Operations Research

On a Real-Time Scheduling Problem

Sudarshan K. Dhall, C. L. Liu,

To cite this article:
Sudarshan K. Dhall, C. L. Liu, (1978) On a Real-Time Scheduling Problem. Operations Research 26(1):127-140. http://dx.doi.org/10.1287/opre.26.1.127

OPERATIONS RESEARCH
Vol. 26, No. 1, January-February 1978

0030-364X/78/2601-0127 $01.25
© 1978 Operations Research Society of America

On a Real-Time Scheduling Problem

SUDARSHAN K. DHALL
Murray State University, Murray, Kentucky

C. L. LIU
University of Illinois, Urbana, Illinois

(Received April 1976; accepted July 1977)

We study the problem of scheduling periodic-time-critical tasks on multiprocessor computing systems. A periodic-time-critical task consists of an infinite number of requests, each of which has a prescribed deadline. The scheduling problem is to specify an order in which the requests of a set of tasks are to be executed and the processor to be used, with the goal of meeting all the deadlines with a minimum number of processors. Since the problem of determining the minimum number of processors is difficult, we consider two heuristic algorithms. These are easy to implement and yield a number of processors that is reasonably close to the minimum number. We also analyze the worst-case behavior of these heuristics.

WE STUDY a problem in which time-critical tasks are to be scheduled on a multiserver system. Our study is motivated by a situation in which computers are used in the control and monitoring of industrial processes to execute certain tasks in response to external signals and to guarantee that each task is completely executed prior to or at a prescribed deadline. The efficient use of computers in this case can only be achieved by a careful scheduling of the time-critical tasks. The problem of devising efficient scheduling algorithms for handling time-critical tasks is of both theoretical and practical interest. A number of authors [3-5] have studied the problem of devising algorithms for scheduling time-critical tasks on a single-processor computing system. We shall study some aspects of the problem of scheduling time-critical tasks on multiprocessor computing systems.

1. THE MODEL

A *time-critical task* consists of a certain number of *requests*, each of which has a prescribed deadline. The scheduling problem is to specify an order in which the requests of a set of tasks are to be executed with the goal of meeting all the deadlines. In this paper we assume the tasks to be scheduled have the following characteristics:

1. The requests of each task are periodic, with constant interval between requests.
2. The deadline constraints specify that each request must be completed before the next request of the same task occurs.
3. The tasks are independent in that the requests of a task do not depend on the initiation or the completion of the requests of other tasks.
4. Run-time for the requests of a task is constant for the task. Run-time here refers to the time a processor takes to execute the request without interruption.

It follows that a task is completely defined by two numbers, the *run-time* of a request and the *request period*. We shall denote a task τ by an ordered pair (C, T), where C is the run-time and T the request period. The *request rate* of a task is defined to be the reciprocal of its request period, and the *utilization factor* of a task (C, T) is defined to be the ratio C/T. The *utilization factor of a set of tasks* is the sum of the utilization factors of the tasks in the set.

A scheduling algorithm provides a set of rules that determine the processor(s) to be used and the task(s) to be executed at any particular point in time. One simple way to specify a scheduling algorithm is to assign priorities to requests so that a higher-priority request has precedence over a lower-priority request. Thus the specification of such algorithms amounts to the specification of a method of assigning priorities to requests. We will consider only *preemptive priority-driven scheduling algorithms.* In these algorithms priorities are assigned to the requests of the tasks, and currently running requests of lower priorities will be preempted whenever requests of higher priorities come in, even though the current lower-priority requests have not been completed. The interrupted requests will be resumed later. We shall assume that the time required to switch a processor from one task to another is zero. This assumption is not unrealistic and is a very good first approximation since the switching time is comparatively small. Moreover, since the run-time of a request can be interpreted as the maximum processing time of the request, switching time due to preemption can be taken into account in the run-time.

Priority-driven scheduling algorithms can be classified into two categories: *static* or *fixed* priority algorithms, and *dynamic* priority algorithms. A *fixed-priority algorithm* is one in which priorities are assigned to tasks and all requests of a task of higher priority have precedence over all requests of a task of lower priority. In a *dynamic-priority algorithm* priorities are assigned to requests of tasks.

We say that a set of tasks can be *feasibly scheduled* by an algorithm, if according to the algorithm the deadlines of all requests will be met. On the other hand, if any of the requests is not fulfilled before its deadline, the set of tasks is said to be *infeasible* with respect to that algorithm.

We shall assume that the first requests of all tasks occur simultaneously at time $t=0$. This assumption takes care of the worst possible case in the sense that if a set of tasks can be feasibly scheduled by a priority-driven scheduling algorithm when first requests of all tasks occur simultaneously, that set can also be feasibly scheduled when the first requests are phased out.

