A Reservation-Based Algorithm for Scheduling Both Periodic and Aperiodic Real-Time Tasks

Kang G. Shin, *Fellow, IEEE*, and Yi-Chieh Chang, *Member, IEEE*

***Abstract*—This paper considers the problem of scheduling both periodic and aperiodic tasks in real-time systems. A new, called *reservation-based* (RB), algorithm is proposed for ordering the execution of real-time tasks. This algorithm can guarantee all periodic-task deadlines while minimizing the probability of missing aperiodic-task deadlines. Periodic tasks are scheduled according to the rate monotonic priority algorithm (RMPA), and aperiodic tasks are scheduled by utilizing the processor time left unused by periodic tasks in each *unit cycle*. The length, *u*, of a unit cycle is defined as the greatest common divisor of all task periods, and a task is assumed to be invoked at the beginning of a unit cycle. For a set *S* of periodic tasks, the RB algorithm reserves a fraction R_S of processor time in each unit cycle for executing aperiodic tasks without missing any periodic-task deadline. The probability of meeting aperiodic-task deadlines is proved to be a monotonic increasing function of R_S. We derive the value of R_S that maximizes the processor time reservable for the execution of aperiodic tasks without missing any periodic-task deadline. We also show that if the ratio of the computation time to the deadline of each aperiodic task is bounded by R_S, the RB algorithm can meet the deadlines of all periodic and aperiodic tasks. Our analysis and simulation results show that the RB algorithm outperforms all other scheduling algorithms in meeting aperiodic-task deadlines.**

***Index Terms*—Real-time systems, task scheduling, periodic and aperiodic tasks, deadline guarantees.**

I. Introduction

Scheduling *both* periodic and aperiodic tasks in real-time systems is a much more difficult problem than scheduling periodic or aperiodic tasks alone [1], [2], [3], [4]. Two common approaches to servicing aperiodic tasks are polling and background processing. In the polling approach, a periodic *polling* task is invoked at regular intervals and services any pending aperiodic task. If there are no aperiodic tasks pending, then the polling task will be suspended until its next period. Since aperiodic-task arrivals are not coordinated with the invocation of a polling task, an aperiodic task may suffer a long delay if it arrives right after the polling task is suspended. In the background processing approach, each aperiodic task is serviced as a background process whenever the processor is idle, but giving low priority to the background process makes the response time of aperiodic tasks neither predictable nor guaranteeable.

To improve the response time of aperiodic tasks while guaranteeing periodic-task deadlines, Lehoczky et al. proposed the priority exchange (PE) and deferrable server (DS) algorithms [2]. In the PE algorithm, a high-priority periodic task is assigned to service aperiodic tasks. To improve the processor utilization, the aperiodic task server can exchange its priority with lower-priority periodic tasks whenever there are no aperiodic tasks to be serviced. In the DS algorithm, the aperiodic-task server will retain its allocated processor time even when there are no pending aperiodic tasks, so it is also referred to as a *bandwidth preserving* algorithm. The authors of [2] showed that both algorithms can improve the average wait and response time of aperiodic tasks over the polling and background process approaches. Sprunt et al. proposed the extended priority exchange (EPE) algorithm based on the PE and DS algorithms to improve aperiodic task response times when the worst-case periodic load is high and little or no unused processor time is left for the aperiodic task server [3].

Based on the concepts similar to the PE and EPE algorithms, Lehoczky and Ramos-Thuel [5] proposed an optimal algorithm for scheduling soft aperiodic tasks in fixed-priority preemptive systems. Their approach, called the *slack stealing* algorithm, does not create any periodic server to service aperiodic tasks. It instead creates a passive task, called the *slack stealer*, which, when prompted for service, attempts to make time for servicing aperiodic tasks by "stealing" all the available processing time from periodic tasks without missing any periodic-task deadline. They then proved that the slack stealing algorithm is optimal in the sense that all available processing time will be exploited for servicing aperiodic tasks while guaranteeing periodic-task deadlines. However, in order to exploit all processing time left unused by a given set of periodic tasks, the slack stealing algorithm uses a trial and error method to find the solution iteratively, and such an algorithm will consume a significant amount of time to find an optimal solution.

One difficulty in scheduling aperiodic tasks (in the presence of periodic tasks) is the lack of prior knowledge of their arrival times and deadlines. One can guarantee aperiodic tasks by treating them as periodic tasks with their minimum interarrival time being equal to their period. However, such a solution will severely under-utilize processor cycles, because the minimum interarrival time is usually much smaller than the corresponding average value. The problem of under-utilizing processor cycles can be eased somewhat by requiring any two aperiodic tasks to be separated by a prespecified minimum *p*. Mok [1] showed that aperiodic tasks can be guaranteed if *p* is large enough, but did not discuss the case when this condition does not hold.

Manuscript received Mar. 11, 1993; revised Sept. 3, 1994.

K.G. Shin is with Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122. e-mail: kgshin@eecs.umich.edu.

Y.-C. Chang is with Quickturn Design Systems, Inc., Mountain View, Calif.

To order reprints of this article, e-mail: transactions@computer.org, and reference IEEECS Log Number C95136.

The main goal of this paper is to propose a new, called *reservation-based* (RB), algorithm that can guarantee all periodic-task deadlines while minimizing the probability of missing aperiodic-task deadlines. Moreover, an upper bound of the ratio of the execution time to the deadline of each aperiodic task is derived and used to guarantee both periodic and aperiodic task deadlines. Under the RB algorithm, periodic tasks are scheduled according to the rate monotonic priority algorithm (RMPA) [6]. Aperiodic tasks are assumed to have lower priority than periodic tasks and are scheduled by utilizing the processor time available after scheduling periodic tasks in each *unit cycle*. The length, u, of a unit cycle is defined as the greatest common divisor of all task periods [7]. The processor utilization, $U(i)$, in unit cycle i by a given set of periodic tasks is calculated by dividing the processor time used in that unit cycle by u. For a given set S of periodic tasks, the RB algorithm reserves a fraction R_S of each unit cycle for aperiodic tasks without missing any periodic-task deadline. The key feature of the RB algorithm is that at least a fraction R_S of each unit cycle is "reserved"—without missing any periodic-task deadline—for aperiodic tasks, such that most, if not all, of aperiodic tasks can still be guaranteed to complete in time even if they are given lower priority than periodic tasks. If there are no aperiodic tasks to be serviced, then R_S can be set to zero, thus degenerating the RB algorithm to the original RMPA. The value of R_S is found to greatly influence the probability of meeting aperiodic-task deadlines even when the average processor utilization by periodic tasks is fixed. For example, let the period and computation time of periodic task τ_1 be 3 and 1.5 unit cycles, respectively. This task can be scheduled by either allocating 1.0 and 0.5 in the first and second unit cycles, or 0.5 in each unit cycle. However, if an aperiodic task v_1 with computation time of 1.0 unit cycle and deadline of 2 unit cycles arrives after starting the execution of τ_1, then the first scheduling scheme can only allocate 0.5 unit cycle for v_1, thus missing its deadline. By contrast, the second scheduling scheme will be able to allocate 1.0 unit cycle to v_1, thus meeting its deadline. One of the most important issues in the RB algorithm is, therefore, to derive the relation between R_S and the probability of guaranteeing aperiodic tasks. Since the probability of meeting the deadlines of aperiodic tasks is a monotonic increasing function of R_S, we will derive the least upper bound of R_S, denoted as R_{lub}, without missing any periodic-task deadline in a given task set S. R_S also influences the processor utilization achievable while meeting the deadlines of all tasks.

The RB algorithm increases the number of aperiodic tasks that can be completed in time while guaranteeing all periodic tasks. Since each periodic task is assumed to be invoked at the beginning of a unit cycle[1] and may have different release times and periods, there may be more tasks invoked in some time intervals than others. For example, $U(i)$ in Fig. 1 varies widely: the processor utilization by periodic tasks is 100% from unit cycle 15 to 18, while it is only 50% from unit cycle 25 to 28. The large variation of $U(i)$ is not desirable in any real-time system, because the probability of meeting an aperiodic-task deadline depends not only on its deadline but also on its arrival time. Since the variation of $U(i)$ results from the RMPA, an alternative scheme must be used to reduce the variation of $U(i)$. Surprisingly, even dynamic-priority scheduling algorithms, such as the earliest-deadline-first (EDF) algorithm, are not suitable for scheduling *both* periodic and aperiodic tasks. (More on this will be discussed in Section V.) Under the RMPA, a low-priority task cannot preempt a high-priority task. Since aperiodic tasks are given the lowest priority and hence cannot preempt any of periodic tasks, their deadlines cannot be guaranteed by the EDF algorithm. On the other hand, if aperiodic tasks are assigned higher priority than periodic tasks, some of the periodic tasks may miss their deadlines as a result of preemption by higher-priority aperiodic tasks; this may cause more deadline misses of the subsequent invocations of periodic tasks. By contrast, the RB algorithm reduces the variation of $U(i)$ in *each* unit cycle and makes more processor cycles available for the execution of aperiodic tasks. By reserving a fraction, $R_S \le R_{lub}$, of each unit cycle for a given set S of periodic tasks, one can increase the chance of meeting the deadlines of aperiodic tasks without missing any periodic-task deadline.

The RB algorithm differs from the PE, DS, or EPE algorithms in that it does not create any periodic server to handle aperiodic tasks and its main objective is to maximize the probability of guaranteeing aperiodic tasks without missing any periodic-task deadline. The RB algorithm may allocate the same amount of processing time as the slack stealing algorithm to service aperiodic tasks, but it is much simpler than the slack stealing algorithm. Section V presents a detailed comparison between the RB algorithm and other related approaches.

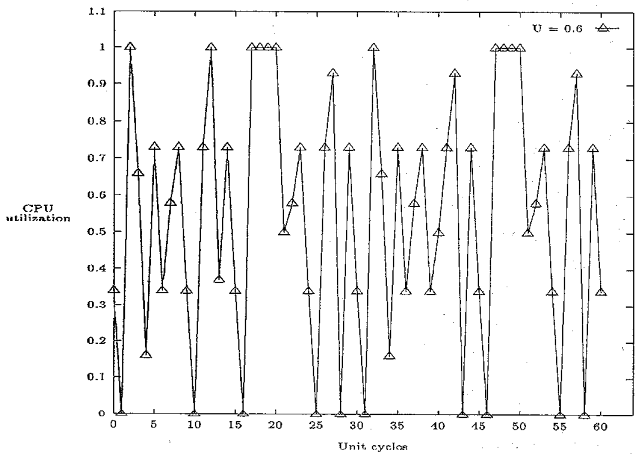


Fig. 1. Processor utilization with $U = 0.6$ and $R_S = 0.0$.

The rest of this paper is organized as follows. Section II states the problem and reviews some related results of [6]. Section III presents the RB algorithm and an analytic model for its performance evaluation. The performance of the RB algorithm is evaluated via both simulations and analytic modeling in Section IV. We derive in Section V an upper bound of ratio of the execution time to the deadline of an aperiodic task with which all of periodic and aperiodic tasks can be guaranteed. The paper concludes with Section VI.

