A Dynamic Real-Time Scheduler for Shared Memory Multiprocessors

S. Sáez†, J. Vila, A. Crespo
Grupo de Sistemas de Tiempo Real
Dpto. Ingeniería de Sistemas Computadores y Automática (DISCA)
Universidad Politécnica de Valencia.
P.O. Box 22012, VALENCIA 46071, SPAIN.
E-mail: ssaez,jvila,alfons@disca.upv.es

Abstract

This paper presents a dynamic scheduling algorithm for multiprocessor systems which is guarantee-oriented. The algorithm is based on dynamic priority policy where priorities are inversely proportional to the latest start time (LST) of tasks. The paper includes a comparative study of the algorithm with other two well known dynamic algorithms: the Earliest Deadline First (EDF) and the Least Laxity First (LLF). The results show that LST is able to schedule some loads where EDF fails and reduces significantly the number of context switches with respect LLF. The algorithm has been implemented and tested on a tool for multiprocessor analysis and simulation which is also presented in this paper.

1. Introduction

The increasing availability of commercial multiprocessors has made possible the development of hard real-time applications on this platforms at reasonable costs.

One of the major advantages of multiprocessor systems is increased computing power, however how to make use of this "extra" power to enhance the response or the ability to accept new tasks of a hard real-time system is difficult and there is no much experience in practical algorithms to solve the problem in an acceptable way.

Theoretical results show that the problem of scheduling with hard timing constraints on multiprocessors is NP-complete in the case of static scheduling except for some reduced sets of hypothesis where it can be solved with polynomial complexity [4]. In dynamic scheduling the complexity is much higher and the main result establishes the lack of an optimal deadline algorithm [3] (for the general case), considering as an optimal algorithm one that is able to find feasible schedules in a set of scenarios which includes the set of scenarios scheduled by any other algorithm. For that reason, research in this area focuses on developing practical algorithms which are optimal under some special hypothesis or, although not optimal, exhibit "good results" at a reasonable computational cost.

Two of the most well known dynamic algorithms for multiprocessor systems are the generalization of the Earliest Deadline First (EDF) [4] and Least Laxity First (LLF) [2] for multiprocessors. But it is also known that these algorithms have important problems in their multiprocessor versions. On one hand, the EDF algorithm is not able to produce feasible schedules in very simple cases [4], but performs few context switches; on the other, LLF reports as schedulables much more cases than EDF, but performs too much context switches.

This paper presents a dynamic scheduling algorithm for multiprocessor systems which is guarantee-oriented. Dynamic scheduling means in the context of this paper that the algorithm has complete knowledge of the currently active set of tasks, but new arrivals may occur in the future, not known to the algorithm at the time it is scheduling the current set. Guarantee-oriented means that the algorithm performs a feasibility check every time a new task arrives to the system in order to accept it. If the feasibility check is passed the new set of accepted tasks is guaranteed to meet their deadlines but the contrary does not hold, since an optimal algorithm does not exist for the general case.

The main advantage of the proposed algorithm is that it is able to produce feasible schedules in similar cases to LLF but reduces significantly the number of
context switches. This is done at a reasonable computational cost.

The algorithm has been implemented and tested on a tool for multiprocessor analysis and simulation which is also presented in this paper.

2. Algorithm description

This section describes the proposed scheduling algorithm and how it performs the feasibility checking.

The assumed system model is a task model in which tasks execute on a shared memory multiprocessor, with a global scheduling algorithm. The system allows task migration between processors at very low cost. Tasks are are preemptible, independent, and have no resource requirements (only a processor for executing the code). They are described by two parameters: the worst case execution time $C_i$, and the deadline $D_i$. According to this, a task will be represented as $T_i(D_i, C_i)$.

The algorithm is a guarantee-oriented dynamic scheduler and consists of two main components which are closely related:

A feasibility checker: It is executed every time a task arrives to the system to the multiprocessor. In that situation, the algorithm considers the set of all currently active tasks plus the newly arrived one and performs a feasibility check to accept or reject the new task. This check only accepts a new task if this decision does not cause timing constraints violations of the already guaranteed tasks.