2. SCHEDULING ON A SINGLE-PROCESSOR COMPUTING SYSTEM

A number of authors [3–5] have studied the problem of devising algorithms for scheduling time-critical tasks on a single processor computing system. Liu and Layland [4] studied a fixed-priority scheduling algorithm, called the rate-monotonic priority assignment algorithm, and a dynamic-priority algorithm, called the deadline-driven scheduling algorithm. Serlin [5] refers to the first one as the intelligent fixed-priority algorithm and to the latter as the relative urgency algorithm. In the rate-monotonic priority algorithm, priorities are assigned to tasks according to their request rates. Tasks with high request rates are assigned high priorities. According to the deadline-driven scheduling algorithm, priorities are assigned to tasks on the basis of the deadline of their current requests. A task is assigned the highest priority if the deadline of its current request is nearest. At any instant the task with the highest priority and yet unfulfilled request is executed. Ties are broken arbitrarily. Liu and Layland [4] also considered a mixed scheduling algorithm that is a combination of the rate-monotonic scheduling algorithm and the deadline-driven scheduling algorithm. According to this algorithm, k tasks that have the k shortest request periods are scheduled according to the fixed-priority rate-monotonic scheduling algorithm, and the remaining tasks are scheduled according to the deadline-driven scheduling algorithm when the processor is not occupied by the k tasks that have the k shortest request periods. It was shown that among all fixed-priority scheduling algorithms the rate-monotonic priority assignment algorithm is a best possible one in the sense that if a set of tasks can be feasibly scheduled according to some fixed-priority scheduling algorithm, that set can also be feasibly scheduled when priorities are assigned to tasks on the basis of their request rates [4].

We recall that the utilization factor of a task (C, T) is defined to be the ratio C/T, which is the fraction of the processor time taken up by the task. Clearly, if the utilization factor of a set of tasks is greater than 1, it will be impossible to feasibly schedule this set of tasks on a single-processor computing system using any algorithm. Hence a necessary condition for a set of tasks to be feasibly scheduled on a single-processor computing system is that its utilization factor be less than or equal to 1.

If tasks are scheduled according to the deadline-driven scheduling algorithm, this condition is also sufficient to guarantee that the task set can be feasibly scheduled [4]. However, in some other algorithms this condition is not sufficient to guarantee that the task set can be feasibly scheduled. To study sufficient conditions for feasible scheduling, we introduce the concept of full utilization. A set of tasks is said to *fully utilize* the processor according to a certain scheduling algorithm if the set of tasks can be feasibly scheduled and any increase in the run-time of any of the tasks will make the task set infeasible with respect to that algorithm. For a given algorithm, the *minimum achievable utilization* is the minimum of the utilization factor over all task sets that fully utilize the processor. This means that any task set whose utilization factor is less than or equal to the minimum achievable utilization can always be feasibly scheduled by the corresponding algorithm. (Sets of tasks with larger utilization factors may or may not be feasibly scheduled according to that algorithm.) Some results about the rate-monotonic scheduling algorithm are given below.

THEOREM 1 [4]. *If a set of m tasks is scheduled according to the rate-monotonic scheduling algorithm, then the minimum achievable utilization factor is* $m(2^{1/m}-1)$.

Note that according to Theorem 1, as m approaches infinity, the minimum achievable utilization approaches ln 2.

THEOREM 2 [1]. *Let* $\tau_1, \tau_2 \cdots, \tau_m$ *be a set of m tasks with* $T_1 \leqq \cdots \leqq T_m$. *Let* $u=\sum_{i=1}^{m-1} C_i/T_i \leqq (m-1)(2^{1/(m-1)}-1)$. *If* $C_m/T_m \leqq 2(1+u/(m-1))^{-(m-1)}-1$, *then the set can be feasibly scheduled by the rate-monotonic scheduling algorithm. As m approaches infinity, the minimum utilization factor of τ_m approaches* $2\,exp(-u)-1$.

3. SCHEDULING ON MULTIPROCESSOR COMPUTING SYSTEMS

Recent progress in hardware technology and computer architecture has led to the design and construction of computer systems that contain a large number of processors. Consequently, it is both of practical and theoretical importance to investigate how to make best use of multiprocessor computing systems for the type of tasks considered. One would naturally hope that a simple extension of the algorithms developed for single-processor systems would give satisfactory results. Unfortunately, this is not the case.

According to a preemptive priority-driven scheduling algorithm for multiprocessor computing systems, all tasks (or requests of tasks) are assigned priorities, and at any instant a processor is free it is assigned to an *active* request (i.e., a request that is ready for execution but not yet executed) of highest priority. Also, if at any instant there is a request of priority higher than that of some request being executed, then this request preempts a request of lowest priority. A processor is left free for a certain period of time if and only if there is no active request within that time period. If we assign priorities to the tasks either according to the rate-monotonic priority assignment or according to the deadline-driven algorithm, then both these algorithms, which perform well for single-processor systems, behave poorly for multiprocessor systems. For example, consider the task set consisting of $m-1$ tasks, each with run-time 2ϵ and request period 1, and a task with run-time 1 and request period $1+\epsilon$. If we schedule this set of tasks on an n-processor computing system $(n<m)$, using either the rate-monotonic scheduling algorithm or the deadline-driven scheduling algorithm, the first request of the task with request period $1+\epsilon$ will not be executed before its deadline. The utilization factor of this set of tasks is $U=2(m-1)\epsilon+1/(1+\epsilon)$. As $\epsilon \rightarrow 0$, $U \rightarrow 1$. Thus even with $n>1$ processors, the minimum achievable utilization is less than or equal to 1. However, if the task with request period $1+\epsilon$ is assigned the highest priority, then this set of tasks can be feasibly scheduled. This example shows that the rate-monotonic scheduling algorithm when applied to scheduling a set of tasks on a multiprocessor computing system is not optimal among fixed-priority scheduling algorithms. Neither is the dead-line-driven scheduling algorithm optimal among all scheduling algorithms, as was the case for a single-processor computing system. It is, therefore, worthwhile to look for better scheduling strategies that will lead to more efficient usage of multiprocessor computing systems.

