An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems

John P. Lehoczky† and Sandra Ramos-Thuel‡

Department of Statistics†
Department of Electrical and Computer Engineering‡
Carnegie Mellon University,
Pittsburgh, PA 15213

Abstract

This paper presents a new algorithm for servicing soft deadline aperiodic tasks in a real-time system in which hard deadline periodic tasks are scheduled using a fixed priority algorithm. The new algorithm is proved to be optimal in the sense that it provides the shortest aperiodic response time among all possible aperiodic service methods. Simulation studies show that it offers substantial performance improvements over current approaches including the sporadic server algorithm. Moreover, standard queueing formulas can be used to predict aperiodic response times over a wide range of conditions. The algorithm can be extended to schedule hard deadline aperiodics and to efficiently reclaim unused periodic service time when periodic tasks have stochastic execution times.[1,2]

1 Introduction

In 1973, Liu and Layland [1] presented an analysis of the rate monotonic algorithm for scheduling periodic tasks with hard deadlines. Recently, this algorithm has gained popularity as an approach to designing predictable real-time systems. Moreover, the algorithm has been modified to allow for the solution of many practical problems which arise in actual real-time systems including task synchronization, transient overload, and simultaneous scheduling of both periodic and aperiodic tasks, among others. The mixed task scheduling problem is important, because many real-time systems have substantial aperiodic task workloads. Moreover, the aperiodic tasks may themselves have a variety of timing requirements, ranging from hard deadlines to soft deadlines. For example, recovery from transient failures may create an aperiodic stream of hard deadline periodic tasks which must be reexecuted (see Ramos-Thuel [2]).

In this paper, we reconsider the problem of jointly scheduling hard deadline periodic tasks and aperiodic tasks. Although the methods presented in this paper apply both to hard deadline and soft deadline aperiodic tasks, we limit our attention to the case of scheduling soft deadline aperiodic tasks. That is, we seek to schedule a mixture of periodic and aperiodic tasks in such a way that all periodic task deadlines are met and the response times for the aperiodic tasks are as small as possible.

There are two standard approaches to this problem. The least effective approach is to service the aperiodic tasks in the background of the periodic tasks (i.e., when the processor is idle). A better approach is to create a periodic polling task with as large a capacity as possible. The polling task will be run periodically, and its capacity will be used to service aperiodic tasks. While the polling server is far superior to background, the periodic polling task is not necessarily coordinated with the aperiodic arrival process, so some aperiodic arrivals must wait for the return of the polling task before they can be executed. This waiting may create unnecessarily long task response times. In addition,

[1]The authors wish to thank Stephen W. Gulyas for his participation in this research effort and Jay K. Strosnider and his research group for their insightful comments and suggestions.

[2]This research is supported in part by a grant from the Office of Naval Research under contracts N00014-84-K-0734 and N00014-91-J-1304N173, by the Naval Ocean Systems Center under contract N66001-87-C-01155, by the Federal Systems Division of IBM Corporation under University Agreement Y-278067, and by AT&T Bell Laboratories under the Cooperative Research Fellowship Program.

the polling task may be ready but have no tasks ready for execution, a situation that wastes the high priority capacity of the polling task.

Recently, new approaches to the joint scheduling problem have been developed including the sporadic server algorithm by Sprunt [3, 4] and the deferrable server algorithm by Strosnider [5]. Although similar in spirit to the polling server, these algorithms allow their capacity to be used throughout the server's period rather than only at the beginning. The two algorithms differ in the way their capacity is replenished, and each has its own individual schedulability analysis; however, in certain circumstances, both can offer up to an order of magnitude improvement in aperiodic response time over the polling approach.

This paper develops a new approach to aperiodic service, and shows that this method can offer substantial improvements over the deferrable and sporadic server algorithms. The new approach, called the ***slack stealing*** algorithm, does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the *slack stealer*, which when prompted for service attempts to make time for servicing aperiodic tasks by "stealing" all the processing time it can from the periodic tasks without causing their deadlines to be missed. This is equivalent to "stealing slack" from the periodic tasks. Note the similarity of this approach to cycle stealing techniques used in memory systems [6].

The slack stealer relies on the exact schedulability conditions given by Lehoczky, Sha and Ding [7] and Lehoczky [8] to provide the maximum possible capacity for aperiodic service at the time it is needed. Substantial improvements in aperiodic task response times will be demonstrated with the slack stealer. It will also be shown to be optimal for the particular fixed priority assignment chosen for the periodic tasks. In addition, the slack stealer can be generalized to handle hard deadline aperiodic tasks, and its functionality can be efficiently augmented by a *reclaimer*. A reclaimer can cooperate with a slack stealer by making available for aperiodic service any processing time unused by the periodic tasks when they require less than their worst-case execution times. The slack stealing algorithm requires a relatively large amount of calculation. Consequently, a direct implementation may not be practical. It does, however, provide a lower bound on aperiodic response which is attainable and a basis for finding nearly optimal implementable algorithms.

2 Framework and Assumptions

Consider a real-time system with n periodic tasks, $\tau_1, \ldots, \tau_n$. Each task, τ_i, has a worst-case computation requirement C_i, a period T_i, an initiation time $\phi_i \geq 0$ or offset relative to some time origin, and a deadline D_i, assumed to satisfy $D_i \leq T_i$. The parameters C_i, T_i, ϕ_i, and D_i are known deterministic quantities. We require that these tasks be scheduled according to a fixed priority algorithm, such as the deadline monotonic algorithm, in which tasks with small values of D_i are given relatively high priority [9]. We assume that the periodic tasks are indexed in priority order with τ_1 having highest priority and τ_n having lowest priority. For simplicity, we refer to those levels as $1, \ldots, n$ with 1 indicating highest priority and n the lowest. The aperiodic tasks can be assigned any priority, and we even permit them to be executed dynamically at different priority levels. We assume that if an aperiodic task executes at priority level k, then it has lower priority than any periodic task with priority $1, \ldots, k-1$ and higher priority than any periodic task with priority $k, k+1, \ldots, n$. Aperiodic task execution at priority level $n+1$ is equivalent to background execution.

A periodic task, say τ_i, gives rise to an infinite sequence of jobs. The k^{th} such job is ready at time $\phi_i + (k-1)T_i$ and its C_i units of required execution must be completed by time $\phi_i + (k-1)T_i + D_i$ or else a periodic task timing fault will occur.

We next introduce the aperiodic tasks, $\{J_k, k \geq 1\}$. Each aperiodic job, J_k, has an associated arrival time α_k and a processing requirement p_k. The tasks are indexed such that $0 \leq \alpha_k \leq \alpha_{k+1}, k \geq 1$. It is useful to define the cumulative aperiodic workload process,

$$W_A(t) = \sum_{\{k \mid \alpha_k \leq t\}} p_k, \qquad (1)$$

which accumulates all the aperiodic work that arrives in the interval $[0, t]$. Any algorithm for scheduling both periodic and aperiodic loads will, for any periodic task set and aperiodic task stream $\{J_k, k \geq 1\}$, create a cumulative aperiodic execution process, $\varepsilon(t)$, giving the cumulative time during $[0, t]$ that aperiodic tasks were executed. $\varepsilon(t)$ is a continuous function which must necessarily satisfy $\varepsilon(t) \leq W_A(t), t \geq 0$, and we require that the associated algorithm must meet all periodic deadlines.

