Deadline Scheduling with Processor Affinity and Feasibility Check on Uniform Parallel Machines

Vahid Salmani
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran
salmani@um.ac.ir

Mahmoud Naghibzadeh
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran
naghib@um.ac.ir

Mohsen Kahani
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran
kahani@um.ac.ir

Abstract

It has been proved that there is no optimal online scheduler for uniform parallel machines. Despite its non-optimality, EDF is an appropriate algorithm to use in such environments. However, its performance significantly degrades in overloaded situations. Moreover, EDF produces a relatively large number of migrations which may prove unacceptable for use on some parallel machines. In this paper a new deadline-based algorithm for scheduling real-time tasks on uniform parallel machines is presented. The performance of this algorithm is then compared with that of EDF algorithm. It is shown that our proposed approach not only demonstrates a performance close to that of EDF in non-overloaded conditions but also has supremacy over EDF in overloaded situations in many aspects. Furthermore, it imposes much less overhead on the system.

1. Introduction

In real-time systems each task has a deadline before or at which it should be completed. A scheduling algorithm is a set of rules that determines which task(s) should be executed in any given instance. Due to the tasks' criticality, scheduling algorithms should be timely and predictable.

As parallel systems are used in real-time applications, scheduling of real-time tasks in such systems is of much significance. Two important types of these systems are identical and uniform parallel machines. In the former the processing power of all processors is the same, whereas, each processor might have a different processing power in the latter case. It has been shown that in general there is no optimal scheduling algorithm for such systems [1].

Two important parameters affecting the performance of scheduling algorithms on parallel machines are preemption and migration. A job executing on a processor can be interrupted at any time, and its execution resumed later on the same or a different processor. If an interrupted task resumes execution on the same processor an (inter-processor) preemption has occurred, and if its execution resumes on a different processor a migration has happened.

Based on the two aforementioned factors, there are two types of scheduling policies in multiprocessor environments named global scheduling and partition scheduling [2]. Global scheduling algorithms put all the arrived tasks with non-zero remaining execution time into a queue that is common among the processing nodes. In a system with \( m \) processors, in every moment, \( m \) tasks having the highest priorities should be executing considering preemptions and migrations, if necessary. Partition scheduling algorithms divide the task set into partitions (subsets) such that all the tasks within a partition are assigned to a processor. In this policy task migrations are not allowed.

2. Related works

A uniform multiprocessor scheduling algorithm is defined to be work-conserving if and only if it satisfies the following conditions [3]:

- No processor is idled while there are active jobs awaiting execution.
- If at some instant there are fewer than \( m \) active jobs awaiting execution (\( m \) denotes the number of processors in the uniform multiprocessor platform), then the active jobs are executed on the fastest processors.
If the execution requirement of tasks is known at their arrival, it is possible to predict whether a task will meet its deadline or not. In a real-time system if a task cannot complete its execution and misses its deadline for any reason, it not only wastes the CPU time but also lessens the chance of the other tasks waiting for execution to be completed [4]. Apparently, a work conserving algorithm should not execute jobs which will finally miss their deadline. As a result, we define a third condition for a work conserving algorithm as follows:

- As long as there are \( m \) jobs having the chance to meet their deadlines, jobs which will not finish on time are not picked up for execution even though they have a relatively nearer deadline.

The earliest deadline first scheduling algorithm (EDF) chooses for execution at each instant in time the currently active job(s) that have the nearest deadlines. The EDF implementation upon uniform parallel machines is according to the following rules [3]:

- No processor is idled while there are active jobs waiting for execution.
- When fewer than \( m \) jobs are active, they are required to execute on the fastest processors while the slowest are idled.
- Higher priority jobs are executed on faster processors.