The problem of devising algorithms to schedule a set of tasks on a fixed number of processors is rather difficult. An alternative approach is to partition the tasks into groups so that each group of tasks can be feasibly scheduled on a single processor according to some scheduling algorithm. The scheduling algorithm to be applied to the individual groups in the partition will influence the partitioning process, since each individual group of tasks must have a feasible schedule on a single processor according to the designated algorithm. The problem is thus reduced to determining a partition of a given set of tasks so that, with respect to the designated algorithm, to be used for the groups in the partition, the number of processors is minimum. Since for a finite set of tasks there is only a finite number of possible partitions, this problem can be solved by an exhaustive search. However, the number of possible partitions increases exponentially with an increase in the number of tasks. Such an exhaustive search will, therefore, require considerable computation time and will offset the advantage gained by using an optimal partition. Hence, it is interesting to consider some "good" partitioning scheme. By a "good" partitioning scheme, we mean a scheme according to which the number of processors required is reasonably close to the minimum number of processors needed.

Recall that if the utilization factor of a set of tasks is less than or equal to 1, then the set can be feasibly scheduled on a single-processor computing system according to the deadline-driven scheduling algorithm. The problem of partitioning a set of tasks with respect to the deadline-driven scheduling algorithm is then the same as the bin-packing problem, where it is desired to pack a set of packages into bins of fixed size so that the sum of the sizes of the packages in a bin does not exceed the size of the bin [2]. This can be seen by imagining a task as a package of size equal to its utilization factor and a processor as a bin of size 1. However, partitioning a given set of tasks with respect to the rate-monotonic scheduling algorithm is not the same as the bin-packing problem. This is true since the rate-monotonic scheduling algorithm does not guarantee a feasible schedule on a single processor if the utilization factor of the set is less than or equal to 1. However, there is a more subtle point that should be noted. A seemingly possible approach, using the result in Theorem 1, would be to consider the size of the bins to be ln 2, and then apply the techniques of the bin-packing problem. This approach is, however, false since the utilization factor of a set of tasks that can be feasibly scheduled on a single processor may turn out to be larger than ln 2. The partitioning problem in this case may thus be viewed as a problem of packing packages of size less than 1 into bins whose size varies between ln 2 and 1.

With this background in mind, the problem of finding an algorithm that produces an optimal partitioning for a set of tasks with respect to the rate-monotonic scheduling algorithm turns out to be rather difficult. In view of this, it is interesting to investigate the performance of some heuristic algorithms. We will consider two such heuristic algorithms and in both cases determine bounds on their worst-case performance relative to the optimal partition.

4. THE RATE-MONOTONIC-NEXT-FIT SCHEDULING ALGORITHM

The first algorithm considered is named the *rate-monotonic-next-fit scheduling* (RMNFS) algorithm. According to this algorithm, tasks are first arranged in descending order of their request rates. (For convenience, they are renumbered as $\tau_1, \cdots, \tau_m$.) The assignment scheme is as follows:

1. Set $i=j=1$.
2. Assign task τ_i to processor P_j if this task together with the tasks that have been assigned to P_j can be feasibly scheduled on P_j according to the rate-monotonic scheduling algorithm for a single processor. If not, assign τ_i to P_{j+1} and set $j=j+1$.
3. If $i<m$, set $i=i+1$ and go to step 2; else stop.

The number j so obtained is the number of processors that will be required under this partitioning algorithm.

While assigning tasks to processors according to step 2, one may use the result of Theorem 1. If task τ_i cannot be assigned to processor P_j

according to Theorem 1, the result of Theorem 2 may be used to see whether this assignment is possible. If this test also fails, then one may see by actual simulation whether the task can be assigned to the processor. (For this purpose, one has only to see whether all the requests made up to the largest request period are satisfied [4].)

We now prove

THEOREM 3. *Let N be the number of processors required to feasibly schedule a set of tasks by the RMNFS algorithm, and N_0 the minimum number of processors required to feasibly schedule the same set of tasks. Then as N_0 approaches infinity,* $2.4 \leqq N/N_0 \leqq 2.67$.

