Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks

MICHAEL L. DERTOUZOS, FELLOW, IEEE, AND ALOYSIUS KA-LAU MOK, MEMBER, IEEE

***Abstract*—The problems of hard-real-time task scheduling in a multiprocessor environment are discussed in terms of a scheduling game representation of the problem. It is shown that optimal scheduling without *a priori* knowledge is impossible in the multiprocessor case even if there is no restriction on preemption owing to precedence or mutual exclusion constraints. Sufficient conditions are derived which will permit a set of tasks to be optimally scheduled at run time.**

***Index Terms*—Deadline, hard-real-time, multiprocessor, scheduling.**

I. Introduction

The availability of inexpensive microprocessors has made it practical to employ large numbers of processors in real-time applications. However, the programming of such multiprocessor systems presents a rather formidable problem. In particular, time-critical tasks must be serviced within certain preassigned deadlines dictated by the physical environment in which the multiprocessor system operates. The scheduling problem which a programmer must face can be solved at compile time, e.g., the *CONSORT* system [1] or at run time, e.g., the *control robotics methodology* [2]. In this paper, we shall investigate some of the problems in run-time scheduling. We shall show that optimal schedulers are not always possible without *a priori* knowledge of the start-times of the tasks. (An optimal scheduler is one which may fail to meet a deadline only if no other scheduler can.) Some sufficient conditions for optimal scheduling will also be given.

II. Definitions and Past Results

In general, a task can be characterized by the integer parameters S (the start-time of the task), C (the number of units of CPU time) and D (the deadline). Additional constraints may be put on the tasks, such as the maximum frequencies of tasks which request service periodically or a precedence relation may be defined on a set of tasks. There may also be resources shared by some tasks. Task preemption may or may not be allowed. For a concise classification of scheduling problems, see Brucker, Lenstra, and Kan [3].

Considerable research has been done in the area of deadline scheduling. For the uniprocessor case, Liu and Layland [4] obtained a necessary and sufficient condition for scheduling periodic task sets (with preemption allowed). The scheduler employed was also shown to be optimal by Dertouzos [2] for arbitrary task sets (not necessarily periodic). Henn [5] generalized the result for a single processor case to cover precedence constraints. Garey and Johnson [6] discovered optimal scheduling algorithms when there are only two processors. Unluckily, the scheduling problem often becomes mathematically intractable (NP-hard and therefore likely requires exponential time to solve) whenever more than two processors are involved, e.g., see [7]. Two notable exceptions are Brucker, Garey, and Johnson [8], and Bratley, Florian, and Robillard [9]. The former deals with the case where a precedence relation in the form of an in-tree is defined on the tasks, and the latter solves the problem of arbitrary start-times and deadlines with no precedence constraint and with preemption allowed.

For the programmer, the known deadline scheduling results have the following implications. When there are more than two processors involved, it is impractical to design optimal run-time schedulers if there is resource sharing or if a general precedence relation may be defined on the tasks. The one multiprocessor scheduling problem which seems to lend itself to on-line solution is where there is no precedence constraint involved and where task preemption is allowed. If all the start-times are known *a priori*, scheduling can be done at compile time using the algorithm of Bratley, Florian, and Robillard [9]. In practice, however, there are many occasions where the start-times of the tasks are not known beforehand. Even if such information is available, sometimes the start-times are so far off into the future that to compute the whole schedule beforehand may be impractical, e.g., periodic tasks whose periods are relatively prime. In other words, we would like to have run-time scheduling algorithms which can be used to determine which tasks to execute next, based on information of the relatively immediate tasks. The algorithm of Bratley, Florian, and Robillard [9] may be used at run-time to schedule all tasks whose start-times have elapsed or are known. However, this algorithm is not optimal when used incrementally. In general, the problem

Manuscript received June 1, 1979; revised July 30, 1989. This work was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under Contract N00014-75-C-0661. A. Mok was supported in part by a research grant from the Office of Naval Research under Contract N0014-85-K-0227.

M. L. Dertouzos is with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

A. K. Mok is with the Department of Computer Science, University of Texas, Austin, TX 78712.

IEEE Log Number 8931481.

of finding an optimal algorithm is not solvable. To gain insight, we shall reformulate the scheduling problem in the form of a game.

III. The Scheduling Game Representation

The status of each task whose start-time has elapsed can be characterized by two parameters: the remaining computation $C(i)$ at time $= i$ and the deadline $D(i)$ by which to complete it. For convenience, we define the *laxity* of a task at time $= i$ by:

$$L(i) = D(i) - C(i).$$

The laxity of a task is a measure of its urgency. Clearly, a task with zero laxity must be executed immediately and without interruption. A negative laxity indicates that a deadline will be missed.

In the literature, solutions to scheduling problems are frequently represented by timing diagrams such as Fig. 1 (up and down arrows denote, respectively, start-times and deadlines) and Fig. 2 (known as a Gantt Chart).

If the scheduling problem evolves with time as in run-time scheduling, it is more helpful to have a succession of graphic representations each of which represents the scheduling problem at a given point in time. The scheduling problem at time i can be modeled by a configuration of "tokens" in the first quadrant of a Cartesian plane with vertical axis C and horizontal axis L. Each task is represented by a token. Specifically, the token representing task j with parameters $C_j(i)$ and $L_j(i)$ at time i is located at the position $L = L_j(i)$, $C = C_j(i)$ on the L–C plane. Fig. 3 shows an example with three tokens (tasks).

Note that more than one token can occupy the same position (tasks 2 and 3), that the number of processors in the system is not explicitly shown in the graphical representation (it will be incorporated into the rules for manipulating the tokens) and that whenever time dependency is not important, the index i (time is discrete) will be left out.

