Dealing With Suspending Periodic Tasks

Ragunathan Rajkumar
IBM T. J. Watson Research Center
Yorktown Heights
rajkumr@watson.ibm.com

June 1991

Abstract

Periodic tasks in preemptive real-time systems are typically assumed to execute continuously until preemption or completion. Unfortunately, periodic tasks do not always adhere to this ideal behavior under several practical conditions. A task may have to suspend pending completion of an I/O activity, and then later resume execution. A task may have to block and relinquish control while it waits to access globally accessible resources in multiprocessors and distributed systems. An output signal or message generated by a task on completion is not always generated on periodic boundaries and is referred to as a signal with jitter. These behavioral patterns can cause heavy penalties on schedulability and cause deadlines to be missed at low levels of processor utilization. In this paper, we present a simple algorithm called the Period Enforcer, which forces tasks to behave like ideal periodic tasks from the scheduling point of view with no associated scheduling penalties. We finally argue that support of this solution in preemptive real-time systems can yield large benefits.

1. Introduction

The preemptive scheduling of periodic tasks has been extensively studied in the literature [2, 3]. In most of these cases, a periodic task is assumed to behave in the following benign fashion. Once the task arrives, it becomes eligible for execution and runs to completion without suspending. In other words, the task cannot suspend itself arbitrarily, resume at a later point in time and ask to be rescheduled again. However, there are several situations where periodic tasks do not strictly adhere to this ideal behavior. The effect of this behavior is that it imposes a scheduling penalty on lower priority tasks which can be unacceptable in practical systems. Unfortunately, this non-ideal behavior can be in one of several forms such as jitter, suspension or can arise in aperiodic task scheduling. The objective of this paper is to develop a simple algorithm that eliminates the scheduling penalties in these cases under the same framework.

A periodic signal is said to encounter jitter when even though an instance occurs once every period, the instances are not separated by equal intervals. For example, suppose that a periodic task scheduled by the rate-monotonic scheduling algorithm generates an output signal after it completes its computations. Since the periodic task can be preempted by higher priority tasks and/or simply because of stochastic execution, it would complete at different times relative to its arrival. As a result, the output signal would exhibit "jitter", where the intervals between successive output signals is not the same. This jitter can also occur on the input side, where it may trigger each instance of a periodic task. If the input signal is jittery, the triggered periodic task instances would not correspond to an ideal periodic task. The general case is where an output signal from one subsystem becomes the input signal for another subsystem.

A task may also behave unlike an ideal periodic task if it has to suspend for some task-dependent reason such as the need to complete an I/O activity or to access a global resource in a distributed or multiprocessor system. In this case, the task executes for a certain amount of time. It then initiates an I/O activity, and suspends. Once the I/O activity completes, the task becomes re-eligible for execution. It executes again for some time, and then may need to access globally shared data. Since the data is being used by other task(s) on some other processor(s), the task is forced to suspend. When the task obtains access to the data, it again becomes ready for execution. During the task's suspensions, the highest priority ready task on the processor begins ex-

ecution to avoid prolonged idleness of the processor. This behavior of alternating durations of execution eligibility and suspension again does not correspond to the ideal behavior of a periodic task.

Scheduling algorithms such as the rate-monotonic algorithm were originally defined for periodic tasks alone. However, aperiodic tasks such as operator requests, emergency signals and exception conditions also exist in a typical real-time system. These aperiodic tasks must be scheduled such that the aperiodic tasks receive good response times without jeopardizing the deadlines of the periodic tasks. This need to integrate the scheduling of periodic and aperiodic tasks has given rise to the class of aperiodic server algorithms such as the Deferred Server [1, 10] and Sporadic Server [8, 9]. The key idea behind these algorithms is the creation of a high priority periodic server task with a given capacity, which is used to service incoming aperiodic tasks. Since the arrival of the aperiodic tasks can be quite random, the times at which they can be serviced can also be rather random. The net effect is that the server execution profile also need not correspond to a pure periodic task. The Sporadic Server, in particular, is designed to avoid the scheduling penalties of this non-ideal behavior.

In this paper, we investigate the problem of scheduling tasks with a tendency to deviate from the ideal behavior of a periodic task, and suggest a simple but effective solution called the *Period Enforcer*. We demonstrate that this solution can be used to address the problems of jitter, suspension, and service to aperiodic tasks. Given this situation, we argue that an implementation of this solution can yield rich benefits in preemptive real-time systems.

The rest of the paper is organized as follows. Section 2 describes the problem of deferred execution and investigates its scheduling penalties. Section 3 defines the period enforcer algorithm and presents a simplification which can be implemented much more efficiently. Section 4 compares the period enforcer with the sporadic server with which it is closely related. Section 5 discusses the implications of the Period Enforcer algorithm to multiprocessor and distributed synchronization protocols, communication and I/O scheduling, and jitter control. Finally, Section 6 presents our concluding remarks.

2. Deferred Execution and Scheduling Penalty

A task is said to *defer* its execution if it suspends during its execution and resumes at a later point in time. A task which can defer its execution is referred to as a *deferrable task*. In a preemptive scheduling environment or due to the stochastic nature of task execution, both the times of suspension and resumption relative to the arrival time of a task instance need not be constant across instances. In this section, we describe the problem caused by deferrable periodic tasks, and compute an upper bound to the scheduling penalty that they can impose on lower priority tasks by such tasks. We shall restrict our attention to deferrable tasks in this section, and discuss issues such as jitter and aperiodic task handling in later sections.