A CPU allocator: It assigns tasks to processors using a dynamic priority scheme based on the concept of latest start time (LST). The LST of a task for a given load conditions is defined as the latest time in which it is possible to start a task so it can meet its deadline. The goal of this policy is to minimize context switches. LST's are reevaluated by the feasibility checker every time a new task is accepted.

The check to accept a new task in a multiprocessor node is based on constructing a feasibility chronogram for each processor. The aim of this chronogram is to calculate the LST of every accepted task and it represents a worst case schedule. The construction of this chronogram is explained below. The feasibility check basically consists of ensuring that the LST of every task is greater or equal than current time. It is worth noting that the LST of task $T_i$, denoted as $LST_i$, may vary every time a new task is accepted.

The schedule produced by the feasibility check does not match (necessarily) the final schedule of the processors, since a task can start executing before its latest start time if some processor is free. However, this schedule is important because it provides a feasibility check at a reasonable cost and because it is the base for a final schedule which is both feasible and efficient. The feasibility of the final schedule comes from the fact that if task $T_i$ is schedulable at $LST_i$, it is also schedulable at an earlier time. The efficiency refers to the low number of context switches.

The construction of feasibility chronogram and the final schedules produced by the algorithm are explained below according to the different situations which may arise:

2.1. Basic algorithm

The description of the basic algorithm is explained using the example of figure 1 which shows one of the most straightforward cases. This example considers a biprocessor system and a task load at instant 0 which consists of $T_1(14, 8), T_2(15, 8)$ and $T_3(13, 7)$.

![Figure 1. Example 1](image_url)

The construction of the feasibility chronogram consists of the following steps: first, tasks are ordered by decreasing deadline ($T_3(15) \rightarrow T_1(14) \rightarrow T_2(13)$) and then, using this order, they are scheduled on the processor where they can meet their deadline with latest start time (LST). This way, $T_3$ is scheduled on $P_1$ with
$LST_1 = 7$, $T_1$ on $P_2$ with $LST_2 = 6$ and $T_3$ on $P_1$ with $LST_3 = 0$. The resulting feasibility chronogram is shown in figure 1 (a).

The basic feasibility check consists of ensuring that $LST_i \geq 0, \forall i$. This is an obvious sufficient condition for schedulability, although somewhat restrictive as it will be seen in the following cases.

The real schedule of tasks is based on using the LST deduced from the feasibility chronogram as a dynamic preemptive priority. According to this, tasks are ordered by LST in a single ready queue: $T_3(0) \rightarrow T_1(6) \rightarrow T_2(7)$. Tasks at the head of the queue are assigned to processors as they become free. This yields the real schedule of figure 1 (b). Note that this schedule only differs from the feasibility chronogram in that $T_2$ can start its execution in $t = 0$, that is, earlier than its LST.

In this case, preemption never occurs since all tasks find a free processor before their LST. As a matter of fact, the goal of the LST policy is to minimize preemption. Preemption could only occur if some new task arrives and it would be assigned an LST earlier than current tasks.

Figures 1 (c) and 1 (d) show the schedule produced by the EDF and LLF algorithms. In this case, the LST schedule matches the EDF schedule and reduces significantly the number of context switches with respect the LLF schedule.

2.2. Improvement 1

There are some load conditions in which the previous check fails. Failing means that while the check reports them as not schedulables (there is some $LST_i < 0$) the LST policy is able to produce a timely schedule however. The aim of this subsection is to characterize these situations and to improve the feasibility check so they can be reported as schedulables.

Figure 2 is an example of such a case. This case only differs from the previous one in that $T_3$ has four more units of computation: $T_1(14,8), T_2(15,8), T_3(13,11)$. According to this, tasks are prescheduled as shown in figure 1 (a) resulting in the following LST's: $LST_1 = 6, LST_2 = 7, LST_3 = -5$. Apparently $T_3$ is too long to produce a timely schedule since $LST_3 < 0$, which violates the previous feasibility check. However this load can be easily scheduled by the LST policy by just replacing negative LST’s with 0 (the highest priority). The resulting priority assignment produced by the feasibility checker in this case would be thus identical to the former case. Nevertheless, the real-time execution, shown in figure 1 (b), differs from the previous one in one main aspect: preemption occurs although no new tasks arrive.