In [3] Baruah et al. came to the conclusion that despite its non-optimality, EDF is an appropriate algorithm to use for online scheduling on uniform multiprocessors. However, their implementation suffers from a great number of migrations due to vast fluctuations caused by finishing or arrival of jobs with relatively nearer deadlines. Task migration cost might be very high. For example, in loosely coupled systems such as a cluster of workstations a migration is performed so slowly that the overhead resulting from excessive migrations may prove unacceptable [5]. Another disadvantage of EDF is that its behavior becomes unpredictable in overloaded situations. That is, there is no guarantee on which jobs will meet their deadline. Therefore, the performance of EDF drops in overloaded conditions such that it can not be considered for use.

3. Real-time system model

This research is concentrated on uniform parallel machines in soft real-time environments. The algorithms being investigated are on-line and use up-to-date information for the scheduling activities during the systems execution.

We have focused on periodic tasks and each task's deadline is equal to its period. The reason for this choice is that it has been proved that a periodic task model is useful for modeling and analysis of majority of real-time systems [6]. Moreover, load factor measurement is easier and more accurate for periodic tasks. All tasks are synchronous i.e. their first request arrive simultaneously at the time zero. Such systems are common and applicable [7].

Tasks are preemptable and in each scheduling event a dispatcher decides which task to be performed next. In addition, a task is not allowed to run concurrently (on more than one processor at a time). Tasks must declare their characteristics and requirements such as interval, deadline and Worst Case Execution Time (WCET) at their arrival. The intervals and execution times are correct multiples of one time slice. The actual execution time of each task is equal to its WCET. Tasks are not removed from the local memory or cache before they migrate to another node.

Scheduling algorithms must prevent simultaneous access to resources and shared devices. We assume the tasks are independent and do not need to do I/O operations. Therefore, the concurrency control matters have not been considered.

4. Proposed approach

We have applied some modifications to the global EDF algorithm to decrease the number of task migrations and also to add predictability to its behavior. In order to decrease the number of migrations we prevent a job from moving to another processor if it is among the \( m \) higher priority jobs. Therefore, a job will continue its execution on the same processor if possible. This concept is known as processor affinity.

Our motivation for exploiting processor affinity derives from the observation that, for many parallel applications, the time spent bringing data into the local memory or cache is a significant source of overhead, ranging between 30% to 60% of the total execution time [8]. While migration is unavoidable in the global schemes, it is possible to minimize migrations caused by a poor assignment of tasks to processors. By scheduling tasks on the processor whose local memory or cache already contains the necessary data, we can significantly reduce the execution time and thus overhead of the system.

It is worth mentioning that still a job might migrate to another processor when there are two or more jobs that were last executed on the same processor. A migration might also happen when the number of ready jobs becomes less than the number of processors. This
fact means that our proposed algorithm is a work-conserving one.

In order to give the scheduler a more predictable behavior we first perform a feasibility check to see whether a job has a chance to meet its deadline. If so, the job is allowed to get executed. Having known the deadline of a task and its remaining execution time it is possible to verify whether it has the opportunity to meet its deadline. More precisely, this verification can be done by examining the tasks’ laxity. The laxity of a real-time task $T_i$ at time $t$, $L_i(t)$, is defined as follows:

$$L_i(t) = D_i(t) - E_i(t)$$

where $D_i(t)$ is the deadline by which the task $T_i$ must be completed and $E_i(t)$ is the amount of computation remaining to be performed. In other words, laxity is a measure of the available flexibility for scheduling a task. A laxity of $L_i(t)$ means that if the task $T_i$ is delayed at most by $L_i(t)$ time units, it will still have the opportunity to meet its deadline. A task with zero laxity must be scheduled right away and executed without preemption or it will fail to meet its deadline. A negative laxity indicates that the task will miss the deadline, no matter when it is picked up for execution.

We call this novel approach the Earliest Feasible Deadline First (EFDF) scheduling algorithm whose details are presented in the following section.

4.1. EFDF scheduling algorithm

Let $m$ denote the number of processing nodes and $n$, $(n \geq m)$ denote the number of available tasks in a uniform parallel real-time system. Let $s_1, s_2, \ldots, s_m$ denote the computing capacity of available processing nodes indexed in a non-increasing manner: $s_j \geq s_{j+1}$ for all $j$, $1 < j < m$. We assume that all speeds are positive – i.e., $s_j > 0$ for all $j$.