We assume aperiodic tasks are processed in FIFO order[3]. The completion time of J_k, denoted T_k, is

[3]In section 4 we consider the shortest remaining processing time queue discipline which will result in lower ***average*** aperiodic task response times.

given by

$$T_k = min\{t \mid \varepsilon(t) = \sum_{i=1}^{k} p_i \}, \qquad (2)$$

and the response time of J_k, denoted R_k, is given by

$$R_k = T_k - \alpha_k. \qquad (3)$$

We seek a scheduling algorithm that will minimize R_k which is equivalent to minimizing T_k. Thus, we need to find a scheduling algorithm whose associated $\varepsilon(t)$ is the supremum or upper envelope of all possible aperiodic execution functions that are associated with algorithms which meet all periodic deadlines. It is important to note that if such an upper envelope can be found, it will lead to the minimum response time *for every aperiodic task*, not just the average response time. We will determine the upper envelope and the associated optimal aperiodic scheduling algorithm in the next section, under the following assumptions:

- *A1:* All overhead for context swapping, task scheduling, etc., is assumed to be zero.
- *A2:* Tasks are ready at the start of their period and do not suspend themselves or synchronize with any other task.
- *A3:* Any task can be instantly preempted.
- *A4:* There is unlimited buffer space for the aperiodic tasks.

3 The Slack Stealing Algorithm

3.1 Formulation

To determine the upper envelope on aperiodic processing, we focus on the *maximum* amount of processing possible such that all periodic deadlines are met. Consider, for example, the j^{th} job of τ_i, or τ_{ij}, which is ready at time $R_{ij} = \phi_i + (j-1)T_i$ and must be finished by $R_{ij} + D_i = D_{ij}$. During $[0, D_{ij}]$ the processor may execute tasks at a priority level equal to or greater than i, tasks at a priority level below i, or may be idle. Under our fixed-priority system any tasks executed at priority level lower than i are equivalent to being idle or inactive relative to level i, thus we refer to ***level-i inactivity*** as processor time spent on activities with priority lower than i. Since level-i inactivity cannot influence the schedulability of any τ_i job, we seek to find the amount of aperiodic work that can be executed at priority level i or higher during [0,t] and still have τ_{ij} finish by D_{ij}.

Since we seek the largest amount of aperiodic processing possible, and are only concerned with τ_{ij}'s deadline, the processor will be busy with level i or higher priority work from 0 until the completion time of τ_{ij}. We now follow the methods developed by Lehoczky, Sha, and Ding [7] and Lehoczky [8] to determine the necessary and sufficient conditions for τ_{ij} to be schedulable.

Suppose $a_i(t)$ is the aperiodic processing at level i or higher during $[0,t]$, $0 \le t \le D_{ij}$, resulting from some algorithm. The job τ_{ij} will finish by D_{ij}, thus meeting its deadline, if and only if there is a time $t \in [R_{ij}, D_{ij}]$ at which all $a_i(t)$ units of aperiodic processing and all periodic jobs of priority i or higher ready before t, including the j jobs of τ_i are completed. Let $P_i(t)$ be the periodic ready work in $[0,t]$, where $P_i(t) = \sum_{j=1}^{i-1} C_j \cdot \lceil max(0, t-\phi_j)/T_j \rceil + jC_i$. The total ready work in $[0,t]$ is then defined by

$$W_i(t) = a_i(t) + P_i(t) + I_i(t), \qquad (4)$$

where $I_i(t)$ is the cumulative level-i inactivity in $[0,t]$. Thus τ_{ij} will meet its deadline if and only if there exists $t \in [R_{ij}, D_{ij}]$ such that $W_i(t) = t$ or equivalently, $W_i(t)/t = 1$. This condition for the feasibility of $a_i(t)$ can be alternatively expressed as

$$min_{\{R_{ij} \le t \le D_{ij}\}} \{W_i(t)/t\} \le 1. \qquad (5)$$

If we assume that the aperiodic workload is sufficiently large so that $W_A(t) > \varepsilon(t)$ for any feasible $\varepsilon(t)$, then we can increase $a_i(t)$ by $I_i(t)$ and the processor will be continually busy with level i or higher priority work up to the completion time of τ_{ij}. Equation (5) can now be rewritten as

$$min_{\{0 \le t \le D_{ij}\}} \{W_i(t)/t\} \le 1. \qquad (6)$$

Given that we want to increase the aperiodic processing time as much as possible, we define A_{ij} to be the largest amount of aperiodic processing possible at level i or higher during $[0, C_{ij}]$, such that $C_{ij} \le D_{ij}$ (C_{ij} refers to the completion time of τ_{ij}). Thus A_{ij} is the largest value such that

$$min_{\{0 \le t \le D_{ij}\}}\{(A_{ij} + P_i(t))/t \} = 1. \qquad (7)$$

A_{ij} is well defined because the periodic task set is assumed to be schedulable and the function being minimized is piecewise continuous and decreasing. A_{ij} is increased until a minimum of 1 is exactly achieved. The completion time of τ_{ij}, or C_{ij}, is the smallest value of t for which equality holds in Equation (7).

Aperiodic processing at level i or higher given by A_{ij} during $[0, t]$ will cause the processor to be constantly busy, but τ_{ij} will meet its deadline. We now need to guarantee that all jobs of τ_i meet their deadline. To ensure the schedulability of τ_i, we define

$$A_i(t) = A_{ij}, \; C_{ij-1} \leq t < C_{ij}, \; j \geq 1, \qquad (8)$$

where $C_{i0} = 0$. The non-decreasing step function $A_i(t)$ gives the largest amount of aperiodic processing in $[0, t]$ at priority level i or higher possible such that the processor is constantly busy with priority level i or higher activity but all jobs of τ_i meet their deadline.

To illustrate, let us consider a task set with two tasks, τ_1 and τ_2, with $C_1 = 1$, $T_1 = 4$, $D_1 = 1$, $\phi_1 = 0$, and $C_2 = 3$, $T_2 = 6$, $D_2 = 6$, $\phi_2 = 0$. Note that the tasks follow a deadline monotonic priority order. We restrict our attention to an interval of time [0, H], where H is the ***hyperperiod*** of the task set, or the time at which the distribution of periodic arrivals repeats itself. The hyperperiod of a periodic task set is equivalent to the least common multiple of the task periods which is 12 for this example. Figure 1.a shows the processor schedule if no aperiodic work is processed. The non-decreasing functions $A_1(t)$ and $A_2(t)$ are shown in Figure 1.b. These are step functions, with *jump points* corresponding to the completion times of the jobs of τ_i, or the C_{ij}'s, and *jump heights* corresponding to the A_{ij} values computed by Equation (7). Note that in this example, all jump points for τ_1 are known *a-priori* because $C_{1j} = D_{1j}$, for all $j \geq 1$. Thus, every job of τ_1 has zero slack, in the interval of time between its arrival and completion. On the contrary, the jobs for τ_2 have non-zero slack, so their execution can be delayed by the processing of aperiodic tasks. As a result, their exact completion times depend on the amount of higher priority aperiodic processing done at run-time. Although the exact completion times for each job of τ_2 cannot be determined *a-priori*, their best- and worst-case values are known. For instance, job τ_{21} in Figure 1.a will complete no earlier than time 4 and no later than time 6, so its jump point is guaranteed to lie somewhere in the time interval [4, 6]. For the particular case in which aperiodics consume all aperiodic processing time possible, the jump point for τ_{21} is 6, as shown in Figure 1.b for $A_2(t)$.