Proof. In order to establish the lower bound, we show that given $\epsilon > 0$, there exists a set of tasks for which N/N_0 is greater than $2.4-\epsilon$. Let $M=12k$, for some integer k. Consider the set of tasks: $(1, 2)$, $(\delta, 2)$, $(1, 3)$, $(\delta, 3)$, $(2, 4)$, $(\delta, 4)$, $(2, 6)$, $(\delta, 6)$, $\cdots$, $(2^{M/2-1}, 2^{M/2})$, $(\delta, 2^{M/2})$, $(2^{M/2-1}, 3\cdot 2^{M/2-1})$, $(\delta, 3\cdot 2^{M/2-1})$. If we apply the RMNFS algorithm, tasks $(1, 2)$, $(\delta, 2)$ will be assigned to the first processor, tasks $(1, 3)$, $(\delta, 3)$ will be assigned to the second processor, tasks $(2, 4)$ and $(\delta, 4)$ will be assigned to the third processor, and so on. Tasks $(2^{M/2-1}, 3\cdot 2^{M/2-1})$ and $(\delta, 3\cdot 2^{M/2-1})$ will be assigned to the Mth processor. Thus the total number of processors required for this set of tasks when the RMNFS algorithm is applied is M.

However, this set of tasks can be feasibly scheduled on $5M/12+1$ processors. Let us divide the tasks into $k+1$ subsets as follows. For $i=1, \cdots, k$, let the ith subset consist of the following tasks: $(2^{6(i-1)}, 2\cdot 2^{6(i-1)})$, $(2^{6(i-1)}, 3\cdot 2^{6(i-1)})$, $(2^{6(i-1)+1}, 2\cdot 2^{6(i-1)+1})$, $(2^{6(i-1)+1}, 3\cdot 2^{6(i-1)+1})$, $\cdots$, $(2^{6(i-1)+5}, 2\cdot 2^{6(i-1)+5})$, and $(2^{6(i-1)+5}, 3\cdot 2^{6(i-1)+5})$. All tasks with run-time δ form the $(k+1)$-th subset. Each of the first k of these subsets can be partitioned into five groups as follows:

1. $(2^{6(i-1)}, 2\cdot 2^{6(i-1)})$, $(2^{6(i-1)+1}, 2\cdot 2^{6(i-1)+1})$
2. $(2^{6(i-1)+2}, 2\cdot 2^{6(i-1)+2})$, $(2^{6(i-1)+3}, 2\cdot 2^{6(i-1)+3})$
3. $(2^{6(i-1)+4}, 2\cdot 2^{6(i-1)+4})$, $(2^{6(i-1)+5}, 2\cdot 2^{6(i-1)+5})$
4. $(2^{6(i-1)}, 3\cdot 2^{6(i-1)})$, $(2^{6(i-1)+1}, 3\cdot 2^{6(i-1)+1})$, $(2^{6(i-1)+2}, 3\cdot 2^{6(i-1)+2})$
5. $(2^{6(i-1)+3}, 3\cdot 2^{6(i-1)+3})$, $(2^{6(i-1)+4}, 3\cdot 2^{6(i-1)+4})$, $(2^{6(i-1)+5}, 3\cdot 2^{6(i-1)+5})$.

Each of these groups can be feasibly scheduled on a single processor according to the rate-monotonic scheduling algorithm. Further, if δ is chosen so small that $M\delta < 2$, then all tasks with run-time δ can be scheduled on one processor. This arrangement will thus take only $5k+1$ processors. Thus $N_0 \leqq 5k+1 = 5M/12+1$, and so $N/N_0 \geqq 12M/(5M+12)$. By selecting M to be sufficiently large, we can make this ratio approach the limit 2.4. In other words, for a given $\epsilon > 0$, there is an integer M such that the ratio $N/N_0 > 2.4-\epsilon$, thus establishing the lower bound.

We now show that the ratio N/N_0 is upper bounded by 2.67. To this end, we define a function f mapping the utilization factor of tasks into the real numbers. Let u be the utilization factor of a task. We define

$$f(u)=\begin{cases}2.67u & 0\leqq u<0.75\\ 2 & u\geqq 0.75.\end{cases}$$

Let $\tau_1, \cdots, \tau_m$ be m tasks with utilization factors $u_1, \cdots, u_m$. Let $\tau_{j1}, \cdots, \tau_{jk_j}$ be the k_j tasks assigned to the jth processor according to an optimal assignment. Then

$$\sum_{r=1}^{k_j} f(u_{jr})\leqq 2.67\sum_{r=1}^{k_j} u_{jr}\leqq 2.67 \qquad \text{for } j=1, \cdots, N_0.$$

Summing up over all processors, we have

$$\sum_{i=1}^{m} f(u_i)\leqq 2.67N_0. \tag{1}$$

In the case of assignment of processors according to the RMNFS algorithm, we will show that if $u_{j1}, \cdots, u_{jk_j}$ are the utilization factors of the k_j tasks assigned to the jth processor, either $\sum_{r=1}^{k_j} f(u_{jr})\geqq 1$, or $\sum_{r=1}^{k_j} f(u_{jr})+f(u_{(j+1)1})\geqq 2$, where $u_{(j+1)1}$ is the utilization factor of the task that could not be assigned to the jth processor.

