# Rewriting History to Exploit Gain Time

G. Bernat, I. Broster and A. Burns
Department of Computer Science,
University of York
YO10 5DD, U.K.
{bernat, ianb, burns}@cs.york.ac.uk

## Abstract

*With modern processors and more dynamic application requirements it is becoming increasingly difficult to produce tight upper bounds on the worst-case execution time of real-time tasks. As a result, at run-time, considerable spare CPU capacity (termed gain time) becomes available that must be usefully employed if cost effective real-time systems are to be engineered. In this paper we introduce a scheme by which gain time is exploited by retrospectively reassigning execution time from a task's own budget to the gain time that later become available. As a result of changing the system's execution history, spare capacity is immediately reallocated and hence preserved. The proposed scheme is shown to work with fixed priority dispatching, the use of servers to provide temporal firewalls, and other capacity sharing approaches. Evaluations are provided via simulations.*

## 1. Introduction[1]

A real-time system is usually constructed as a set of concurrent tasks which may have hard or soft deadlines. Hard tasks have strict timing constraints, soft tasks often have more relaxed QoS (Quality of Service) requirements. A real-time system with this mixture of hard and aperiodic tasks has to ensure that:

- all hard real-time tasks meet their deadlines even when worst-case conditions are being experienced;
- all aperiodic tasks have a good response time. Here, good should mean that the task meets its soft deadline in most of its invocations;
- other soft tasks exhibit good quality of output by gaining access to the maximum computation time available before their soft/firm deadline.

One of the difficulties with hard real-time tasks is that a guaranteed upper bound must be given on their execution times. With modern processors it becomes increasingly difficult to produce these bounds without incorporating excessive pessimism [8]. As a result, at run-time, a hard task may perhaps use only 10% of its allocated CPU time. Even if tight WCET analysis increases this value to 80% there remains significant quantity of CPU resource not being used. In a fixed priority scheme this unused resource will become available to background (or low priority tasks). The objective of the implementation scheme described in this paper is to effectively employ the spare capacity, or *gain time* [1], generated by hard tasks not executing to their worst-case limit. This implies using gain time at the higher priority levels and where necessary preserving the gain time so that it can be used when needed by high priority soft tasks.

This need to *preserve bandwidth* has been explored in many approaches to implementing servers [14, 19, 13, 9, 4] (some of which do address gain time reclaiming [18]).

Notably, gain time reclaiming is studied in power-aware computing, where gain time is used to allow the processor to slow down, according to various on-line and off-line schemes [12, 3, 20, 15]. All these approaches for making use of gain time:

- rely on a gain-point being specifically indicated at the end of the task execution;
- must use the gain time by the by a specific time (usually the period or deadline of the task which produced the gain time).

The distinctive feature of the new scheme described in this paper is that we preserve gain time by *rewriting history*; for example, a soft task that had executed using its own budget is, retrospectively, deemed to have been using a hard task's gain time and hence its own budget is replenished and it can execute further. As Section 3 will describe,

1 This work is supported by the IST Programme of the European Commission under project IST-2001-34820 (FIRST).

this results in a scheme where a gain point does not need to be indicated at the end of a task execution and where the gain time can be preserved and used over a wide interval beyond the deadline of the task.

The details of the proposed scheme are described in Section 3 of this paper. Before that a brief description of the computational model is given. This is a standard fixed priority model. Assessment by simulation is given in Section 4. Some potential generalisation of our results are presented in Section 5 and conclusion are provided in Section 6.

## 2. Task Model

We assume a fixed set of *tasks*[2]. Each task has a potentially infinite number of (non-overlapping) invocations. When referring to an execution of a task we imply a specific invocation. Some constraint on the arrival of each hard task's invocations will be required to bound the load on the system and make it, at least potentially, schedulable. However strict periodic execution is not required.

A task, $\tau_i$ is said to have a maximum execution time of $C_i$, a minimum arrival interval[3] of $T_i$ and a relative deadline $D_i$ (relative to the arrival of any invocation of the task). Tasks also have a priority $P_i$, where the $P_i > P_j$ means that $\tau_i$ has a higher priority than $\tau_j$.