We next determine bounds on the aperiodic processing at each level for which all deadlines of all periodic tasks can still be met. Let $L_i(t)$ denote the total amount of aperiodic processing in $[0, t]$ at priority level $i, 1 \leq i \leq n$. For τ_k to meet all deadlines, it is necessary that

$$L_1(t) + \ldots + L_k(t) \leq A_k(t), \quad 1 \leq k \leq n. \qquad (9)$$

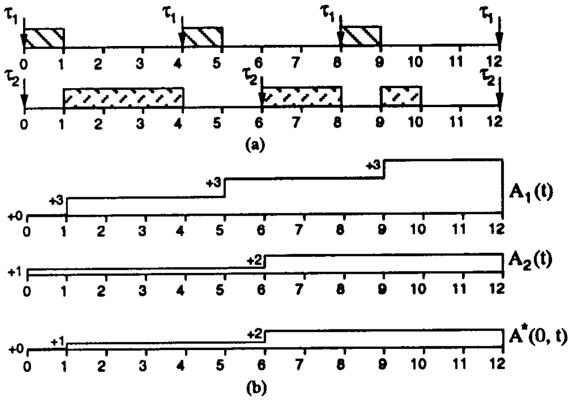


Figure 1: Example 1. Illustrating Slack Stealer operation: (a) Processor schedule in absence of aperiodics; (b) Functions used by the Slack Stealer

Let $A^\star(t) = min_{\{1 \leq k \leq n\}} \; A_k(t)$ and $k^\star(t)$ be the index of the highest priority level satisfying $A_{k^\star(t)} = A^\star(t)$. Thus $L_1(t) + \ldots + L_{k^\star(t)}(t)$ can be no larger than $A^\star(t)$. If this sum does assume its maximum value, then all n inequalities in (9) will hold. Hence, all periodic tasks at all levels with deadlines no later than t will meet their deadlines and the processor will be continuously busy throughout $[0, t]$ executing only tasks of priority $k^\star(t)$ or higher. Hence, all periodic task deadlines before t will be met if and only if

$$L_1(t) + \ldots + L_{k^\star(t)}(t) \leq A^\star(t) \qquad (10)$$

Figure 1.b illustrates the function $A^\star(t)$ for our task set example. Note that priority level 1 places the tightest constraint on aperiodic processing time available in the interval [0, 1), whereas in the interval [1, 12), priority level 2 places the tightest constraint. Therefore, $k^\star(t) = 1$, for $0 \leq t < 1$, and $k^\star(t) = 2$, for $1 \leq t < 12$.

We next address the question of the priority level at which the aperiodic tasks can be executed. If $L_1(t) + \ldots + L_{k^\star(t)}(t) \leq A^\star(t)$, then one can modify $L_1(t) + \ldots + L_{k^\star(t)}(t)$ to $L'_1(t) = L_1(t) + \ldots + L_{k^\star(t)}(t)$, $L'_2(t) = \ldots = L'_{k^\star(t)}(t) = 0$ and still be feasible. In other words, one can carry out all aperiodic processing at the highest priority level without any reduction in aperiodic capacity. Since elevating the priority level of aperiodic processing reduces their response times, it is optimal to service aperiodic tasks at the highest priority level, and the total aperiodic processing time cannot exceed $A^\star(t), t \geq 0$. It follows for the case in which $W_A(t) > \varepsilon(t)$ for all feasible scheduling algorithms, that the upper envelope on aperiodic processing time is given by $A^\star(t), t \geq 0$.

The previous analysis assumed that there was always a sufficiently large amount of aperiodic work to be processed such that aperiodic processing would always use all available slack at all levels. This is, however, not the general case. There may often be times at which aperiodic processing could be done but none is ready. We must modify our analysis to accommodate this case and define the upper envelope. Define

$A(t)$ = cumulative aperiodic processing consumed at any priority level during $[0, t]$

$I_i(t)$ = level-i inactivity during $[0, t]$, for $1 \leq i \leq n$ and $t \geq 0$.

Here, level-i inactivity refers to the cumulative amount of time spent processing periodic tasks of priority $i+1$ or lower or any time the processor is idle during $[0, t]$.

Suppose we now start at time s rather than at 0 and wish to determine the maximum amount of aperiodic processing possible during $[s, t], t \geq s$. The analysis is the same as before. For example, A_{ij} gives the largest possible amount of aperiodic processing at priority level i or higher that can be carried out in $[0, t]$ and still meet the deadline of τ_{ij}. However, during $[0, s]$, $A(s)$ units of processing have already been *used* for aperiodic processing and $I_i(s)$ units of level-i inactive time have taken place, time which was available for level i aperiodic processing but was not used for that purpose. Thus the amount of time *available* for additional aperiodic processing at time t, A_{ij} must be reduced by $A(s) + I_i(s)$. Generalizing Equation (8) to an arbitrary time origin s, we define for $t \geq s$,

$$A_i(s,t) \;=\; A_{ij} - A(s) - I_i(s), \quad C_{ij-1} \leq t < C_{ij}. \quad (11)$$

This quantity gives the maximum amount of aperiodic processing time possible during $[s, t]$ at level i or higher with all τ_i deadlines still being met. The analysis is now the same as the earlier analysis with $s = 0$. Specifically, define

$$A^\star(s,t) \;=\; min_{\{1 \leq i \leq n\}} \; A_i(s,t) \qquad (12)$$

and $A_{k^\star(t)}(s,t) = A^\star(s,t)$ where $k^\star(t)$ is the highest priority such task. As before, the total aperiodic processing during $[s, t]$ cannot exceed $A^\star(s,t)$ and all should be executed at the highest priority level. The function $A_{k^\star(t)}(s,t)$ thus gives the upper envelope on aperiodic processing over any interval $[s, t]$ during which the aperiodic workload does not vanish.

Referring back to our previous example, suppose that no aperiodic work was ready during [0, 5.5] and an aperiodic task, τ_{ap}, requiring 2 units of computation arrives at 5.5. We now use 5.5 as the new time

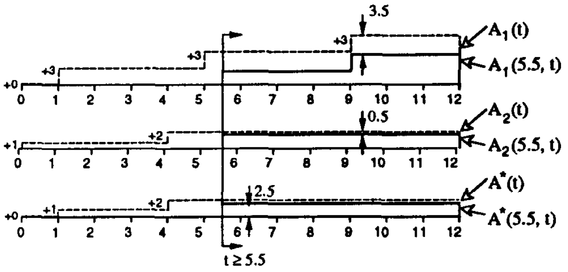


Figure 2: Example 1. Illustrating Slack Stealer operation with a change in the time origin

origin and note that $A(5.5) = 0$, $I_1(5.5) = 3.5$ (corresponding to the level-1 inactivity during [1, 4] and [5, 5.5]) and $I_2(5.5) = 0.5$ (corresponding to the level-2 inactivity during [5, 5.5]). The level-i inactivity values can be visualized in Figure 1.a. Since we have changed the time origin to 5.5, the curves from Figure 1.b must be adjusted for $t \geq 5.5$, to reflect the fact that some aperiodic processing time has been lost due to inactivity. Thus, the functions $A_1(5.5, t)$, $A_2(5.5, t)$, and $A^\star(5.5, t)$ are obtained according to Equations (11) and (12). These are depicted in Figure 2. Given these functions, the slack stealer finds 2.5 units of processing capacity at time 5.5 and immediately allocates 2 units to service τ_{ap}. Consequently, τ_{ap} finishes at time 7.5 and leaves 0.50 units of aperiodic processing available for any aperiodic tasks that may arrive during [5.5, 7.5]. If no other aperiodic tasks arrive, the processor will spend [7.5, 8] on τ_2, [8, 9] on τ_1, [9, 11.5] on τ_2 and then idle during [11.5, 12).

We next define the slack stealing algorithm, an algorithm which achieves the maximum possible amount of aperiodic service time subject to the constraint of meeting all the periodic deadlines. Later we will prove its optimality property using the upper envelope derived in this section.

3.2 Algorithm Description

The slack stealing algorithm uses the functions $A_i(t)$ for each task τ_i, $1 \leq i \leq n$, in determining the capacity that can be allocated to aperiodic service. Because the arrival pattern of the periodic workload repeats itself every H time units, or the task set hyperperiod, it is sufficient to compute all $A_i(t)$ functions for $0 \leq t < H$. Given this, we compute the jump heights associated with each job τ_{ij}, of each task τ_i, according to Equation (7), for $0 \leq t < H$. These jump heights are then stored as pairs of points (i, j), where i is the task priority, $1 \leq i \leq n$ and j is the job number for τ_i, $1 \leq j \leq H/T_i$. This two-dimensional array of values is computed before run-time.