Suppose there are n processors and m tasks ($m > n$) which require computation at time i. We can execute any n of the m tasks at a time. On the L–C plane, this corresponds to moving at most n of the m tokens (representing the tasks being executed) one division downwards and parallel to the C axis and moving the rest of the tokens (representing tasks not executed) one division to the left and parallel to the L axis. Accordingly the new position for a token which has been "executed" at time i is given by $L(i + 1) = L(i)$, $C(i + 1) = C(i) - 1$. For a token that has not been executed, its position at time $i + 1$ will be given by $L(i + 1) = L(i) - 1$, $C(i + 1) = C(i)$. Exactly which tokens are moved downwards is decided by the scheduling algorithm. Since a task with a negative laxity denotes a failure, a token being allowed to move into the second quadrant of the L–C plane means that scheduling algorithm used has missed a deadline. Tokens which reach the L (horizontal) axis without moving across the C (vertical) axis represent tasks whose deadlines are successfully met and can be removed from the L–C plane as soon as they reach the L axis. Thus a schedule (following some scheduling algorithm) can be simulated by a sequence of configurations of tokens on the L–C plane, each configuration representing the scheduling problem at a point in time. The rules for this simulation game can now be stated:

1) The scheduler is provided with an initial configuration of m tokens in the first quadrant of a Cartesian game board (the L–C plane). This configuration models the tasks to be carried out.

2) At each step of the game, the scheduler is allowed to move at most n (corresponding to the number of available processors) of the tokens one division downwards

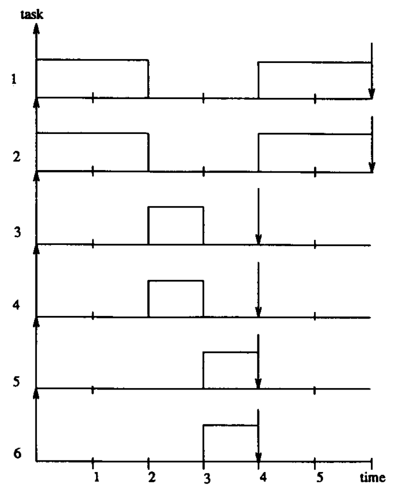


Fig. 1. Timing diagram.

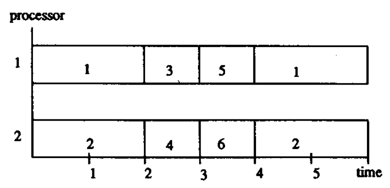
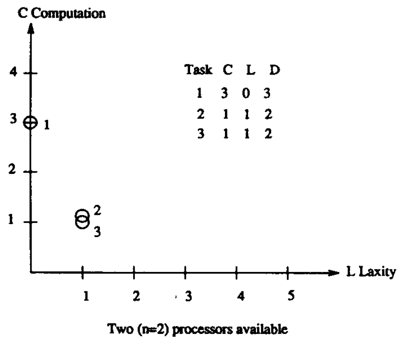


Fig. 2. Gantt chart.



Task	C	L	D
1	3	0	3
2	1	1	2
3	1	1	2

Fig. 3. Scheduling game example.

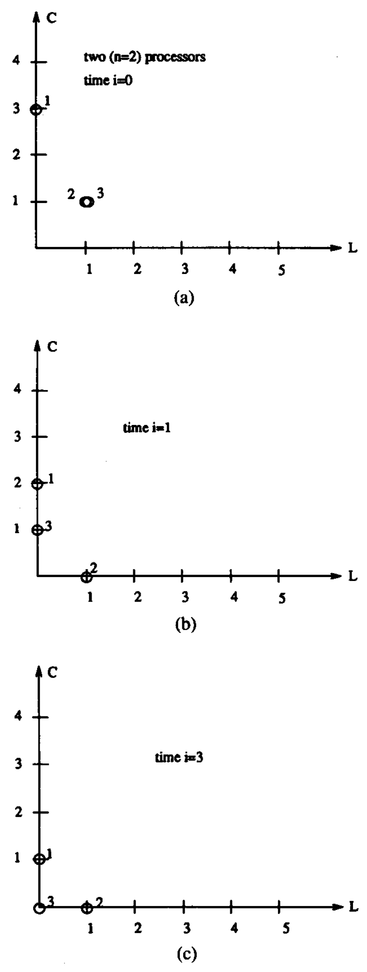


Fig. 4. Solution to scheduling game example.

towards the horizontal axis. The rest of the tokens must be moved one division to the left towards the vertical axis.

3) Any token reaching the horizontal axis can be ignored for the rest of the game (its deadline has been met).

4) The scheduler fails if any token crosses the vertical axis into the second quadrant of the Cartesian game board before reaching the horizontal axis.

5) The scheduler wins (achieves scheduling) if in a succession of steps all tokens are evacuated without incurring failure.

As an example, the scheduling game initialized by Fig. 3 is played by a scheduler who wins it by using two processors ($n = 2$). The sequence of moves is shown in Fig. 4.

In real-time scheduling, new tasks may occur at any time and we permit the addition of new tokens to the game board after any move. If *a priori* knowledge of the start-times is available, we can represent a future task by a "restricted token" which is held stationary at the position corresponding to its preassigned laxity and computation time until its start-time arrives. After that, the token will be free to move parallel to either axis just like ordinary tokens. To distinguish them from the rest, we may write inside each of these restricted tokens a *count* which is the number of time units it will be held stationary, i.e., its start-time with respect to the present. Tokens whose start-times have elapsed have zero count.

IV. Uniprocessor Scheduling

In the case of a single processor, optimal scheduling algorithms exist in the sense that if scheduling can be achieved by any algorithm, then it can be achieved by the optimal algorithm. We list two such algorithms.

A. *Earliest Deadline*

Execute at any time the task whose deadline is the closest. Ties are broken arbitrarily.

B. *Least Laxity*

Execute at any time the task which has the smallest laxity. Ties are broken arbitrarily.

The optimality of the Earliest Deadline algorithm is proved in Dertouzos [2]. The proof depends on the fact that for a single processor system, it is always possible to transform a feasible schedule to one which follows the Earliest Deadline algorithm. This is so because if at any time the processor executes some task other than the one which has the closest deadline, then it is possible to interchange the order of execution of these two tasks, i.e., execute the task with the closest deadline first and make up for the loss of one unit of processor time of the sacrificed task by executing it at a later time when the task with the closest deadline would have been executed. Since the sacrificed task has a more distant deadline, making up for its one unit of lost processor time before the closest deadline certainly does not violate its own deadline. This is illustrated by the timing diagrams of Fig. 5. In the schedule of Fig. 5(a), task 1 which has a deadline at $t = 3$ is executed at $t = 0$ instead of task 2 which has a deadline at $t = 2$. The alternative schedule which follows the Earliest Deadline algorithm is shown in Fig. 5(b). The swapping of the shaded blocks does not cause task 1 to fail. The optimality of the Least Laxity algorithm can be proved in a similar way.