For a correctly specified system all tasks have their deadlines guaranteed by some system-wide test of schedulability. Fixed priority preemptive scheduling is considered. Therefore the most straightforward test for a fixed priority scheme is RTA (Response Time Analysis) [11, 2], where a worst case response time, $R_i$, is calculated for each task and compared to the task's deadline, $D_i$ [6].

To prevent any soft task (or hard task with a soft component, or hard task with error containment) from executing for more than its allocated execution time, budget enforcement is assumed to be provided by the run-time system. There are a number of server techniques that can be used to provide this temporal firewall. The two most popular are the Deferrable Server (DS) [14] and the Sporadic Server (SS) [19]. They have similar behaviour [4]; DS needs a modified schedulability test (see later) and SS has the advantage that it enforces a period of separation between invocations of the same task.

## 3. Exploiting Gain Time

The most important property of any scheme for exploiting gain time is that it must be *safe*; in that it must still be possible to provide scheduling guarantees to all hard tasks. However it is not necessary to apply a (perhaps complex) on-line scheduling test if a simple off-line test will allow gain time to be reused more effectively. In other words, the use of a single, slightly more restrictive schedulability test can be traded against run-time efficiency.

We define *gain time* by noting that a typical invocation of task $\tau_i$ (with worst-case execution time $C_i$) will only consume a lesser amount of CPU time which we denote by $E_i$. Gain time, $g_i$ is simply $C_i - E_i$. Note, due to budget enforcement, $g_i$ is never negative. Gain time generated at priority $P_i$ can clearly only be used by other tasks with priority $P_i$ or lower[4]; it also has a limited 'shelf life'[5]. To fully exploit the available gain time we first maximise its intrinsic shelf life and then look to further preserve its effective shelf life by 'rewriting history'.

### 3.1. Maximising intrinsic shelf life

Consider a typical invocation of task $\tau_i$ that is released at time $t_i^r$. Assume it completes at time $t_i^c$. We know, for a guaranteed system, that $t_i^c \leq t_i^r + D_i$. Indeed if RTA was used we also know that $t_i^c \leq t_i^r + R_i$. Gain time ($g_i = C_i - E_i$) is available at time $t_i^c$. To be able to maximise the time over which this gain can be used by other task, ideally it would be available until the task is next released at time $t_i^r + T_i$. If one considers the RTA equations then any lower priority task (than $\tau_i$) has a maximum level of interference from $\tau_i$ that equates to $C_i$ in every $T_i$. However it is not the case that $g_i$ is available until $t_i^r + T_i$. Gain time cannot safely be given from $\tau_i$ to $\tau_j$ unless there is a $P_j$-level busy period running from $t_i^c$ until $\tau_j$ uses this gain time (which must be before $t_i^r + T_i$). Otherwise, an additional interference would be created. To illustrate this consider the example in Figure 1 which shows the consequences of delaying the allocation of gain time until the next release of the process.

When $\tau_k$ is released at time $t_k^r$ it suffers an immediate interference of $g_i + C_i + C_j$. This value (which is upper bounded by $2C_i + C_j$) is greater than the standard RTA would predict. Task $\tau_k$ is said to suffer back-to-back interference from $\tau_i$. It receives, in effect, interference at the end of one invocation and the beginning of the next.

If we wish to preserve the intrinsic shelf life of the gain time to its maximum extent ($t_i^r + T_i$) then the scheduling test has to accommodate the back-to-back interference that can now occur. Fortunately such analysis already exists as this is exactly the behaviour that occurs with the Deferrable Server (DS) [14].

2 The term *thread* or *task* could equally have been used.

3 This is the period of a periodic task.

4 The scheduling analysis that is undertaken for fixed priority systems correctly assumes that tasks can only suffer interference from tasks of higher priority; if gain time were allocated to tasks with higher priority this could potentially increase the interference on some tasks which would not be safe.

