MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A HARD REAL-TIME ENVIRONMENT[*]

ASHOK KHEMKA
Computer Science Group
Tata Institute of Fundamental Research
Bombay 400 005, India

R.K. SHYAMASUNDAR[†]
Computer Science Group
Tata Institute of Fundamental Research
Bombay 400 005, India
e-mail: shyam@tifrvax.bitnet

ABSTRACT

The problem of preemptive scheduling a set of periodic tasks on multiprocessors is studied from the point of view of meeting their service requirements before their respective deadlines. Sufficient conditions which permit full utilization of the multiprocessor using the given scheduling algorithms are derived. The complexity of the scheduling algorithms in terms of the number of scheduled tasks and the number of processors and upper bounds on the number of preemptions in a given time interval and for any single task are also derived. We also give a schema for modifying existing schedules at little run-time cost when tasks arrive or leave the system on-line.

Keywords: hard real-time, multiprocessor scheduling, preemptions

1. Introduction. *Hard real-time* systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Se-

[*] An earlier version of the paper was presented at the 6th IPPS 1992.
[†] Correspondence Address.

vere consequences will result if timing correctness properties of the system are violated. In other words, satisfying the timing requirements of hard real-time systems demand the scheduling of system resources according to some well understood algorithms, so that the timing behaviour of the system is understandable, predictable and maintainable. Applications of real-time systems can be found in automated factories, robot and vision systems, military command and control systems, process control systems, flight control systems and future systems such as the space-based defense systems, SDI. The use of multiprocessors for control and monitoring functions in such real-time systems has increased recently. Efficient use of these systems can only be achieved by a proper scheduling of *time-critical* functions. Each time-critical function has associated with it a set of tasks, which are either executed in response to external events or in response to events in other tasks. None of the tasks are executed before the event which requests it occurs. The tasks must be serviced within preassigned deadlines dictated by the physical environment. For example, a radar that tracks flights produces data at a fixed rate. A temperature monitor should be read periodically to detect any changes promptly. Some of these periodic tasks may exist from the point of system initialization, while others may come into existence dynamically. An example of a dynamically created task is a task that monitors a particular flight; this comes into existence when the aircraft enters an air traffic control region and will cease to exist when the aircraft leaves the region.

Thus, a scheduling problem in a hard real-time system is defined by the model of the system, the nature of tasks to be scheduled, and the objectives of the scheduling strategy. Not very much is known (cf. [12]) about scheduling algorithms for real-time systems. Most of the existing results either pertain to simplistic situations or single processor systems. For a survey of the existing results, the reader is referred to [3]. In, fact, most of the scheduling problems of interest to practical real-time systems are NP-hard and hence, there is a need for heuristics or approximation techniques. In this paper, we investigate hard real-time scheduling issues at compile time, when the characteristics of tasks are known *a priori*. We also show in a restricted sense how to dynamically update the schedule built at compile time when new tasks arrive or existing tasks leave the system *on-line*. We make the following assumptions about the hard real-time environment in this paper:

1. Requests for tasks are periodic, with constant interval between requests.
2. Each task must be completed before the next request for it occurs.
3. Tasks are independent.
4. Computation time for each task is constant for that task.
   This can be taken as the maximum processing time for the task

including the bookkeeping time necessary to request a successor and the costs of preemptions.

These assumptions allow the characterization of a task by the following four parameters:

1. The arrival time, A: The time at which a task is invoked in the system.
2. The ready time, R: The earliest time at which a task can begin execution. The ready time of a task is equal to or greater than its arrival time. Since we have assumed the tasks to be independent, the ready time of a task is the same as its arrival time.
3. The worst case computation time, C.
4. The deadline D: The time by which a task must finish. In our model of hard real-time environment, the deadline of a periodic task is equal to the arrival time of its next instance.

Let $\{\Gamma_1, \Gamma_2, ..., \Gamma_m\}$ be a set of m periodic jobs with computation times $\{C_1, C_2, ..., C_m\}$, and periodicity[1] $\{D_1, D_2, ..., D_m\}$ respectively. Assumption (2) about the hard real-time environment implies that the k^{th}, $k \geq 1$, instance of a task Γ_i, must be computed in full between the time units $(k-1) * D_i$ and $k * D_i$. We are interested in constructing a *preemptive schedule* of these m tasks on n processors at compile time. We assume that the computation times and periodicity of tasks are expressed as integral multiples of the processor clock tick. Hence a task can be preempted only at integral time units. A valid schedule must satisfy the following two constraints :

A1. At any instant, at most one task can be executed on any single processor.

A2. No single task can be executed on more than one processor at the same instant.

We define the *multiprocessor utilization factor* to be the fraction of total time spent in the execution of the task set. Since C_i/D_i is the fraction of multiprocessor time spent in executing task Γ_i ; for m tasks, the multiprocessor utilization factor is:

$$U = \sum_{i=1}^{m} (C_i/D_i).$$

Although the multiprocessor utilization factor can be improved by increasing the values of the $C_i's$ or by decreasing the values of the $D_i's$, it is constrained by the requirement that all tasks satisfy their deadlines. It is clear from

[1] For the sake of simplicity, we have assumed correspondence between deadline and periodicity and also, we have assumed that the arrival time of a task coincides with the starting time of the task in the first period.

constraint (A1) that the condition $U \leq n$ is necessary for *feasible scheduling* of a set of periodic jobs on n processors. It is very interesting to know how large the utilization factor can be. The uniprocessor case has been considered in [9], where it is shown that for feasible scheduling of a set of periodic tasks, the condition $U \leq 1$ is both necessary and sufficient. The authors of [9] propose two preemptive algorithms. In their first algorithm, called the *rate monotonic algorithm*, static priorities are assigned to tasks based on their periods. They also propose a dynamic priority assignment algorithm, the *earliest-deadline-first algorithm*, which allows full processor utilization. However, the general case whether the condition $U \leq n$ $(n \geq 1)$ is sufficient for feasible scheduling of a set of periodic tasks on n processors remains open. Let the *time slice* of a task Γ_i be $T * C_i/D_i$ where $T = GCD^2\{D_1, D_2, ..., D_m\}$. But the proof is incomplete because the schedule constructed by the above theorem is not shown to satisfy constraint (A2). A preemptive scheduling algorithm to calculate the minimum schedule length for tasks related by precedence constraints has been developed in [11]. In particular, three scheduling approaches are studied here: non-preemptive scheduling, preemptive scheduling and general scheduling. In general scheduling, processors are considered to possess a certain amount of computing capability that can be shared by multiple tasks. Most instances of the scheduling problem for hard real-time systems are shown to be computationally intractable in [7]. A partition approach has been used to solve the problem of scheduling periodic tasks on multiprocessors in [1,2,5]. The main idea is to partition a set of periodic tasks among a minimum number of processors such that each partition of the periodic tasks can be scheduled on one processor.

