Clairvoyant non-preemptive EDF scheduling

Conference Paper *in* Proceedings - Euromicro Conference on Real-Time Systems · August 2006

DOI: 10.1109/ECRTS.2006.7 · Source: IEEE Xplore

1 author:

Cecilia Ekelin
AB Volvo
22 PUBLICATIONS **226** CITATIONS

Clairvoyant Non-Preemptive EDF Scheduling

Cecilia Ekelin*

Department of Electronics & Software
Volvo Technology Corporation
S–405 08 Göteborg, Sweden
cecilia.ekelin@volvo.com

Abstract

It is well-known that although EDF is optimal for preemptive systems this is not the case in non-preemptive ones. The problem is that for a non-preemptive scheduler to be optimal, it must sometimes use inserted idle times. In this paper we show how the performance of non-preemptive EDF can be improved by using a form of lookahead that identifies when idle time insertion is necessary. Experiments show that this modification increases the number of schedulable task sets by up to 100%. Furthermore, by using a form of lazy evaluation the algorithm runs in $O(n log\ n)$ which is the same as plain EDF.

1 Introduction

The problem of scheduling has been studied by mathematicians and computer scientist for several decades. A fundamental property that controls the difficulty of the scheduling problem is whether or not tasks are allowed to preempt each other, i.e., their execution can be interrupted to be continued later on without any penalty. If preemption is allowed the problem is known to be solvable in polynomial time using algorithms such as earliest-deadline-first (EDF) scheduling [1]. Non-preemptive scheduling, however, is an NP-complete problem and it is believed that an optimal polynomial-time scheduling algorithm does not exist.

Within real-time scheduling, tasks are frequently assumed to be preemptive since this is a suitable model of how most operating systems execute tasks. There are situations, however, where non-preemptive execution is preferable. For instance, the overhead required in the operating system to handle the preemption may be regarded as too large. This is especially the case if the system needs to support critical sections where preemption is not allowed. In a preemptive system, this is typically handled by using the priority ceiling protocol or similar [2] while in a non-preemptive system no particular mechanism is needed. Another example can be found in commercial databases for embedded systems [3]. Here non-preemptive transactions are used under the assumption that the time associated with e.g., locking of certain tables or checking consistency, is larger that the time to actually perform the transaction. Scheduling of messages on a communication channel is also typically done non-preemptively. This is because the messages are packed into frames (that contain additional information on how to decode the message) and only whole frames can be sent or received.

Although EDF is non-optimal for non-preemptive systems, other algorithms (of the same complexity) typically cannot guarantee to schedule more task sets than EDF. The underlying problem with EDF is that it does not consider future task arrivals. In a non-preemptive system it is sometimes necessary *not* to dispatch a ready task if it will prevent future tasks to meet their deadlines. (Assuming, of course, that future task arrivals are known.) This procedure is called *inserted idle time*. The problem for any algorithm is to know when to insert idle time due to the NP-completeness of the problem. This is particularly tricky when the scheduling is to be done on-line and algorithm runtime is an issue.

Nevertheless, in this paper we show that it is possible to enhance the performance of EDF by using a lookahead technique that identifies when it is necessary to insert idle time. The algorithm runtime is kept low by using a form of lazy evaluation. The resulting algorithm is called Clairvoyant EDF (CEDF) and is guaranteed to schedule at least as many task sets than EDF and in practice often twice as many. In fact, in our simulations, the performance of CEDF is close to that of an optimal scheduler. Furthermore, CEDF runs in the same complexity as EDF: $O(n log\ n)$.

The rest of this paper is organized as follows. We start by a description of the problem and the system model in Section 2. We then discuss related work in Section 3. Our scheduling algorithm is presented in Section 4 and it is evaluated in Section 5. Finally, we discuss future work in Section 6 and state our conclusions in Section 7.

*The author thanks the anonymous reviewers for their comments which helped to improve the quality of this paper.