5 This is the length of time during which the gain time can be used by other tasks before it becomes stale—unusable.

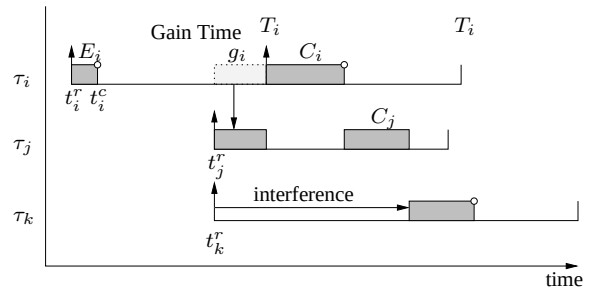



**Figure 1. Assigning Gain Time Later Creates a Double Hit**

A deferrable server is a simple server mechanism which has a capacity that is replenished to the maximum budget at fixed, periodic times. This server has recently been evaluated [4] and shown to perform well over a wide range of application characteristics. The RTA scheduling equation is modified [4] to the following form to accommodate deferrable behaviour:

$$w = C_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{w + T_j - C_j}{T_j} \right\rceil C_j \qquad (1)$$

where $T_j$ and $C_j$ refer to server period and budget and $\mathsf{hp}(i)$ is the set of tasks with higher priority than task $\tau_i$.

In the general task model in use in this formulation we note that budget enforcement is required because soft activities have either unbounded arrival patterns or unbounded execution time requirements. We note that the use of Deferrable Servers is an adequate means of achieving this enforcement. We are therefore able to combine budget enforcement and gain time reallocation efficiently by the use of the DS scheme. Thus we assume the use of DS in the rest of this paper.

However, as noted by Saewong *et al.* [16], "The budget cannot be saved for future use, which means that any unclaimed budget left from the previous replenishment is always thrown away at the next replenishment." Our aim is to prevent this capacity from being thrown away.

### 3.2. Capacity Sharing

With the above scheme gain time is available for 'reuse' from the time that the task finishes ($t_i^c$) until the completion of the task's period ($t_i^r + T_i$). During this time any task of a lower priority that is running, can request extra capacity from any higher priority task with gain available. This is achieved using the capacity sharing algorithm [7, 5] that has recently been investigated.

Capacity sharing is an effective means of reclaiming gain time. However one limitation is that the gain time shelf-life is limited by the time that the end of the task execution occurs (the *gain point* at $t_i^c$). If the gain point is close to the period of the server then the availability of the gain time is far less than if the task finishes its execution early in the server period.

Additionally, it is not always possible to determine the time of the gain point, for example if a deferrable server is managing the load caused by an aperiodic task. In this situation, the earliest time that the gain point can occur is at the server period, in which case no gain time can be reclaimed, even though there is unused capacity available at time $t_i^r + T_i$.

The scheme explained in Section 3.3 is complementary to capacity sharing and can be used both with and without capacity sharing in effect. This method provides a way of exploiting any gain time that is still available at time $t_i^r + T_i$ that would otherwise be lost when the server is replenished. We do this by rewriting history.

### 3.3. Rewriting history

As discussed in the introduction, our aim is to keep the gain time generated at priority $P_i$ as a resource to be used at priority levels close to $P_i$ rather than let the generated CPU capacity be used at low or background priority. We accomplish this by assigning the maximum extra capacity to the task with priority $P_i - 1$. The maximum is bounded by the amount of execution time this task has already consumed (in its current invocation). So if task $\tau_j$ with priority $P_i - 1$ has executed for $e_j$ then some, or all, of this CPU time is retrospectively assumed to be taken from the budget of $\tau_i$ rather than $\tau_j$. This leaves $\tau_j$ executing for longer if it desires. If it does not then it will have more gain time to pass on when it completes, and in doing so, it propagates the shelf-life of the original gain time beyond the period of the original task.

The proposed history rewriting (HisReWri) scheme is defined in terms of the following steps:

1. At the start of a task's ($\tau_i$) invocation ($t_i^r$), it is given a budget, $b_i$, equal to its worst case execution time $C_i$.