In this paper, we answer the question whether the condition $U \leq n$ is sufficient for feasible scheduling on n processors partially. We divide the time interval into *blocks* of equal length such that exactly one instance of each task remains *active* in each block. We attempt to meet the *average* requirement of every task within the block itself. We assume that task arrival times, computation times and deadlines are expressed as multiples of the processor clock ticks. We build an actual schedule when each task's requirement per block is expressible as integral clock ticks. When this is not so, we develop a second algorithm which allots integral time units to each task in a block, so as to meet the computation requirement of every task before its deadline. This scheduling algorithm is shown to work for two particular cases which are described below. We also derive an upper bound on the complexity of the two scheduling algorithms and on the number of preemptions in a given time interval and for any single task. We also give a schema for modifying

[2] Greatest Common Divisor.

the schedule at little run-time cost when tasks arrive or leave the system on-line.

Let $T = GCD\{D_1, D_2, ..., D_m\}$ and $D = LCM^3\{D_1, D_2, ..., D_m\}$. Let $C_i \leq D_i$, for each task Γ_i. Let the utilization factor U of a set of tasks $\{\Gamma_1, \Gamma_2, ..., \Gamma_m\}$ be $\sum_{i=1}^{m}(C_i/D_i)$. We call the time intervals $[0, T], [T, 2T], [2T, 3T], \cdots$ blocks of length T each. Also let $U \leq n$, for some integer n. In this paper, we establish the following *new* results :

1. A schedule of the set of m tasks on n processors when each of the task time slices is an integer.
2. A schedule of the set of m tasks on n processors when either $C_i/D_i \leq (1 - 1/T)$ or $C_i = D_i$, for each task Γ_i.
3. A schedule of the set of m tasks on n processors when $\sum_{i=1}^{m}(C_i/D_i - \lfloor C_i/D_i \rfloor) \leq 1$.
4. An upper bound on the complexity of the scheduling algorithm is $\mathcal{O}(m + n)$, when each of the task time slices is an integer; and is $\mathcal{O}(m * (D/T))$ otherwise.
5. An upper bound on the number of preemptions in a time interval of length D is $(m * D/T + \sum_{j \in \Delta}(D/D_j - 1))$, for some task set Δ containing not more than $(n - 1)$ tasks, when each of the task time slices is an integer; and is $(m + n - 1) * (D/T)$ otherwise. An upper bound on the number of preemptions for any task Γ_i is $(D_i/T + 1)$ when each of the task time slices is an integer; and is $2 * D_i/T$ otherwise.

   In our model, the operation of loading a task for the first time is also treated like a task preemption, as both operations involve near equal cost in physical applications.
6. A schema applicable in a restricted sense for modifying the existing schedule at little run-time cost when new tasks arrive on-line or existing tasks leave the system.

The rest of the paper is organized as follows. Section 2 describes an optimal scheduling algorithm, in the sense that it always constructs a feasible preemptive schedule, if there exists one, on n processors for a set of m periodic tasks having integral task time slices. Upper bounds on the complexity of the algorithm and on the number of task preemptions are also derived. In Section 2.1, we reduce the upper bound on the number of task preemptions by performing a cyclic permutation of the processor schedules. Section 3 describes another scheduling algorithm for tasks having non-integral time slices. It also derives upper bounds on the complexity of the algorithm and on the number of task preemptions. In Section 3.1, we have shown the algorithm to be optimal for two particular cases of task time slice values. Section

[3] Least Common Multiple.

4 describes a schema applicable in a restricted sense for modifying existing schedules at little cost when tasks arrive or leave the system on-line.

2. Integral Task Time Slices. When the average requirement of each task in a block is an integer, then the task is said to have an integral time slice. This means that the task's average requirement per block is exactly an integral multiple of the processor clock tick. In this section, we describe a scheduling algorithm **SA1** for deriving a feasible, multiprocessor schedule, if it exists, satisfying constraints (A1) and (A2) when each of the task time slice is an integer and establish its optimality. We also derive upper bounds on its time complexity and on the number of preemptions in a given time interval and for any single task when **SA1** is used to schedule tasks.

2.1. Scheduling Algorithm SA1. The scheduling algorithm is based on a simple heuristic of allocating the time for the jth task between two processors i and i+1 in a block (the length of the block is the GCD of the deadlines) only if the requirements for the block on hand exceeds the GCD. The number of preemptions are reduced by a simple scheme of cyclic permutation of processors. First, we describe the basic scheduling algorithm.

Algorithm SA1

Let $t_{i,j}$ be the i^{th} time unit corresponding to the j^{th} processor. The steps of the algorithm are described below:

```
1. Initialize : $i, j, k \leftarrow 1$;
   ($i$ denotes the processor, $j$ the time unit and $k$ the task number)
2. If $j + N_k - 1 \geq T$,
   then $i \leftarrow i + 1$; $j \leftarrow j + N_k - T$

       Task $k$ is scheduled on processor $i$ and processor $(i+1)$ for
       $[t_{i+1,1},\ t_{i+1,j+N_k-T-1}]$ and $[t_{i,j},\ t_{i,T}]$ intervals respectively.
   else $j \leftarrow j + N_k$

       Task $k$ is scheduled on processor $i$ only for the interval
       $[t_{i,j},\ t_{i,j+N_k-1}]$.
3. Iterate : $k \leftarrow k + 1$;
   If $k \leq m$, then goto step 2.

       Generate the schedule for the next task.
```

STEPS OF SCHEDULING ALGORITHM SA1

TABLE 1

Tasks:8, Processors:4
$T = GCD[30, 20, 10, 20, 40, 50, 60, 70] = 10$
$D = LCM[30, 20, 10, 20, 40, 50, 60, 70] = 4200$

Task	D_i	C_i	$T * C_i/D_i$
Γ_1	30	21	7
Γ_2	20	8	4
Γ_3	10	5	5
Γ_4	20	8	4
Γ_5	40	8	2
Γ_6	50	35	7
Γ_7	60	12	2
Γ_8	70	63	9

2.1.1. Illustrative Example. Consider the problem of scheduling eight periodic tasks on four processors as described in TABLE 1.