Unfortunately, this optimality proof does not hold in the multiprocessor case. Consider the scheduling problem shown in Fig. 6(a) (the corresponding scheduling game representation is Fig. 3). There are two processors available. Tasks 2 and 3 have deadlines two time units away whereas task 1 has a deadline of three time units. According to the Earliest Deadline algorithm tasks 2 and 3 should be executed first. Token 1 immediately crosses into the second quadrant of the L–C plane where L is negative, indicating a failure to meet the deadline of task 1. However, scheduling is possible if task 1 is executed first, as Fig. 4 illustrates. The timing diagram of Fig. 6(b) shows where the optimality proof fails: the shaded blocks can no longer be swapped because task 1 would then be executed simultaneously on both processors. The scheduling prob-

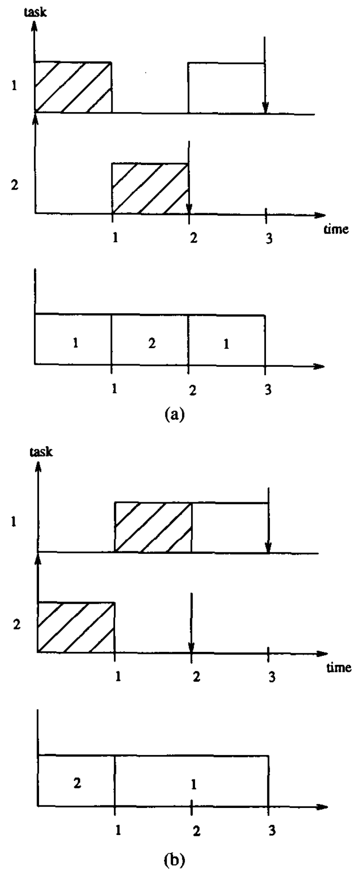


Fig. 5. Pertinent to the optimality proof of the earliest deadline Algorithm.

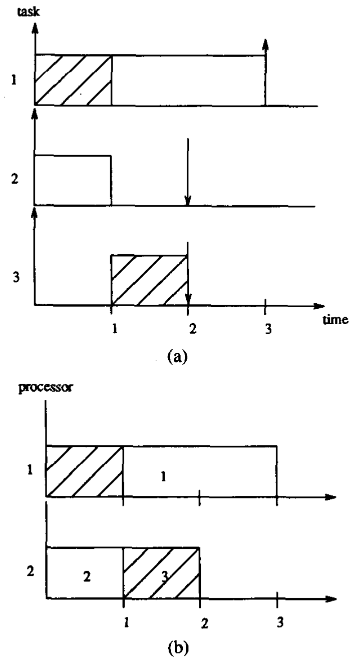


Fig. 6. Breakdown of the optimality proof when there are two processors.

lem of Fig. 6 can be solved if the Least Laxity algorithm is used. However, there are multiprocessor scheduling problems for which the Least Laxity algorithm is not optimal (in fact, the Least Laxity algorithm is not optimal even if the computation times, deadlines and start-times of all the tasks are known *a priori*).

Even though the Earliest Deadline algorithm is not optimal in the multiprocessor case, it is a rather efficient algorithm in terms of minimizing the number of preemptions.

Lemma 1: The system overhead due to context switching required by the Earliest Deadline algorithm is at most twice that required by any other scheduling algorithm.

Proof: We add to the computation time of each task the time to do one "context switch" (i.e., the time needed to save the status of the task being preempted and to load the preempting task) and also the time to restart a task after it has been preempted (counted as another context switch). Then we can charge the cost of each preemption to the preempting task, and the cost of restarting a preempted task to the task which has been completed just before it. Notice that this is possible only in the case of the Earliest Deadline algorithm since each task will preempt another task (because of a more imminent deadline) at most once and a task is restarted only at the completion of some other task. This way, all the switching overhead is accounted for. Since each task must be loaded at least once (we count this as one context switch), the lemma follows.

In practice, a task will be interrupted by the scheduler even though it may not be preempted at all. In the worst case, this can happen as many times as the total number of the other tasks which may request service. We can take this overhead into account by adding it to the computation time of each task. A cleaner solution is to use an extra processor which will be put in charge of scheduling (comparing deadlines) and handling interprocessor communication in real-time. For a discussion of the use of a dual processor, see Mok and Ward [10].

V. The Insufficient Knowledge Problem

The optimality of the Earliest Deadline algorithm in the case of a single processor is a very desirable property since it is a "best effort" algorithm and allows the system to degrade gracefully. Furthermore, the Earliest Deadline algorithm is driven by deadlines alone and does not require *a priori* knowledge of the computation time of the tasks or even their start-times. So it would be especially pleasing if we could find an algorithm which has this property even for multiprocessor systems. Unfortunately, such an algorithm does not exist. In particular, if we do not have *a priori* knowledge of any one of the following parameters: 1) deadlines, 2) computation time, or 3) the start-times, then for *any* algorithm one might propose, one can always find a set of tasks which cannot be scheduled by the proposed algorithm but which can be scheduled by another algorithm. This assertion can be established by "adversary arguments" as follows. Cases 1) and 2) are not surprising, and are included here for completeness.

Lemma 2: There can exist no optimal scheduling algorithm if the computation time of the tasks are not known *a priori*.

Proof: Consider the scheduling situation in Fig. 7. There are three tokens on the game board and the scheduler is allowed to move two tokens downwards at each step (i.e., there are two processors). All three tokens represent tasks with a deadline of two time units. Furthermore, two of the tasks need one unit of computation time and their corresponding tokens are marked by triangles. The other task needs two units of computation time and the corresponding token is marked by a circle. In order to achieve scheduling, the task with two units of computation time must be immediately executed, i.e., the circular token must be moved downwards at once. Since no information about computation time is available, all three tasks appear to be the same to the scheduler. That is to say, the scheduler cannot distinguish between the shape of the tokens. Thus if a scheduling algorithm executes task j first, we can always arrange our example so that task j is represented by a triangular token, in which case the proposed algorithm will fail while some other algorithm will succeed (namely, the one that moves the circular token down first).

