Multiprocessor Scheduling of Hard-Real-Time Periodic Tasks with Task Migration Constraints [*]

Tse Lee and Albert Mo Kim Cheng
Department of Computer Science
University of Houston
Houston, TX 77204-3475
U.S.A.

Abstract

We present a sufficient condition for preemptive scheduling of a set of independent, periodic tasks on n identical processors with task migration constraints so that each task$_i$ can complete its computation time C_i before its deadline D_i. Any set of independent, periodic tasks satisfying this condition are guaranteed a feasible schedule at run time. This condition is optimal for a set of m independent and periodic tasks running on an n-processor system which requires a task migration time of $R (R > 1)$ cycles. Let $T = \gcd(D_1, D_2, \ldots, D_m)$. If the utilization factor (U) of a set of periodic tasks is at most $n * (T - R + 1)/T$, a feasible schedule exists. If $R \leq 1$, then the sufficient condition becomes $U \leq n$. Thus this paper solves the open problem stated as a conjecture in Dertouzos and Mok's paper[1] that the condition $U \leq n$ is both necessary and sufficient for feasible scheduling.

1 Introduction

The increasing availability of multiprocessor computer systems makes it possible to execute many tasks in parallel and thus reduce or eliminate violations of task deadlines in a variety of complex, real-time applications. Dertouzos and Mok [1] states: "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." In our paper, we solve the open problem stated as a conjecture in [1] that the condition $U \leq n$, where U is the utilization factor and n is the number of processors, is both necessary and sufficient for feasible scheduling, and extend it to account for task migration constraints. In [3], Naghibzadeh presents an MITSP algorithm to prove the sufficiency of the condition $U \leq n$ for scheduling a class of sporadic tasks, and to construct feasible schedules for tasks in this class. The MITSP strategy is only a theoretical concept because it allows two different tasks to run in the same quantum. The approximate MMTSP strategy which is adopted from MITSP greatly underutilizes each processor so it is not practical. However, our technique can handle every case as long as the condition $U \leq n$ is true, and maintains a high processor utilization.

Several studies focus on the uniprocessor scheduling of periodic tasks. A partial list of work follows. The earliest real-time uniprocessor scheduling algorithm appears in [4], where Liu and Layland use a rate monotonic algorithm for scheduling m periodic tasks on a uniprocessor. They show that no deadline will be missed if $U \leq \lfloor m(2^{1/m} - 1) \rfloor$. The least upper bound for U is on the order of 70 percent for the feasible scheduling of large task sets. In [6], Shih, Liu, and Liu use a modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines. If the amount of deference is sufficiently long, jobs can be feasibly scheduled as long as their total utilization is at most one. If the deadline of each job is deferred by one period of the job, a set of m independent periodic jobs can be feasibly scheduled if $U \leq \lfloor 1 + m(2^{1/m} - 1) \rfloor / 2$. In this case, U approaches 0.845 when the number of tasks approaches infinity.

Lehoczky [5] reports several simulations on the worst-case utilization bounds of different deadline postponement strategies. He shows that if one additional period is given to tasks to complete their computation requirement, the worst-case schedulable utilization bound increases from 0.690 to 0.811. Jef-

[*]This material is based upon work supported in part by the National Science Foundation under Award No. CCR-9111563 and by the Texas Advanced Research Program under Grant No. 3652270.

fay, Stanat, and Martel [7] show a set of scheduling conditions for scheduling periodic tasks and sporadic tasks on a uniprocessor system. Any set of periodic or sporadic tasks that satisfies those conditions can be scheduled with an earliest deadline algorithm. The following sections focus on the scheduling of periodic tasks on a multiprocessor system.

2 Ideal Scheduling Condition

In [1], Dertouzos and Mok propose a sufficient condition for scheduling a set of independent, periodic tasks. In the same paper, they suspect that the condition $U \leq n$ is a sufficient condition for scheduling independent, periodic tasks on n processors. We show that what they suspect is true in this section. Here, we assume that there is no overhead or latency associated with task interruption or preemption. We also assume that the task migration time R is 0, and that the task periods are equal to the task deadlines. A special processor which is not among the n processors is used for assigning tasks to processors. In Section 3, we shall account for task migration time by deriving a new and more realistic sufficient condition. Note that sporadic tasks can be transformed into an equivalent set of periodic tasks [2] so that both of these scheduling conditions can be applied. In this paper, the scheduling algorithm is omitted due to space limitations.