1. Instead of the beginning of the corresponding task period, a commonly-used assumption.

II. Reservation of a Fraction of Processor Time

We want to guarantee all periodic tasks by using the RMPA while maximizing the probability of meeting aperiodic-task deadlines. For convenience, some of the basic definitions in [6] are re-introduced before presenting the RB algorithm.

Let aperiodic task v_i be represented by a three tuple (a_i, C_i, D_i), where a_i is its release/arrival time, C_i its execution time, and D_i its deadline relative to a_i, all measured in unit cycles (C_i can be a fraction of a unit cycle but D_i is a multiple of unit cycles). If there are more than one aperiodic task to be executed, they are scheduled according to the FCFS policy. A periodic task τ_i is assumed to be released at the beginning[2] of a unit cycle with a deadline equal to one task period after its release. Since the period and the first release time of a periodic task may be different from those of other periodic tasks, there may be more task releases during some time intervals than others. So, we represent τ_i with a three tuple (r_i, C_i, T_i), where r_i is the *first* release time of τ_i, C_i its execution time, and T_i its period. The *j*th invocation of τ_i is released at $r_i + (j-1)T_i$ and ends at $r_i + jT_i$. Thus, T_i is measured from the beginning $(r_i + (j-1)T_i)$ to the end $(r_i + jT_i)$ of the *j*th invocation. Let $S = \{\tau_1, \tau_2, \ldots, \tau_m\}$ denote a set of m periodic tasks. All periodic tasks are assumed to be known *a priori* to the designer and the priority of a periodic task is determined by its period; the shorter the period the higher its priority. Although every periodic task is assumed to arrive at the beginning of a unit cycle, its computation time does not have to be aligned with unit-cycle boundaries. All periodic and aperiodic tasks can be preempted at any time. Since real-time tasks are usually stored in main memory before putting the system in operation, the time to switch between tasks is assumed negligible. Also, note that the computation time of both periodic and aperiodic tasks is the time the processor needs to execute the task with 100% devotion to it. Obviously, if the processor is interrupted before completing a task, the total time to complete the task will be greater than its computation time.

Since periodic tasks are known in advance, they are scheduled according to the RMPA. Using the same notation in [6], the processor utilization, U, by S is

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i}. \qquad (1)$$

The *major cycle*, T_{mc}, for S is the least common multiple of all task periods in S measured in number of unit cycles. For example, the *i*th major cycle starts at $t = (i-1)T_{mc}$ and ends at $t = iT_{mc}$, $i \geq 1$. Equation (1) gives the average processor utilization over one major cycle by the periodic tasks in S. Let $s_i = [(i-1)u, iu)$ denote the *i*th unit cycle within a major cycle. The processor utilization in s_i, denoted by $U(i)$, is calculated by dividing the processor busy time in s_i by u. So, the unused (used) processor time in s_i is $u(1 - U(i))$ $(uU(i))$. From (1) and the definition of $U(i)$, we get

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} = \frac{\sum_{i=1}^{T_{mc}/u} U(i)u}{T_{mc}}.$$

Because whether the deadline of an aperiodic task can be met or not depends on its arrival time,[3] it is not sufficient to analyze the probability of meeting aperiodic-task deadlines by using U alone. For example, consider the average processor utilization over N unit cycles for different time intervals in Fig. 2 while changing N from 2 to 20. Although $U = 0.6$, the average processor utilization over a small N, such as $N < 6$, can be as high as 100% or as low as 1%. Thus, the probability of guaranteeing the aperiodic tasks with deadlines < 6 unit cycles will greatly depend on their arrival times. The difference between the maximum and minimum average processor utilization during N unit cycles reduces to within 30% of each other when $N > 13$.

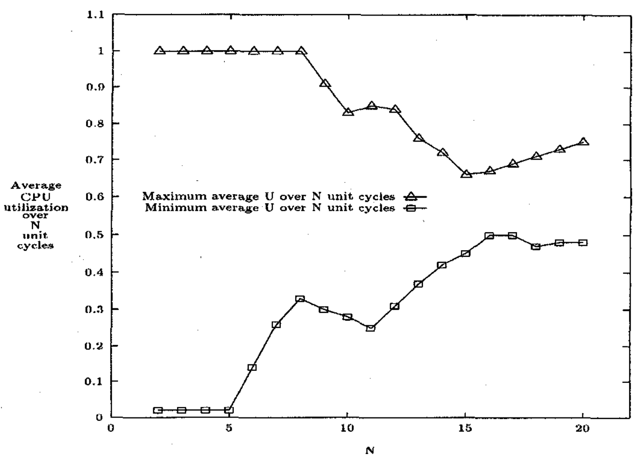


Fig. 2. Variation of average processor utilization during N unit cycles ($U = 0.6$, $R_S = 0.0$).

As mentioned earlier, the fraction of processor time reserved in each unit cycle will greatly affect the probability of meeting aperiodic-task deadlines even when U is fixed. The importance of reserving a fraction of processor time to meet aperiodic-task deadlines can be seen from the example in Fig. 3 (assuming $u = 1.0$). Let $\tau_1 = (0, 1.5, 3)$ and $\tau_2 = (0, 0.5, 5)$. As shown in Fig. 3, for an aperiodic task $v_3 = (0, 0.6, 2)$ only the third scheduling scheme can allocate 0.6 to complete v_3 before its deadline, while all three scheduling schemes guarantee the deadlines of τ_1 and τ_2. However, if the deadline of v_3 is 3 instead of 2, it will be guaranteed by all three scheduling schemes. In this case, even if the execution time of v_3 is increased to 1.0, it will still be guaranteed by all three scheduling schemes. After reserving a fraction, R_S, of the processor time, the fraction of processor time available in each unit cycle after scheduling all periodic tasks will be greater than, or equal to, R_S.

2. This is more general than the usual assumption that all periodic tasks are released at the beginning of their respective periods.

3. And other parameters like its execution time and deadline.

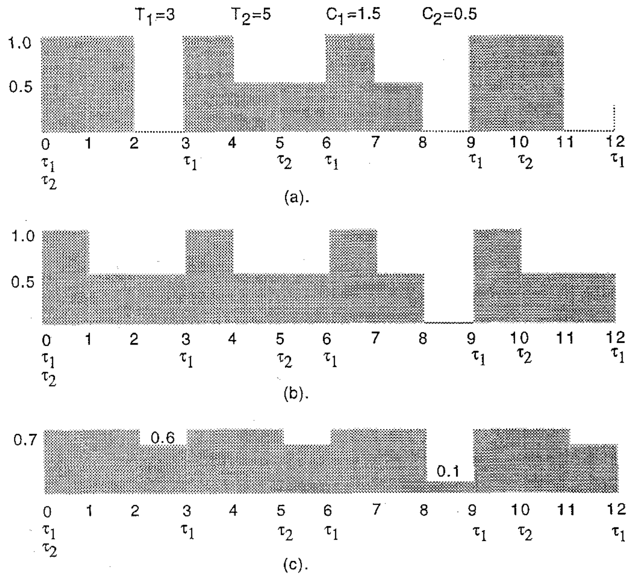


Fig. 3. Three scheduling schemes with different reservation fractions.

Consider the previous example shown in Fig. 3, where no time is reserved in s_1 and s_2, while 0.5 unit cycle is reserved in s_2 in Fig. 3b, and 0.3 unit cycle is reserved in *each* unit cycle in Fig. 3c. The shaded box in Fig. 3 indicates the fraction of the time the processor is busy. The first two schemes cannot guarantee v_3 = {3, 0.6, 2}, but the third scheme can allocate 0.6 unit cycle for the third aperiodic task and meet its deadline. Note that reserving a fraction of each unit cycle does not always guarantee aperiodic tasks. For example, the first and second scheduling schemes can guarantee an aperiodic task $v_4 = (1, 1.0, 2)$, but not the third scheme even though $R_S = 0.3$.

There is also an upper bound of R in order to guarantee the deadlines of all periodic tasks in S. To see this, consider the third scheduling scheme in Fig. 3c; if $R_S = 0.4$ from s_1 to s_3, then only 30% of a unit cycle can be allocated to τ_2 before its next invocation, thus missing the deadline. When U is fixed, the value of R_S will affect the probability of guaranteeing aperiodic tasks with short deadlines and may not affect the tasks with long deadlines. For example, consider the average processor utilization over $2 \le N \le 20$ unit cycles with $R_S = 0.3$, the maximum and minimum $U(i)$ are found to be 70% and 1%, respectively, for $N < 6$, while they were 100% and 1%, respectively, for the example in Fig. 2. However, the difference between the maximum and minimum $U(i)$ is about the same in Fig. 2 for $N > 12$. Since reserving a fraction of each unit cycle does not change U, the value of R_S has less effects on the fraction of time that can be allocated to aperiodic tasks with long deadlines.

Recall that in the RB algorithm, periodic tasks are scheduled according to the RMPA. If there are no aperiodic task arrivals, there is no need to reserve any fraction of time in each unit cycle. However, upon arrival of an aperiodic task at a node, the node will check its available processor time based on the RB algorithm. If the task can be completed in time by the node, it will be scheduled locally; otherwise, this task will be rejected, or transferred to some other node if the task can be guaranteed in that node. (Incorporation of RB scheduling into load sharing is an interesting problem that warrants further investigation. See more on this in Section VI.) Upon completion of an aperiodic task, periodic tasks will be scheduled according to the RMPA if there are no more aperiodic tasks waiting in the queue; otherwise, the RB algorithm will be used to process the remaining aperiodic tasks.

Under the RB algorithm, $U(i) \le 1 - R_S$ for all i after scheduling periodic tasks according to the RMPA. To analyze the performance of the RB algorithm, we must know the probability distribution of aperiodic-task deadlines. The probability of guaranteeing an aperiodic task $v_i = (a_i, C_i, D_i)$ is the probability of allocating processor time $\ge C_i$ after a_i but before or on $a_i + D_i - C_i$. Thus, the deadline distribution of aperiodic tasks is a main factor that determines the performance of the RB algorithm. To implement the RB algorithm, one must consider:

- Derivation of the processor utilization in each unit cycle after scheduling a given set S of periodic tasks, and the relation between R_S and the probability of meeting aperiodic-task deadlines.
- Determination of the value of R_S that maximizes the probability of meeting aperiodic-task deadlines without missing any periodic-task deadline.
- The distribution of aperiodic-task deadlines for performance analysis.

All of these will be discussed in detail in the following sections.

III. THE RB ALGORITHM AND ITS PERFORMANCE MODEL

In this section, we will first introduce the RB algorithm and then derive the processor utilization.