A schedule constructed by **SA1** for the system described in TABLE 1 is shown in Fig.1

2.1.2. Correctness and Complexity. In this section, we establish the correctness of the algorithm and derive the time complexity.

THEOREM 2.1. *Let Γ be a set of m periodic tasks with a utilization factor $U \leq n$. Then a sufficient condition for scheduling Γ on n processors is that $T * (C_i/D_i)$ be integral, for each task Γ_i.*

Proof. We execute task Γ_1 for $T*C_1/D_1$ time units, task Γ_2 for $T*C_2/D_2$ time units, ... , task Γ_m for $T * C_m/D_m$ time units in every block. Let $N_i = T * C_i/D_i$. Since $C_i/D_i \leq 1$, therefore, $N_i \leq T$, for each i. Every task Γ_i is executed for $N_i * D_i/T$ or C_i time units before its deadline D_i. (By the definition of T, D_i/T must be an integer). Now, $\sum_{i=1}^{m} N_i = T * U \leq n * T$. This shows that each task Γ_i can be allocated C_i units of time before its deadline D_i. We need only to prove that there is some schedule satisfying constraints (A1) and (A2).

We claim that algorithm **SA1** constructs a valid schedule of the task set Γ for any single block. To prove the claim, we need to show only that constraints (A1) and (A2) are never violated. Clearly, **SA1** allots any single time unit $t_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq T$, to at most one task, thereby satisfying (A1). Also, since the task time slice in any block is at most T, therefore, if

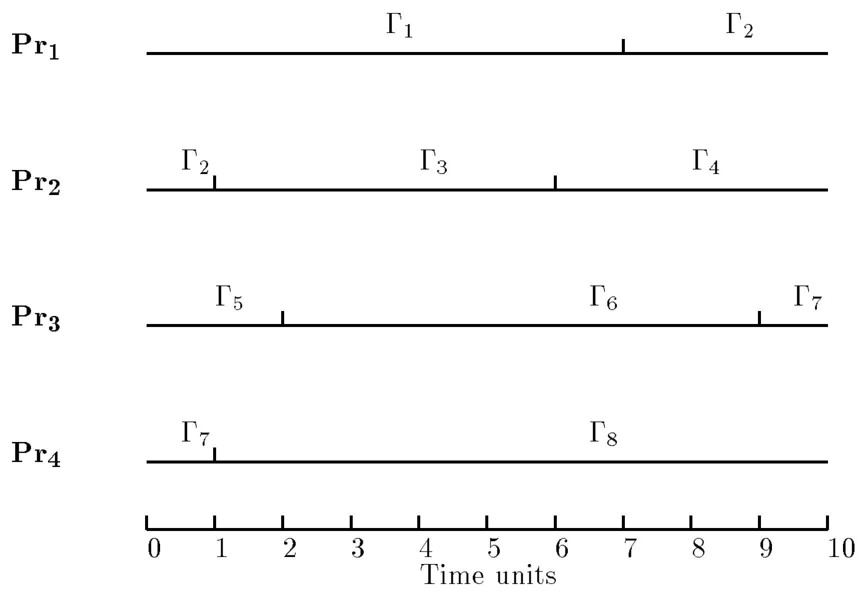

FIG. 1. *Schedule of TABLE 2.1.1 as constructed by* **SA1** *for the first block*

a time unit $t_{i,j}$ is allotted to a task by **SA1**, no other time unit of the type $t_{i',j}$ is allotted to it, for any $i' \neq i$. This is obvious from the construction of the schedule by **SA1**. Hence, constraint (A2) is also satisfied. ■

NOTE: Mok and Dertouzous (cf. [4]) claim that the integer task time slices are sufficient for feasible scheduling in a multiprocessor environment. However, their proof (Theorem 8) is incomplete because the schedule constructed by the theorem is not shown to satisfy constraint (A2).

LEMMA 2.2. *Algorithm* **SA1** *takes $\mathcal{O}(m)$ time to build a complete schedule on n processors for a set of m tasks having integral time slices. Also, the total number of preemptions in any interval of length T is at most $(m+n-1)$.*

Proof. Clearly one pass of algorithm **SA1** takes $\mathcal{O}(1)$ time. To build a schedule for a single block, the algorithm SA1 will have to make m passes, where m is the number of tasks. The schedule of a single block so constructed by **SA1** can be duplicated every T time units. Hence the complexity of SA1 is $\mathcal{O}(m)$.

To get an upper bound on the number of task preemptions in a single block, we note that all m tasks are loaded at least once in every block. **SA1** can preempt at most $(n-1)$ tasks in step 2 when a task is scheduled on two processors. Also, tasks are allotted contiguous time slots of any single processor in any given block. Hence the lemma. ∎

COROLLARY 2.3. *There are at most $2 * D_j/T$ preemptions for any task Γ_j.*

Proof. The corollary follows trivially from the observation that there are a total of D_j/T blocks during the existence of any instance of the task Γ_j and any task is preempted at most once in any single block.
This upper bound is useful in computing the worst case computation time of a task off-line. ∎