2. During execution, the amount of CPU time used from the task's budget is monitored, we denote this as $e_i$; if $e_i$ becomes equal to $b_i$ then the task is suspended (or alternatively, if background scheduling is also used, the task's priority is dropped to background). This is normal deferrable server behaviour.

3*. At the completion of the task (at time $t_i^c$) a *first* gain point is encountered; the gain time is calculated, $g_i := b_i - e_i$. This gain time is now available for other tasks to request up to the end of the task's period . This is the same as capacity sharing.

4. At the end of the task's period ($t_i^r + T_i$), the gain time still available is noted: $g_i^*$. Here, a *second* gain point is apparent and history rewriting is possible.

5. Loop until $g_i^*$ is all allocated ($g_i^* = 0$) or there are no more lower priority tasks:

   - consider task with next lowest priority ($\tau_j$),
   - let $m$ be the minimum of $\tau_j$ execution time ($e_j$) and $g_i^*$,
   - assign $e_j := e_j - m$,
   - assign $g_i^* := g_i^* - m$,
   - if $m > 0$ where necessary, resume task $\tau_j$ or raise priority of $\tau_j$ to $P_j$ (if dropped to background)
   - repeat if $g_i^* > 0$

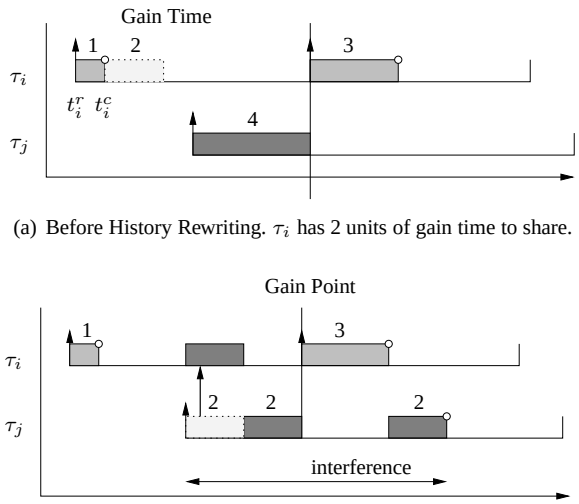
Stage 3* is capacity sharing and is optional. If a gain-point can be identified then use of capacity sharing can increase the effectiveness of gain time reclaiming. (It is also possible to apply history rewriting at the first gain point rather than the second, although it appears to be more advantageous to apply capacity sharing at the first gain point when the first gain point can be identified.)

The effect of reducing the value of $e_j$ is to allocate it extra budget, as the task is now assumed to have executed $m$ from another budget. If it had exhausted its budget and hence was assigned the background priority it has its priority raised back to it usual level. The algorithm reduces $e_j$ rather than increase $b_j$ to ensure than the same execution time quantity is not allocated more than once. Hence $e_j$ always indicates the amount of CPU time $\tau_j$ has consumed from its own budget. When the Deferrable Server replenishes its budget it does this by setting $e_j$ to zero.

Note, gain time is always assigned to tasks with later completion times, and hence it is being preserved beyond the period of the generating task. As time progresses the gain time will either be used or it will eventually propagate to lower and lower priorities. The shelf-life of the gain time is prolonged, but this cannot, of course, be done indefinitely. On some occasions the active invocation of $\tau_j$ will not have executed and hence it cannot receive increased budget. At other times $\tau_j$ will have completed; in which case the new budget will not be used but passed on when the end of the period of $\tau_j$ is reached.

### 3.4. Example

Before a more formal analysis of the scheme two examples will be given. We assume in the following, a gain point *only* at the period of the server, $t_i^r$, thus capacity sharing (stage 3) is not applicable. In both examples, the first




(a) Before History Rewriting. $\tau_i$ has 2 units of gain time to share.

(b) After History Rewriting. $\tau_i$ is considered to have used two units of time that were really used by $\tau_j$. Thus, $\tau_j$ still has 2 units of budget to use after the gain point.