We first state our assumptions. We assume a prioritized, preemptive scheduling environment, and in particular assume that the rate-monotonic scheduling algorithm is being used. We shall assume, unless stated explicitly, that all tasks are periodic and have a fixed priority. Tasks are denoted by $\tau_1, \tau_2, \cdots, \tau_n$ with τ_i having a higher priority than τ_{i+1}. Each periodic task has a period T_i and a worst-case execution time of C_i. Each instance of a periodic task τ must be completed by the time the next instance of τ arrives.

The parameters of an *n*-task set are presented as $\{(T_1, C_1), (T_2, C_2), \cdots, (T_n, C_n)\}$.

The priority of task τ_i is denoted by P_i, with $P_i > P_{i+1}$.

We shall now discuss the deferred execution problem and its schedulability impact. In the ideal case, the worst-case completion time for a periodic task occurs when the task is initiated simultaneously with all higher priority tasks, a phasing called the *critical instant* and the interval from the critical instant until the task completes is called the *critical zone* [3]. However, if a task can defer all or part of its execution during its period, this is no longer true. Consider the following example.

}

Figure 2-1: The Task Set and Time-lines for Example 1.

Example 1:

Consider the 2-task set {(10,4), (18,10)}. In the ideal scenario illustrated in Figure 2-1-a, both tasks meet their deadlines. However, suppose that τ_1 is allowed to defer its execution as shown in Figure 2-1-b. Task τ_1 arrives at time 0, but defers its execution until time 6 when τ_2 arrives. At time 10, the first instance of τ_1 has completed execution but the second instance of τ_1 arrives. Now, this instance as well as the instance arriving at time 20 do not defer their execution but execute immediately. As a result, task τ_2 with its deadline at time (18+6) 24 obtains only 6 units of execution. Thus, τ_2 has been preempted an additional 4 units of time, and is the scheduling penalty imposed by the deferred execution of τ_1.

It has been shown in [1, 10] that when the highest priority task τ_1 can choose to defer its execution, the worst-case phasing for a task τ_i occurs at time t_0 under the following conditions:

- The first instance of task τ_1 defers its execution by $T_1 - C_1$ units and resumes execution at t_0 such that it just meets its deadline at $t_0 + C_1$, and subsequent instances execute immediately on arrival.
- All other higher priority tasks arrive simultaneously at t_0.

We refer to the instant t_0 as the *modified critical instant* of τ_i. Under this modified worst-case phasing, the highest priority task, if it can defer its execution, can impose a penalty of an additional preemption upon lower priority tasks.

Unfortunately, a simple characterization of the worst-case phasing when *all* higher priority tasks can defer their execution does not seem to exist. However, it is possible to determine an upper bound on the penalty imposed by the deferred execution of higher priority tasks on a task τ_j. We do this by determining the worst-case penalty imposed by each higher priority task τ_i on τ_j and adding up the penalties of all the higher priority tasks. This yields a pessimistic result, but shows that the scheduling penalty can be rather high.

We denote the number of time-units that a task τ_i executes during the period of a lower priority

task τ_j (i < j) in its critical phasing by[1] $et_{i,j}$. That is, when τ_j is initiated at time 0 along with all the higher priority tasks, the number of time-units that task τ_i executes in the interval 0 through T_j is denoted by $et_{i,j}$. The value of $et_{i,j}$ can be determined by laying out the critical zone as in [2].

The maximum time that a deferrable task τ_i can execute during a period T_j of a lower priority task τ_j is denoted by[2] $etd_{i,j}$. We now determine an upper bound to $etd_{i,j}$ denoted by[3] $etdu_{i,j}$.

Theorem 1: The upper bound on $etd_{i,j}$, $etdu_{i,j}$, is given by

$$C_i + \lfloor \frac{(T_j - C_i)}{T_i} \rfloor C_i + min(C_i, T_j - C_i - \lfloor \frac{(T_j - C_i)}{T_i} \rfloor T_i)$$

Proof: An upper bound on the execution time of τ_i within a period of τ_j occurs when all tasks with higher priority than τ_i have zero execution times. As a result, when τ_i is eligible to execute, it can preempt any currently executing task. Since τ_i is the highest priority task with a non-zero execution time, the worst-case critical phasing for τ_j arises when it arrives at the modified critical instant. The total number of time-units that τ_i executes during this period of τ_j consists of 3 factors. First, the deferred execution of τ_i to just meet its first deadline is C_i, and the remaining interval in τ_j's period is $T_j - C_i$. Secondly, there are $\lfloor \frac{(T_j - C_i)}{T_i} \rfloor$ complete periods of τ_i in the interval $T_j - C_i$. Correspondingly, there are $\lfloor \frac{(T_j - C_i)}{T_i} \rfloor C_i$ units of τ_i's execution within this interval. Finally, there remains an incomplete period of τ_i in the interval $T_j - C_i$, and this leftover interval is given by $T_j - C_i - \lfloor \frac{(T_j - C_i)}{T_i} \rfloor T_i$. During this leftover interval, τ_i may or may not be able to execute for C_i units of time, and the actual execution time is given by $min(C_i, T_j - C_i - \lfloor \frac{(T_j - C_i)}{T_i} \rfloor T_i)$. The theorem follows by adding these three factors.

The net effect of deferred execution is that a task τ_i can execute longer during a period of τ_j than without deferred execution, that is $etdu_{i,j} \geq etd_{i,j} \geq et_{i,j}$. The difference $etdu_{i,j} - et_{i,j}$ is a scheduling penalty and must be accounted for in the schedulability analysis of τ_j. One way to do this is to add this difference to the execution time of τ_j and checking whether the task can still meet its deadline at its critical instant. This analysis is identical to the blocking factor added to the execution time of a task in the analysis of synchronization protocols [7]. Since schedulability analysis of τ_j already takes into account $et_{i,j}$, the additional execution time of τ_i within τ_j's period is considered a scheduling penalty for τ_j.

