25%
S_B	10000	2500	7500	25%
S_C	20000	5000	15000	25%

Tasks in all three applications access a globally shared resource for a maximum of 350µS. (We note that this time is relatively large in comparison to the capacity of server S_A, this is deliberate to better illustrate the potential effects of long resource access times).

Table 2: Server response times

Server	(1) No Resources	(2) HSRP No Payback	(3) HSRP & Payback
S_A	500	850	850
S_B	3500	5400	4700
S_C	10000	19200	14700

Table 2 gives the worst-case response time of the servers assuming (1) no global resource access, (2) global resource access using the HSRP without the payback mechanism (3) global resource access using the HSRP with the payback mechanism.

Note that where the payback mechanism is employed, as mentioned earlier, we need only consider the server's response time to the exhaustion of its normal capacity. In contrast, in the overrun and no payback case, we must include the server's maximum overrun time and check that the resulting busy period does not exceed the server's period. In our example, the busy periods for the overrun and no payback case exceed the response times given in the second column of Table 2 by 350µS.

It is evident from our schedulability analysis and the values in Table 2 that global resource accesses have a cumulative effect on server response times and busy periods. In particular, permitting server overruns without payback dramatically increases the total interference on lower priority servers. The busy period of server S_C without payback (19550µS) is almost twice that of the non-blocking case (10000µS) and close to being unschedulable. Using the payback mechanism limits this cumulative interference resulting in improved server schedulability.

We now consider task response times. Server S_B executes the set of tasks defined by the parameters in Table 3 below. Task jitter is assumed initially to be zero, increased only by the operation of the server.

Table 3: Task parameters

Task	T	D	C	U
τ_1	25000	25000	2300	9.6%
τ_2	50000	50000	4800	9.2%
τ_3	100000	100000	2400	2.4%

In addition to global resource accesses, these tasks also all access a local shared resource for a maximum of

500µS. Table 4 gives the worst-case response times of the tasks under server S_B assuming (1) no resource access, (2) local resource access using the SRP and global resource access using the HSRP without the payback mechanism (3) local resource access using the SRP and global resource access using the HSRP with the payback mechanism.

Table 4: Task response times

Task	(1) No Resources	(2) HSRP No Payback	(3) HSRP & Payback
τ_1	10800	19000	19350
τ_2	40400	42800	42450
τ_3	89200	90750	90750

There are two interesting conclusions to be drawn from the task response times.

Firstly, resource access can increase the task load sufficient to require a further invocation of the server to complete execution. This effect is responsible for increasing the response time of τ_1 from 10800µS in the non-blocking case to at least 19000µS when resource access is accounted for.

Secondly, using the payback mechanism may or may not improve task response times. Although the interference from higher priority servers in the final server period is either the same or less when the payback mechanism is used, the server induced task jitter is greater. Combined, these two effects may result in task response times that are unchanged, increased or decreased.

The example illustrates all three possibilities. In the case of task τ_1, the task load executed during the final server period that completes execution of task τ_1 is 300µS. Accounting for server blocking and overruns, the busy period is only long enough for one invocation of S_A to interfere before τ_1 is completed. This means the overall task busy period is the same for both overrun without payback and overrun & payback mechanisms at 11500µS. As the server induced task jitter is less in the case of overrun without payback, the overall response time is shorter (19000µS v 19350µS).

For task τ_2, the task load executed during the final server period is 2400µS, leading to interference from three invocations of S_A before τ_2 is completed. The two additional overruns more than counteract the increased server induced jitter and the overall response time is shorter with the payback mechanism (42450µS v 42800µS).

Finally, for task τ_3, the task load executed during the final server period is 1200µS, leading to interference from two invocations of S_A before τ_3 is completed. The additional overrun exactly counteracts the increased server induced task jitter and the response times are the same with and without payback.

6. Recommendations

It is evident from the response time analysis and our example that long accesses to global shared resources in hierarchical fixed priority pre-emptive systems can have a large cumulative impact on the schedulability of both servers and application tasks. Clearly it is advisable to make such resource accesses as short as possible to limit their impact on system schedulability.

In practical applications, there are arguments both for and against employing the payback mechanism within the HSRP. Omitting the payback mechanism has the advantage of simplicity. However including the payback mechanism may improve system schedulability.

Ultimately, the choice whether or not to include the payback mechanism depends upon the parameters of the system. If all of the servers have similar periods, then it is unlikely that the payback mechanism would provide any advantage. If global resource access times are short, then any advantage that the payback mechanism has may be outweighed by the additional complexity of implementation. However, in other systems, typically those with a wider range of server periods and longer global resource access times, the advantages of the payback mechanism can be significant; resulting in the ability to support larger server capacities and hence reduce task response times.

7. Summary and conclusions

In this paper we considered the problem of scheduling a number of applications on a single processor using a set of servers. Application tasks were permitted to make mutually exclusive access to resources that were shared either locally within the same application, or globally, between applications.

The motivation for this work comes from the automotive and avionics 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. These applications typically require mutually exclusive access to both local shared resources such as data buffers within an application and global shared resources such as shared communications devices and other memory mapped on-chip peripherals, as well as executing system calls and other critical sections where interrupts are disabled.

7.1. Contribution

The major contributions of this work are:

- Improved understanding of the impact of resource sharing in server based systems, leading to the realisation that limiting the length of such critical sections is highly desirable – even more so than in monolithic systems.
- Definition of an appropriate resource locking protocol for hierarchical fixed priority pre-emptive systems. This Hierarchical Stack Resource Policy (HSRP) combines ceiling priorities to limit priority inversion and hence blocking of high priority application tasks and an overrun and (optional) payback mechanism to limit interference on low priority applications.
- Extended response time analysis catering for global and local resource access by hard real-time application tasks scheduled under a set of servers.

These contributions make significantly improvements to the techniques and associated analysis available in the design and development of hierarchical multi-application, real-world systems.

7.2. Future work

Alternative approaches have been developed for hierarchical systems with a somewhat different set of assumptions scheduled using dynamic priorities. 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 pre-emptive systems.

8. Acknowledgements

This work was partially funded by the UK EPSRC funded DIRC project and the EU funded FRESCOR project. The authors would like to thank Reinder Bril for his suggestions on an early version of this paper.

9. 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 and 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] D. Niz, L. Abeni, S. Saewong, R. Rajkumar. "Resource Sharing in Reservation-Based Systems" In *proceedings of the IEEE Real-Time Systems Symposium*. pp. 171-180, IEEE Computer Society Press, 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" 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, and J.P. Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". *IEEE Transactions on Computers*, 39(9): 1175-1185, 1990.