Recall that a periodic task $\tau_i = (r_i, C_i, T_i)$ is assumed to arrive at the beginning of a unit cycle, but C_i need not be aligned with unit cycle boundaries. Consider a periodic task set $S = \{\tau_1, \tau_2, \ldots, \tau_m\}$. Since by definition all periodic tasks must be released at least once in a major cycle, the first release time of the tasks in S ranges from 0 to $T_{mc} - 1$ (measured in unit cycles) and $U(i)$ will repeat the same pattern after the second major cycle, i.e., $U(i + T_{mc}) = U(i + nT_{mc})$ for all $n > 1$. Moreover, the authors of [8] and [9] proved that even if some of periodic tasks are released after the second major cycle, we only need to consider the first two major cycles with a scheduling algorithm which does not leave any processor idle as long as there are tasks ready for execution. Note that $U(i)$ in the first major cycle may not be equal to $U(i + T_{mc})$ for some i, because the periodic tasks with $r_i > i$ may not have been released at the corresponding time in the first major cycle. Zhu et al. [9] proved that by creating imaginative task instances in the first major cycle it is sufficient to consider the first two major cycles for the derivation of $U(i)$.

The m periodic tasks in S are sorted in ascending order of their periods, such that $T_1 < T_2 < \ldots < T_m$. In the RB algorithm, the initial value of $U(i)$ is set to R_S for $i = 0, 1, \ldots, 2T_{mc}$. After scheduling all periodic tasks, R_S is subtracted from $U(i)$. Since at most $1 - R_S$ fraction of time in each unit cycle can be allocated to periodic tasks, R_S is the minimum available fraction of time in each unit cycle after scheduling all periodic tasks in S. If the current invocation of a task cannot be completed before its next invocation, the task set is unschedulable and the RB algorithm terminates.

```
RB Algorithm

for $i = 0$ to $2T_{mc}$ $U(i) := R_S$;
for $i = 1$ to $|S|$ do
    for $s_i = r_i$ to $2T_{mc}$ do
        schedule_time := $C_i$;
        $s' := s_i$;
        while schedule_time > 0 do
            $F(s') := (1 - U(s'))u$;
            if schedule_time $\le F(s')$ then
                $U(s') := U(s')$ + schedule_time/$u$;
                schedule_time := 0;
            else
                if $F(s') > 0$ then
                    schedule_time := schedule_time $- F(s')$
                    $U(s') := U(s') + F(s')/u$
                end_if
                $s' := s' + u$;
                if $s' \ge s_i + T_i$ then stop (the task set is unschedulable);
            end_if
        end_do
        $s_i := s_i + T_i$
    end_do
end_do
for $i = 0$ to $2T_{mc}$ $U(i) := U(i) - R_S$;
```

The RB algorithm schedules the periodic tasks in S according to the RMPA while reserving a fraction R_S of each unit cycle for aperiodic tasks. In the RMPA, the task with the shortest period, τ_1, is scheduled first, starting from its first release time r_1. The scheduled time for τ_1 ("schedule_time") is first set to C_1 and then the available processor time at r_1 (denoted as $F(s')$) is compared with τ_1's scheduled time. If $F(s') \geq$ schedule_time, the schedule_time will be added to $uU(s')$ and τ_1 is scheduled; otherwise, $F(s')$ is subtracted from schedule_time and s' is incremented by one. The above procedure continues until schedule_time becomes 0.

The complexity of the RB algorithm can be easily analyzed as follows. In each major cycle, τ_i will be invoked T_{mc}/T_i times and for each invocation of τ_i we need to adjust $U(i)$ for the period of at most T_i unit cycles (because τ_i must be completed within one period after its release). Thus, the maximum total number of unit cycles during which the utilization needs to be adjusted for scheduling τ_i is simply $T_{mc}/T_i \times T_i = T_{mc}$. As a result, the complexity of the RB algorithm for a set of m tasks is mT_{mc} or $O(m)$ (as T_{mc} is a finite constant for any given task set). In Section III.C we will derive the optimal value of R_S thereby eliminating the need to try many different values of R_S in order to achieve the best performance. This is in sharp contrast to the EPE or slack stealing algorithms where one must try many different values of R_S before finding the optimal utilization by aperiodic tasks.

Note that T_{mc} is dependent on the periods of tasks in S and might grow exponentially if the periods are not harmonic. Han and Lin [10] have shown that by adding a distance constraint to the periodic tasks, it is possible to *specialize* a task set so that the task set may be schedulable when the *density* of the specialized task set is less than 1. They have also shown that the specialized task set contains solely multiples of a period, and hence, T_{mc} will not increase exponentially with the periods of the periodic task set.

A. Processor Utilization by a Set S of Periodic Tasks

In order to derive the relation between R_S and the probability of meeting aperiodic-task deadlines, we need to derive $U(i)$. From [6], if $U \le m(2^{1/m} - 1)$ for a set S of m periodic tasks, S is guaranteed to be *schedulable*, but on average a task set is schedulable for up to 88% utilization [11]. Suppose S is schedulable and let $\tau_1, \ldots, \tau_m$ be the tasks in S sorted in ascending order of their periods. After τ_1 is scheduled, $U(i)$ can be calculated as follows. If $C_1 < u(1 - R_S)$ then

$$U(r_i + nT_1) = C_1/u, \quad n = 0, 1, 2, \ldots$$

If $C_1 > u(1 - R_S)$, then

$$\begin{aligned}
&U(i + nT_1) = 1 - R_S,\\
&i = r_1, r_1 + 1, \ldots, r_1 + \left\lfloor \frac{C_1}{u(1 - R_S)} \right\rfloor - 1\\
&U\left(r_1 + \left\lfloor \frac{C_1}{u(1 - R_S)} \right\rfloor + nT_1\right)\\
&= \frac{C_1 - \left\lfloor \frac{C_1}{u(1 - R_S)} \right\rfloor \times (1 - R_S)}{u}, \quad n = 0, 1, 2, \ldots.
\end{aligned}$$

Similarly, $U(i)$ can be calculated after scheduling τ_j, $j = 2, \ldots, k$. Consider the nth invocation of τ_j. Since S is schedulable, there must be a sufficient processing time available for executing τ_j during the period from unit cycle $r_j + (n - 1)\, T_j$ to unit cycle $r_j + n\, T_j$. Let ℓ_n be the number of unit cycles needed to execute τ_j at its nth invocation and $F(i)$ be the available processor time in unit cycle i. For convenience, let A_j be the total available processor time during the period from unit cycle $r_j + (n - 1)\, T_j$ to unit cycle $r_j + (n - 1)\, T_j + \ell_n$, and A_j' be the total available processor time during the period from unit cycle $r_j + (n - 1)\, T_j$ to unit cycle $r_j + (n - 1)\, T_j + \ell_n - 1$. Then,

$$A_j = \sum_{i=r_j+(n-1)T_j}^{r_j+(n-1)T_j+l_n} F(i) \text{ and } A_j' = \sum_{i=r_j+(n-1)T_j}^{r_j+(n-1)T_j+l_n-1} F(i),\ n = 0, 1, \ldots, \infty.$$

If $C_j/u < 1 - U(r_j + nT_j)$ then $U(r_i + nT_j) = U(i + nT_j) + C_j/u$, $n = 0, 1, \ldots$.

If $C_j/u > 1 - U(r_j + nT_j)$ then

$$\begin{aligned}
&U(i + nT_j) = 1 - R_S,\\
&i = r_j, \ldots, r_j + l_n - 1\\
&U(r_j + l_n + nT_j)\\
&= U(r_j + l_n + nT_j) + \frac{A_j - A_j'}{u}, \quad n = 0, 1, \ldots.
\end{aligned} \tag{2}$$

The calculated values of $U(i)$ for a task set with $U = 0.6$ and

$R_S = 0.3$ is plotted in Fig. 4. From these figures, it is found that over many unit cycles, the processor is under-utilized or unused even though the average processor utilization (U) is around 60%. This observation justifies the need of reserving a certain fraction of each unit cycle to improve the overall processor utilization. Another interesting result found in the RB algorithm is that the variation of $U(i)$ decreases as R_S increases, thus making $U(i)$ closer to U. As a result, the probability of meeting aperiodic-task deadlines will depend *less* on their arrival time (see Fig. 4). Recall that when no processor time was reserved, $U(i)$ was as high as 100% in some unit cycles, and thus, the probability of guaranteeing aperiodic tasks depended heavily on their deadlines as well as on their arrival times (see Figs. 1 and 2).

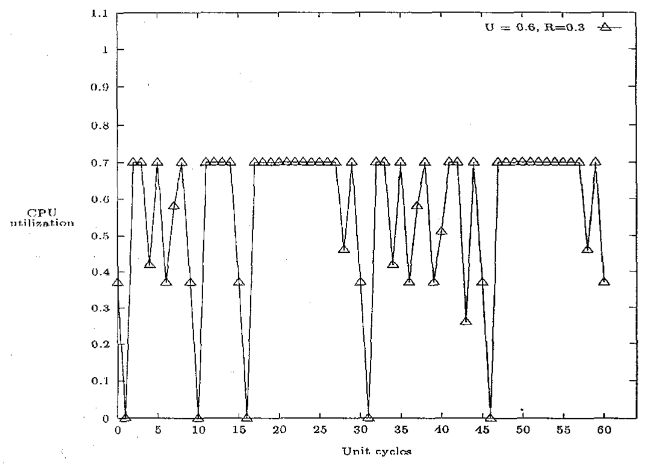


Fig. 4. Processor utilization with $U = 0.6$ and $R_S = 0.3$.

B. Probability of Completing an Aperiodic Task in Time

The probability of meeting the deadline of an aperiodic task $v_\alpha = (a_\alpha, C_\alpha, D_\alpha)$ is the probability that C_α is less than, or equal to, the total available processor time during the interval $[a_\alpha, a_\alpha + D_\alpha]$. For convenience of discussion, C_α is assumed to be exponentially distributed with mean μ. (A similar argument can be made when C_α has a different distribution.) Then,

$$\begin{aligned} P\left(C_\alpha \leq \sum_{i=a_\alpha}^{a_\alpha+D_\alpha} F(i)\right) &= P\left(C_\alpha \leq \overline{F}\, D_\alpha\right) \\ &= \sum_{D=D_{\min}}^{D_{\max}} P(D) \int_0^{\overline{F}D} \mu e^{-\mu t} dt \qquad (3) \\ &= \sum_{D=D_{\min}}^{D_{\max}} P(D)\left(1 - e^{-\mu \overline{F} D}\right), \end{aligned}$$

where

$$\overline{F} = \sum_{i=a_\alpha}^{a_\alpha+D_\alpha} F(i) / D_\alpha.$$

$\overline{F}$ is the average available processor time in the interval $[a_\alpha, a_\alpha + D_\alpha]$ for scheduling only aperiodic tasks. D_{min} (D_{max}) is the minimum (maximum) task deadline (more on the deadline distribution will be discussed in Section IV). $P(D)$ is the probability density function of task deadlines.

We are interested in deriving a condition under which $P\left(C_\alpha \leq \overline{F}\, D_\alpha\right)$ is maximized. From (3), it is easy to see that the probability of meeting aperiodic-task deadlines is a monotonic increasing function of $\overline{F}$. When U is fixed, reserving a fraction of each unit cycle will increase the utilization (or decrease $\overline{F}$) in those unit cycles during which the processor was under-utilized or unused as shown in Fig. 4. Since the probability of completing an aperiodic task in time is increased (decreased) by increasing (decreasing) $\overline{F}$, the increase in the probability of guaranteeing aperiodic tasks in these unit cycles with an increased $\overline{F}$ must be greater than the decrease in the probability of guaranteeing aperiodic tasks in those unit cycles with a decreased $\overline{F}$.