Lemma 3: There can be no optimal scheduling algorithm if the deadlines of the tasks are not known *a priori*.

Proof: By an entirely similar argument as the one above, using the situation shown in Fig. 8.

Lemma 4: There exists no optimal scheduling algorithm if the distribution of the requests in time is not known *a priori*.

Proof: Consider the situation shown in Fig. 9(a). There are three tokens (marked A, B, and C) and the scheduler is allowed to move two of them downward at each step (i.e., there are two processors). Since B has zero laxity, it must be moved down at once. Thus, depending on the scheduler's decision, there are three cases.

Case 1: A and B are moved downwards at $i = 0$. In this case, C will be on the vertical axis at $i = 1$. Consider the situation shown in Fig. 9(b). Two extra tokens (marked D and E) are introduced at $i = 1$. This is permissible since there is no *a priori* knowledge of the distribution of the requests in time, and D and E represent tasks whose requests occur at $i = 1$. Consequently, there are three tokens (C, D, and E) on the vertical axis and since the scheduler can evacuate at most two of them, scheduling cannot be achieved. However, if the scheduler had moved B and C down instead of A and B at $i = 0$, the situation at $i = 1$ would have looked like the one shown in Fig. 9(c). Scheduling could then be achieved by evacuating D and E at $i = 1$ and A subsequently.

Case 2: B and C are moved downward at $i = 0$. In this case, A will be at position ($L = 1$, $C = 2$) at $i = 1$. Consider the situation shown in Fig. 9(d). Two extra tokens (marked F and G) are introduced at $i = 2$. Since F and G are on the vertical axis, they have to be moved downward in the next two time units. Hence it is impossible to evacuate A before it crosses over the vertical axis and the scheduler fails. However, if A and B had been moved downward at $i = 0$ instead of B and C, the situation would have been like the one shown in Fig. 9(e) at $i = 2$. This is so because both A and C could have been evacuated at $i = 1$. Thus the scheduler fails because of a wrong decision made at $i = 0$.

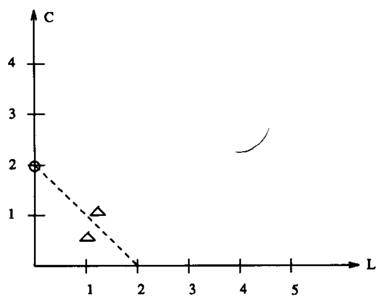


Fig. 7. Scheduled tasks with unknown computation times. Unlike the case of a single processor, it is not enough to know the deadlines. All three tasks in Fig. 7 have deadlines = 2.

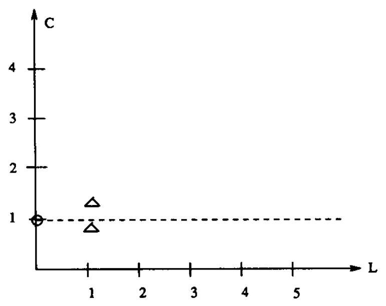


Fig. 8. Scheduled tasks with unknown deadlines.

Case 3: Only B is moved downward at $i = 0$. In this case, extra tokens introduced at $i = 1$ such as D and E in Fig. 9(b) will cause the scheduler to fail unnecessarily.

Thus for any decision that a proposed optimal scheduler makes at $i = 0$, there is a set of future requests (D and E or F and G) which will cause that scheduler to fail, whereas if a different scheduler had been used at $i = 0$, then scheduling would have been achieved. It should be noted that the preassigned computation times and deadlines of the tasks D, E, F, and G may indeed be known at $i = 0$. However, if we add tasks F and G to case 1 and tasks D and E to case 2 so that their start-times are distant enough (after $i = 4$) so as not to affect the completion of the other tasks, then cases 1 and 2 represent two different distributions of start-times for the same set of tasks. Any optimal scheduler must necessarily guess correctly which of the two start-time distributions will actually occur inasmuch as the scheduling decision made at $i = 0$ will cause failure if the wrong guess is made.

Some observations can be made about the above lemma. Focusing on Fig. 9(a), we can see that there are basically two choices that a scheduler can make: either execute the

