Static Scheduling of Pipelined Periodic Tasks in Distributed Real-Time Systems

Gerhard Fohler Krithi Ramamritham

Department of Computer Science
University of Massachusetts
Amherst, MA 01003 *

Abstract

Many distributed real-time applications involve periodic activities with end-to-end timing constraints that are larger than the periods. That is, a new instance of a periodic activity will come into existence before the previous instance has been completed. Also, such activities typically involve communicating modules in a distributed system where some modules may be replicated for resilience. For such activities, pipelined execution allows us to meet the various resource and timing constraints imposed on them.

In this paper, we discuss an approach to dealing with the pipelined execution of a set of periodic activities that have the above characteristics. It can be called a *meta-algorithm* since it works in conjunction with another scheduling algorithm – one that creates the actual schedules. The idea is to exploit the existence of many such scheduling algorithms, which, however, typically work with activities whose deadlines are equal to or less than their periods. Our meta-algorithm invokes such a scheduling algorithm, perhaps multiple times, to generate a pipelined execution for the tasks. Effectiveness of the approach is shown via simulation studies.

Relevant Technical Area: Distributed Real-Time Systems

*This work has been supported by National Science Foundation under grant IRI-9208920.

1 Introduction

Many distributed real-time applications involve periodic activities with end-to-end timing constraints that are *larger* than the periods. That is, a new instance of a periodic activity will come into existence before the previous instance has been completed. For example, consider a computerized assembly plant where a camera observes incoming parts of many types, which arrive at a regular rate on a conveyor belt, and sends commands to one of many robot arms. Each arm is designed to pick up a specific part and so the camera, upon observing a particular part, must instruct the robot designed for that specific part. In this case, the object must be recognized, matching must be done with the models of the parts kept in memory, the robot identified, and command sent to it by the time the object reaches the robot. Clearly, many modules and resources are involved in such activities and it will be inefficient for the system to complete handling one part before considering another. That is, new parts may arrive with a period which is smaller than the deadline for handling a part. For such activities, pipelined execution allows us to meet the various resource and timing constraints imposed on them. This way, instead of waiting for the completion of handling a part before starting another, pipelining allows handling parts as they arrive. Activities such as this are designed as communicating modules where some modules may be replicated for resilience in a distributed real-time system. Thus, besides *periodicity constraints*, the activities and their components have *resource*, *precedence*, *communication*, and *replication constraints*.

In the above example, if the "parts" being handled are hazardous chemicals, then if a robot does not pick up a part in time, it can lead to a catastrophe. The activities then become *safety-critical*. We must make sure that under all circumstances a catastrophic situation will be avoided.

A lot of work has been done for scheduling periodic tasks, but only a subset of the above constraints have been considered. [7] considers the allocation and scheduling of simple periodic tasks having replication requirements on a multiprocessor. The algorithm reported in [1] aims to balance the loads across the sites while allocating replicated periodic tasks in a distributed system. Here periodic tasks are independent, simple entities, without precedence or other constraints. The only requirement is that replicates of a task be on different sites. More recently, several scheduling algorithms have been reported for more complex task models. For example, [9] and [16] discuss the scheduling on single processors of periodic tasks with arbitrary deadlines; these do not consider precedence constraints. Precedence constraints are considered in [10], and [3], but these solutions are for single processors. These, as well as [2] assume tasks whose deadlines are less than or equal to the periods.

From our discussion in the previous paragraph, it should be clear that priority-driven approaches, including those that use static priorities with feasibility checking are not yet mature enough to handle all the constraints mentioned earlier and provide such a guarantee. For this reason, resources needed to meet the deadlines of safety-critical tasks are typically preallocated and the tasks are usually statically scheduled such that their deadlines will be met even under worst-case conditions. In this paper we discuss an approach to statically scheduling the pipelined execution of a set of periodic activities that have the task characteristics described above.

Many scheduling algorithms have been proposed for statically scheduling precedence constrained tasks in distributed systems, where task deadlines are equal to or less than their periods (for example, [18, 14, 6]). Given a set of periodic tasks, such a scheduling algorithm attempts to construct a schedule of length *lcm*, the least common multiple of the task periods. The schedule specifies the exact times at which the tasks will begin execution[1]. A real-time system with the given set of tasks then repeatedly executes its tasks according to this schedule every *lcm* units of time.