At run-time, we distinguish $n+2$ different activities. Activity 0 refers to aperiodic task processing, activities $1,\ldots,n$ refer to periodic task processing at the corresponding priority level, and activity $n+1$ refers to the processor being idle. We also establish accumulators which are initialized to 0 at time 0 and reset at the beginning of each hyperperiod. At any time, $\mathcal{A}$ gives the total aperiodic processing and $\mathcal{I}_i$ gives the level-i inactivity, $1 \leq i \leq n$.

Suppose the processor begins activity $j, 0 \leq j \leq n+1$ at time t_1 and finishes that activity at time $t_2 \geq t_1$. Then,

$$\text{if } j = \begin{cases} 0, & \text{add } t_2 - t_1 \text{ to } \mathcal{A}, \\ 1, & \text{do nothing.} \\ 2 \leq j \leq n, & \text{add } t_2 - t_1 \text{ to } \mathcal{I}_1, \ldots, \mathcal{I}_{j-1}, \end{cases}$$

Any time there is aperiodic work ready to be done, say at time s, we must determine the slack available for aperiodic processing by computing

$$A^\star(s,t) = min_{\{1 \leq i \leq n\}}(\ A_i(s,t) - \mathcal{I}_i(s)\ ) - \mathcal{A}. \quad (13)$$

Suppose W is the amount of aperiodic work to be done. If $A^\star(s,t) \geq W$, then the W units of aperiodic activity can be processed immediately, i.e., in $[s, s+W]$, at the highest priority level. If $A^\star(s,t) < W$, then aperiodic work can be done during $[s, s+A^\star(s,t)]$ at the highest priority level, but no further work can be done until additional slack becomes available. Note that more aperiodic processing capacity can become available only when a periodic job is completed, because these are the only points in time in which the $A_i(s,t)$ functions step up to their next values. Thus, the evaluation of $A^\star(s,t)$ should be done only when aperiodic work arrives to an empty aperiodic queue or when there is aperiodic work ready and a periodic task completes.

3.3 Optimality of the Slack Stealer

In this section, we combine the formulations given in sections 2 and 3.1 and the algorithm given in section 3.2 to provide a proof of optimality of the slack stealer. The result is summarized in the following theorem:

Theorem 1 *For any periodic task set scheduled by a given fixed priority algorithm and any aperiodic arrival stream processed in FIFO order, the slack stealing algorithm minimizes the response time of every aperiodic task among all scheduling algorithms which meet all periodic task deadlines.*

Proof: For the given periodic task set and aperiodic task arrival process $\{J_k, k \geq 1\}$, the slack stealing algorithm will create a sequence of aperiodic busy intervals $\{B_k\}_{k=1}^{\infty}$ with $B_k = [l_k, u_k)$, $l_k < u_k$, $u_k < l_{k+1}$. The B_k denotes intervals of time during which there is aperiodic work available for processing. In the notation of section 2, $l_1 = \alpha_1$, and $W_A(t) = \varepsilon^\star(t)$, for $u_k \leq t < l_{k+1}$, $k \geq 1$. Now consider any other scheduling algorithm $\mathcal{X}$ which uses the same fixed priority assignment for the periodics and meets all such deadlines. Algorithm $\mathcal{X}$ will create a cumulative aperiodic execution function $\varepsilon(t)$. We wish to show that $\varepsilon^\star(t) \geq \varepsilon(t)$, $t \geq 0$, from which, using the argument in section 2, we can conclude that all aperiodic response times are minimized. Clearly, there is no aperiodic activity until α_1, the time of the first arrival, so $\varepsilon^\star(t) = \varepsilon(t) = 0$, for $t \in [0, \alpha_1)$. During $[\alpha_1, u_1)$, $\varepsilon^\star(t) = A^\star(0,t)$, because the slack stealer achieves the upper envelope $A^\star(0,t)$, and throughout $[\alpha_1, u_1]$, $\varepsilon^\star(t) = A^\star(0,t) \geq \varepsilon(t)$ since $A^\star(0,t)$ gives an upper bound on cumulative aperiodic execution. There is no aperiodic work for the slack stealer during $[u_1, l_2)$, thus $\varepsilon^\star(t) \geq \varepsilon(t)$ during this interval. Algorithm $\mathcal{X}$ may also finish all aperiodic work in which case $\varepsilon^\star(l_2) = \varepsilon(l_2)$ or it may not, in which case $\varepsilon^\star(l_2) > \varepsilon(l_2)$. For $t \in [l_2, u_2)$, the slack stealer achieves the upper envelope of cumulative aperiodic execution, $A^\star(l_2, t)$, and there is aperiodic work always available during this interval. Thus the increments in aperiodic execution satisfy $\varepsilon^\star(t) - \varepsilon^\star(l_2) = A^\star(l_2,t) \geq \varepsilon(t) - \varepsilon(l_2)$. Thus recalling that $\varepsilon^\star(l_2) \geq \varepsilon(l_2)$, we have $\varepsilon^\star(t) \geq \varepsilon(t)$, $t \in [l_2, u_2)$. Thus $\varepsilon^\star(t) \geq \varepsilon(t)$, for $0 \leq t \leq u_2$. A simple induction argument over the busy intervals $\{B_k\}_{k=1}^{\infty}$ now proves that the response time of *every* aperiodic task is minimized provided tasks are processed in FIFO order.

4 Extensions of the Algorithm

4.1 Reclaiming Unused Periodic Execution Time

Previous work suggests that the actual execution time of tasks may be very different from their estimated worst-case execution times [10, 11]. As a result, a pre-allocation of processor utilization for periodic tasks based on their worst-case execution times may result in an undesirable waste of processing potential. Some researchers have taken advantage of this pre-allocated but unused execution time to improve average system performance [12] or to redundantly execute the tasks for transient fault detection [13]. In spite of its potential advantages, aperiodic server algorithms such as polling, the deferrable server, and the sporadic server, do not provide the ability to reclaim unused periodic execution time. However, the slack stealing algorithm can be easily extended to reclaim this time.

If the execution time of τ_i is stochastic with C_i being an upper bound, then the actual execution time of τ_i may be smaller than C_i. If τ_{ij} finishes early, the unused time can be used for aperiodic processing. This can be done by adding the unused execution time, call it U_{ij}, to $A_i(t)$, $t \geq C_{ij}$. A simpler way to implement this is to subtract U_{ij} from $\mathcal{I}_k, i \leq k \leq n$, which will make the time available to tasks at or below priority level i.

4.2 Servicing Hard Deadline Aperiodics

The problem of scheduling hard deadline aperiodic tasks where the periodic tasks are scheduled using the Earliest Deadline algorithm was studied by Chetto and Chetto [14]. A similar approach can be used to solve this scheduling problem for the case in which the periodic tasks are scheduled according to a fixed priority algorithm.

The quantity $A^\star(s,t)$ gives the total amount of periodic processing that can occur in an interval $[s,t]$. If one were faced with an aperiodic task arriving at s having a processing requirement of C and a hard deadline of D, one could check to see if $C \leq A(s, s+D)$. If so, that aperiodic task's deadline could be guaranteed. This is the basis for an approach to guaranteeing hard-deadline aperiodic tasks. This topic is the subject of a subsequent paper.

4.3 Managing Aperiodic Capacity: The Allocation Problem

The slack stealing algorithm maximizes the aperiodic processing capacity available during $[0,t], t \geq 0$. This, coupled with the fact that all aperiodic service occurs at the highest priority level, is a first step toward optimality. We must, however, consider how the aperiodic processing capacity should be allocated among the pending aperiodic tasks to minimize their response times on average. If the aperiodic tasks have deterministic processing times which are known at arrival, the solution is straightforward. The aperiodic tasks should be processed according to the SRPT (shortest-remaining-processing-time) algorithm. Thus, the aperiodic task with the shortest remaining processing time should be the one utilizing the aperiodic service capacity if any is available. The optimality of this service policy for minimizing response times is well-known (see, for example Schrage [15]). Because the processor must process both aperiodic and periodic tasks, rather than having a single task type, Schrage's argument must be modified. Nevertheless, the interchange argument presented in [15] still applies, and the proof requires only minor changes in his argument.