Intuitively, the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R_S. However, this is not clear when a large increase of $\overline{F}$ in a time interval causes small decreases of $\overline{F}$ in several other time intervals. So, we have the following theorem.

THEOREM 1. *The probability of guaranteeing aperiodic tasks is a monotonic increasing function of R_S.*

PROOF. Let $\overline{F}_1$ be the average available fraction of time over a time interval, say I_1. If $\overline{F}_1 < R$, after reserving a fraction R_S of each unit cycle the average available fraction of time over I_1 becomes $\overline{F}_1' = R$. Since U is fixed over a major cycle, increasing the available fraction of processor time in I_1 will cause the available fraction of processor time to decrease over some other intervals in which the average available fractions of time were used to be greater than R_S (before the reservation). Let $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$ be those intervals and

$$\overline{F}_{j_1}\left(\overline{F}_{j_1}'\right), \overline{F}_{j_2}\left(\overline{F}_{j_2}'\right), \ \ldots, \ \overline{F}_{j_k}\left(\overline{F}_{j_k}'\right)$$

be the average available fractions of processor time over these intervals before (after) reserving the processor time as shown in Fig. 5. (Note that the distance between I_{j_k} and I_1 is larger for a larger k as shown in Fig. 5.) Then, we have $\overline{F}_1 + \Sigma_{l=1}^{k} \overline{F}_{j_l} = \overline{F}_1' + \Sigma_{l=1}^{k} \overline{F}_{j_l}'$. $\overline{F}_{j_l}' = \overline{F}_{j_l} - R$ if $\overline{F}_{j_l} \geq 2R$, and $\overline{F}_{j_l}' = R$ if $\overline{F}_{j_l} < 2R$ for $\ell = 1, \ldots, k$.

Note that the increase of $\overline{F}_1'$ in I_1 causes many decreases of the available fraction of processor time in intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$. Let $\Delta\overline{F}_{j_l} = \left|\overline{F}_{j_l} - \overline{F}_{j_l}'\right|$ for $\ell = 1, \ldots k$. Then, each of these $\Delta\overline{F}_{j_l}s$ will contribute to $\overline{F}_1'$. So, $\Delta\overline{F}_1 = \left|\overline{F}_1 - \overline{F}_1'\right| = \Sigma_{l=1}^{k} \Delta\overline{F}_{j_l}$. Also, note that the decreases of the available processor time over intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$ due to the increase of $\overline{F}_1'$ will occur first on I_{j_1}, then I_{j_2}, and so on, because in the RB algorithm the task's scheduled time is shifted to an adjacent unit cycle first. Thus, it is sufficient to consider the following three cases.

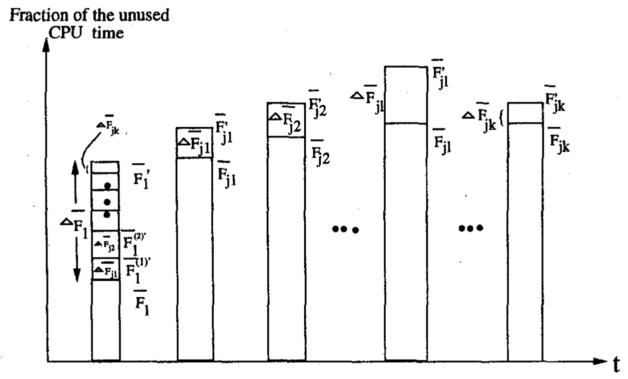


Fig. 5. The variation of the available processor time over some time intervals after reserving a fraction R_S of processor time in each unit cycle.

Case 1. Consider the variation of $\overline{F}_{j_1}$ in I_{j_1} first. For notational convenience, let $\overline{F}_1^{(1)'}$ be the available processor time in I_1 when only $\Delta\overline{F}_{j_1}$ is considered, i.e., assuming $\Delta\overline{F}_{j_l} = 0$ for $\ell = 2, \ldots, k$. We then have $\overline{F}_1^{(1)'} = \overline{F}_1 + \Delta\overline{F}_{j_1}$. The variation of the probability of guaranteeing an aperiodic task $v_\alpha = (a_\alpha, C_\alpha, D_\alpha)$ is computed by:

$$P_1\left(C_\alpha \le \overline{F}_1 D_\alpha\right) - P_1\left(C_\alpha \le \overline{F}_1^{(1)'} D_\alpha\right)$$

$$= P\left(D_\alpha\right)\left(e^{-\mu\overline{F}_1^{(1)'} D_\alpha} - e^{-\mu\overline{F}_1 D_\alpha}\right) < 0$$

$$P_2\left(C_\alpha \le \overline{F}_{j_1}' D_\alpha\right) - P_2\left(C_\alpha \le \overline{F}_{j_1} D_\alpha\right)$$

$$= P\left(D_\alpha\right)\left(e^{-\mu\overline{F}_{j_1}' D_\alpha} - e^{-\mu\overline{F}_{j_1} D_\alpha}\right) > 0.$$

Note that the negative result of the $P\left(D_\alpha\right)\left(e^{-\mu\overline{F}_1^{(1)'} D_\alpha} - e^{-\mu\overline{F}_1 D_\alpha}\right)$ indicates the probability of guaranteeing aperiodic tasks is increased after reserving R_S because $\overline{F}_1^{(1)'} > \overline{F}_1$. So, the increased probability of guaranteeing the task is $P\left(D_\alpha\right)\left(e^{-\mu\overline{F}_1 D_\alpha} - e^{-\mu\overline{F}_1^{(1)'} D_\alpha}\right)$ while the reduced probability of guaranteeing the task is $P\left(D_\alpha\right)\left(e^{-\mu\overline{F}_{j_1}' D_\alpha} - e^{-\mu\overline{F}_{j_1}^{(1)'} D_\alpha}\right)$. The net change is:

$$\begin{aligned}
\Delta P^{(1)} &= P\left(D_\alpha\right)\left[\left(e^{-\mu\overline{F}_1 D_\alpha} - e^{-\mu\overline{F}_1^{(1)'} D_\alpha}\right) + \left(e^{-\mu\overline{F}_{j_1} D_\alpha} - e^{-\mu\overline{F}_{j_1}' D_\alpha}\right)\right] \\
&= P\left(D_\alpha\right)\left[e^{-\mu\overline{F}_1 D_\alpha}\left(1 - e^{-\mu\overline{F}_{j_1} D_\alpha}\right) + e^{-\mu\overline{F}_{j_1}' D_\alpha}\left(e^{-\mu\Delta\overline{F}_{j_1} D_\alpha} - 1\right)\right] \\
&= P\left(D_\alpha\right)\left(1 - e^{-\mu\Delta\overline{F}_{j_1} D_\alpha}\right)\left(e^{-\mu\overline{F}_1 D_\alpha} - e^{-\mu\overline{F}_{j_1}' D_\alpha}\right).
\end{aligned}$$

From the initial condition we have $\overline{F}_1 < R$ and $\overline{F}_{j_1}' \ge R$, thus $\overline{F}_1 < \overline{F}_{j_1}'$.

So, $\Delta P^{(1)}(\textit{guarantee } v_\alpha) > 0$.

Case 2. Consider only the variation of the available processor time in intervals I_{j_1} and I_{j_2}. For notational convenience, let $\overline{F}_1^{(2)'} = \overline{F}_1^{(1)'} + \Delta\overline{F}_{j_2}$. The net change in the probability of guaranteeing v_α is

$$\Delta P^{(2)}(\text{guarantee}\, v_\alpha) =$$

$$P\left(D_\alpha\right)\left(1 - e^{-\mu\Delta\overline{F}_{j_2} D_\alpha}\right)\left(e^{-\mu\overline{F}_1^{(2)} D_\alpha} - e^{-\mu\overline{F}_{j_2}' D_\alpha}\right).$$

Since $\overline{F}_1^{(2)} < \overline{F}_{j_2}', \Delta P^{(2)}(\text{guarantee}\, v_\alpha) > 0$.

Case 3. Generally, consider the variations of the available processor time in intervals l_{j_1} to l_{j_l}, and let $\Delta\overline{F}_{j_l} = \overline{F}_{j_l} - \overline{F}_{j_l}'$ and $\overline{F}_1^{(l)'} = \overline{F}_1^{(l-1)'} + \Delta\overline{F}_{j_l}$. The net change in the probability of guaranteeing v_α is

$$\Delta P^{(l)}(\text{guarantee}\, v_\alpha) =$$

$$P\left(D_\alpha\right)\left(1 - e^{-\mu\Delta\overline{F}_{j_l} D_\alpha}\right)\left(1 - e^{-\mu\overline{F}_1^{(l)} D_\alpha} - e^{-\mu\overline{F}_{j_l}' D_\alpha}\right).$$

Since $\overline{F}_1^{(l)} < \overline{F}_{j_l}', \Delta P^{(l)}(\text{guarantee}\, v_\alpha) > 0$ for $\ell = 1, \ldots, k$.

As a result, the increase in the probability of guaranteeing aperiodic tasks by increasing $\overline{F}$ in I_1 will always be greater than the total reduced probability of guaranteeing aperiodic tasks in $I_{j_1}, \ldots, I_{j_k}$. Since $1 - e^{-\mu\Delta\overline{F}_{j_l}}$ is a monotonic increasing function of $\Delta\overline{F}_{j_l}$, the increase in the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R_S.

In practice, the decreases of the available processor time over some intervals may result from the increases of the available processor time over a multiple number of intervals instead of only one interval I_1 as discussed above. However, we can always identify the decreases of the available processor time over those intervals as a result of the increase of the available processor time in a particular interval. Applying the same argument, we can conclude that the increase in the probability of guaranteeing aperiodic tasks is a monotonic increasing function of R_S. □

C. Deriving the Least Upper Bound of R_S

Since (3) is a monotonic increasing function of R_S, we would like to derive its least upper bound R_{lub}, without missing *any* periodic-task deadline. Note that use of R_{lub} not only maximizes the probability of guaranteeing aperiodic tasks but also maximizes the processor utilization by both periodic and aperiodic tasks. Moreover, analytic derivation of R_{lub} is an important contribution to solving the problem of scheduling both periodic and aperiodic tasks for real-time systems, because the only method reported in the literature to find this bound is based on trial and error, i.e., gradually increasing R_S until the periodic task set becomes unschedulable. In the rest of this subsection, we will show how to derive R_{lub} systematically.