Suppose we consider two tasks t_1 and t_2 with periods 3 and 4 and deadlines 5 and 7 respectively. Then the *lcm* is 12. During this time four instances of t_1 and three of t_2 will arrive. However, at time 12, the fourth instance of t_1 and the third instance of t_2 may still be in the system. So we cannot simply take the schedule generated until time 12 and use it as the static schedule. Thus, since tasks can have deadlines larger than their periods, some subtasks might execute after *lcm*. How do we then construct a feasible static schedule? That is the question answered in this paper with the help of the meta-algorithm.

The meta-algorithm invokes such a scheduling algorithm, perhaps multiple times, to generate a pipelined execution for the periodic tasks. After an invocation, the meta-algorithm determines if the schedules obtained so far are sufficient to produce the static schedule and, if not, it determines the inputs – the tasks and their requirements – for the next invocation. The length of the static schedules produced will be some multiple of the *lcm*.

We have conducted performance studies that indicate that the idea of looking for feasible schedules beyond the traditional *lcm* and furthermore looking for schedules with length greater than the *lcm* is very effective in finding feasible schedules for tasks whose deadlines exceed the periods.

The rest of the paper is organized as follows: Section 2 summarizes the basic ideas underlying our approach. Detailed description of the steps in the algorithm are presented in Section 3. An evaluation of the algorithm can be found in Section 4. The paper is

[1]As opposed to priority driven approaches, which decide which task to run at run-time.

summarized in Section 5.

2 Review of the Basic Ideas Underlying our Approach

In this section, we present the basic ideas underlying our approach to static scheduling. The goal is to produce the *shortest repeating schedule*, one which can be determined off-line and then used repeatedly on-line.

We consider distributed systems consisting of a number of sites, with a set of resources attached to each site. We assume communication media and protocols that have predictable communication delays such that knowing the arrival time and characteristics of a message, we can predict the time by when the message will be delivered [12]. The scheduler takes these worst-case delays into account in determining the start times for communicating tasks.

Before we consider pipelining scenarios, we examine the static scheduling of non pipelining scenarios. For these, the length of the shortest repeating schedule is equal to the *lcm* of the periodic tasks.

2.1 Scheduling Tasks with Deadlines not Greater than Periods

Let us have a closer look at how static schedules are constructed, that is, how the shortest repeating schedule is determined with static scheduling algorithms, e.g., [18, 14, 6]. Given tasks with deadlines, precedence constraints, resource requirements (such as specific requirements for processors, memory, etc.), and replication constraints, these algorithms determine a static schedule which specifies the exact times at which the tasks will begin execution. In contrast, priority driven approaches determine at run-time which task to execute next based on the priorities of the tasks ready at a certain time.

We will illustrate our description with a very simple example. This example is not intended to show the details of how the algorithms cited above work, but is intended to motivate the meta algorithm proposed in this paper, one that capitalizes on the existence of such algorithms.

The system under consideration consists of two processing sites and one communication medium. We assume two tasks, A and B, consisting of one subtask (A_0, B_0) each. The computation times are $c_{A_0} = 3$ and $c_{B_0} = 3$. T^k denotes the k-th instance of task T in the schedule. The periods p_i and deadlines dl_i, $i \in A, B$ are:

	p_i	dl_i	site
A	3	5	0
B	5	6	1

At time 0, all tasks are considered ready to start[2]. The set of ready subtasks in our example is A_0^0, B_0^0. The algorithm then selects ready subtasks, possibly allocating them to the sites in the system. Often, a heuristic function or estimate is used for efficient selection. As subtasks are scheduled, the algorithm keeps track of precedence relations and adds eligible subtasks, namely those with fulfilled precedence constraints, to the set of ready subtasks. At their periods, tasks become ready to start as well, and their subtasks are added to the set that is to be scheduled. In our example we may get the (partial) schedule depicted in figure 1, once time has progressed to 7:

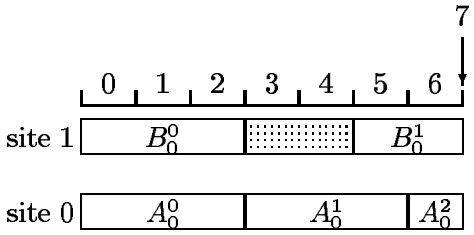

Figure 1: Schedule Construction for Non Pipelined Tasks - Time 7

Thus the search proceeds along the precedence relations until a valid schedule is found or deadlines are violated. In the latter case, limited backtracking is performed, which undoes one or more previous decisions, thereby following a different search path. When the search encounters *lcm*, all tasks have completed execution. The *lcm* in our example is 15, and the corresponding schedule in figure 2:

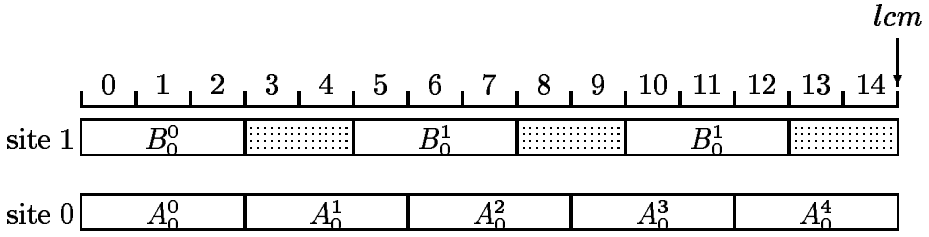

Figure 2: Schedule Construction for Non Pipelined Tasks - Time *lcm*

Five instances of A_0 and three of B_0 have completed. The created schedule is "self contained", i.e., all task executions are completely confined within the *lcm* interval. No task execution goes beyond the *lcm*. Thus the *lcm* is the shortest repeating schedule in this case.

Once tasks become pipelined, we need to look for shortest repeating schedules that are found later than *lcm* or longer than the *lcm*, as we will show next.

[2]We do not consider offsets in this example; they can be handled by the algorithm.

2.2 Scheduling Tasks with Deadlines Greater than Periods

Let us now examine what happens when we apply the above method for constructing the shortest repeating schedule to pipelined tasks. We extend our example by adding a subtask to task A: it now consists of A_0 and A_1. There is a precedence constraint and a message from A_0 to A_1. The transmission of the message takes one time unit. A_1 resides on processing site 1, $c_{A_1} = 1$.

We start as in the non pipelining case, and get the schedule depicted in figure 3 once we reach time 7:

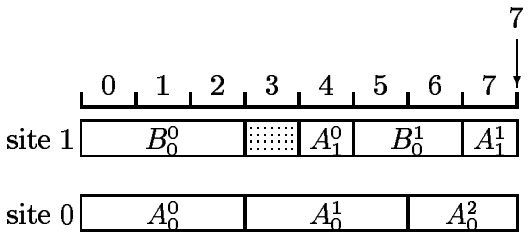

Figure 3: Schedule Construction for Pipelined Tasks - Time 7

Note that task A has not completed when its next instance is ready at 3, thus requiring pipelining. A_1, A's last subtask finished by 5 and thus kept its deadline. Search proceeds and we reach the *lcm*, figure 4:

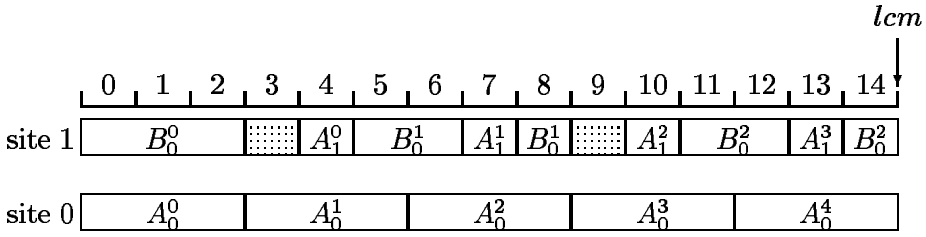

Figure 4: Schedule Construction for Pipelined Tasks - Time *lcm*

The last instances of A_0, A_0^4 and B_0, B_0^2 are completed. However, the last instance of A_1, A_1^4, still needs to be scheduled. Thus we cannot "cut out" this schedule to form the shortest repeating schedule, but need to search on. A_1^4 overlaps the *lcm* boundary. We will call such subtasks *overlapping subtasks* or *overlaps* for short.