2.2. Cyclic Permutation Of Processors. In Section 1, we assumed that the computation times of tasks include the costs of preemption. Thus, it is essential to reduce the upper bound on the number of task preemptions as it leads to a reduction in the worst case computation times of tasks. We can obtain a better upper bound for the number of task preemptions by reallocating processors to tasks in succeeding blocks as follows. If a task is preempted at a processor and re-scheduled at some other processor in step 2 of algorithm **SA1**, then this extra preemption of a task within a single block can be compensated by reallocating processors to tasks in the succeeding block as follows. If some task k is preempted by algorithm **SA1** in the first block, then the time units allotted to it are $[t_{i+1,1},\ t_{i+1,j+N_k-T-1}]$ and $[t_{i,j},\ t_{i,T}]$, for some processor i and time unit j. Such a task is said to be preempted on processor $(i+1)$ at time unit $(j + N_k - T - 1)$ and rescheduled on processor i at time unit j. By allocating to processor i[4] in the next block, the present schedule of processor $(i+1)$, we can compensate for the extra preemption of task k on processor $(i+1)$ in the first block by ensuring contiguous allocation of time units on processor i between the first and second blocks.
Formally, the schedule of a block (other than the first block) is determined from the schedule of the previous block as follows. Let $\{(i_1+1, i_1+2, ..., j_1-1, j_1),\ (i_2+1, i_2+2, ..., j_2-1, j_2),\ \dots, (i_l+1, i_l+2, ..., j_l-1, j_l)\}$ be the processors where a task is preempted and re-scheduled on a second processor in the first block. Here $0 < i_1 < j_1 < i_2 < j_2 < \ \dots\ < i_l < j_l \leq n$. The schedule is obtained by taking the following cyclic permutation of the processors, $\{(i_1, i_1+1, ..., j_1-1, j_1),\ (i_2, i_2+1, ..., j_2-1, j_2),\ \dots, (i_l, i_l+1, ..., j_l-1, j_l)\}$,

$$\{(i_1+1, i_1+2, ..., j_1, i_1),\ (i_2+1, i_2+2, ..., j_2, i_2),\ \dots, (i_l+1, i_l+2, ..., j_l, i_l)\}.$$

The cyclic permutation of $(i_1, i_1+1, ..., j_1)$ to $(i_1+1, i_1+2, ..., j_1, i_1)$ means that the schedule of processor i_1 in a block is the same as the schedule of

[4] If $i = n$, then allot the schedule of processor 1.

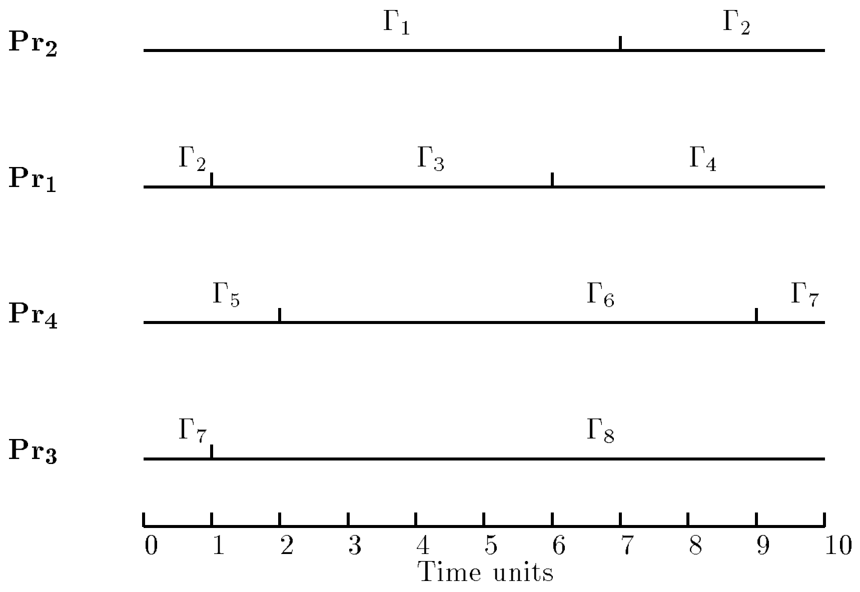

FIG. 2. *Schedule of TABLE 2.1.1 as constructed by modified* **SA1** *for the second block*

processor $(i_1 + 1)$ in the previous block, the schedule of processor $(i_1 + 1)$ in a block is the same as the schedule of processor $(i_1 + 2)$ in the previous block, ... , and the schedule of processor j_1 in a block is the same as the schedule of processor i_1 in the previous block. An example of processor cycles in the schedule of TABLE 2.1.1 as constructed by algorithm **SA1** (Fig. 1). Schedules for next blocks are constructed by taking the (2,1) and (4,3) cyclic permutation of the processor cycles (1,2) and (3,4) in Fig. 1 respectively. The corresponding cyclic permutations of processors follow trivially from Fig. 1. The schedule of the second block resulting from such a cyclic permutation of the schedule of the first block is shown in Fig. 2. The combined schedule of the first two blocks is shown in Fig. 3.

If no task is allotted more than one processor by algorithm **SA1**, then there exists no processor cycles in the schedule of the first block. Hence, the set of tasks can be partitioned into different processors, such that each task is scheduled to completion by just its allotted processor. In such a case, the schedule is simply repeated for succeeding blocks and no cyclic permutation of processors is needed.

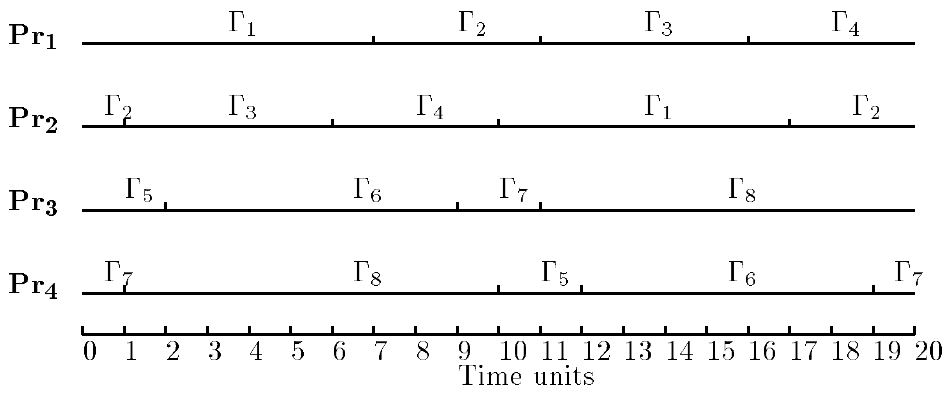

FIG. 3. *Schedule of TABLE 2.1.1 for the first two blocks*

LEMMA 2.4. *The modified* **SA1** *algorithm takes $\mathcal{O}(m+n)$ time to build a schedule on n processors for a set of m tasks having integral time slices. Also, the total number of preemptions of tasks in any interval of length D is at most $(m * D/T + \sum_{j \in \Delta}(D/D_j - 1))$, for some task set Δ containing at most $(n-1)$ tasks.*