It is important to point out that the SRPT would reduce the *average* aperiodic response times; however, if the slack stealer were to process aperiodic tasks using SRPT instead of FIFO ordering, it would no longer possess the strong optimality property of minimizing *every* aperiodic response time. Thus an aperiodic scheduling policy different from the slack stealer using SRPT will have a longer average aperiodic response time, but the response times of some of the aperiodic tasks may be shorter.

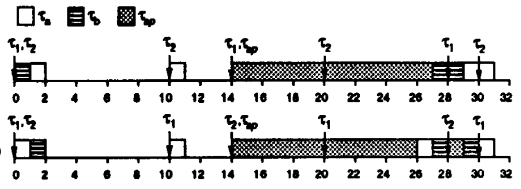
4.4 Finding An Optimal Fixed-Priority Assignment for Joint Scheduling

The slack stealer maximizes the time available for aperiodic processing during any interval of time among all algorithms that use fixed priority for the periodic tasks and meet all periodic deadlines. However, this optimality property is relative to a given fixed-priority order for the periodic tasks. To see that changes in the fixed priority order can alter the aperiodic response times, consider the following example:

Example 2

Consider two periodic tasks τ_a and τ_b with the following timing requirements: $C_a = C_b = 1$, $T_a = 14$, $T_b = 10$, $D_a = 14$, $D_b = 10$, $I_a = I_b = 0$. Assume these tasks are *not* in rate monotonic order, so that $\tau_a = \tau_1$ and $\tau_b = \tau_2$. Suppose an aperiodic task, τ_{ap}, arrives at time 14 and requires 13 units of processing. As illustrated in Figure 3.a, this task can be processed during [14, 27], thus completing at 27 for a response time of 13. τ_1 will be processed during [27, 28], then [28, 29], while τ_2 will be processed during [29, 30] and then [30, 31]. Consequently, all periodic deadlines are met and the response time is 13.

If the priority order were reversed, so that it conforms to the rate monotonic (RM) algorithm, then $\tau_a = \tau_2$ and $\tau_b = \tau_1$, As illustrated in Figure 3.b, the same aperiodic task would be processed during [14, 26] and [28, 29] for a response time of 15, which is longer than the 13 obtained with the other fixed-priority assignment. τ_1 would be processed in [26, 27] then [30, 31], while τ_2 would be processed in [27, 28], then [29,



τ_1 τ_2 τ_{ap}
(a)
τ_1,τ_2 τ_2 τ_1,τ_{ap} τ_2 τ_1 τ_2
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
(b)
τ_1,τ_2 τ_1 τ_2,τ_{ap} τ_1 τ_2 τ_1
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32

Figure 3: Example 2. Illustrating effect of Fixed-Priority Assignment on Aperiodic Response-Times: (a) Non-RM order; (b) RM order

30]. The aperiodic task cannot be processed beyond 26 or else τ_2 would miss its deadline at 28.

In spite of the lack of global optimality, we recommend that the periodic tasks be given a deadline monotonic order.

5 Performance Evaluation

It is important to compare the performance of the slack stealing algorithm[4] against that of current aperiodic scheduling policies, such as background, polling, and the recently proposed server algorithms. To this effect, we would like to answer the following questions: *What are the operating conditions for a real-time system under which the slack stealer significantly outperforms state-of-the-art server algorithms? How much do their performances differ under such conditions?* The answers to these questions can provide valuable insights for assessing the potential value of implementing the slack stealing algorithm as an alternative to current server algorithms.

Another issue of interest is to predict the average response times of aperiodic tasks which are serviced using the slack stealing algorithm. We will show that a very simple approximation, namely using a queueing model which ignores all service delays due to the periodic tasks is very accurate over a wide range of operating conditions. Surprisingly, it is sufficient to compute the aperiodic response times that would be attained if the aperiodic tasks were to have sole access to the processor. Clearly, the response times predicted by such a queueing model could be very optimistic, especially in situations of high periodic loads. However, this model provides the absolute lowest aperiodic response times attainable, thus constituting a lower bound. This lower bound is useful in estimating the degree of interference exerted by the periodic workload on the aperiodic response times. The estimation of periodic interference is interesting, because it gives insight into the cost to aperiodic responsiveness incurred by servicing the periodic and aperiodic workloads on a *shared* processor. Consequently, we also wish to identify the operating conditions in which the performance of the slack stealing algorithm approaches and/or deviates significantly from the lower bound derived from the queueing model.

5.1 Variables Affecting Aperiodic Response Times

There are four important parameters which play a major role in determining the aperiodic task response times and the accuracy of simple queueing formulas for approximating those response times. The parameters are: (1) the periodic load, expressed as a utilization factor, U_p, obtained by summing the utilizations of the individual periodic tasks, (2) the breakdown utilization, U_{BD}, of the periodic task set, a measure of the maximum utilization attainable for the periodic task set without violating any of the tasks' timing constraints, (3) the utilization of the aperiodic tasks, U_{ap}, and (4) the mean computation requirement of the aperiodic tasks, $1/\mu$.

To understand the role of these four parameters, consider first the aperiodic load in isolation. Standard results from queueing theory indicate that job response times will increase with the traffic intensity parameter, $\rho_{aper} = U_{ap}$, and also with the mean computation requirement of the aperiodic jobs. For example, in an M/M/1 queueing system, aperiodic response times are given by $\{\mu(1-\rho_{aper})\}^{-1}$. These effects will carry over to a processor which handles both aperiodic tasks and hard deadline periodic tasks.

Over a long time interval, the processor must devote a fraction of time, U_p, to servicing periodic tasks. The crucial issue is whether there is usually enough slack for the aperiodics to be serviced immediately and to completion or whether the aperiodics will be forced to wait from some periodics to be processed so they can meet their deadlines. The longer the aperiodic mean service times, the more the periodics will be forced to interrupt aperiodic processing thus increasing their response times. We also need a measure of the amount of slack that is typically available for aperiodic use. The larger the value of U_p, the smaller the slack that is available. Moreover, the difference between U_p and U_{BD} is an additional measure of the slack that is typically available. As U_p approaches U_{BD} there will be increasingly long periods of time in which little if any slack is available which will, in turn, increase aperiodic

[4]Henceforth, the terms slack stealing algorithm and optimal algorithm are used interchangeably.

response times. Hence, we should find that aperiodic response times are an increasing function of $1/\mu$ and U_p and a decreasing function of $U_{BD} - U_p$, the proximity of the periodic load to the task set's breakdown utilization.

In order to model the effect of changes in the mean computation requirement of the aperiodic tasks, we define a parameter called the *demand-capacity ratio*. The demand-capacity ratio is the ratio of the mean aperiodic execution time to the maximum capacity or execution time of a server task residing at the highest priority level.

5.2 Simulation Studies

Simulations were conducted to compare the average response times of four aperiodic scheduling policies, namely, background, polling, sporadic server, and the slack stealing or the optimal algorithm. The deferrable server was omitted because its performance is comparable to that of a sporadic server. This section outlines the basic simulation framework and proceeds to present in-depth simulation studies.

Three periodic task sets were considered. Two were randomly generated and one is an application task set obtained from an Inertial Navigation System (INS) [16]. The objective in selecting these task sets is twofold: first, to give some insight into the scope of performance variations that may be attributed to significant differences in the breakdown utilizations of the periodic task sets; and second, to explore the potential performance gains that may be attained in a practical system by using the optimal algorithm.

In all simulations, aperiodic tasks were queued and serviced in FIFO order. The aperiodic workload was modeled by an exponential distribution of execution times and Poisson arrival times. An M/M/1 queueing model can be used to compute the ideal response time bound for such an aperiodic workload, using the M/M/1 formula previously stated. Without loss of generality, we chose mean aperiodic execution times to be a fraction of the maximum execution time for a sporadic server task with a period equal to the smallest period of the periodic task set. The periods for the polling tasks were also fixed in this manner.