**Figure 2. Example of rewriting history where gain is assigned to a task that executed between $t_i^r$ and $t_i^r + T_i$. Server budgets are $b_i = 3, b_i = 4$.**

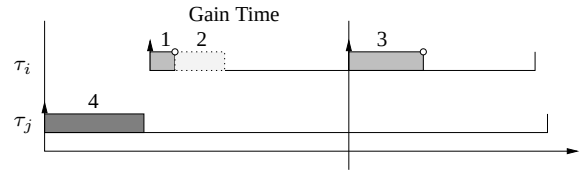
invocation of task $\tau_i$ only executes for 1 unit of its allocated 3; $\tau_j$ benefits by using the gain of 2 units to execute for 6 (rather than 4) units of time.

It is clear, by inspection, that in the first example of history rewriting, shown in Figure 2, no lower priority task will suffer more interference when the gain time has been redistributed than the double hit scenario explored in Section 3.1. Note that we only need to consider the effects after the gain point: rewriting history is done at the gain point and it cannot cause a deadline miss in the past.

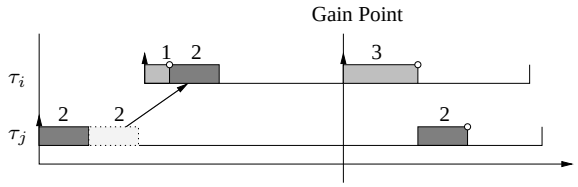
However, the second example, shown in Figure 3, requires a more formal treatment. In this example, the 2 units of gain time from $\tau_i$ are retrospectively assigned to $\tau_j$ even though $\tau_j$ executed *before* $\tau_i$ was released.

**Theorem 1.** *No task that is deemed schedulable by the Deferrable Server scheduling test will miss its deadline when the HisReWri scheme is applied.*

*Proof.* Consider the situations where gain time $g$ is passed at time $t_i^c$ from $\tau_i$ with priority $P_i$ to task $\tau_j$ with priority $P_j$. Note gain time is always passed to lower priority tasks and hence $P_j < P_i$.
Clearly all tasks with priority above $P_i$ are unaffected by this transfer. Moreover any task with priority above $P_j$ but below (or equal) to $P_i$ will experience at most a reduced level of interference and hence will continue to be schedulable.




(a) Before History Rewriting. $\tau_i$ has 2 units of gain time to share.




(b) After History Rewriting. $\tau_i$ is considered to have used two units of time that were really used by $\tau_j$ even though $\tau_j$ executed before $\tau_i$ was released.

**Figure 3. Example of rewriting history where gain is assigned to a task that executed before $t_i^r$. Server budgets are $b_i = 3, b_i = 4$.**

Now consider a task ($\tau_k$) that suffers interference from $\tau_i$ and $\tau_j$. As $g$ is passed from $\tau_i$ to $\tau_j$, $\tau_j$ executed for at least $g$ in its current invocation. The critical instance for $\tau_k$ must either be before or after $\tau_j$ so executed. These two cases are considered separately.

If the critical instance is before then there is a $P_k$-level busy period that includes the execution of $\tau_k$, the execution of $\tau_j$, the execution of $\tau_i$ (which is no greater then $C_i - g$) and then extra execution of $\tau_j$ (which is $g$). Clearly $g$ is just being transferred from $\tau_i$ to $\tau_j$, and hence $\tau_k$ will suffer no increase in interference.

If the critical instance is after the main execution part of $\tau_j$ then the maximum interference that $\tau_j$ can impose on $\tau_k$ is when there is a back-to-back execution in which $g$ occurs just before the completion of one invocation of $\tau_j$ and $C_j$ occurs as $\tau_j$ is released again. But the DS schedulability test accounts for a back-to-back interference of $C_j$ followed by another $C_j$. As $g < C_j$ it follows that the transfer of gain time is no worse than that already accounted for. □

# 4. Evaluation

Before considering, by simulation, the effectiveness of the proposed scheme, we consider the overheads of implementation. One of the motivations for the proposal is that it can be incorporated efficiently into a run-time system (or RTOS). Note that if a Deferrable Server is already in use then no new run-time data need be collected as execution times are already been monitored. All that is required is that at fixed times (at the end of a task's period; which coincides with the task's next release for periodic tasks) a simple reallocation of budgets occurs. The overhead in doing this (see the scheme description given earlier) is small and can indeed be extracted from the gain time itself so that there is no new term that needs to be incorporated into the scheduling test. If there is insufficient gain time to execute the gain time allocation algorithm then it simply does not happen.