2 Problem Description

We assume a system of n tasks where each task τ_i is defined by a triple (r_i, C_i, d_i) where r_i is the release time, C_i the execution time and d_i the absolute deadline of the task. Thus, tasks are not periodic but it is clear that a periodic task easily can be represented as a number of non-periodic ones. That is, n will represent the number of task arrivals within the least common multiple of the task periods. It is assumed that all tasks are independent and known *a priori*. We assume a single processor and that non-preemptive execution is employed. We furthermore assume that C_i is a close estimate of the actual execution time of τ_i.

The problem is to generate a feasible schedule for a given task set. That is, all tasks should meet their deadlines and are not allowed to start before their release times. It is assumed that the schedule is to be generated on-line (or that a short scheduling time is important), thus preventing an optimal (potentially exponential time) algorithm to be used. However, since all tasks are known it is still possible to perform some initial steps off-line.

The runtime complexity of EDF is assumed to be $O(n log\ n)$ since there are n tasks to be scheduled and each task is inserted into the ready queue on arrival.

3 Related Work

It is known that if all tasks have the same release time or if the tasks are ordered such that $r_i \leq r_j \Rightarrow d_i \leq d_j$, EDF is optimal [4]. However, in the general case the problem is NP-complete and EDF fails to schedule all feasible task sets.

Within the real-time community, research on non-preemptive scheduling is focused on finding schedulability bounds or schedulability test for non-preemptive EDF [5, 6, 7, 8]. Although this research provides important knowledge on the complexity of the problem and the behavior of EDF, it does not provide an improvement of the scheduling algorithm itself.

Another line of research is trying to find good heuristic scheduling algorithms, mainly in problems which include additional constraints such as precedence and resource usage [9, 10, 11]. These algorithms often work well on average but usually cannot guarantee to outperform EDF. Moreover, they tend to have a high runtime complexity which makes them less suitable for on-line scheduling. Work has also been done using optimal algorithms such as constraint programming [12] and automata theory [13]. However, these algorithms work solely in an off-line setting and exhibit exponential run-times in the worst case.

Our approach relates to the heuristic approach in that it attempts to improve the scheduling performance. However, our algorithm is guaranteed to schedule all task sets that EDF schedules. Furthermore, it runs in the same complexity as EDF which makes it possible to use on-line.

The concept of inserted idle time is in fact not often used due to the difficulty of knowing when an insertion is beneficial. In [14] a simple rule is used to decide when to insert idle time in a certain type of periodic system adhering to network communication. This is similar to our approach but we address a much more general task model.

4 Clairvoyant EDF (CEDF)

In this section we explain the basic idea behind our scheduling algorithm, called *Clairvoyant EDF (CEDF)*, and show how to implement it efficiently.

4.1 Basic idea

The reason why EDF is non-optimal is that it always executes the first eligible task whenever the processor is free. This implies that the processor may be busy when a more urgent task arrives. Thus, sometimes it is necessary to let the processor be idle although there are tasks awaiting execution. The question is: How do you know when to postpone a task? It can be identified that the situation in which postponing a task could be beneficial appears when there are two tasks τ_i and τ_j with $r_i < r_j$ but $d_i > d_j$. Here we have to make a choice: Should τ_i be scheduled before τ_j (as it would be by EDF) or should τ_i be *postponed* (after τ_j)? The core of our algorithm is to make this decision - but only when we know the answer. To be able to answer the question correctly we need to organize the information we have (the task properties) in a suitable way.