5.2.1 Evaluation of Random Task Sets

Sets of 10 periodic tasks were randomly generated, with periods ranging from 55 to 2,310 and a common hyperperiod of 2,310, until one high and one low breakdown utilization set was found. One set had a breakdown utilization of 77%, which is close to the Liu and Layland least upper schedulability bound for a set of 10 tasks, given by $10(2^{1/10} - 1) \approx 71.77\%$ [7]. This task set is identified as T_{LBD}, because of its low breakdown utilization. The other task set had a high breakdown utilization of $\approx$ 100% and is identified as T_{HBD}.

Load	Task Set	SS Size	Exec. Times	
40%	T_{LBD}	26.49	1.32	6.62
	T_{HBD}	33.00	1.65	8.25
70%	T_{LBD}	5.10	0.26	1.28
	T_{HBD}	16.50	0.83	4.13

Table 1: Mean Aperiodic Execution Times for Demand-Capacity Ratios of 5% and 25%

For a given periodic task set, the computation requirements were scaled to create two periodic utilizations, 40% and 70%. For the case of a 40% periodic load, the aperiodic load was varied from 5%-55%. For a 70% periodic load the aperiodic load was varied from 5%-25%. Note that the maximum total load in both cases was 95%. The demand-capacity ratios considered were 5% and 25%. The sporadic server (SS) sizes and the aperiodic mean execution times corresponding to such demand-capacity ratios are summarized in Table 1. The size of a polling task is equal to that of a corresponding sporadic server.

Figure 4 presents the aperiodic response times for the low breakdown utilization task set, T_{LBD} with a periodic load of 40%. In general, a significant performance degradation is observed as the aperiodic load is increased from 5%-55%. Note that the magnitude of the performance degradation does not solely depend on a particular aperiodic load but also on the demand-capacity ratio. Higher demand-capacity ratios imply that aperiodic jobs arrive less frequently but are larger on average, which increases task waiting times and the probability of preemption by periodic tasks. Figure 4.b shows that at high aperiodic loads and relatively large job sizes, *all* aperiodic service algorithms, including the optimal, eventually have a difficult time accommodating aperiodic requests responsively. Under these conditions, aperiodic response times are dominated by the high accumulation of preemption delays due to the periodic tasks. On the contrary, for relatively small aperiodic job sizes their response times are primarily dominated by the ability of the aperiodic scheduling algorithm to efficiently allocate time on the processor. This observation is supported by Figure 4.a which shows a noticeable performance dif-

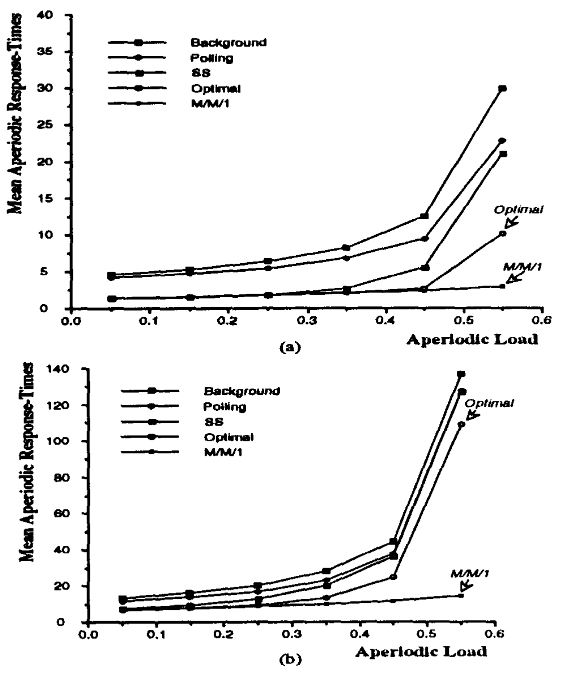


Figure 4: Response-Times for T_{LBD} at 40% and demand-capacity ratios of: (a) 5% ;(b) 25%

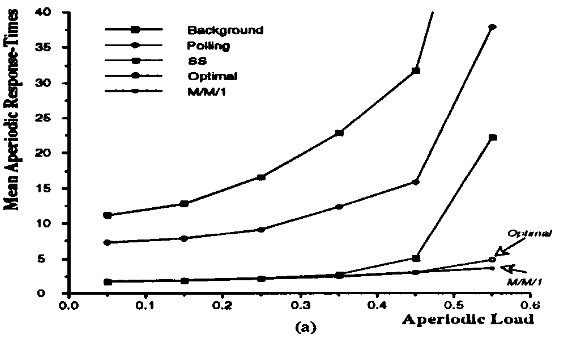
ference among the algorithms over the entire range of aperiodic loads.

The optimal algorithm outperforms all of the other aperiodic scheduling algorithms studied. Furthermore, background scheduling has the worst performance, and the performance of the sporadic server is the closest to the optimal. With a 5% demand-capacity ratio, the performance of the sporadic server is approximately optimal for aperiodic loads up to 25%, or a combined load of 65%. At higher aperiodic loads the performance of the sporadic server tends to degrade to that of a polling task, because there is a high probability that there is always aperiodic work pending upon the arrival of the server task. As a result, the differences in consumption and replenishment of execution time for a polling and sporadic server become indistinguishable. Increasing the demand-capacity ratio to 25% causes the sporadic server to deviate from the optimal even at small aperiodic loads. However, the magnitude of their performance gap is not large because, as stated previously, this is a very stressful condition for all aperiodic service algorithms.

A very interesting result is that the performance of the optimal algorithm is very close to the ideal M/M/1 bound for an appreciable region of operating conditions. Hence, in this region, the optimal algorithm is capable of "masking out" the presence of periodic tasks so efficiently that the responsiveness of aperiodic tasks is almost equal to that attainable on a *dedicated* processor. In Figure 4.a the performance of the optimal algorithm is within a 10% envelope above the M/M/1 bound for combined loads up to 85% and up to 75% in Figure 4.b. On the other hand, the performance of the optimal algorithm deviates significantly from the M/M/1 bound when the aperiodic load is high and/or the job sizes are relatively large. Deviations are due exclusively to the preemption delays caused by the periodic tasks, as any delays due to the aperiodic queue length are accounted for in the queueing model. This highlights our previous observation that preemption delays dominate response times when aperiodic job sizes are large. In addition, we observe that if the aperiodic load is sufficiently high, the response times are affected by preemption delays even if the jobs are small (see Figure 4.a at a 55% aperiodic load). The cause is a large incidence of *spill-overs*, or cases in which the server execution time is not long enough to empty the aperiodic queue before the next preemption by periodics [4]. In conclusion, results indicate that T_{LBD} has a particular operating region under which the M/M/1 queueing model can be used to predict average aperiodic response times within a reasonably small error margin. Furthermore, the optimal algorithm guarantees that this operating region, delimited by a particular aperiodic load and mean aperiodic job size, is maximized.

Now consider a task set with a high breakdown utilization, such as T_{HBD}. Simulation results for T_{HBD} with a periodic load of 40% are presented in Figure 5. The general trends observed are the same as those described for T_{LBD} in Figure 4. However, unlike T_{LBD}, the performance gains from using server algorithms over background are more significant. Task sets with a high breakdown utilization tend to have larger aperiodic server execution times as compared to those with a low breakdown utilization, as illustrated in Table 1. A larger server execution time reduces the probability of spill-overs, so the aperiodic response times are shorter on average, yielding a more dramatic performance advantage over background. By the same token, the performance differences among the polling and sporadic server algorithms in Figure 5 are more noticeable for the same demand-capacity ratios depicted in Figure 4.