It is possible to envisage other history rewriting schemes that may perform better for certain applications but which have higher overheads. We contend that a very simple scheme (always allocate to the next priority level) is straightforward and robust and likely to be an effective scheme for a wide range of applications.

To evaluate the use of history rewriting as a gain time reclaiming mechanism, we consider two metrics. Firstly, we evaluate how much gain time can actually be reclaimed by this mechanism, by measuring the increase in utilisation. Secondly, we compare how effective gain time reclaiming is at reducing the response times of aperiodic activities.

Both of these metrics together are needed to assess the effectiveness of history rewriting; neither of them alone is useful. To understand this consider alternative schemes. Background scheduling (lowering a priority to background when a server is exhausted), for example will always be able to 'reclaim' 100% of gain time, however the time taken for this gain time to become available means that response times are large. On the other hand, scheduling aperiodic activities before hard tasks will result in very short response times, but does not easily allow much gain time to be reclaimed.

## 4.1. Quantity of Gain Time Reclaimed

To evaluate how much gain time can be reclaimed, a small set of 6 tasks is simulated, consisting of 3 hard tasks (which execute for a random time which is less than their budget) and three soft tasks. The specific soft tasks used for this section are *unbounded*, i.e. they will use any bandwidth that they are allowed to use.

This soft tasks in the set are motivated by the use of imprecise computation models [17], such as *anytime algorithms* [10]. Anytime provides a useful scheme for integrating complex and unbounded calculations into real-time systems. We may describe an anytime algorithm as one which may be interrupted at any time to give an answer. If the algorithm is left to run for longer then a more accurate result is returned. Anytime algorithms are frequently used for exploring large search spaces and in artificial intelligence. Many complex calculations/algorithms are amenable for transformation to an anytime algorithm structure [21]. In the following, an anytime process is termed an *unbounded task*.

The priority ordering of the hard (H) and unbounded (U) tasks is such that they are interleaved (highest priority first): *HUHUHU*. Thus, any spare bandwidth unused by a hard task could potentially be reclaimed by any subsequent unbounded task. Deferrable servers are used to limit the bandwidth used by each task.

The tasks and servers are listed in Table 1. There is one task per server. $C_i$ and $T_i$ refer to the server budget and period. $U_i = C_i/T_i$ is the maximum utilisation contribution of the server. The budgets of each server are selected such that the system is schedulable, where schedulable means that the worst case response time, $R_i$, calculated from equation (1), guarantees that the full server budget for each server is guaranteed to be available before the next period of the server.

| **Task** | **Server Parameters** | | | |
|---|---|---|---|---|
| | $C_i$ | $T_i$ | $U_i$ | $R_i$ |
| H0 | 100 | 1000 | 0.1 | 100 |
| U1 | 150 | 1200 | 0.125 | 350 |
| H2 | 250 | 1400 | 0.1786 | 750 |
| U3 | 450 | 2600 | 0.1731 | 1950 |
| H4 | 550 | 4500 | 0.1222 | 4250 |
| U5 | 700 | 8000 | 0.0875 | 8000 |

**Table 1. Processes and Server Parameters**

In the simulations, if the hard (H) tasks consume their worst case execution time (equal to the server budget) then we would expect to see the utilisation of each task to be equal to the server utilisation from Table 1. This forms an upper bound on the maximum utilisation that any reclaiming algorithm can achieve without resorting to background scheduling. It is 0.79 for this task set.

Note that in these experiments we do not make use of background scheduling, because any gain achieved here would be entirely dependent on the utilisation of the system including any low priority tasks that are not budgeted. Therefore, we do not consider any possible gain from background work, aiming instead to fully utilise server budgets.