Let C_i and D_i respectively denote the computation time and deadline of $task_i$. If we can execute $task_i$ for $T*C_i/D_i$ time units for every T time units, then $task_i$ is allocated $(T*C_i/D_i)*(D_i/T) = C_i$ units of computation time in every deadline. Thus, in any time slice S_j, the total number of computation units allocated is $T* \sum_{i=1}^{m}(C_i/D_i)$, which we know should be at most $T*n$(the maximum computation power of an n-processor system in a time slice S_j). The first time slice(S_1) starts at time 0, the second time slice(S_2) starts at time T and so on. The following lemma states that no task will execute on two different processors simultaneously.

Lemma 1: Each $task_i$ can be allocated $T*C_i/D_i$ units in each time slice S_j of the Gantt Chart without two or more processors executing the same task at the same time if $U \leq n$.

Proof: Tasks are assigned to processors according to the order of the task IDs. When a task $task_i$ cannot be assigned to $processor_j$ because $processor_j$ no longer has time to accommodate the task, $task_i$ is assigned to $processor_{j+1}$. In the worst case, $task_i$ can request T(when $C_i = D_i$) computation units, and the first available $processor_j$ has X units left in the time slice. Here, we assign $task_i$ to $processor_j$ from time $T-X$ to T, and the remaining computation of $task_i$ to the next available $processor_{j+1}$ from time 0 to $T-X$. Thus, both $processor_j$ and $processor_{j+1}$ will not execute $task_i$ at the same time. □

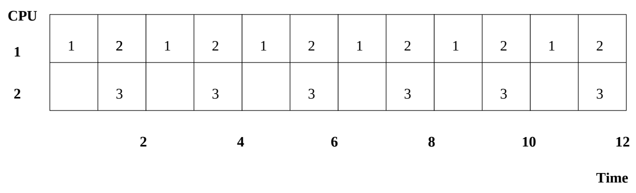

Figure 1: The Gantt Chart for the layout of the integral parts.

Ideally, we allocate $T*C_i/D_i$ time units in every time slice for every $task_i$. Because $T*C_i/D_i$ is not necessarily an integer, this method will allow two different tasks to execute in one CPU unit. However, this scenario is not possible in current systems. Thus, we use a swapping technique to make every computation block (CB) an integer block.

We allocate $\lfloor T*C_i/D_i \rfloor$ time units for every $task_i$ in every time slice in the Gantt Chart. Thus, in every time slice, for every $task_i$, the fractional part of $T*C_i/D_i$ is X_i, which is equal to $T*C_i/D_i - \lfloor T*C_i/D_i \rfloor$ and $0 \leq X_i < 1$. $Task_i$ will be allocated $\lfloor T*C_i/D_i \rfloor + 1$ time units in any X_i*D_i/T time slices before D_i, while in other time slices before D_i, $task_i$ is allocated only $\lfloor T*C_i/D_i \rfloor$ time units. After we perform the transformation, the total number of computation time units is the same for $task_i$ before the deadline D_i. The following lemma states that this can be done without causing any task to miss its deadline.

Lemma 2: The earliest deadline algorithm can be used to reschedule the fractional part of $T*C_i/D_i$ in each time slice without allowing any $task_i$ to miss its deadline.