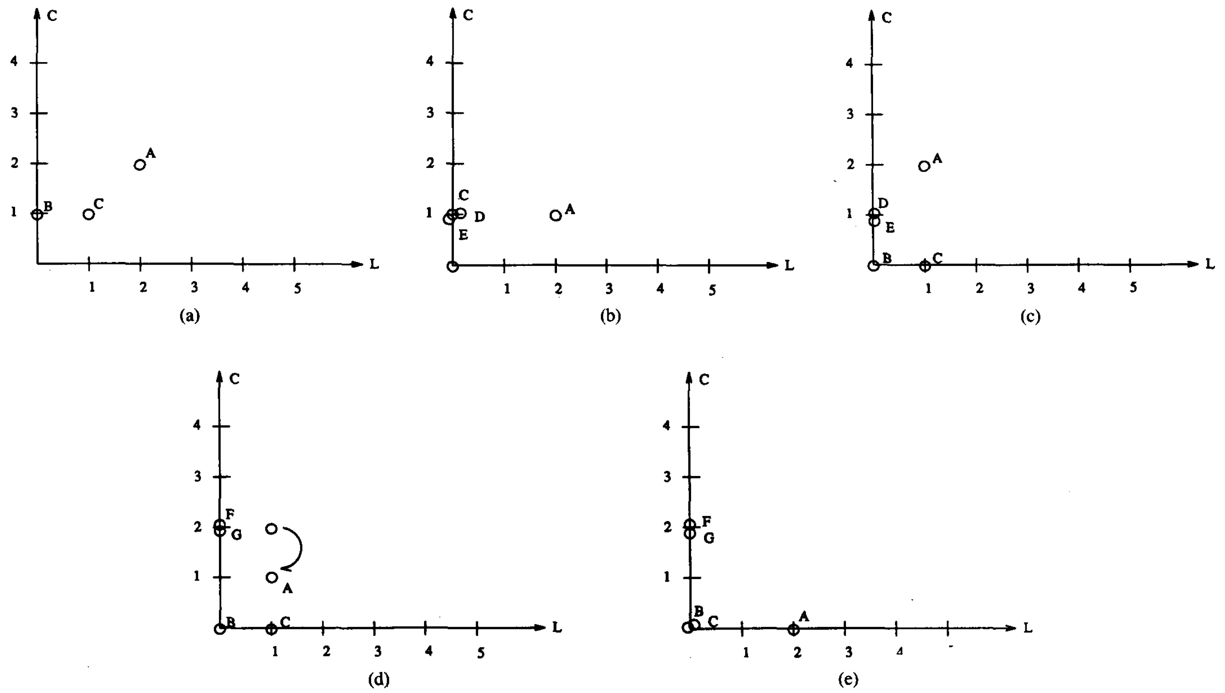


Fig. 9. Scheduled tasks without *a priori* information about start times. The tasks D and E conflict with F and G in the sense that the initial move of the scheduler must anticipated which of them will occur first.

shorter tasks (B and C) and run the risk of letting one processor idle at $i = 1$ which is what happens in Fig. 9(d); or execute the longer task (A) first and run the risk that short but urgent tasks may occur in the near future [at $i = 1$ in Fig. 9(b)]. The former case corresponds to a strategy of minimizing the number of unfinished tasks as soon as possible and the latter a strategy of minimizing the overall completion time of the schedule. These are conflicting objectives. Although the above proof deals with only two processors, the lemma remains valid for more processors since we can always keep extra processors busy by introducing tasks with zero laxity.

Theorem 5: For two or more processors, no deadline scheduling algorithm can be optimal without complete *a priori* knowledge of the 1) deadlines, 2) computation times, and 3) start-times of the tasks.

As a result of the above theorem, the known multiprocessor scheduling algorithms which require *a priori* knowledge of start-times will not be optimal when used on line.

A. Conflicting Task Sets

Although optimal schedulers exist for the uniprocessor case, we cannot expect to find one when more processors are involved. The inevitable failure of any run-time scheduler may be attributed to the possible existence of two or more sets of future tasks which "conflict" with one another in the sense that the scheduler is forced to make an early commitment to meet the deadlines of one set of tasks at the expense of all others, even though not all the tasks which the scheduler is prepared to schedule may actually request service in the immediate future. In practice, additional knowledge may be available for deciding which one of the conflicting task sets will occur next and they can be scheduled accordingly. If no such *a priori* knowledge is available, optimal scheduling is possible only if the set of tasks does not have subsets which conflict with one another. For example, if all tasks have unit computation time, then the Earliest Deadline algorithm which does not require *a priori* knowledge of the start-times is optimal even in the multiprocessor case. In this case, the optimality proof of the Earliest Deadline algorithm is independent of the number of processors since the "swapping" of tasks that is essential in the proof will not result in the same task being executed simultaneously on two or more processors. Nevertheless, the requirement that all tasks have unit computation time is often too restrictive.

We observe from the last section that no feasible schedule exists if the conflict task sets request service simultaneously. Intuitively, this has to be true of any conflicting task sets. In other words, if there are enough processors to successfully schedule a set of tasks when they request service simultaneously, then it is reasonable to expect that the problem of conflicting subsets of tasks will not occur. However, care must be taken so that the scheduler will select for execution only those tasks whose start-times have passed. In the following section, we shall show that if a schedule exists to meet the deadlines of a set of tasks when their start-times are the same, then the same set of tasks can be scheduled when their start-times are different and furthermore, it is unnecessary to know about the start-times *a priori*. Our proof extends a result of Knut Nordbye [11] who has shown a necessary and sufficient condition for scheduling tasks with the same start-time in terms of the scheduling game model invented by the second author.

B. Sufficient Conditions for Conflict Free Task Sets

Some notation is in order. We shall denote the jth task by J_j and use the subscript j in all its parameters. For the ease of counting, the L-C plane is divided into three regions as illustrated in Fig. 10. For every integer k, all the tokens on a game board can be partitioned into three distinct sets:

$$R_1(k) = \{J_j : D_j \le k\}$$
$$R_2(k) = \{J_j : L_j \le k \text{ and } D_j > k\}$$
$$R_3(k) = \{J_j : L_j > k\}.$$

Next we define for every positive integer k, the function

$$F(k) = k*n - \sum_{R_1} C_j - \sum_{R_2} (k - L_j) \qquad k > 0.$$

The function $F(k)$ defined above is a measure of the "surplus" computing power of the system in the next k time units. To emphasize that $F(k)$ is really a function of time, its time dependence will be explicitly written out, whenever necessary, as $F(k, i)$ which will be understood as the value of $F(k)$ computed on the scheduling situation at time $= i$.

Our strategy is to first prove a necessary condition for scheduling a set of tasks whose start-times are the same. Given this necessary condition, we shall then show that scheduling strategies exist which will not cause a deadline to be missed in the next time instant and which also preserve the necessary condition in the assumption. Only tasks whose start-times have elapsed will be selected for execution and that their selection can be made without any knowledge of the start-times of future tasks. Knowledge of the preassigned laxities of all tasks alone is enough for scheduling. In other words, a set of tasks is conflict-free if their deadlines can be met when their start-times are the same.

Lemma 6: A necessary condition for scheduling to meet the deadlines of a set of tasks whose start-times are the same (at time $i = 0$) is that for all $k > 0$, $F(k, 0) \ge 0$.