In this section we present the five steps of the EFDF algorithm. Obviously, each task which is picked up for execution is not considered for execution by other processors.

1. Perform a feasibility check to specify the tasks which have a chance to meet their deadlines and put them in set $A$. Put the remaining tasks in set $B$.
2. Sort both task sets $A$ and $B$ according to their deadline in a non-descending order.
   Let $k$ denote the number of tasks in set $A$ – i.e. the number of tasks that have the opportunity to meet their deadline.
3. For all processor $j$, $(j < \min(k,m))$ check whether a task which was last running on the $j^{th}$ processor is among the first $\min(k,m)$ tasks of set $A$. If so assign it to the $j^{th}$ processor.

4. For all $j$, $(j < \min(k,m))$ if no task is assigned to the $j^{th}$ processor, select the task with earliest deadline from remaining tasks of $A$ and assign it to the $j^{th}$ processor.
   If $k \geq m$, each processor have a task to process and the algorithm is finished.
5. If $k < m$, for all $j$, $(k < j \leq m)$ assign the task with smallest deadline from $B$ to the $j^{th}$ processor.
   The last step is optional and all of the tasks from $B$ will miss their deadlines.

4.2. Time complexity

The EDF algorithm simply sorts the tasks with regard to their deadlines and picks up the $m$ jobs having the nearest deadlines at each scheduling event. Therefore, the time complexity of EDF is equal to that of a typical sorting algorithm which is $O(n, \log n)$.

The time complexities of the five steps of EFDF are $O(n)$, $O(n, \log n)$, $O(m)$, $O(m)$, $O(m)$, respectively. Since the steps are independent of each other and $n \geq m$, the complexity of the EFDF algorithm is $O(n, \log n)$ too.

5. Performance evaluation

In this section, we study the performance of our proposed algorithm (EFDF) based on simulation and compare it with that of global EDF algorithm. This evaluation can be conducted in two major ways [4]:

• Examining the effect of a specific parameter (e.g. load factor) on a variety of performance metrics as dependent variables.
• Investigating the effect of different parameters as independent variables on a specific performance metric (e.g. success ratio).

Since presenting the result of the above methods together requires a lot of space, we choose the former approach. The latter method can be used in a separate paper. Therefore, the load factor is considered as main parameter and its effect on the performance metrics below as dependent variables is presented [9]:

- success ratio
- response time
- preemptions and migrations
- CPU utilization
- load balance

In the results to be presented, in order to minimize the influence of exceptional situations, each
experiment was repeated 100 times using different task sets and the results were averaged out. The simulation time is equal to a meta period which is equivalent to the smallest common multiple of all tasks’ periods. The presented results are in fact the average of the obtained values from all processors. In the following experiments, the values of the parameters are considered as below, unless mentioned otherwise.

- The number of tasks in the system is a random number with a uniform distribution between 30 and 150.
- The period of tasks is a random number with a uniform distribution between 10 and 10000.
- The WCET of each task is a random number between 1 and 40% of its period.
- Tasks are preemptable.
- The number of processors is a random number with a uniform distribution between 4 and 8.
- The processing capacity of each processor is at most 4 times as much as that of the slowest one.
- The overall processing capacity of the system is equal to 20.
- The load factor fluctuates between 0 and 40 in order to evaluate the performance of algorithms in both non-overloaded and overloaded conditions.

Load factor of task $T_i$ is defined as the ratio of its WCET ($E_i$) to its request period ($P_i$). For $n$ periodic tasks, load factor is equal to:

$$L = \sum_{i=1}^{n} \frac{E_i}{P_i}$$

In parallel environments, the overall load factor is the sum of all processors’ load factor.