Proof: The urgent tasks trade the fractional part of their computation shares in later time slices with the less urgent tasks at the current time slice. Note that the swapping technique will assign one more time unit each to those tasks which have earlier deadlines. Since the urgent tasks will yield the fractional parts of their

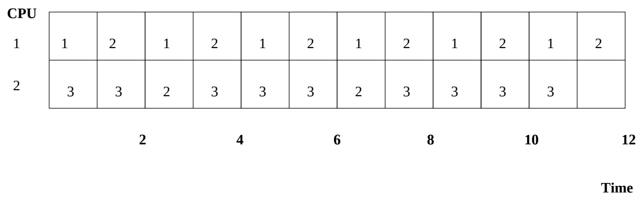

Figure 2: The complete layout of the Gantt Chart.

computation shares to the sacrificed tasks before the urgent tasks' deadlines and the sacrificed tasks should have later deadlines, swapping shares will not cause the sacrificed tasks to miss their deadlines. □

The same argument can be applied to the least laxity algorithm. The following theorem states the most general, feasible condition for scheduling periodic tasks.

Theorem 3: A necessary and sufficient condition for scheduling periodic tasks is $U \leq n$.

Proof: The necessary condition is trivial. Here, we prove that the sufficient condition for scheduling periodic tasks on n processors is also $U \leq n$. From Lemma 1, we know that we can squeeze the periodic tasks into the Gantt Chart if $U \leq n$. After applying Lemma 2, the periodic tasks can have integral computation blocks (CBs) in every time slice, and every task meets its deadline. When we apply this result to Lemma 1's allocation of CBs, we guarantee that no computation blocks of any task will overlap at any time in the Gantt Chart though the size of the CBs may change. Thus, the theorem is proved. □

Example: Consider the following task set. The number of processors n is 2.

task	*computation*	*period*
$task_1$	2	4
$task_2$	4	6
$task_3$	3	4

Note that $U = 2/4 + 4/6 + 3/4 < 2$, $T = gcd(4,6,4) = 2$, and $T' = lcm(4,6,4) = 12$. In each time slice, $task_1$ has $\lfloor 2*2/4 \rfloor = 1$ time unit, $task_2$ has $\lfloor 2*4/6 \rfloor = 1$ time unit, and $task_3$ has $\lfloor 2*3/4 \rfloor = 1$ time unit. Thus we obtain the Gantt Chart layout in Figure 1. The empty slots in Figure 1 show the space for the scheduling of the fractional parts. In the first time slice, the deadline of $task_3$ is 4, while the deadline of $task_2$ is 6. Thus, in T_1, $task_3$ is the most urgent task. In the second time slice, $task_3$ meets the computation requirement which is 3 time units before deadline 4. Thus, $task_2$ should get its share now. The following time slices follow the same idea. The Gantt Chart from S_1 to S_6 is shown in Figure 2.

3 Extended Model with Task Migration

Most published results assume that a task can be interrupted at any time and an interrupted task can be resumed immediately on a new processor without any latency or overhead. However, this assumption may not be realistic if each processor has its own cache in a multiprocessor system. In the following lemmas and theorems, we consider the time needed for task migration and data movement between different processors. Here, we assume that there is no overhead or latency associated with task interruption or preemption. Also, I/O processors are used for task migration, and a special processor which is not among the n processors is used for assigning tasks to processors.

Lemma 4: If a set of m periodic tasks satisfies the condition $U \leq n$, then within each time slice this set of tasks can meet the task migration requirement without missing any task deadlines in a system which requires one time unit for task migration between two processors.

Proof Sketch: We sort the integral computation blocks of each $task_j$ according to their sizes in non-decreasing order in every time slice T. For each $task_i$ which has a computation block size equal to T, we allocate the processor to $task_i$ in this time slice exclusively. For those tasks which have computation block sizes less than T, we can use a similar argument used in the proof of Lemma 1 to prove that within each time slice, a task migration time of one time unit can be satisfied. □

Lemma 4 guarantees that, within each time slice T, every $task_i$ can satisfy the system requirement if task migration occurs. Next, we consider the migration requirement between time slices. The following lemma shows that the task migration between time slices can satisfy the system migration requirement with only one time unit.

Lemma 5: If a set of m periodic tasks satisfies the condition $U \leq n$, then between time slices this set of tasks can meet the task migration requirement without missing any task deadline in a system which requires one time unit for task migration.