An upper bound on the scheduling penalty imposed by a task τ_i on a lower priority task τ_j is

---

[1] *et* stands for "execution time".

[2] *etd* stands for "execution time w/ deferral".

[3] *etdu* stands for "upper-bound to execution time w/ deferral".

denoted by[4] $dep_{i,j}$ and is given by[5] $etdu_{i,j} - et_{i,j}$. The value of $dep_{i,j}$ yields the scheduling penalty of a single higher priority task τ_i on task τ_j. The following theorem specifies the penalty imposed by all higher priority tasks on τ_j.

Theorem 2: An upper bound on the scheduling penalty imposed on task τ_j by all the higher priority tasks is given by $\sum_{i<j} dep(i,j) = \sum_{i<j} (etdu_{i,j} - et_{i,j})$.

Proof: The Theorem follows from the fact that the worst-case penalty imposed by τ_i on τ_j, $dep_{i,j}$, cannot get worse as all tasks execute together.

The following example illustrates the determination of $dep_{i,j}$.

}

Figure 2-2: The Task Set and Computations of Example 2-2.

Example 2:

Consider the 3-task set {(10, 4), (14, 6), (28, 4)} shown in Figure 2-2. The Liu & Layland critical zone for this task set is given in Figure 2-2-a which shows that all 3 tasks meet their deadlines. Now, suppose that each task can also defer its execution. Task τ_1 can always meet its deadline as long as it does not defer its execution longer than the time needed to meet its deadline (i.e. 10−4 time-units). The scheduling penalty of τ_1 on τ_2 is determined as follows. From Figure 2-2-a, $et_{1,2}$=8, and from Figure 2-2-b, $etdu_{1,2}$=8. Hence, the penalty $dep_{1,2}$=8−8=0.[6] From Figure 2-2-a, we have $et_{1,3}$=12 and $et_{2,3}$=12. From Figure 2-2-c, we have $etdu_{1,3}$=16. Finally, from Figure 2-2-d, we have $etdu_{2,3}$=18. Thus, $dep_{1,3}$=16−12=4 and $dep_{2,3}$=18−12=6 and the pessimistic upper bound on the scheduling penalty on τ_3 is (4+6)10. That is, in the worst-case, τ_3 would get no execution time. One phasing where τ_3 does not get any execution time at all is shown in Figure 2-2-e.

It can also be seen that in Figure 2-2-e, the highest priority task τ_1 is the only task that defers execution while τ_2 and τ_3 do not. As a result, the scheduling penalty is the reduction of C_3 from 4 to 0, an actual penalty of 4 which agrees with the modified critical zone result. This scheduling penalty due to deferred execution corresponds to a utilization loss of 4/28=14% but can be much higher in other cases. For example, consider a relatively large task set of more than 8 tasks where each of the higher priority tasks can defer their execution. The least priority task can get

---

[4] *dep* stands for "deferred execution penalty".

[5] This upper bound is pessimistic and can be greater than C_i (but is less than $2C_i$). However, we conjecture that the maximum penalty that can be imposed by a deferrable task τ_i on a lower priority task τ_j has an upper bound of C_i. The point of this section is that the scheduling penalty of deferred execution can be rather high, and therefore must be avoided. A proof of this conjecture does not refute this position.

[6] The {(10,4), (14,6)} task set corresponds to the Liu and Layland worst-case 2-task set. Indeed, the scheduling penalty due to deferred execution on Liu & Layland worst-case task sets is zero. It must be borne in mind that this result has no significance in itself because the worst-case task set changes with the introduction of deferred execution.

some execution time only at low levels of processor utilization. In the worst case, if there are no constraints on the periods, the worst-case schedulable utilization is 50% [1] as opposed to the Liu and Layland bound of 69%.

3. The Period Enforcer

As seen in the previous section, the complexity of the computation of the scheduling impact of deferred execution is relatively low but is pessimistic. However, it is clear that the scheduling impact of deferred execution can cause substantial schedulability degradation. A solution that eliminates deferred execution penalties without much burden on the application and with little overhead is therefore very desirable. We propose below the period enforcer algorithm as such a solution.

With deferred execution, a task can execute its C_i units of execution in discrete amounts $C_{1,1}$, $C_{1,2}$, $\cdots$ with suspension inbetween $C_{1,i}$ and $C_{1,i+1}$. Without any loss of generality, we shall assume that a task τ_i can defer its entire execution time but not parts of it. That is, a task τ_i executes for C_i units with no suspensions once it begins execution. Any task that does suspend after it executes for a while can be considered to be two or more tasks each with its own worst-case execution time. The only difference is that if a task τ_i is split into two tasks τ_i' followed by τ_i'', then τ_i'' has the same deadlines as τ_i'. We shall address this deadline issue in Section 3.2.

We shall also use the following notation and terminology in our discussion.

P_p: The priority level at which the processor is currently executing.

$s_{i,j}$: The time at which the j^{th} instance of task τ_i attempts to execute at the processor. (If this task is resuming after a deferral, $s_{i,j}$ corresponds to the time of resumption.)

The period enforcer algorithm avoids scheduling penalties due to deferred execution by delaying the execution of task instances if a penalty may be caused. In other words, under the period enforcer algorithm, a task instance may not be eligible to execute at $s_{i,j}$.

$ET_{i,j}$: (Eligibility Time) The time at which the j^{th} instance of task τ_i is eligible to be activated on the processor.