Proof: Referring to Fig. 11, it can be seen that any token in R_1 must be evacuated in k steps since all of their deadlines are no bigger than k. This requires $\Sigma_{R_1} C_j$ units of processor time for all the tokens in R_1. For a token (with a laxity of L_j) in R_2, it can be allowed to move parallel to the L (horizontal) axis for at most L_j divisions within the next k steps and still be in the first quadrant. For the rest of the time, it must be moved parallel to the C (vertical) axis until it reaches the L axis. They cannot be stationary since their start-time (which is zero) has elapsed. For a token in R_2, $C_j = D_j - L_j > k - L_j$. Thus in the next k steps, each token in R_2 must be moved at least $k - L_j$ divisions downward. The minimum amount of processor time required by all the tokens in R_2 needs is thus $\Sigma_{R_2} (k - L_j)$. In k steps, the maximum amount of processor time we can get from an n-processor system is $k*n$. Hence, to satisfy the minimum computation requirement of the tasks in R_1 and R_2, we must have

$$F(k) = k*n - \sum_{R_1} C_j - \sum_{R_2} (k - L_j) \ge 0.$$

Since this must hold for all $k > 0$, the lemma is proved.

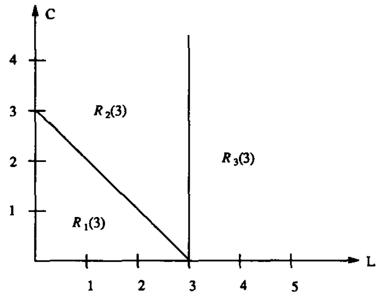


Fig. 10. Pertinent to the definition of $F(k)$. Tokens in $R_1(k)$ have deadlines no later than k. Tokens in $R_2(k)$ have laxities no bigger than k, but with deadlines later than k. Tokens in $R_3(k)$ need no immediate attention within the next k steps.

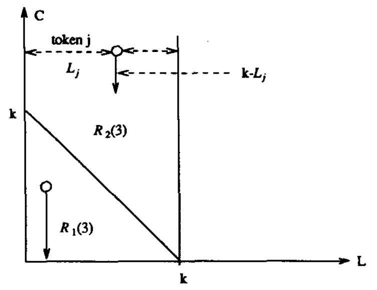


Fig. 11. Necessary conditions for scheduling. Tokens in $R_2(k)$ must be evacuated in k steps. Tokens in $R_2(k)$ require $k - L_j$ downward moves. Tokens in $R_3(k)$ will not fail in k steps.

Theorem 7: If a schedule exists which meets the deadlines of a set of tasks whose start-times are the same, then the same set of tasks can be scheduled at run-time even if their start-times are different and not known *a priori*. Knowledge of the preassigned deadlines and computation times alone is enough for scheduling. One successful run-time scheduling algorithm is the Least Laxity algorithm.

Least Laxity Algorithm: From among the tasks whose start-times have elapsed, execute as many as possible (up to n, the number of processors) tasks with the smallest laxities. Ties are broken arbitrarily.

Proof: By the previous lemma, $F(k, 0) \ge 0$ if there is a schedule which meets the deadlines of the tasks when their start-times are the same. ($F(k, 0)$ is computed according to the preassigned laxities and computation times of all the tasks. By the convention described earlier, a task whose start-time has not arrived is represented by a token which is held stationary at the position corresponding to its preassigned laxity and computation time. Start-times do not affect the computation of $F(k, 0)$ as long as this convention is observed.) With this assumption, we shall show that no deadline will be missed in the immediate future and that after one move, our induction hypothesis will again hold. Specifically,

1) $F(k, i) \geq 0$ implies that the number of tokens on the C (vertical) axis is no bigger than n (the number of processors).

2) $F(k, i) \geq 0$ for all $k > 0$ implies that $F(k, i + 1) \geq 0$ for all $k > 0$ if the Least Laxity algorithm is used to select the tokens to move.

We observe that $F(1, 0) \geq 0$ implies that $n \geq \Sigma_{R_1(1)} C_j + \Sigma_{R_2(1)} (1 - L_j)$. However, the right-hand side of this last inequality is exactly the number of tokens on the C (vertical) axis. Since these tokens have zero laxity, they will be moved down the C axis according to the Least Laxity algorithm if their start-times have elapsed. Hence, all the tokens on the C axis will either be moved down or stay stationary. No immediate failure will occur and proposition 1) is proved.

To verify proposition 2), the strategy is to prove that for all every $k > 0$, there is a k' such that $F(k, i + 1) \geq F(k', i)$. Since by hypothesis, $F(k', i) \geq 0$ for any $k' > 0$, it can then be concluded that $F(k, i + 1) \geq 0$. Referring to Fig. 12, there are two cases to consider. The arrows in Fig. 12 show the movements of the tokens according to the Least Laxity algorithm. Where an arrow is missing, the token is stationary.

Case 1—Fig. 12(a): In this case, all the tokens left of the line $L = k$ either have been stationary or arrived at their current positions at time $i + 1$ by a vertical (parallel to the C axis) move. We pick $k' = k$, i.e., we want to prove $\delta F = F(k, i + 1) - F(k, i) \geq 0$.

Owing to the possible migration of tokens from R_3 to the line $L = k$, there may be a gain of tokens in R_1 and R_2. For these tokens, $k - L_j = 0$ and so they do not affect δF. Other tokens from R_2 may have moved to the line $L + C = k$ which is in R_1. They contribute equally to both $F(k, i)$ and $F(k, i + 1)$ since $k - L_j(i) = C_j(i) - 1 = C_j(i + 1)$. Tokens which have remained in R_2 have the same laxities as before and do not affect δF. All other tokens in R_1 add to $F(k, i + 1)$ by subtracting one from C_j if they move downward, or contribute equally to both $F(k, i + 1)$ and $F(k, i)$ by remaining stationary. Thus for case 1, $\delta F =$ the number of tokens in R_1 which have been moved downwards in the recent move. Therefore, $F(k, i + 1) \geq 0$.

Case 2—Fig. 12(b): In this case, some of the tokens left of the line $L = k$ have arrived at their current positions by a horizontal (parallel to the L axis) move. We pick $k' = k + 1$, i.e., we want to prove $\delta F = F(k, i + 1) - F(k + 1, i) \geq 0$.