4.2 Task properties

In ordinary EDF the scheduler only considers the deadline, d_i. In CEDF, we consider the following properties for a task τ_i:

- Deadline: d_i
- Execution time: C_i
- The earliest possible start time of the task: s_i^{min}
- The latest possible start time of the task: s_i^{max}

Initially $s_i^{min} = r_i$ and $s_i^{max} = d_i - C_i$ but, in contrast to d_i and C_i, these properties are not constant. Instead they vary depending on the properties of the other tasks and the scheduling decisions. Clearly, the interval $[s_i^{min}, s_i^{max}]$ represents the time period where τ_i must begin its execution. Note that this interval cannot be increased but only decreased as a result of the other tasks. In particular, if we know that a task τ_i must be postponed, i.e., delayed for a certain amount of time, its s_i^{min} parameter would be increased. Similarly, if this increase means that another task τ_j cannot start as late, s_j^{max} would be decreased.

Instead of a single ready queue we will actually have three different data structures to keep track of the tasks. First we will have an *index structure* where we store the information on each task. Second we have a *critical queue*

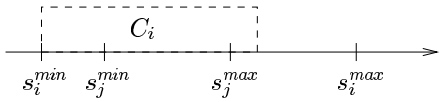

Figure 1: Illustration of postpone condition 1.

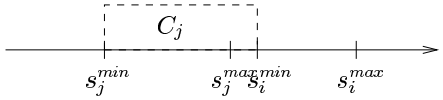

Figure 2: Updating the s_i^{min} parameter.

which contains all tasks ordered based on their s^{max} parameters. Third we have the regular *ready queue* which contains the tasks that have arrived and are ready for execution. These tasks are ordered based on their d_i parameters. To save space, the two queues will contain links to the information in the index structure.

4.3 Insertion of idle time

The postpone operation is actually equivalent to deciding whether to insert idle time or not.[1] Given that τ_i is the first task in the ready queue and τ_j the first task in the critical queue, τ_i *must* be postponed if the following conditions hold.

Postpone conditions:

1 $s_i^{min} + C_i > s_j^{max}$

2 $\tau_i \neq \tau_j$

3 $s_j^{min} \leq s_j^{max}$

This is because what condition 1 says is that even if we execute τ_i as early as possible, there will not be enough time to execute τ_j afterwards. Condition 2 tells us that we are really considering two different tasks and condition 3 says that τ_j has not already missed its deadline. Thus, conditions 2 and 3 are used to tell us when to disregard the postpone operation. That is, if the ready task is also the critical task it should not be postponed. Similarly, if τ_j already is unschedulable there is no point in delaying τ_i. Condition 1 is illustrated in Figure 1.

Note that if τ_i is not postponed when the conditions are satisfied, a deadline miss will occur. Thus, the postpone operation MUST be done to make the task set schedulable.

Postponing a task means updating its s_i^{min} parameter such that $s_i^{min} := s_j^{min} + C_j$. (See Figure 2.) That is, a timer will be set to trigger when the task is to be released again and the information in the index structure will be updated. However, the postponing of τ_i also affects τ_j - it may no longer be allowed to start as late as previously in order to have enough room to execute τ_i afterwards. Thus, $s_j^{max} := min(s_j^{max}, s_i^{max} - C_j)$. (See Figure 3.) In fact, this goes for *all* tasks τ_j in the critical queue for which condition 1 holds. However, updating every affected task in the critical queue would require n updates in the worst case.

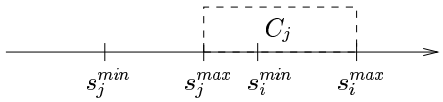

Figure 3: Updating the s_j^{max} parameter.

4.4 Lazy evaluation

To prevent unnecessary updating, we can use a form of lazy evaluation. This is done as follows. If $s_i^{min} + C_i > s_i^{max}$ we remove and reinsert τ_i in the critical queue. On insert we use $s_i^{min} + C_i$ as the key for where to insert τ_i.[2] After adding τ_i (with the s_i^{max} parameter) an update note is made that all tasks τ_j before τ_i in the queue should have their latest start time parameter updated as $s_j^{max} := min(s_j^{max}, s_i^{max})$. The update is not actually performed until the queue is traversed due to some other operation. The concept is the same as can be found in [15] and will be explained in detail in Section 4.7. Note that this update does not affect the order or the tasks in the queue. This is the reason for not using the C_j parameter in the update which otherwise would enable a further reduction of the start time interval.