$AT_{i,j}$: (Activation Time) The time at which the j^{th} instance of task τ_i is actually activated on the processor (by addition into the ready queue on the processor). Clearly, we must have $AT_{i,j} \geq s_{i,j}$ and $AT_{i,j} \geq ET_{i,j}$.

A priority level P_i is said to be *active* if $P_p \geq P_i$. That is, the task currently executing on the processor has an equal or higher priority than P_i.

Conversely, a priority level P_i is said to be *idle* if $P_p < P_i$. That is, the task currently executing on the processor has a lower priority than P_i.

a_t^i: If priority level P_i is active at time t, the earliest time before t since which P_i has been active continuously, else this is equal to t. That is, $a_t^i \leq t$.

The value of a_t^i is used as follows. Whenever the j^{th} instance of a task τ_i attempts to execute at

$t=s_{i,j}$, $a^i_{s_{i,j}}$ is determined and represents if and how long the processor has been at priority level P_i.

3.1. Definition of The Period Enforcer Algorithm

When the j^{th} instance of a task τ_i ($j = 1, 2, \cdots$) attempts to execute at time $s_{i,j}$, the values $ET_{i,j}$ and $AT_{i,j}$ are computed as follows.

- $ET_{i,0} = -T_i$.
- $ET_{i,j} = \max(ET_{i,j-1} + T_i, a^i_{s_{i,j}})$.
- $AT_{i,j} = \max(ET_{i,j}, s_{i,j})$.

The instance is activated and added to the processor ready queue only at time $AT_{i,j}$.

Note that the value of $AT_{i,j}$ under the period enforcer algorithm implies that $AT_{i,j} \geq s_{i,j}$ and $AT_{i,j} \geq ET_{i,j}$, conditions required by the semantics of $AT_{i,j}$.

The period enforcer algorithm is illustrated by the following example.

}

Figure 3-1: The Task Set of Figure 2-2 under the Period Enforcer.

Example 3:

Consider the 3-task set of example 2 and the same phasing of incoming tasks as in Figure 2-2-e where τ_3 does not get any execution time. Let the tasks be scheduled now under the period enforcer algorithm. Since τ_1 is the highest priority task, priority level P_1 can be active only if τ_1 executes. As a result, when τ_1 tries to execute, the period enforcer always finds that $a^1_t = t$. The sequence of events illustrated in Figure 3-1 is as follows:

- At time t=0, τ_1 arrives but defers its execution by 6 units.
- At time t=6, τ_1 tries to resume execution. Hence, $s_{1,1}$=6, and we have $ET_{1,1}$=6, and $AT_{1,1}$=max(6,6)=6. Hence, τ_1 is activated immediately, and begins execution. Task τ_2 also arrives and finds that $s_{2,1}$=6, $ET_{2,1}$=6, and $AT_{2,1}$=max(6,6)=6. Hence, τ_2 is activated immediately but has to wait for τ_1 to complete.
- At time t=10, the first instance of τ_1 completes execution, and the next instance of τ_1 arrives and again defers execution for 6 units. Task τ_2 begins execution.
- At time t=16, task τ_2 completes execution and the second instance of τ_1 tries to resume execution. We now have $s_{1,2}$= 16, $ET_{1,2}$=max(16, 16)=16, and $AT_{1,2}$=max(16,16)=16. Hence, τ_1 begins execution immediately. Also, task τ_3 arrives and finds that $ET_{3,1} = a^3_{16} = 6$, and $AT_{3,1} = \max(6,16) = 16$. It is therefore activated but has to wait for τ_1 to complete.
- At time t=20, the second instance of τ_1 completes. Its next instance also arrives

and tries to execute immediately. However, we now have $s_{1,3}$= 20, $ET_{1,3}$=max(16+10, 20)=26, and $AT_{1,3}$=max(26,20)=26. As a result, τ_1 is scheduled by the enforcer to become eligible for execution at t=26. The next instance of task τ_2 arrives and finds that a_{20}^2=6 since the processor has been at priority level 2 from t=6. The period enforcer determines that $s_{2,2}$=20, $ET_{2,2}$=max(6+14,6)=20, $AT_{2,2}$=max(20,20)=20. Hence, τ_2 is activated immediately and begins execution since τ_1 is not eligible to execute and τ_3 has lower priority.

- At time t=26, τ_2 completes execution, and the third instance of τ_1 is activated and begins execution.
- At time t=30, the third instance of τ_1 completes execution. Its next instance arrives and tries to execute immediately. Again, we have $s_{1,4}$=30, $ET_{1,4}$=max(26+10, 30)=36 , $AT_{1,4}$=max(36,30)=36. Hence, τ_1 is scheduled to become eligible for execution at t=36. Since task τ_3 is the only task eligible to execute, it begins execution.
- At time t=34, τ_3 completes execution thereby meeting its deadline which is at t=16+28=44. The next instance of τ_2 arrives and finds a_{34}^2=34. Also, $s_{2,3}$= 34, $ET_{2,3}$=max(34,34)=34, and $AT_{2,3}$=max(34,34)=34. As a result, τ_2 is eligible for execution and begins execution.

The sequence proceeds with instances of τ_1 *always* executing at least 6 time-units since their respective arrival.

The above example illustrates some key features of the period enforcer algorithm.

- Deferred execution imposes a scheduling penalty because it is possible for one instance of a deferrable task to defer its execution by some amount of time and for the next instance to defer execution by a shorter amount of time. The period enforcer algorithm disallows this condition and delays the second intruding instance if necessary to avoid the scheduling penalty. Such an enforcement on instances of τ_1 occurs at t_{20} and t_{30}.
- Tasks which do not defer any part of their execution may or may not need to use the period enforcer algorithm and the choice can be made to suit convenience of implementation. As we shall prove shortly, the execution behavior of periodic tasks is identical whether they use the period enforcer algorithm or not. For example, in the sequence of events in Figure 3-1, τ_2 and τ_3 would perform exactly the same if they did not use the period enforcer algorithm.