8.1. Success ratio

Success ratio is defined as the ratio of the jobs that have been successfully completed to the jobs that arrived to the system [3]. As illustrated in figure 1, both algorithms show a near optimal performance in non-overloaded situations. Nonetheless, in overloaded conditions, the performance of the both methods descends. Part of this performance degradation is due to the fact that the system does not have the capacity to meet all deadlines. By performing feasibility check, however, EFDF tries to fully utilize the computing capacity of available processing nodes, and therefore shows a better performance.

8.2. Response time

Response time is defined as the time between arriving a request and completion of its processing [10]. Obviously it is not influenced by the tasks which fail to meet their deadlines. A point to be considered is that due to presence of the tasks with different periods, the absolute numeric values are meaningless. Therefore, we use response ratio instead of response time and define it as the ratio of a task's response time to its period. In calculation of these values, scheduling overhead and communication time between processors has been ignored.

Figure 2 depicts the observed response ratio. In non-overloaded situations the two algorithms have close performances. In overloaded conditions, however, EFDF algorithm shows a better performance. The diagram suggests that in overloaded conditions tasks are completed relatively sooner when scheduled by EFDF.
8.3. Preemptions and migrations

One of the most significant factors influencing scheduling overhead is the number of produced preemptions and migrations, and our aim here is to measure their values for each of the compared algorithms. In this case, due to different number of tasks in diverse conditions, applying absolute numeral values is meaningless. As a result, we use preemption ratio and migration ratio, instead of preemption and migration, and define them as the ratio of the total number of preemptions to the total number of arrived requests that have the chance to be executed, and the ratio of the total number of migrations to the total number of arrived requests that are picked up for execution, respectively.

Figure 3 illustrates the produced (inter-processor) preemption ratio indicating that the supremacy is with EDF algorithm. A preemption ratio of 1.0 means that on average jobs are preempted once per release. Since preemption ratio for both algorithms is less than 1.0 it is acceptable and the two algorithms incur a very trivial overhead in term of preemptions.

8.4. CPU utilization

CPU utilization is the percentage of CPU time in which the CPU has not been idle with respect to the time passed. Therefore, it does not include the times in which CPU has had idle processing or has been processing the jobs which have ultimately been missed. In figure 5 the CPU utilization of the two algorithms has been illustrated. Both algorithms have approximately the same performance in low load factor conditions and use the maximum possible CPU resources. However, in overloaded conditions the EFDF algorithm almost fully utilizes the processors. This considerable improvement is due to performing feasibility check.

8.5. Load balance

Load balance means steady distribution of load among processors in such a way that minimizes the load difference. Regular load balance among
processors not only decreases the response time, but also increases system's reliability which is very significant in real-time systems. Another advantage of a balanced system is the minimized total power consumption. The length of schedule in balanced case is also minimized. We apply the formula below for defining the system's load balance [11]:

\[
\frac{1}{m} \sum_{j=1}^{m} |U_j - \overline{U}| \times \frac{1}{m \times U}
\]

in which \( m \) is the number of processors. \( \overline{U} \) denotes the average CPU utilization and \( U_j \) represents the \( j \)th processor's utilization. Figure 6 illustrates the load balance for both algorithms. Apparently, EFDF algorithm results in a balanced schedule in overloaded conditions.

![Figure 6. Load balance](image)

9. Conclusion

In this paper a new deadline-based algorithm, called EFDF, for scheduling real-time tasks on uniform parallel machines is presented. Applying processor affinity and feasibility check to the global EDF decreases the number of task migrations and also gives the scheduler a more predictable behavior. The performance of this algorithm is then compared with that of EDF algorithm. It is shown than our proposed approach not only demonstrates a performance close to that of EDF in non-overloaded conditions but also it has supremacy over EDF in overloaded situations in many aspects. We show that traditional global EDF algorithm which ignores the location of tasks when assigning them to processors, incurs a significant performance penalty on the system. Since EFDF imposes much less overhead on the system, it could be more appropriate for use on parallel machines in which the cost of migrations is relatively high.

10. References