The execution times of the hard tasks in the simulations are set to follow a uniform (rectangular) distribution between 5 and $C_i/2$. This is a reasonable assumption for even moderately complex tasks. The unbounded tasks still continue to use all available CPU time. Thus, if we use no reclaiming of any sort, then the utilisation of this simulation forms a lower limit which any reclaiming algorithm can be compared against. This is 0.49 in this task set.

Using history re-writing (alone, with no capacity sharing), simulation results appear in Table 2. Naturally, the simulations also confirm that no hard deadlines are missed.

The results indicate that the history rewriting mechanism is capable of reclaiming a high percentage of available gain time. Therefore we regard history rewriting as a useful mechanism.

| | |
|---|---|
| $E_i = C_i$ (No Gain Time Available) | $U = 0.79$ |
| $E_i = \mathrm{rand}(5, \frac{C_i}{2})$ but no reclaiming | $U = 0.49$ |
| Utilisation available for reclaiming | $0.79 - 0.49 = 0.30$ |
| Utilisation with history rewriting on | $U = 0.77$ |
| Utilisation Reclaimed | $0.77 - 0.49 = 0.28$ |
| Fraction Reclaimed | $0.28/0.30 = 93\%$ |

**Table 2. Quantity of Gain Time**

In comparison with capacity sharing and background scheduling, we note that in this system, both schemes together would be able to reclaim 100% of gain time. However, the 'quality' of the gain time (how early it becomes available) is considered in Section 4.2.

Although we make no claim that this task set is representative of any system, similar results have been observed with other task sets.

## 4.2. Effect on Response Times

The second set of simulations concerns the problem of reducing response times for aperiodic tasks. A system similar to that of Section 4.1 is considered, except that the unbounded tasks (U) are replaced by aperiodic tasks (A) which use 5 units of computation time every invocation. The server parameters are not altered, therefore there may be many invocations of an aperiodic task to one server period.

The aperiodic tasks are released randomly using a Poisson distribution of arrivals. The average bandwidth for the aperiodic tasks is set to be 1.4 times the available bandwidth of the respective servers. This factor is so that there is sufficient unused bandwidth from the hard tasks for all aperiodic tasks to be scheduled. Thus the system represented is experiencing an overload situation for its aperiodic tasks; the scheduling algorithm and budgeting mechanism must ensure that all hard tasks continue to meet their deadlines whilst providing the best possible service to the aperiodic tasks.

A number of experiments with combinations of history rewriting and capacity sharing were performed. Results appear in Table 3. For completeness, we consider the effect of applying history rewriting at both the first (1) and second (2) gain points noted in Section 3.3. Capacity sharing can only occur at the first gain point. For servers with aperiodic processes, only one gain point exists and is at always at the period.

The results indicate that the combination of capacity sharing and history rewriting enable gain time to be reclaimed earlier than capacity sharing (or indeed history rewriting) alone.

| $\tau$ | Average Response Time | | | | |
|---|---|---|---|---|---|
| | DS | H(1) | H(2) | CS | CS&H(2) |
| A1 | $\infty$ | 88.0 | 88.0 | 21.53 | 8.0 |
| A3 | $\infty$ | 21.8 | 21.2 | 18.18 | 14.6 |
| A5 | $\infty$ | 49.9 | 49.9 | 45.37 | 43.1 |

**Key**

**DS** Deferrable server only
**H** History Rewriting on
**CS** Capacity Sharing on
**(1)** Gain point at task completion
**(2)** Gain point at server period

**Table 3. Response Times of Aperiodic Processes**

In these experiments, the effect was most noticeable for high priority tasks, indicating that capacity is being preserved at higher priorities.

## 5. Discussion of Possible Extensions

The scheme described in this paper links history rewriting with the Deferrable Server. This has a number of advantages that have already been discussed. It is logical to link gain time reallocation with real-time systems that have a mixture of hard and soft activities. For these systems, budget control (temporal firewalling) is essential. However there are other budget control schemes. The Sporadic Server (SS) [19], for example, has the advantage that it enforces separation (for sporadic tasks) and is incorporated into the POSIX standard. History rewriting can be be used with Sporadic Servers in two different ways:

- with no change to the run-time protocol described here, but the use of the DS schedulability test; or
- with some modification to the run-time protocol and the use of the SS schedulability test.