Proof: For every time slice, we sort the integral computation blocks of each task according to its size in non-decreasing order. There are two scenarios to consider. First, the computation block size of $task_i$ is equal to T. If we find a $processor_j$ which executes $task_i$ in the end of the previous time slice, then we allocate the $processor_j$ exclusively to $task_i$ in the current time slice. In this case, there is no task migration for $task_i$ between time slices. Otherwise, this task is assigned to any available processor left in the end exclusively. This situation arises if $task_i$ has computation block size $T-1$ in the previous time slice, and is scheduled from time 0 to time $T-1$ in the previous time slice. Although there may be a task migration between time slices for $task_i$, the migration occurs from time $T-1$ to T in the previous time slice.

Second, for any $task_i$ which has a computation block size less than T, we find the $processor_j$ which executes $task_i$ in the end of the previous time slice. We assign as many computation time units of $task_i$ as possible to the currently available processor. If the currently available processor cannot provide enough processing power for $T*C_i/D_i$, then the rest of $T*C_i/D_i$ will be assigned to $processor_j$ starting from time 0 of the current time slice. In this case, there is no task migration between time slices. However, in other cases, $task_i$ is only executed by the current processor. There are two possible situations:(1) $task_i$ starts at time 0 in the current processor, and (2) $task_i$ starts at another time in the current processor. In (1), since the current processor is $processor_j$, there is no task migration between time slices. Although task migration occurs in (2), task migration can be done from time 0 to time 1 in the current time slice. □

Theorem 6: A necessary and sufficient condition for scheduling periodic tasks on n processors is $U \leq n$ if the task migration time is one time unit.

Proof Sketch: With the property of Lemma 4 and Lemma 5, we prove this theorem by induction on the time slice. This property is obvious. □

Theorem 6 shows that the condition $U \leq n$ can be applied to any system with a migration overhead of one time unit.

We now extend this task migration constraint to more general cases. The basic idea of meeting task migration constraints is that we can trade idle pro-

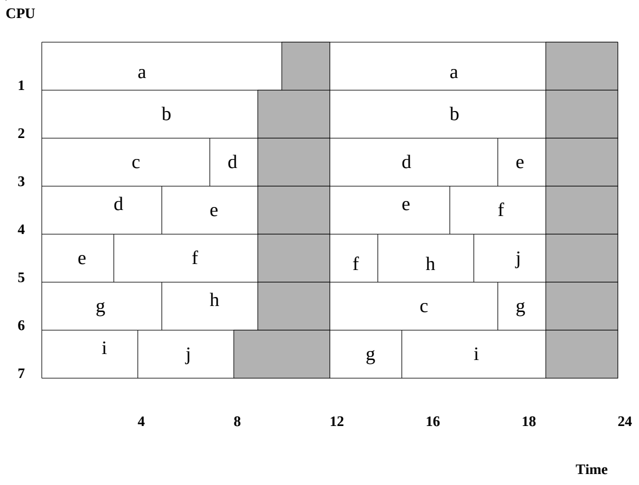

Figure 3: The Gantt Chart before shifting computation blocks.

cessor time for task migration time. Ideally, every $processor_j$ has $R-1$ idle processor time units in each time slice if a system needs R time units for task migration. When we combine the fact of Theorem 6 and the reservation of $R-1$ idle processor time units in every $processor_j$, we derive the following theorem.

Theorem 7: For a set of m periodic tasks running on n processors with a utilization factor U, let time slice $T = \gcd(D_1, D_2, \ldots, D_m)$ and the system requirement for task migration time between processors be $R(>1)$ process units. A sufficient condition for scheduling m periodic tasks is $U \leq n*(T-R+1)/T$.

Proof: Similar to the approach given in the proof of Lemma 4, we allocate one processor exclusively to $task_i$ in the current time slice for any $task_i$ which has a computation block size greater than or equal to $T-R+1$. The remaining tasks with computation block sizes less than $T-R$ will be assigned in the same way as we do in the proof of Lemma 4, but only from time 0 to time $T-R$ for the rest of the processors in the Gantt Chart. The Gantt Chart in Figure 3 shows only one process time unit for task migration. However, we can make use of the idle processor time in $processor_j$. We shift $task_i$ in $processor_j$ to the right of the current time slice. Referring to Figure 4, tasks c, d, e, f, and g are shifted right, making it possible for these tasks to meet the task migration constraint R. Tasks a, b, and f in the first time slice are shifted to minimize context-switching time.