In the example when $t = LST_2 = 7$ arrives, $T_2$ preempts the task with most laxity, that is $T_1$. In the case of preemption the LST of the preempted task ($T_1$) is reevaluated building a new chronogram for each processor where only ready tasks are taken into account. In this case $LST_1$ is reassigned to 13 and $T_1$’s execution is terminated later on $P_1$ after $T_3$’s execution.

Obviously, not all cases with negative LST’s are schedulables: there must be enough free time in other processors to accommodate the “excess of computation” of tasks with negative LST’s before their deadlines. This could be stated in terms of the feasibility check as follows: $\sum_{i=1}^{n} EST_i \geq 0$ where $EST_i$ is the earliest LST for processor $P_i$, and $n$ is the number of processors. In this case $\sum_{i=1}^{2} EST_i = -5 + 7 = 2$.

Figures 1 (c) and 1 (d) show the schedule produced by the EDF and LLF algorithms. In these kind of scenarios EDF often fails to produce timely schedules.

2.3. Improvement 2

This subsection introduces a new improvement in the algorithm whose goal is to produce timely schedules for some load cases that the previous check reports as not schedulables and where the former LST assignment would lead to deadline violations.

This improvement applies to cases where the construction of the feasibility chronogram contains “gaps” (free CPU time between tasks). Figure 3 is an illus-
A tractive example of such situation. This scenario only differs from the former one in the introduction of a new task $T_4$: $T_1(14, 8), T_2(15, 8), T_3(13, 11), T_4(3, 2)$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.png}
\caption{Example 3}
\end{figure}

According to the rules for the construction of the feasibility chronogram, tasks are prescheduled as shown in figure 1 (a): $T_1$ would be scheduled on $P_2$ with $LST_1 = 6$, $T_2$ on $P_1$ with $LST_2 = 7$, $T_3$ on $P_1$ with $LST_3 = -4$, and $T_4$ on $P_2$ with $LST_4 = 1$. This case would be rejected by the previous feasibility check since $EST_1 + EST_2 = -4 + 1 = -3$. However, note that the schedule of the feasibility check is far from being optimal, since it leaves unused free time between $T_4$ and $T_2$. A feasible schedule which takes advantage of this gap is proposed next.

The improvement of the feasibility chronogram is based on the following observation: Some task A with LST after some gap anticipates its LST to take advantage of some portion $\Delta$ of that gap and yields task B, with the earliest LST, the same $\Delta$ before B's deadline. In this case A could be $T_1$ while B is $T_3$.

According to this, the feasibility chronogram is reorganized as shown in figure 3 (b) where $T_1$ uses all the three units of the gap, and yields $T_3$ those three units before $T_3$'s deadline. This reorganization produces the following LST assignment: $LST_1 = 1, LST_2 = 7, LST_3 = -1$ (which would be reassigned to 0), and $LST_4 = 0$.

Not always the free CPU time of a gap can be fully used. The crucial question in this case is how much can be the LST of a task delayed when there are gaps in the feasibility chronogram. The answer is that the LST of the task must be delayed the minimum of:

- The gap size.
- The freedom of the task: $(D_i - LST_i) - C_i$.
- The maximum movement needed: $D_{gap} - LST_i$.

where $D_{gap}$ is the deadline of the tasks who has produced the gap. In this case is $T_4$.

In this case the new $LST_3$ should be reassigned to the $LST_3 + \min(3, 6, 7) = -1$ so the original $LST_3 = -4$ is effectively reassigned to $-1$, as previously argued, and then to 0.

After the proposed LST reassignment the same feasibility check than in the previous case applies: $\sum_{i=1}^{n} EST_i \geq 0$. In this case, $EST_1 + EST_2 = -1 + 1 = 0$.

The real-time schedule of this case is shown in figure 3 (c). Preemption occurs in similar situations as in the previous case: $T_1$ is preempted by $T_2$ when $LST_2$ arrives.

Figures 1 (d) and 1 (e) show the schedule produced by the EDF and LLF algorithms. As in previous case, the EDF algorithm has difficulties to find a feasible schedule.

3. Description of Multiprocessor Real-Time Tool

The scheduling algorithm presented in the previous section has been implemented and tested on a tool for multiprocessor analysis and simulation. This tool executes on an XView window system and the EDF, LLF, LST algorithms are built-in. This section describes the main capabilities of this tool which has been developed as an aid to perform the analysis of the proposed scheduler and to develop new ones.