Proof. Algorithm **SA1** takes $\mathcal{O}(m)$ time to construct a schedule for the first block. The schedules for succeeding blocks is obtained by determining the processor cycles in the schedule of the first block, which takes $\mathcal{O}(n)$ time. Hence it takes $\mathcal{O}(m+n)$ time to build a complete schedule.
The total number of times tasks are loaded in an interval of length D is $m * D/T$, as m tasks are loaded in each block. The preemption of a task in a block is compensated in the modified **SA1** algorithm by allocating contiguous time units to the tasks between consecutive blocks. However we need to account for those task preemptions in a block when there is a new occurrence of the task in the succeeding block. This happens $(D/D_j - 1)$ times for task Γ_j in an interval of length D. There is at most $(n-1)$ tasks which are preempted in a single block, hence the result on the maximum number of possible preemptions in an interval of length D using the modified **SA1** algorithm. ∎

COROLLARY 2.5. *There are at most $(D_j/T + 1)$ preemptions for any task Γ_j using the modified* **SA1** *algorithm.*

Proof. The corollary follows trivially from the simple observation that only the preemption, if any, within the first block of the task's existence is left unaccounted. All succeeding preemptions within a block are accounted for by allotting contiguous time units to the task between two consecutive blocks. ∎

Hence, by the method of cyclic permutation of processors, we have been

TABLE 2

Task Requirements

Tasks: 6; Processors:3

$T = GCD[10, 20, 30, 20, 30, 60] = 10$

$D = LCM[10, 20, 30, 20, 30, 60] = 60$

Task No.	D_i	C_i	$T * C_i/D_i$
1	10	6	6
2	20	11	$5\frac{1}{2}$
3	30	23	$7\frac{2}{3}$
4	20	6	3
5	30	5	$1\frac{2}{3}$
6	60	37	$6\frac{1}{6}$

able to reduce the upper bound on the number of task preemptions in an interval of length D from $(m+n-1)*D/T$ to $(m*D/T+\sum_{j\in\Delta}(D/D_j-1))$, for some task set Δ containing at most $(n-1)$ tasks. Therefore the upper bound is reduced by $((n-1)*D/T-\sum_{j\in\Delta}(D/D_j-1))$ preemptions. It is easy to see that using the modified **SA1** algorithm, the upper bound is reduced by at least $(n-1)$ preemptions irrespective of the composition of the task set Δ. Also the upper bound on the number of preemptions for any task Γ_j is reduced by $(D_j/T-1)$. Hence, the worst case computation time of a task is reduced by the modified **SA1** algorithm.

3. Non Integral Task Time Slices. The hard real-time environment may not necessarily consist of tasks with integral time slices only. There may be some tasks with non-integral computational requirements per block. This implies that we cannot apply algorithm **SA1** to such a task set, as we cannot allot non-integral clock ticks to tasks in a block. An example of such a task set is shown in TABLE 3. In this section, we shall build a schedule for the case when tasks have non integral time slices. Let $N_i = \lfloor T*C_i/D_i \rfloor$[5] and $F_i = T*C_i/D_i - \lfloor T*C_i/D_i \rfloor$, for each task Γ_i. Clearly, $T*C_i/D_i = N_i+F_i$ and $0 \leq F_i < 1$. We apply algorithm **SA2** to such a task set, so as to allot integral time units to tasks in a block.

Let the requirement of a task Γ_k before a block be r_k time units. $r_k = T*C_k/D_k$, for each task Γ_k at the start of the first block. In general r_k is non-integral and its value is different for different blocks. r_k is a constant for all blocks iff $F_k = 0$. Let $r_k = n_k + f_k$, where $n_k = \lfloor r_k \rfloor$, and $f_k = r_k - \lfloor r_k \rfloor$.

[5] Greatest integer less than or equal to.

TABLE 3

Allotment of time units to tasks of TABLE 3 by **SA2**

a_i denotes the integral time units allotted to task Γ_i in a block.

Block	Γ_1		Γ_2		Γ_3		Γ_4		Γ_5		Γ_6	
No.	r_1	a_1	r_2	a_2	r_3	a_3	r_4	a_4	r_5	a_5	r_6	a_6
1	6	6	$5\frac{1}{2}$	6	$7\frac{2}{3}$	8	3	3	$1\frac{2}{3}$	1	$6\frac{1}{6}$	6
2	6	6	5	5	$7\frac{1}{3}$	8	3	3	$2\frac{1}{3}$	2	$6\frac{1}{3}$	6
3	6	6	$5\frac{1}{2}$	6	7	7	3	3	2	2	$6\frac{1}{2}$	6
4	6	6	5	5	$7\frac{2}{3}$	8	3	3	$1\frac{2}{3}$	2	$6\frac{2}{3}$	6
5	6	6	$5\frac{1}{2}$	6	$7\frac{1}{3}$	8	3	3	$1\frac{1}{3}$	1	$6\frac{5}{6}$	6
6	6	6	5	5	7	7	3	3	2	2	7	7

3.1. Scheduling Algorithm SA2. The steps of the algorithm are described below:

1. $F \leftarrow n * T - \sum_{i=1}^{m} n_i$.
   (F denotes the number of free time units in a block after meeting integer time unit requirements of all tasks.)
2. Choose the first F tasks from the list of tasks having current $f_i > 0$, and $n_i < T$.
   (If there are less than F tasks, then all tasks satisfying the above two conditions are chosen.)
   Allot to these chosen tasks $(n_i + 1)$ time units, and to all other tasks n_i units.
3. Apply algorithm **SA1** using integer time units of tasks so computed in step (2).
4. **If** task Γ_i is allotted $(n_i + 1)$ time units in step (2)
   then $r_i = (N_i + F_i) - (1 - f_i)$
   else $r_i = (N_i + F_i) + f_i$

The algorithm **SA2** is applied D/T times to get a schedule for an interval of D time units, which is sufficient to determine the schedule completely. An example showing the allotment of time units to tasks having non-integral time slices is shown in TABLE 3.1.