We now try to continue constructing the schedule in the same way as before. The situation encountered at *lcm* is similar to that at time 0: both A and B are ready to start. The difference is that we have to schedule the overlap instance A_1^4 as well. So we add the overlap to the set of tasks ready at time 0, and correctly reflect the situation at *lcm*. The set of subtasks ready to be scheduled is now A_1^4, A_0^5, B_0^4. We proceed with schedule construction and get the schedule shown in figure 5 once we reach the next *lcm*, $2 \times lcm = 30$:

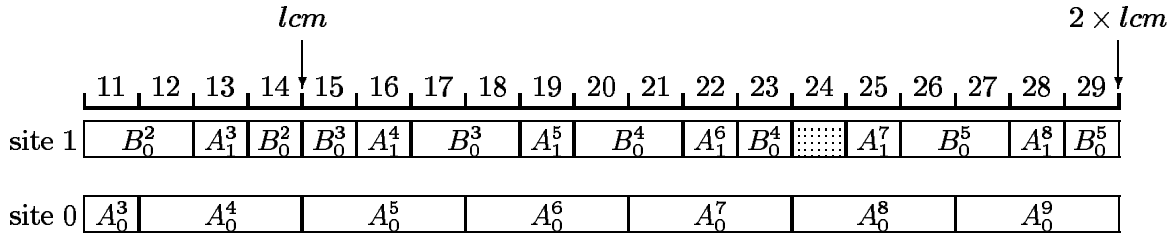

Figure 5: Schedule Construction for Pipelined Tasks - Time $2 \times lcm$

The situation at the end of the schedule is similar to the previous one, the last instance of A_1, A_1^9, has not completed. Counting the number of instances completed in the interval $[lcm, 2 \times lcm)$ gives five instances for A_0 and A_1 and three for B_0. These are the numbers required for an interval of length lcm. What happened is that the overlap instance of A_1, at $2 \times lcm$, A_1^9 was made up for by the overlap instance A_1^4 at lcm. The set of overlaps at the lcm boundary did not increase, or in other words, the set of overlaps at $2 \times lcm$ is a *subset* of that at lcm. We can see further, that all deadlines and precedence constraints are kept in the new schedule. That means, if we "cut out" $[lcm, 2 \times lcm)$ and execute this schedule repeatedly at runtime we will get a feasible schedule. We have found a shortest repeating schedule. Note that the schedule for $[0, lcm)$ is needed once at system start.

So, we execute the following steps to schedule pipelined tasks: Firstly, a normal search within the first lcm is performed. If no solution is found then, we arrange the task set to reflect the situation at lcm, by adding the overlap subtasks to the the ones we started with at time 0. Again, we search for a schedule within lcm, but with the new set of tasks. If no schedule is found and the next lcm is encountered, we check for an additional termination condition: if the set of overlaps is a subset of that at the previous lcm and the overlaps executed, we have found a feasible schedule which can be repeated at runtime.

As we will detail in section 3, these steps form a meta algorithm, which applies the static scheduling algorithm for non pipelined tasks and takes over control at lcm boundaries. As will be explained later, this algorithm is also capable of finding schedules at higher lcms; further, the schedule lengths can be multiples of lcm.

3 Detailed Description of the Algorithm

We will now give details about the determination of the tasks to consider at the beginning of each lcm, the termination condition, and about the other steps of our meta algorithm.

3.1 Determining the Tasks to be Scheduled at the Beginning of an lcm

Our goal is to determine the set of tasks to be scheduled at the beginning of each lcm so that it allows static scheduling algorithms for non pipelined tasks to be used.

The new set of tasks to be scheduled must consist of the current overlaps along with all tasks that become ready at lcm, the latter being the same as that at time 0. The timing and other parameters of the overlaps will remain the same as in the previous lcm. For subtasks that executed partially during the previous lcm, the execution time is reduced by the time they have already executed. Those of the additional tasks will reflect the fact that they start after time lcm.

If non preemptive scheduling is used, we need to take special care of subtasks that started execution before lcm and did not complete by lcm. Specifically, by adjusting its start time and deadline, such a subtask is forced to be continued immediately after lcm and at the same site.

3.2 Termination of the Algorithm

A schedule is successfully found at $(l \times lcm)$ when either (a) all the subtasks of the task set have been scheduled or (b) the set of unscheduled subtasks is a subset of the set of unscheduled subtasks at the end of some previous lcm boundary, say $(k \times lcm)$. The resulting schedule will have a length of $((l-k) \times lcm)$. Note that schedules of length $n \times lcm$, $n = 1, 2, \ldots$ are covered by this condition. If no schedule is found at $l \times lcm$, the meta algorithm rearranges the task set as detailed in the previous section and tries again.