3.2. Properties of the Period Enforcer Algorithm

The execution profile of a periodic task does not change when it uses the period enforcer algorithm. In addition, deferrable tasks do not impose any scheduling penalty on lower priority tasks when they use the period enforcer algorithm. We prove these properties of the period enforcer algorithm below.

Theorem 3: The execution behavior of a non-deferrable periodic task τ_i using the period enforcer algorithm is identical to that when it is scheduled normally.

Proof: Task τ_i executes at the same priority level under both cases, and has the same period and execution time. In order to prove the Theorem, we just need to show that any instance of τ_i is eligible to execute on arrival under the period enforcer algorithm. In other words, we need to show that $\forall\, j$, $j = 1, 2, \cdots$, $AT_{i,j} = s_{i,j}$. By the period enforcer algorithm, $AT_{i,j} = \max(ET_{i,j}, s_{i,j})$. Hence, to show that $\forall\, j$, $j = 1, 2, \cdots$, $AT_{i,j} = s_{i,j}$, we only need to show that $\forall\, j$, $j = 1, 2, \cdots$, $ET_{i,j} \le s_{i,j}$. We prove this by mathematical induction on j.

When j=1, $ET_{i,1} = a^i_{s_{i,1}}$. Since $a^i_{s_{i,1}} \le s_{i,1}$, we have $ET_{i,1} \le s_{i,1}$.

Now, suppose that $ET_{i,j} \le s_{i,j}$. That is, $\max(ET_{i,j}) = s_{i,j}$. We want to show that $ET_{i,j+1} \le s_{i,j+1}$. By the period enforcer algorithm, we have

$$ET_{i,j+1} = max(ET_{i,j} + T_i, a^i_{s_{i,j+1}})$$

Therefore,

$$max(ET_{i,j+1}) = max(max(ET_{i,j}) + T_i, max(a^i_{s_{i,j+1}}))$$
$$max(ET_{i,j+1}) = max(s_{i,j} + T_i, s_{i,j+1})))$$

Since $s_{i,j} + T_i = s_{i,j+1}$,

$$max(ET_{i,j+1}) = s_{i,j+1}$$

The Theorem follows.

Theorem 4: A schedulable task τ_i that uses the period enforcer algorithm does not impose any scheduling penalty on lower priority tasks when it becomes a deferrable task.

Proof: We prove this by showing that in the worst case, a deferrable task acts like a non-deferrable periodic task with the same period and execution time but with a different phasing. Hence, lower priority tasks do not incur any scheduling penalty.

Let the first instance of the deferrable task τ_i be ready for execution at time t_0 after deferral if applicable. Let $ET_{i,1}=a^i_{t_0}=t_a$ which is less than or equal to $s_{i,1} = t_0$. By the period enforcer algorithm, the instance is immediately activated at t_0. We also have $ET_{i,j} = \max(ET_{i,j-1} + T_i, a^i_{s_{i,j}})$, i.e. $ET_{i,j} \ge ET_{i,j-1}+T_i$.

Therefore, the second instance of τ_i cannot become eligible to execute before $ET_{i,1} + T_i = t_a + T_i$. Again, if the second instance of τ_i does become eligible to execute at $ET_{i,1} + T_i = t_a + T_i$, the third instance cannot become eligible to execute before $ET_{i,1} + 2T_i = t_a + 2T_i$. That is, in the interval between t_a and $t_a + 2T_i$, τ_i behaves identical to a periodic task with period T_i and C_i that arrives at t_a. As a result, the first two instances impose no additional scheduling penalty on lower priority tasks. The argument is repeatable for all subsequent instances of τ_i and the Theorem follows.

We shall now address the issue of the ability of the deferrable tasks to meet their deadlines. Consider an instance of a deferrable task τ_i under its worst-case conditions. Let its worst-case execution deferral relative to its arrival be t_d. That is, if an instance arrives at t_0 it defers its

execution to resume at $t_r = t_0+t_d$ in the worst case. The deadline for the instance, nevertheless, is t_0+T_i and needs to be met. The instance meets its worst-case conditions when t_r is its critical instant, i.e. instances of all higher priority tasks are also initiated at the same time.

A deferrable task may be prevented from imposing any scheduling penalty on lower priority tasks by delaying its execution in a very pessimistic fashion (such as a delay of more than a period). However, this would jeopardize the deadlines of the deferrable task. The period enforcer algorithm eliminates the scheduling penalty on lower priority tasks without endangering the deadlines of the deferrable task. We now prove that the period enforcer algorithm does not affect the deadlines of deferrable tasks.

Theorem 5: A deferrable task that is schedulable under its worst-case conditions is also schedulable under the period enforcer algorithm.

Proof: Let an instance of τ_i which arrives at t_a defer its execution to resume at $t_r = t_a+t_d$ in the worst case. Given that this instance is schedulable without the period enforcer algorithm. Under the algorithm, higher priority tasks, deferrable or not, cannot impose any scheduling penalty on τ_i. Hence, to show that τ_i is also schedulable under the algorithm, we just need to show that any instance of τ_i is activated within t_d time units of its arrival. If the first instance of τ_i arrives at time t_0, since t_d is the longest deferral, $\forall\, j$, $j = 1, 2, \cdots, s_{i,j} \le t_0 + (j-1)T_i + t_d$. We need to show that $\forall\, j$, $j = 1, 2, \cdots, AT_{i,j} \le t_0 + (j-1)T_i + t_d$. We prove this again using mathematical induction on j.