Suppose $\sum_{r=1}^{k_j} f(u_{jr})<1$. This means that $u_{jr}<0.75$ for $r=1, \cdots, k_j$. Therefore, $\sum_{r=1}^{k_j} f(u_{jr})=2.67\sum_{r=1}^{k_j} u_{jr}$. Thus, $\sum_{r=1}^{k_j} u_{jr}<1/2.67<k_j(2^{1/k_j}-1)$. Let $U_j=\sum_{r=1}^{k_j} u_{jr}$. Then, according to Theorem 2, $u_{(j+1)1}$, the utilization factor of the task that could not be assigned to the jth processor, must be greater than $2(1+U_j/k_j)^{-k_j}-1$, which in turn is greater than $2\exp(-U_j)-1$. If $u_{(j+1)1}\geqq 0.75$, we have $f(u_{(j+1)1})=2$, and hence $\sum_{r=1}^{k_j} f(u_{jr})+f(u_{(j+1)1})>2$. Otherwise, we have $\sum_{r=1}^{k_j} f(u_{jr}) +f(u_{(j+1)1})\geqq 2.67(U_j+2\exp(-U_j)-1)\geqq 2$ since for $x<1/2.67$, the expression $x+2\exp(-x)-1\geqq 2/2.67$.

Let N be the total number of processors used according to the RMNFS algorithm. Let processor P_j, $1\leqq j<N$, be the first processor for which $\sum_{r=1}^{k_j} f(u_{jr})<1$. Then, $\sum_{r=1}^{k_j} f(u_{jr})+\sum_{r=1}^{k_{j+1}} f(u_{(j+1)r})\geqq 2$.

We then repeat the above argument on processors P_{j+2} to P_N. All this implies that $\sum_{j=1}^{N}\sum_{r=1}^{k_j} f(u_{jr})\geqq N-1$. Thus,

$$\sum_{i=1}^{m} f(u_i)=\sum_{j=1}^{N}\sum_{r=1}^{k_j} f(u_{jr})\geqq N-1. \tag{2}$$

Combining the result of (1) and (2), we have $N-1\leqq \sum_{i=1}^{m} f(u_i)\leqq 2.67N_0$ or, $\lim_{N_0\to\infty}(N/N_0)\leqq 2.67$.

Note that in establishing the upper bound, we have made use of the result of Theorem 2. Therefore, even if only Theorem 2 is used in implementing step 2 of the algorithm, the upper bound will not increase.

5. RATE-MONOTONIC-FIRST-FIT SCHEDULING ALGORITHM

In the RMNFS algorithm, while assigning the next task to a processor, we check only the last processor used to see whether or not this task can be assigned to that processor, even though this task may possibly be assigned to one of the processors used earlier. If earlier processors used are also considered for assignment of the next task, it may be possible to reduce the number of processors used. This approach is adopted in the second algorithm considered, the RMFFS algorithm.

According to this algorithm, the tasks are first arranged in descending order of their request rates and renumbered for convenience. The assignment procedure is as follows:

```
Step 1. Set $i=N=1$
Step 2. (a) Set $j=1$.
        (b) If task $\tau_i$ together with the tasks that have been assigned
            to processor $P_j$ can be feasibly scheduled on processor $P_j$ according
            to the rate-monotonic scheduling algorithm, assign $\tau_i$
            to $P_j$ and go to step 3.
        (c) Set $j=j+1$ and go to step 2(b).
Step 3. If $j>N$, set $N=j$.
Step 4. If all tasks have been assigned, then stop; else set $i=i+1$ and
        go to step 2.
```

N is the number of processors required for scheduling the given set of tasks according to the RMFFS algorithm.

Note that step 2(b) may be implemented in the same way as step 2 of the RMNFS algorithm.

Since the RMFFS algorithm does not assign a task to the jth processor until it has been determined that the task cannot be assigned to the first $j-1$ processors, one would expect some improvement in the worst-case performance of this algorithm over the RMNFS algorithm. This in fact is the case since

THEOREM 4. *Let N be the number of processors required to feasibly schedule a set of tasks by the RMFFS algorithm and N_0 be the minimum number of processors required to feasibly schedule the same set of tasks. Then, as N_0 approaches infinity, $2 \leqq \lim_{N_0\to\infty} N/N_0 \leqq 4\times 2^{1/3}/(1+2^{1/3})$.*

Before we prove this theorem, we establish a series of lemmas.

LEMMA 1 [1]. *If m tasks cannot be feasibly scheduled on $m-1$ processors according to the RMFFS algorithm, then the utilization factor of the set of tasks is greater than $m/(1+2^{1/3})$.*

We now define a function f mapping the utilization factor of tasks into the real interval [0, 1]. If u is the utilization factor of a task, let

$$f(u)=\begin{cases} 2u & 0\leqq u\leqq \tfrac{1}{2} \\ 1 & \tfrac{1}{2}\leqq u\leqq 1. \end{cases}$$

LEMMA 2. *If tasks are assigned to the processors according to the RMFFS* *algorithm, among all processors to each of which two tasks are assigned, there is at most one processor for which the utilization factor of the set of the two tasks is less than* $\frac{1}{2}$.