It should be mentioned that tasks that arrive during the execution of another tasks will have their s^{min} parameters updated to the current time when they are inserted in the ready queue. Moreover, when a task is dispatched it is removed from the critical queue.

4.5 Complexity of CEDF

By using the postpone operation together with the lazy evaluation, the algorithm runs in $O((x+n)log\ n)$. The postpone operation will lead to that the same task may be considered several times for execution. Thus, $x+n$ scheduling decisions will be made where x are the number of postponed tasks. Each scheduling decision require the lookup of the ready task, the lookup of the critical task and possibly the re-insertion of the ready task in the critical queue. All these operations require $log\ n$ steps. An interesting property is that $x \leq n$. The reason is that the postpone versus the dispatch operations act as a sorting mechanism. If τ_i once has been postponed after τ_j this decision will not have to be repeated again. This is because if τ_j later on is postponed

[1] Of course, there may be other tasks ready that will be able to use the idle time inserted by the postpone operation.

[2] Note that during the entire update procedure we use the old value of s_i^{min}.

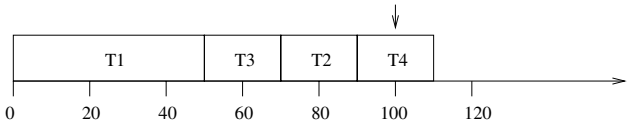

Figure 4: EDF schedule for example 1

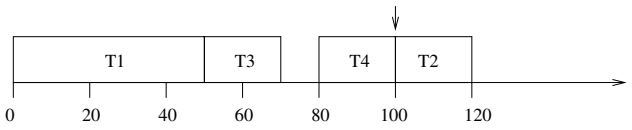

Figure 5: CEDF schedule for example 1

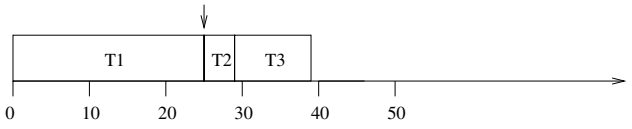

Figure 6: EDF schedule for example 2

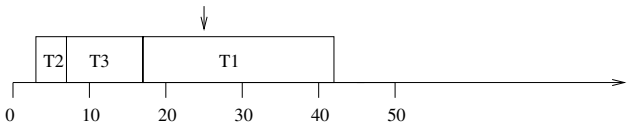

Figure 7: CEDF schedule for example 2

due to another task τ_k, the only possibility for $s_j^{min} > s_i^{min}$ again is if $s_k^{min} + C_k > s_i^{min}$. However, if this is the case the postponing of τ_j will lead to $s_j^{min} > s_j^{max}$ which means that τ_j is unschedulable. Thus, in the (potentially repeated) postpone operation, condition 3 will not be satisfied and τ_i will not be postponed due to τ_j again.

4.6 Illuminating examples

To demonstrate how CEDF works we show some examples.

Example 1 Consider the following task set: $\tau_1 = (0, 50, 148), \tau_2 = (25, 20, 145), \tau_3 = (40, 20, 125), \tau_4 = (80, 20, 100)$. The EDF schedule for this task set is shown in Figure 4. As can be seen, τ_4 misses its deadline.