When j=1, $AT_{i,1} = \max(a^i_{s_{i,1}}, s_{i,1}) = s_{i,1}$. Hence, the condition is met.

Suppose that $AT_{i,j} \le t_0 + (j-1)T_i + t_d$. We want to show that $AT_{i,j+1} \le t_0 + jT_i + t_d$. By the period enforcer algorithm,

$$ET_{i,j} \le AT_{i,j}$$
$$s_{i,j} \le AT_{i,j}$$
$$ET_{i,j+1} = max(ET_{i,j} + T_i, a^i_{s_{i,j+1}})$$

Therefore,

$$max(ET_{i,j+1}) = max((max(ET_{i,j}) + T_i, max(a^i_{s_{i,j+1}}))$$
$$max(ET_{i,j+1}) = max(AT_{i,j} + T_i, s_{i,j+1})$$

We therefore have,

$$max(AT_{i,j+1}) = max(AT_{i,j} + T_i, s_{i,j+1}, a^i_{s_{i,j+1}}) = max(AT_{i,j} + T_i, s_{i,j+1})$$

Hence, by the assumed bound for $AT_{i,j}$ and the given value of $s_{i,j+1}$,

$$AT_{i,j+1} \le t_0 + jT_i + t_d$$

The Theorem follows.

3.3. Simplification of The Period Enforcer

The period enforcer algorithm defines the earliest time that a deferrable task can be activated without imposing a scheduling penalty on lower priority tasks. However, it requires the determination of a^i_t when a task attempts to execute at time t. The determination of a^i_t means that the times at which *every* priority level becomes active or idle must be monitored by the system con-

tinually. Since the current priority level of the processor P_p changes whenever a task completes execution or is preempted, the priority level data structures must be updated on every context switch. This update may be a costly operation particularly if the system supports a large number of priorities. Ways of speeding this update are possible. However, not all systems may prefer to implement such schemes. A simpler algorithm which still has the advantages of the period enforcer algorithm may be more desirable in these systems.

A *vanilla period enforcer* algorithm can be defined as follows. If a_t^i is always defined to be t, priority levels no longer need to be tracked and the period enforcer algorithm becomes simply

$$AT_{i,0} = -T_i.$$

$$AT_{i,j} = ET_{i,j} = \max(AT_{i,j-1} + T_i, s_{i,j}).$$

The vanilla period enforcer algorithm is intuitive once the problems of deferred execution are understood. If an instance of task τ_i defers execution until time t_r, the scheduling penalty of deferred execution is prevented if the next instance of the task is not allowed to execute until $t_r + T_i$.[7] The $s_{i,j}$ term is a sanity factor which ensures that an instance is activated only when it has actually arrived.

Under this vanilla algorithm, a task that defers execution by a certain amount of time during any instance would be forced to resume subsequent execution at least that far from its arrival time for all future arrivals. This property of the algorithm will be referred to as the *minimum separation property* and ensures that lower priority tasks do not encounter any additional penalty due to deferred execution. This also means that if the longest deferral for a task is t_d units from its arrival (as in the proof of Theorem 5), then *all* its subsequent instances would also be delayed from their arrival times from exactly t_d units[8]. Hence, if the task is schedulable during its longest deferral, it still remains schedulable under the vanilla algorithm. Thus, both the desirable properties of the period enforcer algorithm are still maintained by this simpler version at possibly a much smaller cost.

Also note that the minimum separation property of the vanilla version can be used to enforce the requirement of a sporadic task [4] that there be a minimum interarrival time between successive instance of the task.

4. Period Enforcer *vs* Sporadic Server

The sporadic server [8] is used to provide excellent response times to aperiodic tasks. In this section, we investigate what the period enforcer algorithm means to the sporadic server and vice-versa. The period enforcer algorithm determines when the next instance of a deferrable task may execute based upon the current instance. The sporadic server determines when the server capacity must be replenished and by how much.

---

[7] It may be possible to execute earlier and the Period Enforcer is more sophisticated because it takes care of this possibility.

[8] This behavior is true for only the highest priority task in the period enforcer algorithm

4.1. Implementing the Sporadic Server with the Period Enforcer

We shall now show how the sporadic server can be defined in terms of the period enforcer.

A straightforward implementation of the Sporadic Server using the Period Enforcer is as follows.

- The Sporadic Server maintains a pool of available service units.
- Each service unit comprising the Sporadic Server capacity uses its own Period Enforcer.
- When an aperiodic task requests one unit of service from the Sporadic Server at time t, the server pool is checked. If a service unit is available, $s_{i,j}$ of this unit is t. Else, the task waits until a service unit is available[9] at t_r, and $s_{i,j}=t_r$. The values $ET_{i,j}$ and $AT_{i,j}$ for this service unit are then computed for this unit. The aperiodic task begins execution at the sporadic server priority at $AT_{i,j}$.
- The service unit is replenished in the server pool at $ET_{i,j}+T_{SS}$, where T_{SS} is the period of the Sporadic Server.

When two or more service units are required by the aperiodic task and are available in the server pool, the period enforcer parameters of these service units can be updated simultaneously.

When a unit is replenished and reused, $ET_{i,j+1}$ is recomputed as $\max(ET_{i,j}+T_{SS}, a^i_{s_{i,j}})$. However, since a unit becomes replenished only at $ET_{i,j}+T_{SS}$, it is guaranteed that when it is to be used for service again by an aperiodic task at $s_{i,j}=t_r$, $t_r \geq ET_{i,j}-T_{SS}$. With this observation, an efficient approximation is also possible if, similar to the vanilla period enforcer, we assume that $a^i_{s_{i,j}}) = s_{i,j}$.