Proof. Suppose the contrary is true. Let τ_{j1} and τ_{j2} denote the two tasks assigned to processor P_j, and τ_{k1} and τ_{k2} the two tasks assigned to processor $P_k (j<k)$, such that

$$u_{j1}+u_{j2}<\tfrac{1}{2} \tag{3}$$

and

$$u_{k1}+u_{k2}<\tfrac{1}{2}. \tag{4}$$

We have the following 3 cases:

Case 1. Tasks τ_{k1} and τ_{k2} were assigned to processor P_k after task τ_{j2} had been assigned to processor P_j. Since a set of three tasks with utilization factor less than or equal to $3(2^{1/3}-1)$ can be feasibly scheduled on a single processor according to the rate-monotonic scheduling algorithm (Theorem 1), we must have $u_{j1}+u_{j2}+u_{k1}>3(2^{1/3}-1)$ and $u_{j1}+u_{j2}+u_{k2}>3(2^{1/3}-1)$. Hence,

$$u_{k1}+u_{k2}>6(2^{1/3}-1)-2(u_{j1}+u_{j2})>6(2^{1/3}-1)-1$$

or $u_{k1}+u_{k2}>\frac{1}{2}$, which is a contradiction of (4).

Case 2. Tasks τ_{k1} and τ_{k2} were assigned to processor P_k after task τ_{j1} had been assigned to processor P_j, but before task τ_{j2}. We have $u_{j1}+u_{k1}>2(2^{1/2}-1)$ and $u_{j1}+u_{k2}>2(2^{1/2}-1)$. Hence

$$u_{k1}+u_{k2}>4(2^{1/2}-1)-2u_{j1}>4(2^{1/2}-1)-1>\tfrac{1}{2},$$

which is again a contradiction of (4).

Case 3. Task τ_{k1} was assigned to processor P_k after task τ_{j1} had been assigned to processor P_j, and τ_{k2} was assigned to P_k after τ_{j2} had been assigned to P_j. We have $u_{j1}+u_{k1}>2(2^{1/2}-1)$ and $u_{j1}+u_{j2}+u_{k2}>3(2^{1/3}-1)$. Once again, we have $u_{k1}+u_{k2}>2(2^{1/2}-1)+3(2^{1/3}-1)-\frac{1}{2}-\frac{1}{2}>\frac{1}{2}$, which contradicts (4).

LEMMA 3. *Let N_0 be the minimum number of processors required to schedule the set of tasks $\tau_1, \cdots, \tau_m$ with utilizations $u_1, \cdots, u_m$. Then $\sum_{i=1}^{m} f(u_i) \leqq 2N_0$.*

Proof. Let $\tau_{j1}, \cdots, \tau_{jk_j}$ be the set of tasks assigned to processor P_j. Since $\sum_{r=1}^{k_j} u_{jr} \leqq 1$, we have $\sum_{r=1}^{k_j} f(u_{jr}) \leqq 2\sum_{r=1}^{k_j} u_{jr} \leqq 2$. Hence

$$\textstyle\sum_{i=1}^{m} f(u_i) = \sum_{j=1}^{N_0} [\sum_{r=1}^{k_j} f(u_{jr})] \leqq 2N_0.$$

We now introduce some definitions. Let $\tau_{j1}, \cdots, \tau_{jk_j}$ be k_j tasks assigned to processor P_j, and let $\sum_{r=1}^{k_j} u_{jr} = U_j$. The *deficiency* δ_j of processor

P_j is defined as

$$\delta_j=\begin{cases}0 & \text{if } U_j\geqq k_j(2^{1/k_j}-1)\\ 2(1+U_j/k_j)^{-k_j}-1, & \text{otherwise.}\end{cases}$$

The *coarseness* α_j of the processor P_j is defined to be

$$\alpha_j=\begin{cases}0 & j=1\\ \max_{1\leqq r\leqq j-1}\delta_r & j>1.\end{cases}$$

LEMMA 4. *Suppose tasks are assigned to processors according to the RMFFS algorithm. If a processor with coarseness greater than or equal to* $1/6$ *is assigned three or more tasks, then* $\sum f(u_i)\geqq 1$, *where the sum is over all tasks assigned to the processor.*

Proof. First we observe that if the coarseness of a processor is α, then the utilization factor of every task on this processor is larger than α. This follows directly from the definition of the coarseness and Theorem 2. Thus, the utilization factor of each of the tasks on this processor is larger than $1/6$. If any one of the tasks assigned to the processor has a utilization factor larger than or equal to $1/2$, the result is immediate. Otherwise, $\sum f(u_i)\geqq 2\times 3\times 1/6=1$.

LEMMA 5. *Suppose tasks* $\tau_{j1},\cdots,\tau_{jk_j}$, $k_j>2$, *are assigned to processor* P_j, *whose coarseness* α_j *is less than* $1/6$. *If* $\sum_{r=1}^{k_j}u_{jr}\geqq \ln 2-\alpha_j$, *then* $\sum_{r=1}^{k_j}f(u_{jr})\geqq 1$.

Proof. If any one of the tasks has utilization factor larger than or equal to $1/2$, the result is immediate. We therefore assume that all tasks have utilization factor less than $1/2$. We then have

$$\sum_{r=1}^{k_j}f(u_{jr})=2\sum_{r=1}^{k_j}u_{jr}\geqq 2\ (\ln\ 2-\alpha_j)>2\ (\ln\ 2-1/6)>1.$$

LEMMA 6. *Let processor* P_j *with coarseness* α_j *be assigned tasks* $\tau_{j1},\cdots,\tau_{jk_j}$, *with utilization factor* $u_{j1},\cdots,u_{jk_j}$, *respectively, and let* $\sum_{r=1}^{k_j}f(u_{jr})=1-\beta$, *where* $\beta>0$. *Then*

(i) $k_j=1$ *and* u_{j1} *is less than* $1/2$;

or (ii) $k_j=2$ *and* $u_{j1}+u_{j2}$ *is less than* $1/2$;

or (iii) $\sum_{r=1}^{k_j}u_{jr}\leqq \ln 2-\alpha_j-\beta/2$.