In CEDF we start by ordering the tasks in the critical queue. Thus, the critical queue will contain $\tau_4, \tau_1, \tau_3, \tau_2$ (based on $s_1^{max} = 98, s_2^{max} = 125, s_3^{max} = 105, s_4^{max} = 80$). At time 0, τ_1 will arrive and since τ_4 is the critical task, CEDF will perform the postpone test: Is $0 + 50 > 80$? The answer is no which means that τ_1 will be dispatched (and τ_1 is removed from the critical queue). At time 50 CEDF will add τ_3 and τ_2 to the ready queue and make the postpone test: Is $50 + 20 > 80$? The answer is again no, selecting τ_3 to be dispatched. At time 70, τ_2 is the only ready task and the postpone test says: Is $70 + 20 > 80$? This time the answer is yes and since the critical task is not the same as the ready task and $80 \leq 80$, τ_2 is postponed. That is, $s_2^{min} := 80 + 20 = 100$. We then check whether $s_2^{max} < s_2^{min} + C_2$ $(125 < 70 + 20)$. Since this is not the case no updates of other values are necessary. The processor will now be idle until time 80 when τ_4 arrives. The postpone test says: Is $80 + 20 > 80$? Yes, but the first critical task is the same as the ready task so τ_4 is dispatched. At time 100 τ_2 arrives again and the postpone test says: Is $100 + 20 > 125$? No, so τ_2 is dispatched. All tasks meet their deadlines as shown in Figure 5.

Example 2 Consider the following task set: $\tau_1 = (0, 25, 45), \tau_2 = (3, 4, 25), \tau_3 = (6, 10, 25)$. With EDF, both τ_2 and τ_3 misses their deadlines as can be seen in Figure 6.

In CEDF the critical queue will be ordered as τ_3 $(s^{max} = 15)$, τ_1 $(s^{max} = 20)$, τ_2 $(s^{max} = 21)$. At time 0, τ_1 arrives and the postpone conditions will be tested: Is $0 + 25 > 15$? Since the answer is yes, τ_1 will be postponed to time $6 + 10 = 16$. Since $s_1^{min} + C_1 > s_1^{max}$ we will also have to update the s^{max} parameters for all tasks that satisfy condition 1 with regard to τ_1. Thus, we reinsert τ_1 in the critical queue with $s_1^{min} + C_1 = 25$ as the key.[3] This means that τ_1 will be the last task in the critical queue. We then make a note that tasks before τ_1 should have their s_j^{max} parameters updated to $min(s_1^{max}, s_j^{max})$. In this case only τ_2 will be affected and get $s_2^{max} = 20$. The processor will then be idle until time 3 when τ_2 will arrive. Since the task set now is ordered, no more postpone operations will occur. The final schedule can be seen in Figure 7.

4.7 Implementation details

A key in achieving a low runtime complexity is to use suitable data structures and to update these data structures efficiently. It is well known that a queue can be maintained in $O(log\ n)$ by implementing it as a balanced tree, e.g., an AVL tree [16]. However, this assumes that only one element in the queue is affected, i.e., added, modified or deleted. In CEDF we (in the worst case) need to update *all* elements prior to a given element. However, it is only when we actually request to see the new value, that the update must have been carried through. Hence, we can make a note that the value needs to be updated and then perform the actual update later on. This is done as follows (assuming that the critical queue is implemented as an AVL tree). When τ_i is reinserted in the critical queue, a note containing s_i^{max} is attached to all tasks directly to the left on the path to τ_i. When

[3] Note that s_i^{max} will be the parameter that is actually inserted.

PROCEDURE: **CEDF**
Input: Ready queue $\mathcal{R}$, Critical queue $\mathcal{C}$
Output: A schedule

(1) $\tau_i := first(\mathcal{R})$
(2) $remove(\tau_i, \mathcal{R})$
(3) $\tau_j := first(\mathcal{C})$
(4) IF $s_i^{min} + C_i > s_j^{max}$ AND $\tau_i \neq \tau_j$
AND $s_j^{min} \leq s_j^{max}$ THEN
(5) IF $s_i^{min} + C_i > s_i^{max}$
(6) $remove(\tau_i, \mathcal{C})$
(7) $insert(s_i^{min} + C_i, \tau_i, \mathcal{C})$
/* update notes will be posted */
(8) END
(9) $s_i^{min} := s_j^{min} + C_j$
(10) Set timer for τ_i to s_i^{min}
(11) ELSE
(12) $remove(\tau_i, \mathcal{C})$
(13) Dispatch τ_i
(14) END