To take into account the task migration between time slices, we use a similar argument as in the proof

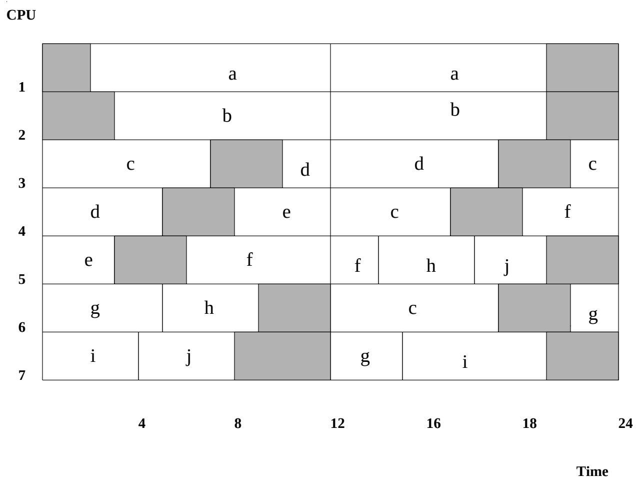

Figure 4: The Gantt Chart after shifting computation blocks.

of Lemma 5. Consider the situation shown in Figure 3, tasks c, g, h, and j have task migration between time slices. We use the same inductive argument as in the proof of Theorem 6 to show that the sufficient condition ($U \leq n*(T-R+1)/T$) guarantees that this set of tasks can meet the task migration constraint without missing any task deadlines. □

There is one problem with this sufficient condition. When T becomes 1, the overhead becomes too large to be practical. We use two approaches to solve this problem. The first is to combine two adjacent time slices into a larger time slice. The second is to use dynamically adjustable deadlines to obtain a larger T value. Based on the above lemmas and theorems, a scheduling algorithm has been implemented for periodic tasks with task migration constraints. Due to space limitations, we do not describe this algorithm here.

4 Conclusion

In this paper, we proved that $U \leq n$ is a sufficient condition for the feasible scheduling in an n-processor system under the assumption that the cost for task and status transfer between processors is at most one clock cycle. For the more general case where this cost is more than one clock cycle, we derived a new sufficient condition to account for variable task migration time(R). The sufficient condition is $U \leq n*(T-R+1)/T$. The algorithms based on these scheduling conditions have been implemented. We can also consider the cost of context switching in our model. In general, we can rewrite the computation in a period as: $C_i' = C_i + Y_i X_i$, where C_i is the original computation requirement, Y_i is the context switching cost each time it occurs, and X_i is the number of time slices in a period D_i. Ongoing work investigates the use of dynamically adjustable deadlines to obtain a larger T value.

References

[1] M. L. Dertouzos and A. K. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks," *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1497-1506, Dec. 1989.

[2] A. K. Mok, "The design of real-time programming systems based on process models," *in Proc. 5th IEEE Real-Time Systems Symp.*, pp. 5-17, Dec. 1984.

[3] M. Naghibzadeh, "Analytic design and verification of overrun-free uniprocessor and multiprocessor real-time computing systems," Ph.D Dissertation, Dept. of EE-Systems, Univ. Southern Calif., June 1980.

[4] C. L. Liu and J. W. Layland, "Scheduling Algorithm for multiprogramming in a hard-real time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[5] J. P. Lehoczky, "Fixed priority scheduling of periodic tasks set with arbitrary deadlines," *in Proc. 11th IEEE Real-Time Systems Symp.*, pp. 201-209, Dec. 1990.

[6] W. K. Shih, J. W. S. Liu, and C. L. Liu, "Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines," *IEEE Trans. Software Eng.*, vol. 19, no. 12, pp. 1171-1179, Dec. 1993.

[7] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," *in Proc. 12th IEEE Real-Time Systems Symp.*, pp. 129-139, Dec. 1991.