Hence, the period enforcer expressions become

$$AT_{i,j} = ET_{i,j} = s_{i,j}$$

As can be seen, $ET_{i,j}$ and $AT_{i,j}$ are independent of the values for the $(j\text{-}1)^{th}$ instance and the suffixes i,j can be ignored. Hence the algorithm can be redefined as follows.

- Let an aperiodic task request service of C units from the Sporadic Server at time t. If the server has this capacity, $s = t$. Else, the task waits until the server capacity is replenished at t_r, and $s=t_r$.
- We have $AT = s$ and the aperiodic task is activated.
- The capacity assigned to this aperiodic task (or the capacity actually consumed, if smaller) is replenished in the server at $ET + T_{SS}$, where T_{SS} is the period of the Sporadic Server.

The original sporadic server defined in [8] needs to compute replenishment times whenever the priority level of the server becomes active or idle. In the definition above, these computations occur only when an aperiodic task arrives and finds sufficient server capacity to execute. Hence, an implementation based on this definition would incur less overhead.

---

[9]The task may instead choose to execute in background mode just as in the original Sporadic Server.

4.2. Implementing the Period Enforcer with the Sporadic Server

It is also possible to implement the period enforcer in terms of the sporadic server. The period enforcer corresponds to a special case of the sporadic server in that the period enforcer for a deferrable task is equivalent to the deferrable task being serviced by a dedicated sporadic server task with the same period as the deferrable task and a capacity equal to the execution time of the deferrable task.

The differences between this approach and using the period enforcer directly is as follows. With the period enforcer, an additional separate sporadic server need not be created, and the execution time consumed by the deferrable task need not be monitored. In fact, the period enforcer allows the algorithm parameters $ET_{i,j}$, $ET_{i,j-1}$ and $AT_{i,j}$ to be defined as part of the task control block in operating systems for greatest implementation efficiency.

4.3. The Sporadic Server-Period Enforcer Relationship

As we have shown above, the period enforcer is in a sense virtually identical to the sporadic server. The sporadic server is itself a deferrable task, and the algorithm determines how much server capacity must be replenished at what times without affecting the schedulability of lower priority tasks. Similarly, the period enforcer algorithm determines at what times a deferrable task becomes eligible to execute without affecting the schedulability of lower priority tasks. Since execution times are no longer monitored in the period enforcer, it is actually a simplification of the sporadic server. However, this simplification permits the distillation of the key concept behind the sporadic server into a basic mechanism which can then be used as a primitive to solve all problems related to deferred execution in an efficient fashion. The end result is that the objectives of the period enforcer form a superset of the sporadic server's goals.

Due to the difference in objectives between the sporadic server and the period enforcer, the deadline satisfaction properties of the sporadic server have not been studied even though Theorem 5 applies to the sporadic server as well. In addition, this difference also means that not all simplifications possible on the sporadic server can always be made to the period enforcer. For example, the sporadic server capacity can be replenished at *any* time *after* $ET + T_{SS}$ as defined in Section 4.1. For instance, instead of replenishing used capacity at the earliest possible time as defined by $ET + T_{SS}$, the replenishment time may be set only after some pre-defined portion of its total capacity (say 0.25 or 0.5) is used up at t_u. The replenishment time can also be computed to be t_u+T_s, where T_s is the period of the sporadic server. In contrast, under the period enforcer, the replenishment time will be *at most* $t_u-C_u+T_i$ where C_u is the capacity used up. A later replenishment time for the sporadic server means only that the response time to aperiodics would degrade but not considerably [9]. However, deferrable periodic tasks suspending for reasons like I/O and data access can use a simpler algorithm only at the risk of missing their deadlines. The vanilla algorithm defined in section 3.3 is the simplest period enforcer policy which can still guarantee that the deadlines of deferrable periodic tasks will be met.

5. Implications of the Period Enforcer

In this section, we consider what the period enforcer brings to the domain of synchronization protocols in multiple processor systems, I/O scheduling, communication media scheduling and jitter problems.

5.1. Implications to Multiple Processor Systems

In multiple processor real-time systems such as shared memory systems and distributed systems, the need to share resources across processors or communicate between processors is unavoidable[10].

In the common presence of resource sharing in such systems, the need for synchronization to access global resources leads to schedulability problems. The primary problem is the need to avoid an unbounded duration of waiting to access a global resource. Synchronization protocols which avoid such unbounded waiting durations have been defined for use in shared memory systems [6] that use test-and-set type of primitives on shared memory, and in distributed systems that use remote procedure calls [5]. In these cases, the deferrable task problem is unavoidable in that a task may be forced to suspend waiting to access a global ressource and will resume at a later point in time. The scheduling penalty of deferred execution in these environments can be enormous to the extent that it can make such protocols almost useless in several cases. The development of the period enforcer algorithm was primarily motivated by the need to reduce or eliminate this penalty with a small, if not negligible, overhead. If tasks using these synchronization protocols use the period enforcer, their worst-case blocking durations would be considerably smaller and the schedulable utilization correspondingly higher.

5.2. Communication Scheduiling

Communication between processors also pose a deferred execution problem. The interprocessor communication has to be initiated on a processor, transmitted on a shared communication medium (such as a bus or a token ring), and finally delivered to and processed by another processor. There can be jitter caused by each of these 3 phases. Due to preemption and/or stochastic execution, the instants at which the communication is initiated will not be strictly at periodic intervals. Additionally due to non-instantaneous preemption on the communication medium, the instants at which delivery is made to the destination will also not be strictly periodic. Finally, the instants at which the receiving process processes the delivered communication will not be exactly periodic. The period enforcer can be used to eliminate the resulting scheduling penalties on the communication medium and the destination processor. This is done by using two period enforcers. One period enforcer on the source processor regulates the instants at which communication is initiated and could be done by the (hardware or software) interface module to the communication medium. Another software period enforcer at the destination processor controls the instants at which the receiving process processes the message.