Proof. (i) If $k_j=1$ and u_{j1} is greater than or equal to $1/2$, then $f(u_{j1})=1$, which contradicts the fact that $\beta>0$.

(ii) If $k_j=2$ and $u_{j1}+u_{j2}\geqq 1/2$, then again we have $f(u_{j1})+f(u_{j2})\geqq 1$, which contradicts the fact that $\beta>0$.

(iii) If neither (i) nor (ii) holds, then $k_j\geqq 3$. By Lemma 5 we have $\sum_{r=1}^{k_j}u_{jr}<\ln 2-\alpha_j$.

Let $\sum_{r=1}^{k_j}u_{jr}=\ln\ 2-\alpha_j-\lambda$, where $\lambda>0$. Let us replace tasks $\tau_{jr}=(C_{jr},\ T_{jr})$, $r=1,\ 2,\ 3$, by tasks $\tau'_{jr}=(C'_{jr},\ T_{jr})$, such that $C'_{jr}\geqq C_{jr}$, and

$\sum_{r=1}^{3} C'_{jr}/T_{jr} = \sum_{r=1}^{3} C_{jr}/T_{jr} + \lambda$ and $C'_{jr}/T_{jr} \leqq \frac{1}{2}$, for $r = 1, 2, 3$. Since the utilization factor of the set of tasks $\tau'_{j1}, \tau'_{j2}, \tau'_{j3}, \tau_{j4}, \cdots, \tau_{jk_j}$ is less than ln 2, this set can be feasibly scheduled on a single processor (Theorem 1). By Lemma 5

$$\sum_{r=1}^{3} f(u'_{jr}) + \sum_{r=4}^{k_j} f(u_{jr}) \geqq 1$$

$$\sum_{r=1}^{k_j} f(u_{jr}) \geqq 1 - f(\lambda) = 1 - 2\lambda$$

$$1 - \beta \geqq 1 - 2\lambda, \qquad \sum_{r=1}^{k_j} u_{jr} \leqq \ln 2 - \alpha_j - \beta/2.$$

Proof of Theorem 4. Suppose that for a given set of tasks N processors were used according to the RMFFS algorithm. Let $P_{i_1}, \cdots, P_{i_s}$ denote the processors for which $\sum f(u)$ is strictly less than 1, where the sum is over all tasks assigned to the processor.

Let us divide all these processors into 3 sets:

1. Processors to which only one task is assigned. Suppose there are p of them.
2. Processors to which two tasks are assigned. According to Lemma 2, there is at most one such processor. Let us denote this number by q, where $q = 0$ or 1.
3. Processors to which more than two tasks are assigned. Suppose that there are r of them. Clearly, $p + q + r = s$.

Note that coarseness of each processor in set (3) is less than $\frac{1}{6}$ (Lemma 4). For convenience, let us relabel the processors in set (3) as $Q_1, \cdots, Q_r$. Let α_j be the coarseness of processor Q_j. Further, for processor Q_j, let $\sum_{i=1}^{k_j} f(u_{ji}) = 1 - \beta_j$, where $\beta_j > 0$, for $j = 1, \cdots, r$. According to Lemma 6, for the r processors in set (3) we have

$$U_j = \sum_{i=1}^{k_j} u_{ji} \leqq \ln 2 - \alpha_j - \beta_j/2$$

Also, $\alpha_{j+1} \geqq \delta_j \geqq \ln 2 - U_j$, for $j = 1, \cdots, r-1$. Thus, $\alpha_j + \beta_j/2 \leqq \ln 2 - U_j \leqq \alpha_{j+1}$ for $j = 1, \cdots, r-1$. Hence, $\frac{1}{2}\sum_{i=1}^{r-1} \beta_i \leqq \alpha_r - \alpha_1 < \frac{1}{6}$. Thus, for the first $r-1$ processors in set (3),

$$\sum f(u) = (r-1) - \sum_{i=1}^{r-1} \beta_i \geqq (r-1) - \tfrac{1}{3} = r - \tfrac{4}{3}.$$

For the processors in set (1), since the p tasks cannot be scheduled on $p-1$ processors, by Lemma 1,

$$\sum_{i=1}^{p} u_{i1} \geqq p/(1 + 2^{1/3}). \qquad (5)$$

Also, since $f(u)$ for each of these processors is less than 1, each of these tasks has utilization factor less than $\frac{1}{2}$. Hence $\sum_{i=1}^{p} f(u_{i1}) \geqq 2p/(1 + 2^{1/3})$. Further, no task in this set can have utilization factor less than or equal to $\frac{1}{3}$ because a task with utilization factor less than or equal to $\frac{1}{3}$ can be feasibly scheduled on a single processor together with a task of utilization factor less than $\frac{1}{2}$ (Theorem 1). Hence in any optimal partition of the set of tasks, no more than two of these tasks can be assigned to a single processor. Therefore, $N_0 \geqq p/2$. Now,

$$\begin{aligned}\textstyle\sum_{i=1}^{m} f(u_i) &\geqq (N-s)+(r-\tfrac{4}{3})+2p/(1+2^{1/3})\\ &=N-(p+q+r)+(r-\tfrac{4}{3})+2p/(1+2^{1/3})\\ &=N-p(1-2/(1+2^{1/3}))-\tfrac{4}{3}-q.\end{aligned}$$