Figure 8: Pseudo code for CEDF

CEDF later on will request the critical task, the tree will be traversed and any note that is on the path will be moved to the subtrees of the tasks on the path. Furthermore, the tasks τ_j on the path will have their s^{max} parameters updated as $min(s_j^{max}, s_{note}^{max})$. If $s_j^{max} \leq s_{note}^{max}$ already before the update, the note does not need to be moved any further. Moreover, if a subtree already contains a note the notes will be merged such that the new note contains only the minimum s_{note}^{max}. Any AVL rotations that will be performed to keep the tree balanced will not affect the validity of the notes. This is because in order to carry out the rotations the tree must be traversed. Thus, the notes will be updated and the rotations will only concern subtrees that are up to date. (Notes within subtrees are valid regardless of the position of the tree - this is a fundamental AVL-tree property.) Note that for this concept to work we must not use the index structure directly when requesting the s^{max} parameter of a task but the task must be located through the critical queue. However, the only time we use the s^{max} value is in the postpone conditions for the critical task so this is not a problem.

This lazy evaluation concept is the same as used in [15] where it is described in even more detail.

The pseudo code for CEDF can be seen in Figure 8.

4.8 Algorithm properties

CEDF exhibits a number of interesting properties that should be noted.

- The postpone conditions provide a necessary test for when to insert idle time in the scheduling. That is, if the conditions are satisfied, any algorithm that does not postpone the concerned task, will cause a deadline miss. In contrast, if the conditions are not satisfied, the scheduling algorithm may or may not cause a deadline miss by dispatching the concerned task but this cannot be determined by the test. The proof of this claim follows directly from the way the parameters are updated. The consequences of this are that a) CEDF will schedule all task sets that EDF schedules and b) for some task sets that EDF does not schedule CEDF will be able to correctly insert idle times in order to schedule the task set. Of course, this assumes that the execution times are accurately estimated. In particular, if they are too pessimistic a task may be postponed, leaving the processor idle, even though the task actually did not have to be postponed.

- The information contained in the s^{min} and s^{max} parameters implies that CEDF is cognizant of how its scheduling decisions affect the schedulability of future tasks. That is, information gathered from a simple pairwise comparision of tasks will be propagated to other affected tasks in a way the improves the scheduler's knowledge of the global system situation. Thus, the possibility to make a correct scheduling decision is increased compared to the case when only static parameters are used.

- CEDF is an on-line algorithm which means that any changes to the task set do not imply that a new schedule needs to be generated. It is even possible to add new tasks on the fly, e.g., when a new application that contains a number of tasks arrives to the system.

5 Simulation

It is clear that CEDF does not always succeed in scheduling feasible task sets. This is because the update of the s^{max} parameters is not omniscient and the scheduler may dispatch a task that actually should have been postponed. However, to get some idea of the performance of CEDF we compare it against (non-preemptive) EDF and the optimal scheduler from [12].

5.1 Experimental setup

The evaluation is done based on randomly generated task sets. Of course, the characteristics of the task sets greatly affects the results. For example, if the task sets are very loosely constrained, EDF will always succeed and the abilities of CEDF will not be tested. Similarly, if the task sets are too heavily constrained both EDF and CEDF will fail. Therefore, we strive to generate task sets that fall somewhere in between these two categories.