By the Least Laxity algorithm, n tokens must have been moved downwards left of the line $L = k + 1$. (Tokens on

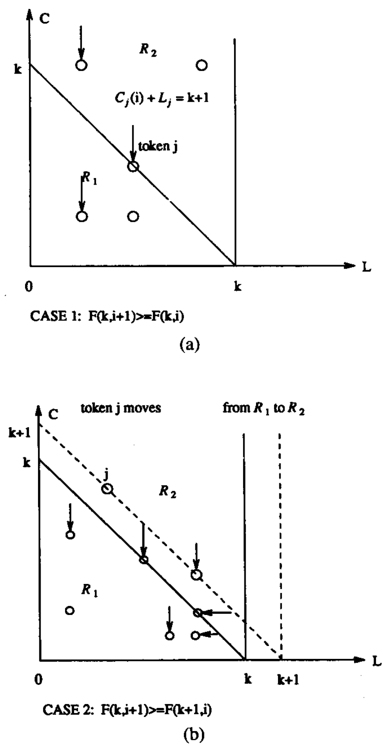
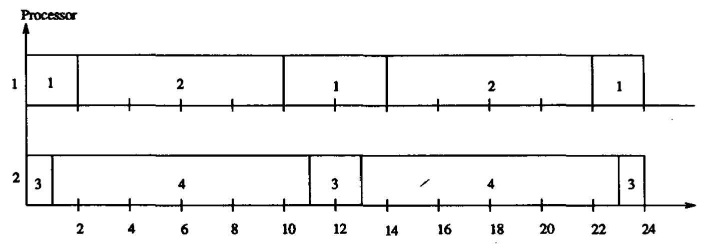


Fig. 12. Sufficient conditions for optimal scheduling.

the line $L = k$ might have moved in either direction.) Consider the tokens which have been moved downward. If it is in R_2, then it contributes $-(k - L_j(i + 1)) + (k + 1 - L_j(i)) = 1$ to δF since $L_j(i + 1) = L_j(i)$. If the token is on the line $L + C = k$, then it must have been moved from the line $L + C = k + 1$ and is therefore in R_1 both before and after the last move. It contributes positively to δF since its computation has decreased by one. If the token has been in R_1, its contribution to δF is obviously one. Hence, in all cases, a token which has been moved downward contributes positively to δF and there are n of them. Next, consider the tokens which have moved horizontally. If it is in R_2, then it contributes $-(k - L_j(i + 1) + (k + 1 - L_j(i)) = 0$ to δF. If it is on the line $L + C = k$, then it must have moved from the line $L + C = k + 1$. Its contribution to δF is zero since it is in R_1 both before and after the last move and its computation has not decreased. If it has been in R_1, then its contribution to δF is clearly zero. So tokens which have moved horizontally do not affect δF. Similarly, tokens which have remained stationary make zero or unit contribution to δF depending on whether $L_j + C_j < k$. It is interesting to note that stationary tokens on the line $L + C = k + 1$ "move" from $R_1(k + 1)$ at time i to $R_2(k)$ at time $i + 1$. However, it contributes positively to δF since $-(k - L_j(i + 1)) + C_j(i) = -(k - L_j(i)) + (k + 1 - L_j(i)) = 1$. Thus the final tally for δF is given by $F(k, i+ 1) - F(k + 1, i) = k*n - (k + 1)*n + at$



Two processors
Four tasks

Tasks	C_j	D_j
J_1	2	6
J_2	4	6
J_3	2	12
J_4	20	24

T = GCD (6,6,12,24) = 6
t = GCD (6*2/6, 6*4/6, 6*2/12, 6*20/24, 6)
= GCD (2,4,1,5,6) = 1

Fig. 13. Example of time slicing.

least $n \geq 0$. This inequality is strict if all stationary tokens are positioned left of the line $L + C = k$.

For every $k > 0$, either of the above two cases must hold true. (For the vacuous case where no token is positioned left of the line $L = k$, $F(k) = k*n$.) In all cases, $F(k, i + 1) \geq 0$ and the theorem is proved.

The Least Laxity algorithm is conservative in the sense that some of the tasks it selects for execution are not really urgent. In fact, if it is possible to meet all the deadlines when the start-times are the same, then a scheduler will succeed if it obeys the following criterion (see [12, pp. 48-53]).

C. Criterion for Successful Schedules

At any time $= i$ and for all $k > 0$, at least $n - F(k, i)$ of the tasks with laxity less than k must be selected for execution. If $n - F(k, i) < 0$, then it is not necessary to execute any of the tasks with laxity less than k at time $= i$.

In run-time scheduling, only the preassigned laxities and computation times of all the tasks can be known *a priori*. However, this is enough information to compute $F(k)$ as required by the above criterion. Furthermore, a task whose start-time has not arrived may be counted as one of the n-$F(k, i)$ tasks selected to satisfy the scheduling criterion (while the task actually receives no processor time at time $= i$) if the sum of its preassigned laxity and computation time is not smaller than k.

D. Periodic Tasks

We have shown that the Least Laxity algorithm is an optimal scheduler if the task set satisfies a sufficient condition which also allows it to be scheduled without *a priori* knowledge of their start-times. The Least Laxity scheduler, however, is nonoptimal if the tasks are allowed to repeat their requests for computation. Let us define the utilization factor U of a set of tasks to be $\Sigma_i\, C_i/D_i$ where C_i and D_i are, respectively, the computation time and deadline of the ith task, and every task is permitted to request new computation as soon as its current deadline expires, i.e., D_i is also the period of the ith task. Obviously, the condition $U \leq n$ is a necessary condition for scheduling a set of task on n processors. It is an open problem whether $U \leq n$ is also a sufficient condition for scheduling. However, we can give a partial answer to this problem in the following theorem. (We remind the reader that in our model, task preemption is allowed only at integral time boundaries.)

Theorem 8: Suppose Γ is a set of m periodic tasks with a utilization factor $U \leq n$. Let $T = \text{GCD}\,(D_1, \cdots, D_m)$ and let $t = \text{GCD}\,(T, T*C_1/D_1, \cdots, T*C_m/D_m)$, where C_i and D_i are, respectively, the computation time and period of the ith task in the task set Γ. Then a sufficient condition for scheduling Γ on n processors is that t be integral.