The SS test can theoretically guarantee task sets with higher utilization than the DS test, but it has a higher run-time overhead. To rewrite history and keep with the SS test requires more information to be maintained at run-time. Gain time cannot be reallocated to a task until a period of time has elapsed after that task has executed. Whilst it is possible to develop such a protocol, we currently believe that the overheads are likely to be too high and hence we advocate the first approach identified above. However this is still the topic of further study.

## 6. Conclusions

Although it is not possible to define a single abstract real-time application, it is clear that many domains will generate systems that combine hard and soft activities—such systems may be considered *mixed criticality*. These combinations will sometimes be of quite independent activities, but they may also involve quite integrated hard and soft computations within the same application module. Another common characteristic of these systems, particularly when they execute on modern hardware, will be that the hard components will rarely (if ever) execute up to their worst-case upper bound. And hence it is crucial for the effective engineering of such real-time applications that gain time released by the system's hard tasks is used constructively by the system's soft tasks. To achieve this requires algorithms within the soft tasks for exploiting gain time, and protocols within the run-time system for maximising the amount of gain time that can be made available. This paper has described a scheme for addressing the second of these challenges.

The proposed protocol identifies those tasks that did execute when a hard task was dormant (not executing up to its maximum allocation) and retrospectively assigns their execution time to the hard task. In this way gain time is reallocated and exploited at priority levels close to those that actually generated the gain time. By always assigning the gain time to tasks that have yet to reach the end of their period, the bandwidth of the gain time to extended—it has a prolonged shelf life. As execution time is retrospectively reallocated we describe the protocol as history rewriting (HisReWri).

The use of history rewriting is orthogonal to other protocols that have been proposed to maximise spare capacity and to share out server capacity. Taken together these schemes now provide the application/system programmer with an effective and comprehensive means of structuring mixed criticality real-time system. However a multi-server scheme still has a number of free parameters to fix for the quality of behaviour desired for a specific application. This optimisation problem remains an open research question, and will be the subject of further study.

## References


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

[2] N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[3] H. Aydin, P. Mejía-Alvarez, D. Mossé, and R. Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*. IEEE Computer Society, 2001.

[4] G. Bernat and A. Burns. New results on fi xed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium*, Phoenix. USA, Dec 1999.

[5] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems Journal*, 22:49–75, 2002.

[6] A. Burns and A. J. Wellings. Engineering a hard real-time system: From theory to practice. *Software-Practice and Experience*, 25(7):705–26, 1995.

[7] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings 21st IEEE Real-Time Systems Symposium*, 2000.

[8] A. Colin and S. M. Petters. Experimental evaluation of code properties for WCET analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, December 3–5 2003.

[9] R. I. Davis and A. J. Wellings. Dual priority scheduling. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 100–109, 1995.

[10] T. Dean and M. Boddy. An analysis of time dependent planning. In *Proceedings of the Seventh National Conference on Artifi cial Intelligence*, pages 49–54, 1988.

[11] M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.

[12] C. M. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*. IEEE Computer Society, 2000.

[13] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fi xed-priority preemptive systems. In *Proceedings 13th IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[14] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[15] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, Banff, Canada, 2001. ACM, ACM Press.

[16] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fi xed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, pages 173–181, Vienna, Austria, June 2002. Computer Society, IEEE.

[17] W. K. Shih, J. W. S. Liu, and J. Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. In *Proceedings 10th IEEE Real-Time Systems Symposium*, 1989.

[18] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings 9th IEEE Real-Time Systems Symposium*, pages 251–258, 1988.

[19] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1:27–69, 1989.

[20] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):686—700, 2003.

[21] S. Zilberstein. *Operational Rationality through compilation of anytime algorithms*. PhD thesis, Computer Science Division, University of California, 1993.