LEMMA 3.1. *The algorithm* **SA2** *takes $\mathcal{O}(m * D/T)$ time to build a complete schedule for a set of m tasks on n processors, with task computation times C_i's and deadlines D_i's satisfying usual conditions*[6] *. Also, the total number of task preemptions in any interval of length D is at most $(m + n - 1) * D/T$.*

Proof. The proof is the same as for Lemma 2.2, but here we have to consider an interval of size D. This is because tasks may be allotted different time units in a block, due to the non-integral time slices of tasks. We obtain a complete schedule by just considering D/T blocks from which the lemma follows. ■

COROLLARY 3.2. *There are at most $2 * D_j/T$ preemptions for any task Γ_j.*

Proof. Same as for Corollary 2.3. ■

3.2. Sufficient Conditions For Feasible Schedulability. In this section, we state and prove two sufficient conditions for algorithm **SA2** to be optimal, i.e., to produce a valid, feasible schedule whenever there exists one, when tasks have non-integral time slices.

The following two observations would be helpful in understanding the proofs of Theorems 3.3 & 3.4:

OBSERVATIONS :

O1. If $T * C_i/D_i$ is an integer for some task Γ_i, then $r_i = T * C_i/D_i$ at the start of each block.
This is because algorithm **SA2** allots exactly r_i time units to task Γ_i in each block.

O2. $(T * (C_i/Di) - 1) < r_i < (T * (C_i/Di) + 1)$, for all tasks Γ_i and at the start of each block.
This is obvious from step (4) of algorithm **SA2** where you either subtract $(1 - f_i)$ or add f_i to the task's original requirement of $T * C_i/D_i$ time units per block.

THEOREM 3.3. *Let Γ be a set of m periodic tasks with a utilization factor $U \leq n$. Then a sufficient condition for scheduling Γ on n processors using algorithm* **SA2** *is $(C_i/D_i) \leq (1 - 1/T)$, or $C_i = D_i$, for each task Γ_i.*

Proof. The following two assertions hold at the start of each block when algorithm **SA2** is applied to such a task set :

1. $-1 < r_i \leq T$, for all tasks Γ_i.
2. $\sum_{i=1}^{m} r_i \leq n * T$.

Induction : Initially, at the start of the first block, (1) and (2) both hold, because $r_i = T * C_i/D_i$. Suppose (1) and (2) hold at the start of some block k. We shall now prove that (1) and (2) also hold at the start of block $(k+1)$.

[6] $C_i \leq D_i$, for each task Γ_i, and $U \leq n$.

Suppose $C_i = D_i$ for some task Γ_i. Then, $T * C_i/D_i = T$ is an integer and from observation (O1) above, $r_i = T$ at the start of block $(k+1)$ also. Next, suppose $C_i/D_i \leq (1 - 1/T)$. This implies $0 < T * C_i/D_i \leq (T-1)$. From observation (O2) above, it follows that $-1 < r_i \leq T$ at the start of each block. Therefore, assertion (1) holds at the start of block $(k+1)$ and is hence an invariant.

To prove assertion (2), we observe that if no time unit of block k remains idle, i.e., if all time units of block k are allotted to some task by algorithm **SA2**, then (2) holds at the start of block $(k+1)$ also. This is because $\sum_{i=1}^{m} r_i{}^{7} - \sum_{i=1}^{m} r_i{}^{8} = \sum_{i=1}^{m}(N_i + F_i) - n * T \ = U - n * T \ \leq \ 0$.
Now suppose that there is some idle time unit in block k. This implies that at the start of block $(k+1)$, $r_i \leq N_i + F_i$, for each task Γ_i, as all tasks are either computed fully or in excess of their requirements in block k. This is because algorithm **SA2** allots at least r_i time units to each task Γ_i at the start of block k. This is possible by the invariant (1) above, as no task's requirement exceeds T time units. Hence, in this case too, at the start of block $(k+1)$, $\sum_{i=1}^{m} r_i \ \leq \ \sum_{i=1}^{m}(N_i + F_i) \ \leq n * T$.
Therefore, assertion (2) also holds at the start of each block.

We observe that the time units allotted to tasks by algorithm **SA2** in each block satisfy the conditions of Theorem 2.1, therefore, an actual schedule can be built by **SA2** for each block using algorithm **SA1**. It only remains to be proved that no deadlines are violated or no task is computed before its arrival. This is trivial from the observation that if a fractional amount (less than one) of a task is not computed in some block, then the deadline for that task has not been crossed at the completion of that block. This is because the total computation requirement C_i of a task Γ_i before its deadline D_i and the time units allotted to it so far are both integral. For the same reason, if a task Γ_i is computed in excess of its requirement r_i in a block, there is no danger of computing the next instance of the task before its arrival, as only a proper fractional amount is computed in excess. ∎

An example where the conditions of Theorem 3.3 is satisfied is given in TABLE 3. In real-time system applications, the computation times of tasks are much less than their respective deadlines and the deadlines of tasks are not usually relatively prime. Hence, there is a very good chance of a task set satisfying the conditions of Theorem 3.3.

[7] at the start of block $(k+1)$.
[8] at the start of block k.

THEOREM 3.4. *Let Γ be a set of m periodic tasks with a utilization factor $U \leq n$. A sufficient condition for scheduling Γ on n processors using algorithm* **SA2** *is, $\sum_{i=1}^{m} F_i \leq 1$.*

Proof. In this case, the following two assertions hold at the start of each block when algorithm **SA2** is applied to such a task set :

1. $-1 < r_i < (T+1)$, for all tasks Γ_i.
2. $\sum_{i=1}^{m} r_i \leq n * T$.

Induction : Initially, at the start of the first block, both (1) and (2) hold, because $r_i = T * C_i/D_i$.
Suppose (1) and (2) hold at the start of some block k. We shall now prove that (1) and (2) also hold at the start of block $(k+1)$.
From observations (O1) and (O2) above, assertion (1) holds at the start of block $(k+1)$, because non-integral task time slices cannot exceed T time units.