Proof: If t is integral, then T, $T*C_1/D_1, \cdots, t*C_m/D_m$ must all be integers. Our strategy is to execute task 1 for $T*C_1/D_1$ time units, task 2 for $T*C_2/D_2$ time units, $\cdots$, task m for $T*C_m/D_m$ time units every T time units. For task j, every request requires C_j time units and this has to be scheduled in D_j time units. We can divide up D_j time units into D_j/T (which is guaranteed to be an integer by definition of T and the sufficient condition) time slices, each of size T. In each slice, we allocate $T*C_j/D_j$ (which is an integer) units of computation time to task j. Hence, task j is allocated $(D_j/T) * (T*C_j/D_j) = C_j$ units of computation every period. In each time slice of size T, the total computation time allocated to all the tasks in the task set Γ is $T*(\Sigma_i\, C_i/D_i) = T*U$. The fact that $U \leq n$ and the time allocated to each task in a time slice does not exceed the size of the time slice guarantees feasibility.

For an example, see Fig. 13.

VI. Concluding Remarks

We have demonstrated in this paper a difficulty unique to run-time scheduling. For the system designer, our results suggest that it is unrealistic to specify an "optimal scheduler" without providing information about all the tasks which the multiprocessor system will have to handle. It was shown that except for the case of a uniprocessor, *a priori* knowledge of the start-times of the tasks is necessary for optimal scheduling. The fundamental problem is that the set of tasks may have two or more subsets which require premature commitment from a scheduler even before their start-times. If the start-times are not known *a priori*, then additional information must be made available to decide which of the conflicting subsets of tasks will require computation before the others. On the other hand, some task sets may meet certain conditions which guarantee that they can be scheduled optimally at run-time. A simple case is when all the tasks have unit computation time. For tasks with arbitrary computation times, we developed a sufficient condition which guarantees that a set of tasks does not have subsets which are in conflict with one another. Specifically, we proved that if a set of tasks can be successfully scheduled when their start-times are the same, then they can be scheduled at run-time even if their start-times are different. The important point here is not that the tasks can be scheduled (which is what we would expect), but that they can be scheduled without *a priori* knowledge of their start-times.

It should be pointed out that some task sets can be optimally scheduled at run-time even though their deadlines cannot be met when all of them have the same start-time. For example, consider three tasks all of which have unit computation time and zero laxity, i.e., all three have immediate deadlines. Obviously, they cannot be scheduled on two processors if all three request computation simultaneously. On the other hand, whenever it is possible to schedule the three tasks (given *a priori* knowledge of their individual start-times), then they can be scheduled at run-time by the Earliest Deadline algorithm. Thus we have for further research the interesting question of determining whether a task set is conflict free. In other words, if the preassigned laxities and computation times of a set of tasks are given, are there necessary and sufficient conditions for optimal scheduling without *a priori* knowledge of the start-times.

In general, we can extend the scheduling problem to allow a task to repeat its requests for computation. If the requests are strictly periodic, i.e., the start-times are evenly spaced apart and are therefore known *a priori*, then the flow maximization algorithm in [9] can be used to solve the problem. The time complexity of this approach, however, is exponential in the worst case. We have given a sufficient condition, testable in polynomial time for the periodic task scheduling problem. It is our conjecture that this same condition ($U \leq n$) is both necessary and sufficient for feasible scheduling in the multiprocessor case.

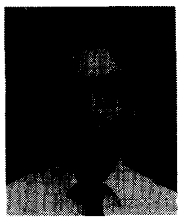
Acknowledgment

The authors would like to thank K. Nordbye for helpful discussion and J. Kempf for pointing out an error in an earlier version of this paper.

References

[1] S. Ward, "An approach to real-time computation," in *Proc. Seventh Texas Conf. Computing Systems*, Houston, TX, Oct. 1978, pp. 5.26–5.34.
[2] M. Dertouzos, "Control robotics: The procedural control of physical processes," in *Proc. IFIP Cong.*, 1974, pp. 807–813.
[3] P. Brucker, J. Lenstra, and A. H. G. Kan, "Complexity of machine scheduling problems," Mathematisch Centrum, Amsterdam, The Netherlands, Rep. BW 43/75, 1975.
[4] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *JACM*, vol. 20, no. 1, Jan. 1973.
[5] R. Henn, "Antwortzeitgesteuerte prozessorzuteilung unter strengen zeitbedingungen," *Computing*, vol. 19, pp. 209–220, 1978.
[6] M. Garey and D. Johnson, "Two-processor scheduling with start-times and deadlines," *SIAM J. Comput.*, vol. 6, pp. 416–426, 1977.
[7] ——, "Complexity results for multiprocessor scheduling under resource constraints," in *Proc. Eighth Annu. Princeton Conf. Information Sciences and Systems*, 1974, pp. 168–172.
[8] P. Brucker, M. Garey, and D. Johnson, "Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness," 1977, private communication.
[9] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints," *Nav. Res. Log. Quart.*, vol. 18, pp. 511–519, Dec. 1971.
[10] A Mok and S. Ward, "Distributed broadcasting channel access," *Comput. Networks*, vol. 3, no. 5, pp. 327–335, Nov. 1979.
[11] K. Nordbye and A. Mok, "Results in deadline scheduling with preemption," Real Time Systems Group., Lab. Comput. Sci., M.I.T., Internal Working Paper.
[12] A. Mok, "Task scheduling in the control robotics environment," Lab. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, Rep. TM-77, Sept. 1976.

Michael L. Dertouzos (S'58–M'65–SM'74–F'76), photograph and biography not available at the time of publication.



Aloysius Ka-Lau Mok (M'84) received the S.B. and S.M. degrees in electrical engineering and computer science, and the Ph.D. degree in computer science in 1983, all from the Massachusetts Institute of Technology, Cambridge.

He is currently Associate Professor of Computer Science at the University of Texas at Austin. His current interests include the design of robust distributed real-time systems and performance issues of real-time rule-based programs. He has been on the research staff of the Division of Sponsored Research at MIT, and has consulted for government and industry on real-time system design.

Dr. Mok is a member of Eta Kappa Nu and Tau Beta Pi.