Consider the case of two periodic tasks, as shown in Fig. 6a, $\tau_1 = (C_1, T_1)$ and $\tau_2 = (C_2, T_2)$, and let $T_1 < T_2$ and $q_{2,1} = T_2 \bmod T_1$. As discussed in [6], the critical instant occurs when a task is released simultaneously with all other higher-priority tasks. So, the most difficult situation in scheduling τ_2 occurs when it is released at the same time as τ_1. It was shown in [6] that the maximum U occurs when $C_1 = T_2 - T_1 \lfloor T_2/T_1 \rfloor$, leading to

$$U_{\max} = \frac{\left\lceil \frac{T_2}{T_1} \right\rceil C_1 + C_2}{T_2}.$$

Thus,

$$R_S = 1 - U_{\max} = 1 - \frac{\left\lceil \frac{T_2}{T_1} \right\rceil C_1 + C_2}{T_2}.$$

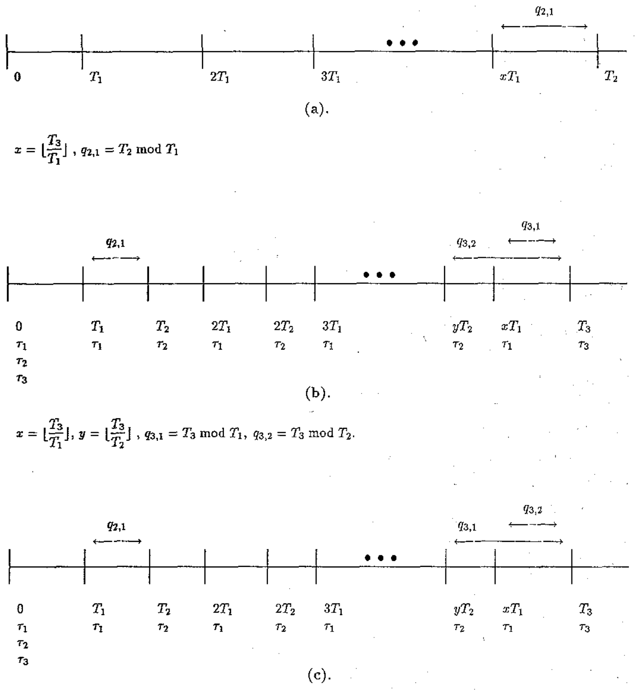
However, the R_S derived from the above formula is the most pessimistic bound, because when $q_{2,1} > 0$, τ_1 does not necessarily use all $\left\lceil \frac{T_2}{T_1} \right\rceil C_1$ in T_2 unit cycles. In fact, the last invocation of τ_1 before the second invocation of τ_2 can only use up to a fraction, $1 - R_{lub} = U_{max}$, of processor time if $C_1 > q_{2,1}\, U_{max}$. The following expressions can be used to calculate R_{lub} for the case of two periodic tasks.

$$U_{\max} = \begin{cases} \dfrac{\left\lceil \frac{T_2}{T_1} \right\rceil C_1 + C_2}{T_2} & C_1 < q_{2,1} U_{\max} \\[2ex] \dfrac{\left\lfloor \frac{T_2}{T_1} \right\rfloor C_1 + C_2 + q_{2,1} U_{\max}}{T_2} & C_1 > q_{2,1} U_{\max} \end{cases} \tag{4}$$

$$R_{lub} = 1 - U_{max}. \tag{5}$$

The R_{lub} derived from (5) is the least upper bound of R_S without missing any periodic-task deadline (because $U_{max} + R_{lub} = 1$). So, it is impossible to reserve a fraction, $R_S \geq R_{lub}$, of processor time at the critical instant. Also, both periodic tasks can be guaranteed because the U_{max} derived from (4) is the maximum utilization needed for these two tasks at the critical instant. Although it is possible to reserve R_S higher than R_{lub} at a non-critical instant, R_S must be reduced to R_{lub} when a critical instant occurs; otherwise, one of the periodic tasks will miss its deadline. Thus, R_{lub} is the maximum fraction of processor time that can be reserved for aperiodic tasks without missing any periodic-task deadline at *any* instant.

Since the calculation of U_{max} depends on itself, we must resort to an iterative approach to finding the final value of U_{max}. Initially, U_{max} is set to 1 and (4) is used to calculate a new U_{max}. Then, the newly-calculated U_{max} is used as the initial value of the next iteration. The difference between this value of U_{max} and its final value will be reduced by $1/T_2$ at each iteration. When $T_2 > 1$, U_{max} is shown to converge to its final value only in a few iterations. If $T_2 = 1$, $U_{max} = U = C_1/T_1 + C_2/T_2$, there is no need to use (4) to calculate U_{max}. The more iterations are used in the calculation of U_{max}, the closer the computed U_{max} will be to its final value, but the U_{max} calculated at each iteration will always be greater than its final value. Thus, $R_{lub} = 1 - U_{max}$ computed at any iteration can guarantee all periodic tasks. In fact, even if $T_2 = 2$, R_{lub} is shown to get close to 95% of its final value within four iterations.



$q_{2,1}$
0
T_1
$2T_1$
$3T_1$
xT_1
T_2
(a).
$x = \lfloor \frac{T_2}{T_1} \rfloor$, $q_{2,1} = T_2 \bmod T_1$
$q_{2,1}$
$q_{3,2}$
$q_{3,1}$
T_2
$2T_2$
yT_2
T_3
τ_1
τ_2
τ_3
(b).
$x = \lfloor \frac{T_3}{T_1} \rfloor$, $y = \lfloor \frac{T_3}{T_2} \rfloor$, $q_{3,1} = T_3 \bmod T_1$, $q_{3,2} = T_3 \bmod T_2$.
(c).

Fig. 6. Illustration of deriving R_{lub} for two and three periodic tasks.

For the case of three periodic tasks, the approach to finding R_{lub} is more complicated than the two-task case. As shown in Fig. 6b and Fig. 6c, the critical instant occurs at the second invocation of τ_2 or τ_3. If $\dfrac{\left\lceil \frac{T_2}{T_1} \right\rceil C_1 + C_2}{T_2} > U$, we need to consider the instant of the second invocation of τ_2; otherwise, we only need to consider the instant of the second invocation of τ_3. If both cases need to be considered, (4) and (5) can be used to find U_{max} at the first critical instant. The U_{max} at the second critical instant can be computed by deriving the exact amount of processor time needed for the xth invocation of τ_1 and yth invocation of τ_2, where $x = \left\lfloor \frac{T_3}{T_1} \right\rfloor$ and $y = \left\lfloor \frac{T_3}{T_2} \right\rfloor$. R_{lub} is then computed simply by subtracting the maximum utilization at the critical instant from 1.

At the second critical instant as shown in Fig. 6b and Fig. 6c, one has to find the exact utilization required by the xth invocation of τ_1 and yth invocation of τ_2 before the second invocation of τ_3. Obviously, if $C_1 < q_{3,1}\, U_{max}$, the total compu-

Algorithm 1

```
$U := \sum_{i=1}^{m} \frac{C_i}{T_i}$; $U_{max} := 1$; $U_{max}^{(1)} := 0$;
If $\frac{\lceil \frac{T_2}{T_1} \rceil C_1 + C_2}{T_2} > U$ then
    use Eqs. (4) and (5) to find $U_{max}$ at the second invocation of $\tau_2$;
    repeat the above four times;
    $U_{max}^{(1)} := U_{max}$;
end_if
for $k = 3$ to $m$ do
    if $\sum_{i=1}^{k} \frac{\lceil \frac{T_k}{T_i} \rceil}{C_i} > U$ then do
        for $j = 1$ to $k - 1$  $q_{k,i} := T_k \bmod T_i$;
        for $\ell = 1$ to 4 do
            $CET := \sum_{i=1}^{k} \lfloor \frac{T_k}{T_i} \rfloor C_i$;
            use Eqs. (4) and (5) to find the exact computation time of the $\lfloor \frac{T_k}{T_1} \rfloor^{th}$ invocation of $\tau_1$.
            $CET := CET +$ the exact computation time of the $\lfloor \frac{T_k}{T_1} \rfloor^{th}$ invocation of $\tau_1$.
            for $i = 2$ to $k - 1$ do
                find the exact computation time of the $\lfloor \frac{T_k}{T_i} \rfloor^{th}$ invocation of $\tau_i$.
                $CET := CET +$ the exact computation time of the $\lfloor \frac{T_k}{T_i} \rfloor^{th}$ invocation of $\tau_i$.
            end_do
            $U_{max} := \frac{CET}{T_k}$;
        end_do
        if  $U_{max} > U_{max}^{(1)}$ then $U_{max}^{(1)} := U_{max}$;

    end_if
end_do
$U_{max} := U_{max}^{(1)}$; $R_{lub} := 1 - U_{max}$;
```

tation time needed for all three periodic tasks before the second invocation of τ_3 will be $\left\lfloor \frac{T_3}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T_3}{T_2} \right\rfloor C_2 + C_3 + C_1$. Otherwise, the total computation time will be $\left\lfloor \frac{T_3}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T_3}{T_2} \right\rfloor C_2 + C_3 + q_{3,1} U_{\max}$. To find the exact utilization by τ_2, we have to know if $q_{3,2} > q_{3,1}$. If $q_{3,2} > q_{3,1}$ as shown in Fig. 6b, the time available for executing τ_2 from unit cycle yT_2 to T_3 is equal to the remaining processor time available after executing τ_1. However, if $q_{3,2} < q_{3,1}$ as shown in Fig. 6c, there may not have any time available for the execution of τ_2 from unit cycles yT_2 to T_3, or the available time will be at most $q_{3,2}\ U_{max}$, instead of all the time available from unit cycle xT_1 to T_3 as shown in Fig. 6c. Thus, the total computation time needed for these three tasks from unit cycle 0 to T_3 can be calculated accordingly.

Generally, R_{lub} for a set of m periodic tasks (C_1, T_1), ..., (C_m, T_m) can be derived by Algorithm 1 (see next page). Let $T_1 < T_2 < \ldots < T_m$ and assume all tasks are released at time 0.

The R_{lub} derived from Algorithm 1 is the least upper bound of R_S as reasoned below. No periodic tasks will miss their deadlines if a fraction, $R_{lub} = 1 - U_{max}$, of each unit cycle is reserved, where U_{max} is the maximum processor utilization by periodic tasks. Note that when all periodic tasks are released at the same time, say time 0, the critical instant occurs at the second invocation of τ_i $i > 1$. Since the U_{max} derived from Algorithm 1 is the maximum utilization at one of these critical instants, all periodic tasks will be completed before their next invocation after reserving a fraction, $R_{lub} = 1 - U_{max}$, of each unit cycle. Thus, the R_{lub} derived from the above algorithm is the least upper bound of R_S.

The complexity of Algorithm 1 is $O(m^2)$. This is based on the observation that there are $m - 1$ critical instants to be considered, i.e., at the second invocation of each τ_i for $i = 2$, ..., m. At each of these critical instants, one needs to consider up to $k - 1$ tasks for $k = 2$, ..., m. Note that a constant number (4) of iterations is used in Algorithm 1 to calculate U_{max} at each critical instant, so the iteration loop only contributes a constant factor in the complexity of this algorithm.