We generate 100 task sets for each experiment. In each experiment we generate n tasks. Each task gets a randomly generated execution time, release time and deadline such

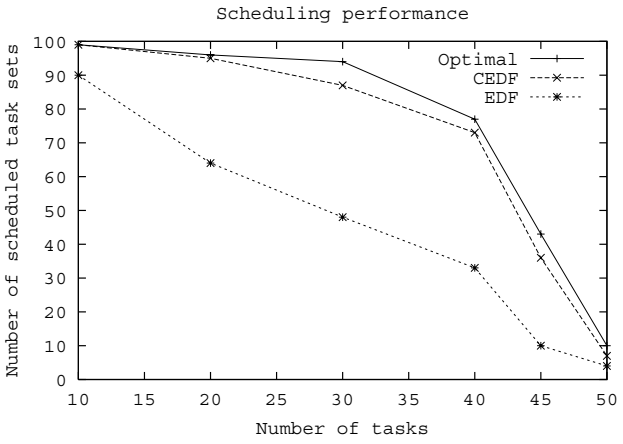

Figure 9: Performance for scheduled task sets. (Category A+B)

that $C_i \in [1, 20], r_i \in [0, 400], d_i \in [r_i, r_i + 200]$. A uniform distribution is used. This implies that the utilization of a task set is approximately $U = \frac{10n}{600}$ since the scheduling period is $[0, 400 + 200]$ and there will be n tasks with an average execution time of 10. Out of the 100 task sets, not all are necessarily feasible.

It should be noted that the overall utilization does not necessarily reflect the difficulty of the problem. The most important factor is instead the relationship between arrival times and deadlines. For example, for a task set with two tasks $\tau_1 = (0, C_1, d_1)$ and $\tau_2 = (1, 1, C_1)$ where $C_1 > 1$ and $d_1 > C_1 + 1, d_1 \to \infty$ means that $U \to 0$ but the task set is still unschedulable with EDF.

5.2 Experimental results

We have divided the results into five categories:

A Number of runs where both CEDF and EDF schedules the entire task set.

B Number of runs where CEDF schedules the entire task set and EDF does not.

C Number of runs where CEDF schedules more tasks than EDF.

D Number of runs where CEDF schedules the same amount of tasks as EDF.

E Number of runs where CEDF schedules fewer tasks than EDF.

The results from category A and B are shown in Figure 9 while the results for category C, D and E are shown in Table 1. As can be seen in Figure 9, CEDF clearly is able to schedule more task sets than EDF and in fact performs close to the optimal scheduler. As expected, the largest gap

Table 1: Performance for unscheduled task sets.

Category	n					
	10	20	30	40	45	50
C (CEDF > EDF)	1	1	6	12	22	31
D (CEDF = EDF)	0	4	7	11	30	44
E (CEDF < EDF)	0	0	0	4	9	14

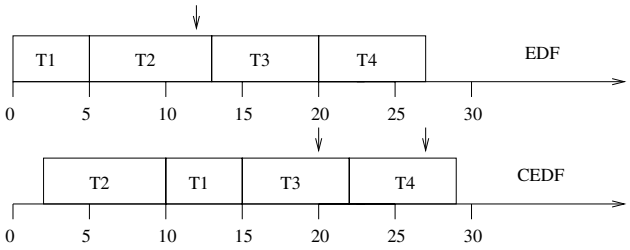

Figure 10: CEDF and EDF schedule for $\tau_1 = (0, 5, 15)$, $\tau_2 = (2, 8, 12)$, $\tau_3 = (10, 7, 20)$, $\tau_4 = (15, 7, 27)$.

appears for medium utilized task sets where the postpone operation is capable of handling the occasional "out of order" arrivals. Remember that task sets that EDF schedules are also scheduled by CEDF. This is a fundamental property since the postpone operation is only performed in those cases where EDF is certain to cause a deadline miss.

However, for overconstrained systems where the goal is to minimize the number of deadline misses, CEDF is not guaranteed to outperform EDF (although these experiments show that it probably will). The reason is that the postpone operation may prevent the current deadline-miss situation but at the cost of creating more ones in the future. This is illustrated by Table 1. Here we can see that although CEDF in most cases is able to meet more deadlines than EDF (category C and D), there are also task sets where CEDF misses more deadlines than EDF (category E). An example that illustrates why this may happen is shown in Figure 10. *Note that this situation only applies to task sets which are already unschedulable by EDF.*

We also investigated how much decision power we lose by not considering the execution times in update of the s^{max} parameters. It turns out that, for the task sets used in our simulations, the increase in the number of scheduled task sets is less than 1%.