An interesting observation is that for a fixed demand-capacity ratio and a given aperiodic load, the mean aperiodic execution times for T_{HBD} are



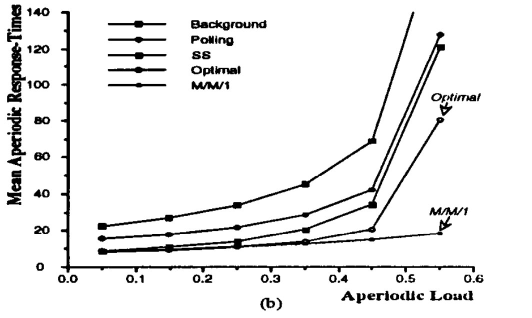
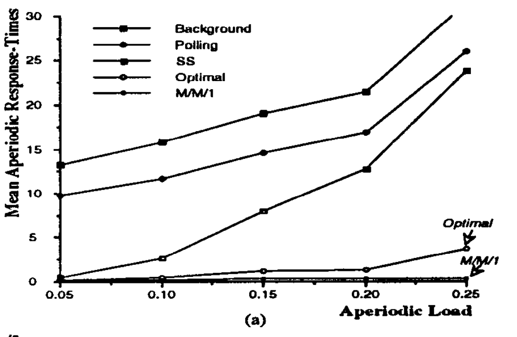


Figure 5: Response-Times for T_{HBD} at 40% and demand-capacity ratios of: (a) 5% ;(b) 25%



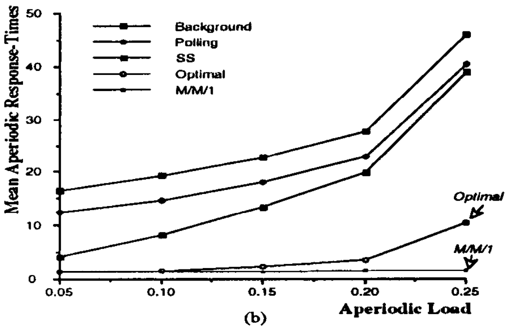


Figure 6: Response-Times for T_{LBD} at 70% and demand-capacity ratios of: (a) 5% ;(b) 25%

larger than those for T_{LBD}. Thus, one would expect that larger job sizes would tend to mask out performance differences due to a larger server execution time. However, noticeable performance differences for the two task sets are observed, nonetheless. These results are further evidence of the significant impact of breakdown utilization on the performance of different scheduling algorithms in allocating time for aperiodic service.

The high breakdown utilization of T_{HBD} guarantees that there is no utilization loss when the server task is added to the task set; the total utilization of the periodic task set and the server task is 100%, regardless of the periodic load. Given this, the performance of the sporadic server algorithm promises to be ideal in that none of the aperiodic processing is relegated to background. Hence, the performance of the sporadic server lies close to the optimal for an appreciable range of aperiodic loads (e.g., up to 35% in Figure 5.a). Yet, the optimal algorithm outperforms the sporadic server as the aperiodic load is increased, even under these ideal conditions. This phenomenon illustrates a major shortcoming of current server algorithms.

The underlying philosophy behind current server algorithms is to transform processor idle times (which would otherwise appear as background service times), into a more favorable distribution of service opportunities for aperiodics. Background time is carefully transformed into a high priority server task with *periodic* timing attributes such that schedulability is not violated. As a result, the distribution of service opportunities provided by such server tasks tends to become periodic as the aperiodic load increases. Evidence of this is the prior observation that the performance of the sporadic server and that of a polling server are indistinguishable at sufficiently high aperiodic loads, as depicted in Figure 4.b.

The optimal algorithm circumvents this shortcoming by allocating time for aperiodic service in the most aggressive way possible subject only to schedulability constraints. Hence, the artificial timing constraints imposed by a periodic server abstraction are eliminated. Moreover, the distribution of service opportunities is optimal, so aperiodic response times are guaranteed to degrade as gracefully as possible.

Consider an increase in the periodic load. Figure 6 presents the aperiodic response times for T_{LBD} with a periodic load of 70%. The performance curves for background are not shown because they have degraded an order of magnitude relative to all other algorithms. Recall that the aperiodic load varies from 5% to 25%.

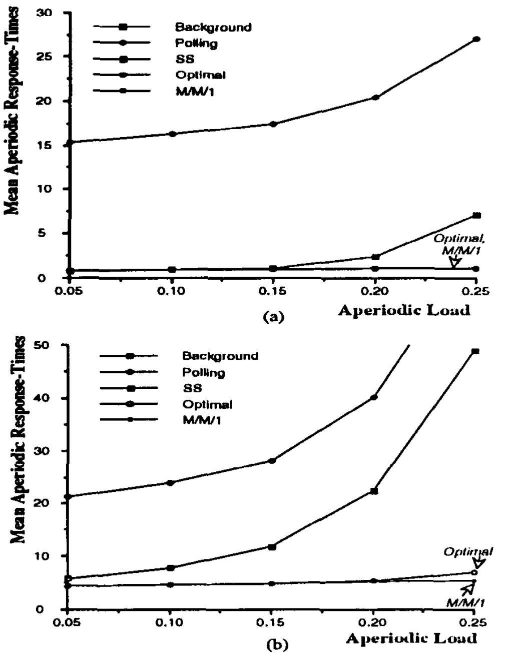


Figure 7: Response-Times for T_{HBD} at 70% and demand-capacity ratios of: (a) 5% ;(b) 25%

There are two outstanding differences with respect to the results presented for a 40% periodic load in Figure 4. First, note the dramatic performance gains of the optimal algorithm as compared to all other algorithms. Although the sporadic server continues to outperform polling and background, its performance deviates from the optimal even at low aperiodic loads. This performance gap is primarily caused by the small server size (5.10) which is a function of the low breakdown utilization. The performance gains of the optimal algorithm, on the other hand, illustrate the superiority of allocating time for the aperiodics on demand, without the constraints of a periodic server task. Second, we observe that the optimal curve departs from the ideal M/M/1 bound at lower aperiodic loads than those observed for the 40% load. Evidently, at a 70% periodic load, the preemption delays experienced by the aperiodics have a significant impact on their response times.

Increasing the utilization of T_{HBD} to 70% has a different effect on aperiodic response times, as shown in Figure 7. Note the difference in the performance gaps between the sporadic server and the optimal for a 5% demand-capacity ratio, shown in Figures 6.a and 7.a. Because of its high breakdown utilization, T_{HBD} has a much larger server execution time than T_{LBD} (16.50 vs. 5.10), which allows the sporadic server to maintain near optimal performance for aperiodic loads below 15% and relatively small job sizes. In addition, it is observed that the performance of the optimal algorithm is remarkably close to the M/M/1 bound for the entire range of aperiodic loads. These results suggest that aperiodic tasks can experience an ideal level of responsiveness even when sharing the processor, as long as the periodic task set has a high breakdown utilization and the aperiodic execution times are not very large.

Task	Period	Exec.	Description
τ_1	2.5	1.18	Update Ship Attitude
τ_2	40	4.28	Update Displacement
τ_3	62.5	10.28	Send Attitude Message
τ_4	1000	20.28	Send Navigation Message
τ_5	1000	100.28	Update Status on Screen
τ_6	1250	25	Update Ship Position

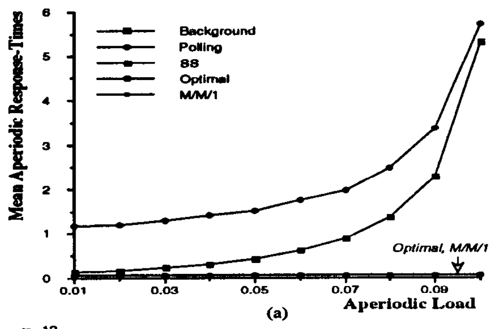
Table 2: Timing Requirements for the Inertial Navigation System

5.2.2 Evaluation of the INS Task Set

The INS task set consists of 6 periodic tasks as described in Table 2. It has a periodic load of $\approx$ 88%, a hyperperiod of 5,000 and a breakdown utilization of 99.4%. The sporadic server size is 0.28, yielding mean aperiodic execution times of 0.028 and 0.069 for demand capacity ratios of 25% and 50%, respectively. The aperiodic load was varied from 1%-10%, for a maximum combined load of 98%. Simulation results are shown in Figure 8.