D. Calculating the Probability of Meeting Aperiodic Task Deadlines

Substituting the R_{lub} derived from (5) into (3), one can maximize the probability of guaranteeing aperiodic tasks, because $P\left(C_\alpha \le \overline{F}\, D_\alpha\right)$ is a monotonic increasing function of R_S. Since the deadlines of aperiodic tasks are not known until their actual arrival, $P(D)$ in (3) can be calculated and the probability of guaranteeing aperiodic tasks can then be derived from (3). However, any *a priori* performance evaluation requires the distribution of aperiodic-task deadlines. A uniform or exponential distribution of deadlines has been commonly used [12], [13], [14]. In case an exponential distribution is used, randomly-generated deadlines are close to their mean value, and thus, the variation of deadlines is not large. The probability of guaranteeing aperiodic tasks is closely related to the distribution of deadlines. Considering the fact that the distribution of aperiodic-task deadlines is application-dependent, one can let

$$D = D_{min} + \lfloor (1 + \alpha X)\, C_i \rfloor, \qquad (6)$$

where D_{min} is the minimal deadline of an aperiodic task, α is an integer number, and X is a random variable uniformly distributed in [0, 1]. As stated earlier, C_i is assumed to be exponentially distributed with mean μ.

The deadline distribution in (6) allows us to adjust the control parameters, D_{min} and α, to choose an appropriate range of deadlines for aperiodic tasks. D_{min} serves a purpose similar to the minimal separation p in [1]. By varying D_{min}, one can control the worst-case probability of missing deadlines, thus guaranteeing the deadlines up to any acceptable level. $\lfloor (1 + \alpha X)\, C_i \rfloor$ determines the range of deadlines; a large α will generally result in a wide range of distribution. Since it is assumed that X is uniformly distributed and C_i is exponentially distributed, the range of deadlines is a composite function of uniform and exponential distributions. Using (6) to specify the deadline of each aperiodic task can avoid the drawbacks of assuming either a loose bound or a tight bound of deadlines. Moreover, the randomly-generated deadlines are uniformly distributed in the range determined by $\lfloor (1 + \alpha X)\, C_i \rfloor$, and thus, the deadlines are more evenly distributed in a specified range than the exponential distribution. To be consistent with the case of periodic tasks where the deadlines are equal to the task period (an integer number of unit cycles), the deadlines of aperiodic tasks are truncated to be integer numbers (represented in unit cycles) in (6).

Considering $D_\alpha = D_{min}$ as an example, we get

$$\begin{aligned} &P(D_\alpha = D_{\min}) \\ &= P\left(C_\alpha < \frac{1}{1+\alpha}\right) + P((1+\alpha X)C_\alpha < 1)\,P\left(\frac{1}{1+\alpha} \le C_\alpha < 1\right) \qquad (7) \\ &= 1 - e^{-\frac{\mu}{1+\alpha}} + \frac{\mu}{\alpha}\int_{\frac{1}{1+\alpha}}^{1} \frac{e^{-\mu t}}{t}\,dt + \frac{1}{\alpha}\left(e^{-\mu} - e^{-\frac{\mu}{1+\alpha}}\right). \end{aligned}$$

If $C_\alpha < \dfrac{1}{1+\alpha}$, the second term in (6) is always less than 1, thus giving the first term in (7). If $1/(1 + \alpha) < C_\alpha < 1.0$, then D_α is equal to D_{min} when $(1 + \alpha X)\, C_\alpha < 1.0$. Since X is a uniformly-distributed random variable in [0, 1], $P((1+\alpha X)C_\alpha < 1) = \frac{1}{\alpha}(1/C_\alpha - 1)$, thus giving the second term in (7). If $C_\alpha > 1.0$, D_α is always greater than D_{min}. Similarly, $P(D_\alpha = D_{min} + 1)$ can be shown as:

$$\begin{aligned} &P(D_\alpha = D_{\min} + 1) \\ &= P((1+\alpha X)C_\alpha \ge 1)\,P\left(\frac{1}{1+\alpha} \le C_\alpha < \frac{2}{1+\alpha}\right) \\ &+ P(1 \le (1+\alpha X)C_\alpha < 2)\,P\left(\frac{2}{1+\alpha} \le C_\alpha < 2\right) \qquad (8) \\ &= \left(1 + \frac{1}{\alpha}\right)\left(e^{-\frac{\mu}{1+\alpha}} - e^{-\frac{2\mu}{1+\alpha}}\right) \\ &- \frac{\mu}{\alpha}\int_{\frac{1}{1+\alpha}}^{\frac{2}{1+\alpha}} \frac{e^{-\mu t}}{t}\,dt + \frac{\mu}{\alpha}\int_{\frac{2}{1+\alpha}}^{2} \frac{e^{-\mu t}}{t}\,dt. \end{aligned}$$

When $C_\alpha \in \left[\frac{1}{1+\alpha}, \frac{2}{1+\alpha}\right)$, $(1 + \alpha X)\, C_\alpha$ is always less than 2. So, the probability of $D_\alpha = D_{min} + 1$ is $(1 + \alpha X)\, C_\alpha > 1$, thus giving the first term in (8). When $C_\alpha \in [2/(1 + \alpha), 2)$, $D_\alpha = D_{min} + 1$ if $(1 + \alpha X)\, C_\alpha \in [1, 2)$; this is the second term in (8). When $C_\alpha > 2$, D_α is always greater than $D_{min} + 1$. Generally, for $P(D_\alpha = D_{min} + k)$, $k = 2, \ldots, D_{max}$, we get:

$$\begin{aligned} &P(D_\alpha = D_{\min} + k) = \\ &P((1+\alpha)C_\alpha \ge k)\,P\left(\frac{k}{1+\alpha} \le C_\alpha < \frac{k+1}{1+\alpha}\right) + \\ &P(k \le (1+\alpha)C_\alpha < k+1)\,P\left(\frac{k+1}{1+\alpha} \le C_\alpha < k\right) \qquad (9) \\ &= \left(1 + \frac{1}{\alpha}\right)\left(e^{-\frac{k\mu}{1+\alpha}} - e^{-\frac{(k+1)\mu}{1+\alpha}}\right) - \\ &\frac{k\mu}{\alpha}\int_{\frac{k}{1+\alpha}}^{\frac{k+1}{1+\alpha}} \frac{e^{-\mu t}}{t}\,dt + \frac{\mu}{\alpha}\int_{\frac{?}{1+\alpha}}^{k} \frac{e^{-\mu t}}{t}\,dt. \end{aligned}$$

From (6), the deadline of an aperiodic task is generated based on its execution time which was randomly-generated from an exponential distribution. The effect of D_{min} and α on the probability of guaranteeing an aperiodic task will be discussed in the next section.

IV. The RB Algorithm for Hard Real-Time Systems

Missing any critical-task deadline in a hard real-time system may lead to catastrophe [15], [16], [17]. Since all periodic tasks are guaranteed under the RMPA part of the RB algorithm, we only need to consider aperiodic tasks. However, due to their random-arrival nature, aperiodic tasks cannot always be guaranteed without imposing some restrictions on their behavior. The following theorem defines an upper bound of the ratio of the computation time to deadline of each aperiodic task in order to guarantee all aperiodic tasks with the RMPA.

THEOREM 2. *All aperiodic tasks v_i, for $i = 1, \ldots, k$ can be guaranteed by the RMPA (i.e., $R_S = 0$) if*

$U_a = \sum_{i=1}^{k} \frac{C_i}{D_i} \le 1 - U_{\max}$, *and min* $D_i \ge T_{mc}$, $\forall$ *i, where* U_{max} *is derived from Algorithm* 1.

PROOF. Consider the case of one aperiodic task. Let $D_a < T_{mc}$. Even if $C_a/D_a \le 1 - U$, this task may not be guaranteed if it arrives at those unit cycles with 100% utilization by periodic tasks as shown in Fig. 1. For example, suppose an aperiodic task arrives at the 16th unit cycle and with $D_a = 4$. Since the system is 100% utilized by periodic tasks from unit cycles 16 to 20, this task cannot be guaranteed no matter how small C_a/D_a is. However, if $D_a \ge T_{mc}$, the system can always allocate up to a fraction, $1 - U$, of the processor time for this task, regardless of the time of its arrival.

It is straightforward to apply the above argument to the case with an arbitrary number of aperiodic tasks, and thus, the theorem follows. □

Let us define the *total* processor utilization by periodic and aperiodic tasks as $U_T = U + U_a$. Note that periodic tasks are known *a priori* and preallocated before their actual release. The problem of the RMPA is two-fold. First, the deadlines of all aperiodic tasks have to be greater than, or equal to, the major cycle of a given set of periodic tasks. This is unrealistic because the deadlines of aperiodic tasks are not known until they actually arrive. Moreover, T_{mc} for a given set of periodic tasks may be very long, thus severely limiting the number of aperiodic tasks that can be guaranteed. The second problem is that the set of aperiodic tasks that can be guaranteed will depend on the set of periodic tasks already scheduled, because the processor utilization, U, by periodic tasks may vary from one set of periodic tasks to another. To avoid the second problem, a set of aperiodic tasks must be chosen based on the set of periodic tasks with the highest processor utilization, thus severely reducing the average processor utilization.

In contrast to the RMPA, in the RB algorithm the set of aperiodic tasks that can be guaranteed depends only on R_S, and we have the following theorem.

THEOREM 3. *All aperiodic tasks* v_i, *for* $i = 1, \ldots, k$, *can be guaranteed by the RB algorithm by reserving a fraction,* $R_S \ne 0$, *of the processor time if*

$$U_a = \sum_{i=1}^{k} \frac{C_i}{D_i} \le R_S,$$

where k is the number of aperiodic tasks queued for execution.

PROOF. This theorem is proved by induction.

1) When $k = 1$. Since a fraction, R_S, of processor time is reserved in each unit cycle for aperiodic tasks, regardless of its arrival time an aperiodic task can be allocated up to $R_S\, D_1 \ge C_1$ of processor time for its execution, so the theorem follows.

2) When $k = 2$. Consider the (worst) case when both $v_1 = (a_1, C_1, D_1)$ and $v_2 = (a_1, C_2, D_2)$ arrive at the same time. For convenience, assume $D_1 < D_2$ and v_1 is scheduled to execute first. Since v_1 is scheduled first, the processor time that can be allocated to v_2 is $R_S\, D_2 - C_1$ and this has to be greater than, or equal to, C_2. From the condition of Theorem 3, we get $\frac{C_1}{D_1} + \frac{C_2}{D_2} \le R_S$. Multiplying D_2 on both sides, we get

$$C_1 \frac{D_2}{D_1} + C_2 \le R_S D_2 \Rightarrow R_S D_2 - C_1 \frac{D_2}{D_1} \ge C_2 .$$

Since $D_2 > D_1$, the above inequality still holds for $R_S D_2 - C_1 \ge C_2$, thus proving the theorem.