6 Future Work

For future work it would be interesting to characterize the task sets which are expected to benefit most from CEDF and those for which CEDF is a poor choice. For example, in the update of the s^{max} parameters we could have made

the reduction sharper by using the minimum execution time of all tasks in the computation. Although this did not seem beneficial in our simulations, there may exist other task sets where this is of importance.

In this paper we assume a more general model than periodic tasks. If a strict periodic model was used, it might be possible to enhance the performance of CEDF further since more knowledge of the tasks is available. Hence, future work could involve looking at more specialized task models.

A reason why inserted idle times schedulers are not used more often, is that they are difficult to analyze. Thus, a possible future work could be to find a schedulability test for CEDF.

7 Conclusions

In this paper we have presented a non-preemptive scheduling algorithm called Clairvoyant EDF (CEDF). CEDF is an inserted-idle-times algorithm and uses a lookahead technique to determine when a task must be postponed to a later time. CEDF runs with the same complexity as EDF but is capable of scheduling 100% more task sets in many cases. Moreover, it is guaranteed to schedule all task sets that EDF schedules.

References

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

[2] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[3] McObject LLC, "eXtremeDB Database System," http://www.mcobject.com.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman Company, San Francisco, 1979.

[5] K. Jeffay, D. F. Stanat, and C. U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proc. of the IEEE Real-Time Systems Symposium*, San Antonio, Texas, Dec. 4–6, 1991, pp. 129–139.

[6] S. Baruah, "The Non-Preemptive Scheduling of Periodic Tasks upon Multiprocessors," *Journal of Real-Time Systems (to appear)*, 2005.

[7] R. R. Howell and M. K. Venkatrao, "On Non-Preemptive Scheduling of Recurring Tasks Using Inserted Idle Times," *Information and Computation*, vol. 117, no. 1, pp. 50–62, Feb. 1995.

[8] L. Georges, P. Mühletahler, and N. Rivierre, "A Few Results on Non-Preemptive Real-Time Scheduling," INRIA Research Report 3926, May 2000.

[9] K. M. Zuberi and K. G. Shin, "Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications," *Proc. of the IEEE Real-Time Technology and Applications Symposium*, Chicago, Illinois, May 15–17, 1995, pp. 240–249.

[10] P. Pop, P. Eles, and Z. Peng, "An Improved Scheduling Technique for Time-Triggered Embedded Systems," *Proc. of the Euromicro Conference*, Milan, Italy, Sept. 1999, pp. 303–310.

[11] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in a Hard Real-Time System," *IEEE Transactions on Software Engineering*, vol. 13, no. 5, pp. 564–577, May 1987.

[12] C. Ekelin and J. Jonsson, "Evaluation of Search Heuristics for Embedded System Scheduling Problems," *Proc. of the International Conference on Principles and Practice of Constraint Programming*, Paphos, Cyprus, Nov. 26–Dec. 1, 2001, pp. 640–654.

[13] P. Chen and W. Wonham, "Real-Time Supervisory Control of a Processor for Non-Preemptive Execution of Periodic Tasks," *Real-Time Systems*, vol. 23, pp. 183–208, Mar. 2002.

[14] L. Almeida and J. Fonseca, "Analysis of a Simple Model for Non-Preemptive Blocking-Free Scheduling," *Proc. of the Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 13-15, 2001, pp. 233–241.

[15] B. Andersson and C. Ekelin, "Exact Admission-Control for Integrated Aperiodic and Periodic Tasks," *Proc. of the IEEE Real-Time Applications and Embedded Technology Symposium*, San Francisco, California, Mar. 13-15, 2005, pp. 76–85.

[16] N. Wirth, *Algorithms+Data Structures=Programs*, Prentice Hall, Upper Saddle River, NJ, 1978.