As in Theorem 3.3, if no time unit of block k remains idle, i.e., if all time units are allotted to some task by algorithm **SA2**, then assertion (2) holds at the start of block $(k+1)$ also.
Now, suppose that there is some idle time unit in block k. All tasks Γ_i with their r_i values less than or equal to T at the start of the block, are either computed fully or are computed in excess. If there is no task whose requirement at the start of block k is greater than T, then assertion (2) again holds at the start of block $(k+1)$.
So let there be some task Γ_j whose requirement, r_j, at the start of block k is greater than T and also let there be some idle time unit in block k. If for a task, say task Γ_i, the requirement is not greater than T, then it is allotted at least N_i time units in block k by algorithm **SA2**. This is because at the start of block k, its requirement, r_i, is greater than $(N_i - 1)$ time units from observation (O2). Now consider a task, say task Γ_j, with its requirement, r_j, greater than T at the start of block k. Such a task is allotted T or $(N_j + 1)$ time units by algorithm **SA2**. This is because, $N_j = (T-1)$ for such a task Γ_j. Therefore, the total time units allotted by algorithm **SA2** in block k is at least $\sum_{i=1}^{m}(N_i) + 1$. From this it follows that the total requirement at the start of block $(k+1)$ is at most,

$$\sum_{i=1}^{m}(N_i + F_i + r_i{}^{2}) - \sum_{i=1}^{m}(N_i) - 1,$$
$$= \sum_{i=1}^{m}(F_i + r_i) - 1 ,$$
$$\leq \sum_{i=1}^{m} r_i, \quad (\textstyle\sum_{i=1}^{m} F_i \leq 1, \text{ by assumption})$$
$$\leq n * T \quad \text{(by the inductive hypothesis)}$$

Hence assertion (2) holds at the start of block $(k+1)$.

[2] Requirement at the start of block k.

TABLE 4

Tasks:6; Processors: 3

$T = GCD[10, 20, 30, 20, 30, 60] = 10$

$D = LCM[10, 20, 30, 20, 30, 60] = 60$

Task No.	D_i	C_i	$T * C_i/D_i$
1	10	6	6
2	20	11	$5\frac{1}{2}$
3	30	24	8
4	20	6	3
5	30	4	$1\frac{1}{3}$
6	60	37	$6\frac{1}{6}$

Therefore, both assertions (1) and (2) hold at the start of each block.
Here again, as in Theorem 3.3, the time units allotted by algorithm **SA2** to tasks in each block satisfy the conditions of Theorem 2.1. Therefore, a schedule can be built for each block using the allotted time units. As in Theorem 3.3, no task deadlines are violated and no task is computed before its arrival. Hence the theorem. ∎

An example of a task set satisfying the constraints of Theorem 3.4 is shown in TABLE 3.2.

4. Modifying Schedules On-Line. In a system that is static, the characteristics of the real-time system are assumed to be known *a priori*, and, hence, the schedule can be built at compile time. Such systems are quite inflexible even though they may incur low run-time (not necessarily constant) overheads. In practice, most applications involve a number of components that can be statically specified along with many dynamic components. A proper design should ensure high resource utilization and low overheads for such applications. Whereas a large proportion of current real-time systems are static in nature, next generation systems will have to adopt more dynamic and flexible solutions. Most of the algorithms which are optimal for static scheduling are not optimal for dynamic scheduling. In particular, Mok and Dertouzos [4] showed that for multiprocessor systems, there can be no optimal algorithm for scheduling preemptable tasks if either the arrival time or deadline or computation time of tasks is not known *a priori*. They also showed that, if the set of all possible tasks that will ever arrive in a system can be scheduled initially, then the set can be scheduled at run-time also. The use of this approach is limited, because in most dynamic systems, the probability that all possible arriving tasks can be scheduled initially is low. We now show how to modify the schedules built at compile-time when either new tasks arrive or existing tasks leave the system on-line at little run-time cost.

4.1. Integral Task Time Slices. First consider the case of integral task time slices. Assume that new tasks arrive at time units $T, 2*T, 3*T, ...,$ and so on, and their deadlines are integral multiples of the block length T. The scheduling algorithm **SA1** is applied when some new tasks arrive on-line. If no current task leaves the system, then one need not run algorithm **SA1** all over again. The variables i and j in algorithm **SA1** denoting the processor and the time unit respectively are kept as global variables and the schedule for a new task is obtained by just a single pass of algorithm **SA1**. Hence, a new task is schedulable in $\mathcal{O}(1)$ time. If some current task leaves the system, then an attempt is made to allot the time units corresponding to each single outgoing task to one or more new tasks. But if a new task's requirement per block exceeds the time units allotted to any of the outgoing tasks, then algorithm **SA1** has to be applied to the whole task set in order to guarantee the new task and to maintain the properties of the schedule on the maximum number of preemptions allowable for any task and during any fixed time interval.

4.2. Non-Integral Task Time Slices. Now consider the case of non-integral task time slices. Here we assume that new tasks arrive at time units $D, 2*D, 3*D, ...,$ and so on, and their deadlines are integral multiples of the block length T. Suppose that the total idle time units in the first block after steps (1) and (2) of algorithm **SA2** are applied to the original task set is at least $\lceil T*C_l/D_l \rceil$[3], where Γ_l denotes the new task arriving on-line. For such a task set, task Γ_l needs no more than $\lceil T*C_l/D_l \rceil$ time units in each block, and there are at least $\lceil T*C_l/D_l \rceil$ idle time units in each block after allotting time units to the tasks in the original task set in step (2) of algorithm **SA2**. The variables i and j denoting the processor and the time unit respectively are kept as global variables for each of the D/T blocks and the schedule for the new task is obtained by just a single pass of algorithm **SA1** for each block. Hence, the schedule for each block can be revised in just $\mathcal{O}(1)$ step. Since the schedule needs to be revised over D/T blocks, i.e., for an interval of length D, the complete revised schedule is obtained in just $\mathcal{O}(D/T)$ time. If some current task leaves the system or when the requirements of a new task entering the system does not satisfy the property above, then there is no simple method by which the existing schedule can be revised at little run-time cost, and algorithm **SA2** has to be applied to the complete task set.