This tool allows the implementation of scheduler prototypes, being able to analyze their behaviors with different set of real-time tasks. There are two ways to define the set of tasks depending on the selected execution mode: analysis or simulation. The tool capabilities in each mode and the statistical results offered are described in the following subsections.

3.1. Analysis mode

The analysis mode is used to analyze the scheduler algorithm in its development phase, where the user is
mainly interested in the correctness of the prototype, more than in the statistical results.

The tool allows the scheduling algorithm to be debugged by providing a feasibility checking for the best-effort algorithms and statistics of rejected tasks for the guarantee-oriented ones.

![Figure 4. Task Set Descriptor panel.](image)

Task can be specified as periodic, pure aperiodic or aperiodic with an exponential arrival rate. The tool also allows to specify the worst case execution time, the deadline and an initial offset as shown in figure 4. Once the set of tasks has been specified, the user can visualize the produced scheduling in two ways: a timing diagram where all task features are displayed (figure 5) or a Gantt chart showing the load distribution between processors (figure 6). The simulation length and its start point can be specified on both of these windows. It is also possible to locate missed deadlines in an automatic way.

![Figure 5. Timing diagram.](image)

![Figure 6. Gantt chart.](image)

where the user can compare the behavior of all provided scheduling algorithms. This blackboard can visualize the time required by the task set, the processor time that the system must have served at each time slot, the maximum power of the system, and the behavior of every implemented algorithm in the tool.

### 3.2. Simulation mode

The simulation mode has been developed to analyze the performance of the already tested scheduling algorithms. The main subject of this mode is to allow the user to specify a workload in a flexible way, to perform an exhaustive simulation of the prototype.

The workload descriptor allows the user to characterize the load parameters using some random distributions. This way, the worst case execution time, deadline, arrival rate and jitter of every task can be specified using constant, uniform, normal or exponential distributions as shown in figure 7.

![Figure 7. Task Templates panel.](image)

Once the workload has been specified, the tool can perform an exhaustive simulation of the provided algorithms, producing a wide set of statistics. The simulation parameters such simulation length, number of tasks generated and number of simulation repetitions can be specified in the Simulation Control Panel (figure 8).

### 4. Statistical results

This section presents a comparison between LST, EDF and LLF algorithms, in its multiprocessor version, showing some statistical results. Simulations have been performed using the tool described in previous section.

The simulations are oriented to prove that the algorithm design goals have been minimally achieved. This way we expect that as we increase the weight of task,
the LST algorithm is able to produce as least as good utilizations as LLF while EDF reduces this utilization because it reports as not schedulables more load cases. We also expect to show that in the general case the LST algorithm performs less context switches than LLF algorithm. As great the requested utilization is, great the difference is.

In order to show these expectatives two different simulations have been studied. The first one compares the system utilization achieved by the algorithms against the weight of the tasks. The weight of task is defined as the relation between its worst case execution time (C) and its deadline (D).

The workload used to perform this simulation is defined by the following parameters: \( C = \text{normal}(\text{Mean} = 20, \text{StdDev} = 5) \) and \( S = \text{exponential}(\text{Mean} = 9) \), where \( S \) is the arrival rate (Shoot) and deadlines are set to vary the tasks weights from a mean of 25% to 95%. This parameters produce a workload with a requested utilization around 230 percent.

Figure 9 shows that the LST is close enough to the LLF performance, while EDF becomes worse when the tasks weight is increased.

The second one compares the number of context switches performed by all three algorithms as a function of the requested system utilization. The workload used to perform this simulation is defined by the following parameters: \( C = \text{normal}(\text{Mean} = 20, \text{StdDev} = 5) \), \( D = 100 \times C/\text{uniform}(\text{Min} = 40, \text{Max} = 60) \) and \( S = \text{exponential}(\text{Mean} = 7..20) \).

Figure 10 shows that EDF performs the least number of context switches, and that LST algorithm reduces the number of context switches performed by the LLF algorithm in a significant way.

5. Conclusions

Future work will follow two main trends. The first one is to improve the algorithm to take into account the case where the release time (earliest start time) of tasks is greater than the arrival time. The second one is the application of the algorithm to an expert real time system based on a blackboard architecture [1].

References