Once again, the performance of the sporadic server approaches the optimal at low aperiodic loads and small aperiodic execution times. However, the sporadic server deviates from the optimal at a very low aperiodic load (less than 3% in Figure 8.a), which reaffirms the higher susceptibility of the sporadic server to the periodic load relative to the optimal algorithm. In fact, the optimal algorithm is capable of maintaining aperiodic response times equal to those of the M/M/1 bound for the entire range of aperiodic loads and both demand-capacity ratios shown in Figure 8.

The performance results for the INS task set confirm that a periodic workload with a high breakdown utilization is very favorable to aperiodic responsiveness, as concluded in the random task set study. A surprising result is that the optimal algorithm's performance is essentially equal to that of an M/M/1



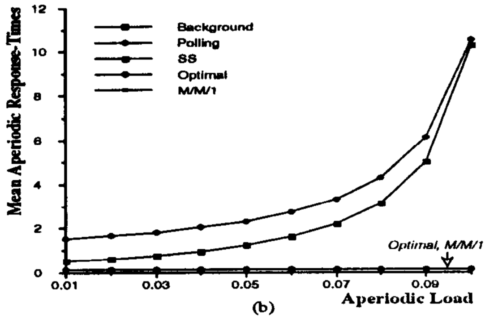


Figure 8: Response-Times for the INS Task Set and demand-capacity ratios of: (a) 25% ;(b) 50%

queue with the periodic workload ignored, even when the periodic load is as high as 88%. It seems even more surprising that such a performance is maintained for all reasonable aperiodic loading levels, even for a moderate demand-capacity ratio of 50%. This is a significant result because it shows a promising direction for finding some solutions to the analytical prediction of aperiodic response times, which is an open research issue.

5.3 Performance Summary

Simulation data shows that all aperiodic scheduling algorithms are sensitive to the breakdown utilization of the periodic task set. A low breakdown utilization suggests that the schedulability constraints of the task set make it inherently more difficult to steal large blocks of time for aperiodic service relative to a task set with a high breakdown utilization.

The sporadic server can attain near optimal performance when both the periodic and aperiodic loads are relatively low. In contrast, neither background nor polling approach optimal performance under any of the circumstances considered.

The performance of the sporadic server degrades significantly when the periodic load is increased, even if the periodic task set has a high breakdown utilization. This performance degradation is due to the rapid increase in the incidence of spill-overs as the server execution time is decreased.

The optimal algorithm, on the other hand, is much less sensitive to the periodic load than any other algorithm. Its main performance degradations are due to increases in aperiodic load. Even then, its performance degrades more slowly than the other algorithms because it guarantees that the distribution of service opportunities for aperiodics is optimal.

The largest performance gains of the optimal algorithm occur at high periodic loads, with aperiodic loads reaching their upper limit as imposed by the total resource utilization. On the contrary, the most strenuous condition for the optimal algorithm, as well as for all the algorithms, is a high aperiodic load superimposed on a relatively low periodic load. In addition, the aperiodic response times observed for all algorithms was increased by an increase in aperiodic execution times.

The performance of the optimal algorithm approaches the ideal M/M/1 bound within a 10% error margin for a relatively large range of total loads. In our simulation studies, performance deviations from M/M/1 occurred at a minimum total load of 65% in Figure 4.b and a maximum total load of 98% in Figure 8. In most cases, optimal performance tends to deviate significantly from the ideal when the aperiodic load is high relative to the periodic load. Under these conditions, preemption delays due to the periodic tasks tend to dominate the aperiodic response times. Since this situation is not likely to occur at high periodic loads, the performance of the optimal algorithm remains close to the M/M/1 bound for a wide range of operating conditions.

6 Implementation Issues

An implementation of the slack stealing algorithm requires one level-i inactivity counter per priority level ($\mathcal{I}_i$), one counter for the cumulative aperiodic processing time ($\mathcal{A}$), one completion status flag per task, where the status can be *active*, if the task has a pending job, or *inactive*, if it has no pending job, and a two-dimensional array of $A_i(t)$ values, as described in section 3.2. These requirements give a total of: m level-i inactivity counters, where $1 \leq i \leq m$ and $m \leq n$, (n is the number of periodic tasks), 1 counter for $\mathcal{A}$, n task completion status flags, and a table of size $x \times n$, where $x = max\{x_1, x_2, \ldots, x_n\}$ and x_i is the number of jobs of τ_i in a hyperperiod. Note that $x = x_1$, if the tasks are in deadline monotonic order. Clearly, the largest implementation overhead of the slack stealer is this table, which could be very large for task sets with long hyperperiods. To reduce this overhead, we have developed an approximate algorithm which provides near optimal performance and requires that only n values be stored for the table. This algorithm is the topic of a subsequent paper.

7 Summary

This paper focuses on the problem of jointly scheduling hard deadline periodic tasks and soft aperiodic tasks. The periodic tasks are assumed to be scheduled according to some fixed priority assignment scheme and the aperiodic tasks are allowed to be serviced any time, subject to the condition that all periodic deadlines be met. We develop a new aperiodic scheduling algorithm called the *slack stealer*, which is proved to be optimal in the sense that it guarantees aperiodic response times are minimized over all aperiodic service methods. We conducted simulation studies to evaluate the potential performance gains of the slack stealer over conventional aperiodic service methods such as background, polling, and the sporadic server. Results indicate that in some circumstances, the sporadic server can attain near optimal performance while in other circumstances all conventional aperiodic service methods are significantly outperformed by the slack stealer. Moreover, it is shown that the aperiodic response times that the slack stealer yields are very close to those predicted by a queueing model, under a wide range of conditions. In addition to its performance advantages, the slack stealer can be extended to service hard deadline aperiodic tasks and to reclaim processing time unused by the periodic tasks when they require less than their worst-case execution times. Although the implementation requirements for the slack stealer can be high in some cases, it provides the basis for approximate algorithms which can be implemented efficiently.

References

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

[2] S. Ramos-Thuel and J.K. Strosnider. The Transient Server Approach to Scheduling Time-Critical Recovery Operations. In *RTSS*, pages 286–295, 1991.

[3] B. Sprunt, L. Sha and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[4] B. Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon Univ., 1990.

[5] J.K. Strosnider. *Highly Responsive Real-Time Token Rings*. PhD thesis, Carnegie Mellon Univ., 1988.

[6] J.P. Hayes. *Computer Architecture and Organization*, Chapter 6. McGraw-Hill, 1988.

[7] J.P. Lehoczky, L. Sha and Y. Ding. The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *RTSS*, pages 166–171, 1989.

[8] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *RTSS*, pages 201–209, 1990.

[9] J.Y.-T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[10] M. Woodbury. Analysis of the Execution Time of Real-Time Tasks. In *RTSS*, pages 89–96, 1986.

[11] M.H. Woodbury and K.G. Shin. Evaluation of the Probability of Dynamic Failure and Processor Utilization for Real-Time Systems. In *RTSS*, 1988.

[12] C. Shen, K. Ramamritham and J. Stankovic. Resource Reclaiming in Real-Time. In *RTSS*, pages 41–50, 1990.

[13] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner and J. Reisinger. Tolerating Transient Faults in MARS. In *FTCS*, pages 466–473, 1990.

[14] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on SW Eng.*, 15(10):466–473, 1989.

[15] L. Schrage. A Proof of the Shortest Remaining Processing Time Discipline. *Operations Research*, 16:687–690, 1968.

[16] K. Fowler. Inertial Navigation System Simulator: Top-level Design. Technical Report CMU/SEI-89-TR-38, Software Engineering Institute, 1989.