5.3. I/O Device Scheduling

I/O scheduling is a special case of global synchronization where an I/O device must typically be used exclusively by a single task until the service completes. Deferred execution penalty can arise in the analysis of the I/O device schedulability due to "jittery" service requests as well as in the processor scheduling model where tasks resume at irregular intervals after I/O completion. Again, one period enforcer is required to regulate the requests to the I/O device and another to control the resumption of tasks completing I/O.

---

[10]If this were not true, one would have several independent uniprocessors rather than a single cohesive system.

In each of the above cases, by Theorem 5, a task or subtask will always be able to meet its deadlines as long as it is known to meet its deadline when it defers its execution by the longest duration of time.

5.4. Output Jitter Control

We have so far dealt with jittery signals whose output forms an input to a different resource where it can be controlled to avoid scheduling penalties. However, there are situations where the output signals must be strictly periodic and just controlling the jitter sequence to avoid scheduling penalty is not sufficient. For example, consider an output signal that is generated by a task that is used to trigger a sampling/tracking device. The handling of the device results may be much easier if the sampling triggers occur exactly at periodic intervals. One inexact but approximate way of doing this using the period enforcer is as follows. Let the task that computes the values of the output signal be τ_i. Create a highest priority task τ_H that merely transmits the output signal values, and let it use a vanilla period enforcer with the desired signal period. The highest priority of τ_H ensures that whenever it is activated, it will execute immediately and generate the signal. The first instance of τ_i arriving at t_0 must trigger the *first* instance of τ_H at t_0+t_D, where $t_D \geq$ the longest time that any instance of τ_i will take to complete its computations relative to its arrival. The minimum separation property of the vanilla period enforcer combined with the fact that the deferrals cannot be longer ensure that all instances of τ_H will always be activated at t_D units of τ_i's arrival.

This approach is inexact because if there are two or more highest priority tasks generating their respective jitter-free output signals, it is possible that two or more of these signals need to be transmitted at the same time. As a result, one or more would be forced to be delayed by a small amount of time. Processing time of other high priority interrupts can also cause additional minor jitter. A perfect solution may need hardware support (such as one which implements a vanilla period enforcer in hardware dedicated to each output signal).

6. Concluding Remarks

Periodic tasks in preemptive real-time systems may suspend during execution to perform I/O activity or to access shared resources with synchronization constraints, and then resume later. These practical requirements cause part or all of a task's execution time to be deferred, causing deviation from the ideal behavior of periodic tasks. Unfortunately, these deferrals can impose a scheduling penalty on lower priority tasks which can quickly become unacceptable. Non-ideal behavior of periodic tasks can also arise in communication media and in I/O scheduling activities. In this paper, we have proposed the Period Enforcer algorithm which eliminates the scheduling penalty caused by such behavior. The close relationship between this algorithm and the sporadic server is also investigated in depth.

The pervasive nature of the deferred execution problem and its potential serious effects on schedulability argue for the inclusion of the period enforcer algorithm or the simpler vanilla version in all preemptive real-time systems. One major reason for resistance to more popular acceptance of the sporadic server is its perceived implementation complexity. However, the ability of the period enforcer to simulate the sporadic server while addressing the impact of the often unavoidable causes for deferred execution can make it a desirable solution. The definition of the vanilla period enforcer, if not the period enforcer itself, is very simple and can be implemented

with relative ease.

There remain some outstanding issues with respect to the period enforcer. The required interaction between the period enforcer algorithm and jitter-free output signals needs to be better understood. Also, while the period enforcer can be applied to dynamic priority scheduling algorithms such as the earliest deadline scheduling algorithm, the corresponding schedulability impact has not been studied.

Acknowledgements

The author would like to thank Dr. Lui Sha and Prof. John Lehoczky for their comments during the early stages of this work.

References

[1] Lehoczky, J. P., Sha, L. and Strosnider, J.
Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

[2] Lehoczky, J. P., Sha, L. and Ding, Y.
The Rate Monotonic Scheduling Algorithm --- Exact Characterization and Average-Case Behavior.
IEEE Real-Time Systems Symposium , Dec, 1989.

[3] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[4] Mok, A. K.
Fundamental Design Problems of Distributed Systems For The Hard Real Time Environment.
PhD thesis, M.I.T., 1983.

[5] Rajkumar, R., Sha, L., and Lehoczky J.P.
Real-Time Synchronization Protocols for Multiprocessors.
Proceedings of the IEEE Real-Time Systems Symposium :259-269, 1988.

[6] Rajkumar, R.
Real-Time Synchronization Protocols for Shared Memory Multiprocessors.
The Tenth International Conference on Distributed Computing Systems , 1990.

[7] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
IEEE Transactions on Computers :1175-1185, September, 1990.

[8] Sprunt, H.M.B., Sha, L., and Lehoczky, J.P.
Aperiodic Task Scheduling on Hard Real-Time Systems.
The Real-Time Systems Journal , June, 1989.

[9] Sprunt, H. M. B.
Aperiodic Task Scheduling for Real-Time Systems.
PhD thesis, Carnegie Mellon University, August, 1990.

[10] Strosnider, J.K.
Highly Responsive Real-Time Token Rings.
PhD thesis, Carnegie Mellon University, August, 1988.

Table of Contents

List of Figures