[3] Least integer greater than or equal to.

5. Conclusion. In this paper, we have given for the first time linear multiprocessor scheduling algorithms which compute feasible, preemptive schedules at compile time and also arrived at upper bounds on the number of preemptions for any single task and during a fixed time interval. We have also shown in certain cases a schema for modifying existing schedules at little run-time cost when new tasks arrive on-line or when existing tasks leave the system. Our results come closer to the open problem: whether any idle time of the multiprocessor system could be used for feasible scheduling of new periodic tasksΓ Whether our scheduling algorithm fails for tasks not satisfying the sufficient conditions of & 3 remains open. It would appear that the **SA1** and **SA2** algorithms defined in this paper are also non-preemptive scheduling algorithms. The algorithms compute the actual time-slots at which a task must execute, and one could use this for non-preemptive scheduling as opposed to preemptive scheduling. But non-preemptive scheduling has its problems when the actual run times of tasks is less than its worst case computation time. It is shown in [6, 10] that the schedule length may increase, if the actual run-time of some task is less than its worst case computation-time, which may result in some tasks missing their deadlines. Since algorithm **SA1** caters to the situation where each time slice is an integer, one way of applying **SA1** to all task sets is to artificially increase the computation times of tasks when necessary to ensure that the corresponding time slice is an integer. This entails lesser processor utilization, but then it becomes very easy to guarantee new tasks on-line. The primary practical problem with the **SA1** and **SA2** algorithms is that it is based on the GCD and the LCM of the task periods. In order to keep these numbers manageable (for instance consider task periods which are relatively prime), task periods may have to be shortened (lengthening of task periods may not be possible because of application requirements), and the utilization of the task set is only artificially increased. A small GCD (of a single clock tick) can cause the schedule to be long and tedious to manage. The fact that tasks can execute on more than one processor during their existence means that tasks may have to be sent from one processor to another. The communication time between two processors is taken care of by the fact that the algorithms **SA1** and **SA2** allot time units to a preempted task within a block in such a way that there are $(T - T * C_i/D_i)$ time units available for communication to take place between the processors. If the above time units is not enough to allow for communication, then we may schedule the task on a single processor, keeping the remaining time units of the processor idle. So there is a tradeoff between achieving high multiprocessor utilization and avoiding high communication costs. In this paper, we have assumed that tasks in a hard real-time environment are independent. However simple precedence constraints among tasks can be modeled as follows. Let a task Γ_i occur after a fixed number (say n) of occurrences of another task Γ_j. This is modeled by choosing the periods of tasks Γ_i and Γ_j so that the period of Γ_i is n times the period of Γ_j and the $(n+1)^{th}$ request for Γ_j coincides with the 1^{st} request for Γ_i, and so on.

The fact that the multiprocessor can be fully loaded and still guarantee task service requirements within a preassigned time has tremendous scope in real-time applications. Most real-time tasks arrive on-line and a good on-line strategy is therefore needed to determine its schedulability and then to arrive at a schedule in good time. Current methods of determining the schedule of a new task arriving on-line in a multiprocessor environment is based on heuristics. In fact, under the constraints of the task characteristics as discussed above, the criterion that the multiprocessor utilization be not more than its capacity to guarantee feasible scheduling of a new, periodic task is an optimal one. An optimal design for next generation real-time systems would be to revise the schedules dynamically at little run-time cost. We feel that our approach has potential for multiprocessor hard real-time on-line scheduling, when the arrival times and deadlines of newly arriving tasks are truly random. The work in this direction is progressing. Further, the preliminary work related to scheduling for imprecise computations and its relation to reclaiming the time left unutilized in a schedule (cf. [10]) is reported in [8].

Acknowledgements. The authors thank the referees for their suggestions.

REFERENCES

[1] J.A. Bannister and K.S. Trivedi, *Task Allocation In Fault-Tolerant Distributed Systems*, Acta Informatica, Springer-Verlag, 1983.

[2] S. Davari and S.K. Dhall, *An On-Line Algorithm For Real-Time Tasks Allocation*, IEEE Real-Time Systems Symposium, Dec. 1986.

[3] S.C. Cheng, J.A. Stankovic and K. Ramamritham, *Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey*, in Tutorial on Hard Real-Time Systems, edited by J.A. Stankovic and K. Ramamritham, IEEE Press, 1988.

[4] M.L. Dertouzos, and Aloysius Ka-Lau Mok, *Multiprocessor On-Line Scheduling Of Hard-Real-Time Tasks*, IEEE Transactions on Software Engineering, vol. 15, no. 12, Dec.1989.

[5] S.K. Dhall and C.L. Liu, *On A Real-Time Scheduling Problem*, Operations Research, vol. 26(1), 1978.

[6] M.R. Garey and Johnson, D.S., *Complexity Results For Multiprocessor Scheduling Under Resource Constraints*, SIAM Journal of Computing 4, 1975.

[7] R.L. Graham, *Bounds For Certain Multiprocessor Anomalies*, Bell Syst. Tech. J. 45 (1966), 1563-1581.

[8] A. Khemka, K.V. Subrahmanyam and R.K. Shyamasundar, *Multiprocessor Scheduling for imprecise computations in a hard real-time environment*, to be presented at IPPS 93, New Port Beach, California, April 1993.

[9] C.L. Liu, and James W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Assoc. for Computing Machinery, vol. 20, no. 1, Jan. 1973.

[10] G.K. Manacher, *Production and Stabilization of Real-Time Task Schedules*, Journal of the Assoc. for Computing Machinery, vol. 14, no.3, July 1967.

[11] R.R. Muntz and E.G. Coffman, *Preemptive Scheduling Of Real-Time Tasks On Multiprocessor Systems*, Journal of the Assoc. for Computing Machinery, vol. 17, no. 2, April 1970.

[12] J.A. Stankovic, *Real-Time Computing Systems: the Next Generation*, in Tutorial on Hard Real-Time Systems, edited by J.A. Stankovic and K. Ramamritham, IEEE Press, 1988.