3) Assume the theorem holds for $k = n$. Consider the worst case when all of these $n + 1$ tasks arrive at the same time. For convenience, let $D_1 < D_2 < \ldots D_n < D_{n+1}$ and the tasks are scheduled in the sequence of $\{\tau_1, \tau_2, \ldots, \tau_n, \tau_{n+1}\}$. From the condition of Theorem 3, we get $\sum_{i=1}^{n+1} \frac{C_1}{D_1} \le R_S$. Multiplying D_{n+1} on both sides, we get

$$\sum_{i=1}^{n+1} C_i \frac{D_{n+1}}{D_i} \le R_S D_{n+1} \Rightarrow R_S D_{n+1} - \sum_{i=1}^{n} C_i \frac{D_{n+1}}{D_i} \ge C_{n+1} .$$

Since $D_{n+1} > D_n > \ldots D_1$, the above expression can be written as:

$$R_S D_{n+1} - \sum_{i=1}^{n} C_i \ge C_{n+1} .$$

Thus, all of these $n + 1$ tasks can be guaranteed. □

The chief advantage of employing the RB algorithm in hard real-time systems is that aperiodic tasks can always be guaranteed as long as the combined utilization by all aperiodic tasks is less than the reserved fraction, R_S, of processor time. Moreover, the R_{lub} derived from Algorithm 1 is shown to be the maximum (or optimal) R_S that can be reserved without missing any periodic-task deadline. Thus, by reserving a fraction, R_{lub}, of processor time for aperiodic tasks, the RB algorithm becomes an optimal solution to the problem of scheduling both periodic and aperiodic tasks in hard real-time systems. By contrast, the RMPA requires an additional, unrealistic restriction, $D_i > T_{mc}$, $\forall$ i, to get the same result as the RB algorithm.

V. COMPARATIVE ANALYSIS OF RB ALGORITHM

The RB algorithm is intended for use in scheduling both periodic and aperiodic real-time tasks to meet their deadlines. In order to show its advantages and limits, we analyze and compare the RB algorithm against the other methods using a concept (similar to ours) of reserving a fraction of the processor time for aperiodic tasks, such as the PE, DS, or slack stealing algorithms.

A. Performance Analysis

The RMPA is chosen as part of the RB algorithm to schedule periodic tasks. Aperiodic tasks are scheduled according to the FCFS policy by using the reserved (solely for aperiodic tasks) and unused[4] processor time in each unit cycle.

The performance of the RB algorithm is also evaluated by simulation and compared with the analytic results. Since the value of R_{lub} derived analytically in Section III is used to reserve the processor time, we want to know how close R_{lub} is to the average $R_{avg} = 1 - U$, because the processor utilization cannot exceed 100%. The ratio of R_{lub} to $1 - U$ is plotted in Fig. 7. The total processor utilization is the sum of R_{lub} and U, also shown in this figure. Note that R_{lub} can be as high as 95% of the R_{avg} when $U < 0.5$. Even when U is as high as 0.95, R_{lub} is still about 60% of R_{avg}. Since R_{lub} is very close to R_{avg}, the total processor utilization remains almost constant at 97% regardless of the processor utilization by periodic tasks.

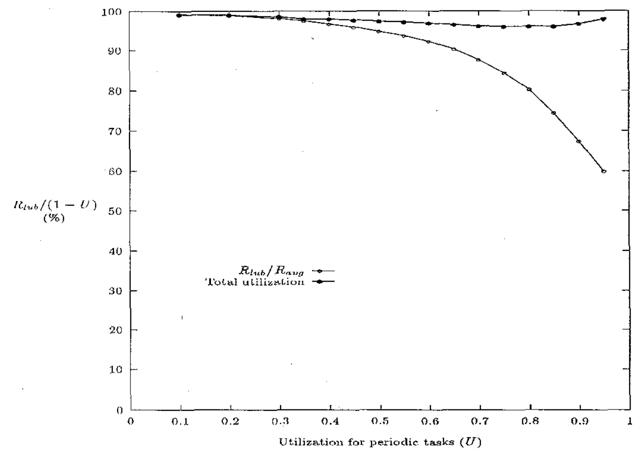


Fig. 7. Ratio of R_{lub} to $R_{avg}(= 1 - U)$ and the total processor utilization.

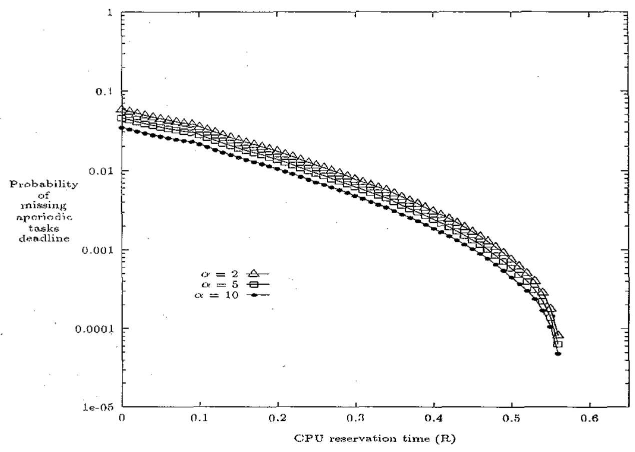


Fig. 8. Probability of missing aperiodic-task deadlines vs. processor reservation ($U = 0.4$, $D_{min} = 5$).

As shown in Fig. 8, the probability of missing aperiodic-task deadlines is found to decrease as R_S increases. as expected from Theorem 1. Increasing the value of α reduces the probability of missing aperiodic-task deadlines. The effect of increasing D_{min} for aperiodic tasks can be seen in Fig. 8. The probability of missing a deadline can be reduced to as small as 10^{-4} when $U = 0.4$, $D_{min} = 5$, and the processor time is maximally reserved (i.e., $R_S = R_{lub}$) in each unit cycle (Fig. 8). The effects of varying D_{min} and R_S on the probability of missing deadlines is also studied. It is found that the probability of missing aperiodic-task deadlines increases as U increases. Thus, U must be kept below a certain level in order to reduce the probability of missing aperiodic-task deadlines to a prespecified acceptable level.

4. More precisely, allocated to, but unused by, periodic tasks. This happens because one might use the worst-case execution times of periodic tasks when scheduling them.

B. Comparison With Other Related Work

The concept of reserving a fraction of processor time for aperiodic tasks has also been considered by others. For example, Lehoczky et al. proposed the priority exchange (PE) and deferrable server (DS) algorithms [2] to improve the response times of aperiodic tasks while guaranteeing periodic-task deadlines.

The objective in the PE and DS algorithms is to improve the response times of aperiodic tasks, while the objective in the RB algorithm is to enhance the probability of meeting their deadlines. (Hence the latter is cognizant of the deadlines of aperiodic tasks.) Due to this difference in objectives, the periodic server for aperiodic tasks in the PE and DS algorithms can significantly improve the response times of aperiodic tasks as reported in [2] but not necessarily improve the probability of guaranteeing aperiodic tasks. The periodic server in the PE and DS algorithms is scheduled according to the RMPA with the rest of regular periodic tasks. In order not to miss the deadline of any periodic task, this server can only be allocated up to a fraction, R_{lub}, of time in each unit cycle. Recall that R_{lub} is the maximum fraction of time that can be reserved for aperiodic tasks without missing any periodic-task deadline. Moreover, as reported in [2], the period of this server cannot be too long because aperiodic tasks may otherwise need to wait for a long time if its arrival does not coincide with the beginning of the periodic server. Due to these two restrictions, the maximum time that can be allocated to the periodic server is $R_{lub} \times T_a$, where T_a is the period of this server. So, only aperiodic tasks with computation time less than $R_{lub} \times T_a$ and deadline longer than T_a can be guaranteed. Obviously, this is an undesirable restriction in scheduling aperiodic tasks. Sprunt et al. [3] later proposed the extended priority exchange (EPE) algorithm based on the PE and DS algorithms to improve the response times of aperiodic tasks when the worst-case periodic load is high and little processor time is left for the aperiodic task server. However, the aforementioned basic restrictions on the periodic server still exist, because the capability of guaranteeing aperiodic tasks is limited by the short period of the server which is necessary to ensure the short response times of aperiodic tasks.

It should be noted that the slack stealing algorithm proposed by Lehoczky and Ramos-Thuel [5] is shown to be optimal for scheduling soft aperiodic tasks in fixed-priority preemptive systems. In this algorithm, the slack stealer does not create a periodic server for aperiodic tasks. It instead creates a passive task which, when prompted for service, attempts to make time for servicing aperiodic tasks by "stealing" all the available processing time from periodic tasks without missing any periodic-task deadline. They proved that the slack stealing algorithm is optimal in the sense that all available processing time will be exploited for servicing aperiodic tasks while meeting all periodic-task deadlines.

The basic concept behind the slack stealing algorithm is similar to the derivation of R_{lub} in the RB algorithm. As shown in Fig. 7, the derived R_{lub} is close to, or higher than, 95% of R_{avg} when U is less than 50% and can still be as high as 80% of R_{avg} when U is 80%. But in the slack stealing algorithm, one needs to gradually increase the processor time that can be reserved without missing any periodic-task deadline until the optimal solution is found. However, in the RB algorithm, this optimal solution can be derived systematically using Algorithm 1. The preliminary results of the RB algorithm was also reported at the 1991 Real-Time Systems Architecture Workshop [18], while the slack stealing algorithm was presented a year later at the 1992 Real-Time System Symposium.

We claim several important contributions via the development and evaluation of the RB algorithm. First, we proposed an analytic model to evaluate the performance of the RB algorithm; this is in sharp contrast to most of the early work that solely relies on simulations. Second, although the concept of reserving a fraction of processor time for aperiodic tasks has been proposed and used by others, the analytic approach developed in this paper is the first of the kind that treats the problem of systematically deriving the maximum processor time to be reserved for the execution of aperiodic tasks without missing any periodic-task deadline. Moreover, we can derive the least upper bound of R_S with Algorithm 1 *before* putting the system in operation, thus incurring no scheduling overhead at the time of arrival of each aperiodic task. (That is, there is little on-line scheduling overhead for aperiodic tasks.) Third, as proven in Theorem 3, all aperiodic tasks can be guaranteed as long as their combined utilization is less than R_{lub}. If the total number of aperiodic tasks waiting for execution is n, we only need n divisions and n additions to determine whether a newly-arrived aperiodic task can be guaranteed or not. This is much faster than any other existing methods. Even if this restriction cannot be satisfied, the probability of guaranteeing aperiodic tasks can still be maximized by reserving a fraction, R_{lub}, of the processor time in each unit cycle, because this probability is shown to be a monotonically increasing function of $R_S \leq R_{lub}$. Finally, the most important result of the RB algorithm is the analytic derivation of R_{lub}. Since R_{lub} is fixed for a given set S of periodic tasks and since it can be calculated *a priori*, scheduling aperiodic tasks is more predictable than the case when there is no knowledge of how much of the processor time can be used for the execution of aperiodic tasks. Moreover, using the relationship $R_{lub} = 1 - U_{max}$, one can determine whether S is schedulable or not without actually employing the RMPA. As shown by Liu and Layland [6], if the utilization by a set of periodic tasks is less than log 2 then the task set is always schedulable, but if the utilization is greater than log 2 then the task set may, or may not, be schedulable. For example, if the R_{lub} derived from Algorithm 1 is not positive for a given set of periodic tasks, the task set is not schedulable under the RMPA, because the task set requires utilization greater than 100% at each critical instant.