Also, from Lemma 3 we have $\sum_{i=1}^{m} f(u_i) \leqq 2N_0$. Therefore,

$$N \leqq 2N_0+p(2^{1/3}-1)/(1+2^{1/3})+\tfrac{4}{3}+q,$$

or

$$\begin{aligned}N/N_0 &\leqq 2+p(2^{1/3}-1)/(1+2^{1/3})N_0+(\tfrac{4}{3}+q)/N_0\\ &\leqq 2+7/3N_0+2(2^{1/3}-1)/(2^{1/3}+1).\end{aligned}$$

When N_0 is sufficiently large, we have $N/N_0 \leqq 4 \times 2^{1/3}/(1+2^{1/3})$.

To establish the lower bound, we show that for a given $\epsilon>0$, there exists an arbitrarily large set of tasks for which $N/N_0>2-\epsilon$.

Let us choose a set of N tasks as follows:

$$\begin{aligned}\tau_1&=(1,\ 1+2^{1/N})\\ \tau_2&=(2^{1/N}+\delta,\ 2^{1/N}(1+2^{1/N}))\\ \tau_3&=(2^{2/N}+\delta,\ 2^{2/N}(1+2^{1/N}))\\ &\vdots\\ \tau_N&=(2^{(N-1)/N}+\delta,\ 2^{(N-1)/N}(1+2^{1/N}))\end{aligned}$$

where δ is such that no task has utilization factor greater than $\tfrac{1}{2}$.

This set of tasks, when scheduled according to the RMFFS algorithm, will require N processors because no two of these tasks can be feasibly scheduled on a single processor using the rate-monotonic scheduling algorithm for a single processor. However, since no task has a utilization factor greater than $\tfrac{1}{2}$, any pair of these tasks can be feasibly scheduled on a single processor, according to the deadline-driven scheduling algorithm [3]. Thus all these tasks can be feasibly scheduled on $\lceil N/2 \rceil$ processors. Thus $N_0 \leqq \lceil N/2 \rceil$ and so $N/N_0 \geqq N/\lceil N/2 \rceil$. Taking N sufficiently large, we can make the ratio $N/N_0>2-\epsilon$.

6. CONCLUDING REMARKS

We have considered the problem of scheduling time-critical tasks on multiprocessor computing systems. In particular, we have studied two heuristic algorithms for feasibly scheduling given sets of tasks. These algorithms do not use more than a fixed percentage of the minimum number of processors that are needed. We have obtained the bounds on the worst-case behavior of these algorithms. In the case of the RMNFS algorithm, we showed that worst-case behavior is lower bounded by the constant 2.4 and is upper bounded by the constant 2.67. We suspect that the upper bound can be improved to 2.4, although we have not yet been able to prove it.

In the case of the RMFFS algorithm, we showed that in the worst case the ratio N/N_0 is lower bounded by the constant 2 and is upper bounded by the constant $4\times 2^{1/3}/(1+2^{1/3})$. Again, we suspect that the upper bound can be improved to 2. Dhall [1] showed that if the ratio of the longest request period to the shortest request period is less than or equal to 2, then the bound $m/(1+2^{1/3})$ in Lemma 1 can be increased to $m/(1+2^{1/m})$. It is conjectured that this result is true for an arbitrary set of m tasks. If this can be proved, then by substituting $p/(1+2^{1/p})$ for $p/(1+2^{1/3})$ in (5), we see that the upper bound for the RMFFS algorithm can be reduced to 2.

There are still many open questions in connection with the scheduling of time-critical tasks. For example, it would be interesting to investigate other heuristic algorithms, such as rate-monotonic-best-fit. It would also be interesting to study the scheduling problem when the processors are not identical. Furthermore, we limited our study to preemptive scheduling algorithms only. The problem of non-preemptive scheduling still remains largely unexplored.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under Grant NSF MCS73-03408-A01.

REFERENCES

1. S. K. Dhall, "Scheduling Periodic-Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems," Ph.D. dissertation, University of Illinois, Urbana, 1977.
2. D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-Case Performance Bounds for Simple One Dimensional Packing Algorithms," *SIAM J. Comput.* **3,** 299–325 (1974).
3. J. Labetoulle, "Some Theorems on Real Time Scheduling," in *Computer Architecture and Networks,* pp. 285–298, E. Gelenbe and R. Mahl (Eds.). North Holland Publishing Co., 1974.
4. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. Assoc. Comput. Machinery* **20,** 46–61 (1973).
5. O. Serlin, "Scheduling of Time Critical Processes," *Proceedings of the Spring Joint Computers Conference* **40,** 925–932 (1972).