VI. Conclusions and Future Work

We proposed a new algorithm to schedule *both* periodic and aperiodic real-time tasks. Periodic tasks are scheduled according to the RMPA and their deadlines are guaranteed if the task set is schedulable (or $U <$ log 2) as specified in [6]. Aperiodic tasks are scheduled by utilizing the reserved and unused (by periodic tasks) processor time in each unit cycle. We have shown that the value of R_S greatly affects the probability of guaranteeing aperiodic tasks even when the processor utilization is fixed. The relation between R_S and the probability of guaranteeing aperiodic tasks is established for the case when the execution time of aperiodic tasks is exponentially-distributed. The least upper bound of R_S (R_{lub}) is derived for this case. Since the probability of guaranteeing an aperiodic task is a monotonic increasing function of R_S, this probability is maximized when we reserve a fraction, R_{lub}, of each unit cycle. Moreover, if the utilization by aperiodic tasks is restricted to below R_{lub}, all aperiodic tasks can be guaranteed by the RB algorithm as discussed in Theorem 3, regardless of their arrival time and their required computation time. Thus, the RB algorithm can be used to schedule both periodic and aperiodic tasks in a hard real-time system as long as Theorem 3 is satisfied.

The RB algorithm proposed in this paper suggests many interesting issues that warrant further investigation. Some of these issues are briefly discussed below. First, if the task switching time is not negligible, this overhead in the RB algorithm may affect the derivation of R_{lub}. Generally, the number of task switchings increases when a fraction, R_S, of a unit cycle is reserved, as aperiodic tasks will preempt periodic tasks during its reserved period of time. If this number increases as R_S is increased, then the probability of guaranteeing aperiodic tasks will no longer be a monotonic increasing function of R_S. Instead, there may exist an optimal $R_S < R_{lub}$, because increasing R_S will increase the switching overhead, thus reducing the amount of processor time to be allocated for the execution of aperiodic tasks. Thus, one needs to derive a bound for the task switching overhead as a function of R_S.

Second, when $U < 70\%$, it is possible to have more than one aperiodic task waiting for execution. A good scheduling algorithm may improve the probability of guaranteeing aperiodic tasks, but, as discussed in Section V, the EDF policy does not necessarily perform better than the FCFS policy. It is therefore desirable to find a simple algorithm which can satisfy the condition of Theorem 3 while incurring minimal scheduling overhead. This situation can be seen by the following example. If there are k aperiodic tasks waiting for execution and their combined utilization $\sum_{i=1}^{k} \frac{C_i}{D_i}$ is less than R_{lub}, then these tasks can be guaranteed according to Theorem 3. However, if an early-arrived task a is scheduled to its completion using a fraction, R_{lub}, of processor time, an aperiodic task arriving later but before the task a's completion with a shorter deadline than task a may miss its deadline, even when their combined utilization is less than R_{lub}. This phenomenon can be explained as follows. Since $\sum_{i=1}^{k} \frac{C_i}{D_i} < R_{lub}$, we have $\frac{C_i}{D_i} < R_{lub}, \forall i$. But, if one of the tasks, say a, is allocated up to a fraction, R_{lub}, of processor time during the first few unit cycles, this task has, in fact, utilized the processor time more than it is supposed to, i.e., the ratio of $\frac{C_a}{D_a}$ over the D_a unit cycles. So, if another aperiodic task, say b, arrives before the completion of a, task b may not be able to meet its deadline even when $\frac{C_a}{D_a} + \frac{C_b}{D_b} < R_{lub}$, because task a has consumed all the reserved fraction of processor time before its completion.

A simple adaptive RB algorithm may be used to resolve the above problem. If each of these tasks is allocated up to a fraction, $\frac{C_i}{D_i}, \forall i$, of processor time in each unit cycle, all of these tasks can be guaranteed, regardless of the order of their arrival. In other words, when a fraction $\frac{C_i}{D_i}$ of processor time is reserved for each of these tasks, all of them can be guaranteed as long as their combined utilization is less than R_{lub}. However, the model for analyzing the probability of guaranteeing aperiodic tasks needs to be modified since each aperiodic task may have a different R_S instead of a constant R_{lub}. Also, the switching overhead will be much higher in the adaptive RB algorithm as many aperiodic tasks will be executed in the same unit cycle during their reserved period. How to construct an analytic model to evaluate the probability of guaranteeing aperiodic tasks as well as the switching overhead of the adaptive RB algorithm needs further investigation.

Third, it is practically important to extend the RB algorithm to multiprocessor/distributed systems. For example, consider a distributed system with N nodes. Since each node has its own set of periodic tasks assigned *a priori* and aperiodic tasks arriving randomly, it is desirable to schedule these tasks so that a maximum number of aperiodic tasks in the entire system may be guaranteed without missing any periodic-task deadline. Obviously, it is impossible to obtain an optimal global solution by simply combining N independent nodes, each with an optimal local scheduling algorithm. Due to the variation of $U(i)$ as shown in Fig. 1, an aperiodic task that cannot be guaranteed on one node might be guaranteed on another node. Thus, one way to solve the above problem for a distributed system is to use the concept of load sharing (LS) as proposed in [19]. There are many issues to be resolved before LS is employed along with the RB algorithm to achieve an optimal global scheduling algorithm in distributed systems. For example, we must explore ways of collecting state information and the type of state information to be collected. If a node cannot guarantee an aperiodic task, then when and where to transfer this aperiodic task in order to meet its deadline is the main issue. Another related question is "should only aperiodic tasks be transferred or even periodic tasks should be considered for transfer?" For example, during bursty arrivals of aperiodic tasks at a particular node, it might be more beneficial to transfer some of periodic tasks to other nodes so that the node's utilization by periodic tasks may be reduced in order to guarantee more aperiodic tasks locally. After the bursty arrivals die out, these periodic tasks can be transferred back to its original node.

The foregoing problems are matters of our future inquiry.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by the Office of Naval Research under Grant N00014-J-92-1080 and the National Science Foundation under Grant MIP-9203895. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agencies.

REFERENCES

[1] A. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," PhD thesis, Massachusetts Institute of Technology, 1983.

[2] J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," *Proc. Eighth Real-Time System Symp.*, pp. 261–270, 1987.

[3] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting unused periodic time for aperiodic service using the extened priority exchange algorithm," *Proc. Ninth Real-Time System Symp.*, pp. 251–258, 1988.

[4] K. Jeffay, R. Anderson, and C. Martel, "On optimal, non-preemptive scheduling of periodic and sporadic tasks," Technical Report TR-90-019, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, Apr. 1990.

[5] J.P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," *Proc. 13th Real-Time Systems Symp.*, pp. 110–123, 1992.

[6] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.

[7] M.H. Woodbury and K.G. Shin, "Evaluation of the probablity of dynamic failure and processor utilization for real-time systems," *Proc. Ninth Real–Time System Symp.*, pp. 222–231, 1988.

[8] J.Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[9] J. Zhu, T.G. Lewis, and J.-Y. Colin, "Scheduling hard real-time constrained tasks on one processor," Tech. Rep. #93-60-16, Computer Science Dept., Oregon State Univ., June 1993.

[10] C.-C. Han and K.-J. Lin, "Scheduling distance-constrained real-time tasks," *Proc. 13th Real-Time System Symp.*, pp. 300–308, 1992.

[11] J.P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," Technical report, Dept. of Statistics, Carnegie Mellon Univ., 1987.

[12] J.A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real–time systems," *IEEE Transactions on Computers*, vol. 34, pp. 1,130–1,143, Dec. 1985.

[13] K. Ramamritham, J.A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*, vol. 38, pp. 1,110–1,123, Aug. 1989.

[14] J. Hong, X. Tan, and D. Towsley, "A performance analysis of minimum laxity and earliest deadline scheduling in a real–time system," *IEEE Transactions on Computers*, vol. 38, pp. 1,736–1,744, Dec. 1989.

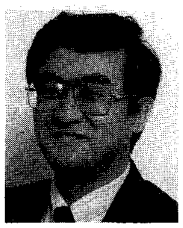
[15] K.G. Shin, C.M. Krishna, and Y.-H. Lee, "A unified method for evaluating real–time computer controllers and its application," *IEEE Trans. on Auto. Contr.*, vol. 30, pp. 357–366, Apr. 1985.

[16] D.W. Leinbaugh, "Guaranteed response times in a hard–real–time environment," *IEEE Transactions on Software Engineering*, vol. 6, pp. 85–93, Jan. 1980.

[17] C.M. Krishna, K.G. Shin, and I.S. Bhandari, "Processor tradeoffs in distributed real–time systems," *IEEE Transactions on Computers*, vol. 36, pp. 1,030–1,040, Sept. 1987.

[18] Y.-C. Chang and K.G. Shin, "Scheduling periodic tasks with consideration of load sharing of aperiodic tasks," *Workshop on Architecture Support for Real-Time Systems*, pp. 91–95, Dec. 1991.

[19] K.G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state change broadcasts," *IEEE Transactions on Computers*, vol. 38, pp. 1,124–1,142, Aug. 1989.

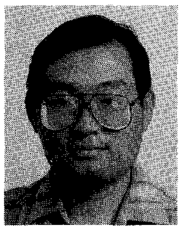


Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970 and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, N.Y., in 1976 and 1978, respectively. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, N.Y. He has held visiting positions at the U.S. Air Force Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California at Berkeley, International Computer Science Institute, Berkeley, Calif., and IBM T.J. Watson Research Center, Yorktown Heights, N.Y. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning January 1991.

He is currently professor and director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor. He has authored/coauthored over 350 technical papers (more than 150 of these in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. He is currently writing jointly with C. M. Krishna a textbook *Real-Time Systems* which is scheduled to be published by McGraw Hill in 1996. In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called HARTS, to validate various architectures and analytic results in the area of distributed real-time computing.

He has also been applying the basic research results of real-time computing to intelligent vehicle highway systems and manufacturing-related applications, ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes. Recently, he has initiated research on the open-architecture information base for machine tool controllers and middleware services for real-time fault-tolerant embedded systems.

He is an IEEE fellow, was the program chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the general chairman of the 1987 RTSS, the guest editor of the 1987 August special issue of *IEEE Transactions on Computers* on Real-Time Systems, a program cochair for the 1992 *International Conference on Parallel Processing*, and served on numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-93, is a distinguished visitor of the IEEE Computer Society, an editor of *IEEE Transactions on Parallel and Distributed Computing*, and an area editor of *International Journal of Time-Critical Computing Systems*.



Yi-Chieh Chang (S'84) received the BS and MS degrees in electrical engineering from National Taiwan University, Taipei, Republic of China, in 1979 and 1984, respectively. He received his PhD degree in electrical engineering and computer science from the University of Michigan, Ann Arbor in 1991. He was an assistant professor in the Electrical and Computer Engineering Department at the University of Texas at El Paso until May 1995. He is now with Quickturn Design Systems, Inc., Mountain View, Calif. He has published more than 20 technical papers in the areas of reconfigurable fault-tolerant array processors, parallel computer architecture, and distributed real-time systems. His research interests include computer architecture, VLSI systems, parallel processing, and distributed real-time systems.