We need to extend (b) to handle partially executed overlaps. If a subtask has only been partially executed at $k \times lcm$, its subset condition is fulfilled if the remaining execution time at this lcm is smaller than or equal to that at a previous lcm and at least a part of it has been scheduled. The latter requirement is necessary to avoid the situation where a subtask is not scheduled at all within an lcm. If it is not scheduled at all, its remaining time is the same at the end and the beginning of the lcm, thus fulfilling – incorrectly – the subset condition.

There are three reasons why it may not be possible to find a feasible schedule for a task set. Firstly, the set might be infeasible. Secondly, given our task characteristics, knowing if there is a feasible schedule is an NP-hard problem, that is, a computationally intractable. For this reason, search-based scheduling algorithms are designed to be terminated beyond a certain point. However, this means that in some cases, the scheduling algorithm may be

terminated prematurely, even if there is a feasible schedule. Thirdly, with respect to the meta-algorithm, in pathological cases the set of overlaps may not converge or may even oscillate. This is the reason for the introduction of *maxlcm*, which is used to terminate the meta algorithm at some predefined, arbitrary multiple of *lcm*. In practice we find that in a very large majority of the cases, a feasible schedule is produced with our approach.

3.3 Backtracking

Within *lcm* boundaries, backtracking is provided by the features of the basic algorithm. That is, if a search-based static scheduling algorithm, such as the one described in section 2.1 is unable to find a schedule by following a particular search path, typically, it would backtrack in the search tree and try an alternative path.

In order to enable backtracking over *lcm* boundaries, we need to be able to reconstruct the state of the task set at some previous *lcm* so that a different schedule could be attempted for that *lcm*. This can be achieved, e.g., by a stack of task sets, with one element for each *lcm* encountered.

3.4 Preparing Shortest Repeating Schedules

We cannot always simply cut out a feasible schedule found at *lcm* boundaries and use it as the static schedule. Depending on why the subset condition found a feasible schedule, we may need to adjust the number of instances and execution times. There are two possibilities to consider.

One possibility is that the sets of overlaps of the pair of *lcm* points under consideration – for application of the subset condition for termination detection – are exactly the same, including identical remaining execution times, as in the example in section 2.2. In that case, the repeated execution of the cut out schedule will produce the correct number of instances and execution times: it can be used without modifications.

The alternative situation occurs when the overlaps at the current *lcm* are a proper subset of that of a previously encountered *lcm*, the remaining execution times at the current *lcm* are smaller, or both. In that case, we need to eliminate superfluous executions of overlaps or adjust scheduled execution times.

Let O_i denote the set of overlaps at $i \times lcm$. A task A has to execute $\frac{(l-k)\times lcm}{p_A}$ times in a schedule between *lcm*s k and l. Suppose $O_l \subset O_k$ and $A \in O_k$ and $A \notin O_l$ as illustrated in figure 6. A schedule is found at *lcm* l, and in the last *lcm*, the algorithm managed to schedule A before the *lcm* boundary, which was not the case at *lcm* k. Therefore, the number

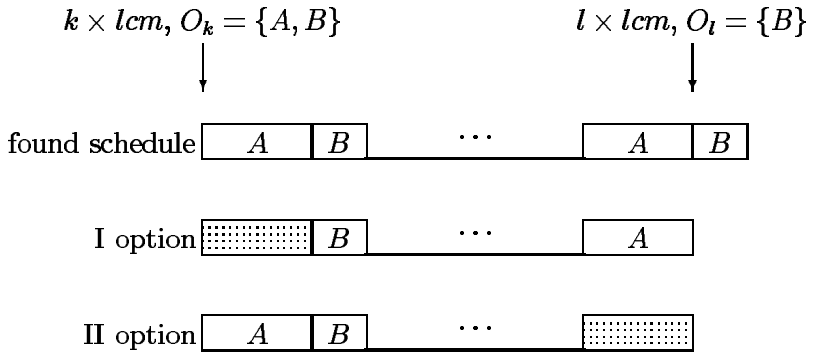

Figure 6: Adjusting the Number of Instances

of executions of A is now $\frac{(l-k)\times lcm}{p_A} + 1$, that is, one too many. Either the overlap instance of A at lcm k or the last one before lcm l can be eliminated. Note that both instances fulfill all timing requirements and the number of instances in the total schedule is correct in either case. Thus the repetition of the schedule will have feasible task executions.

Let us now assume a schedule found between lcms k and l, $k < l$, and an instance of A is an overlap at lcm l, A^{lcm_l} with remaining execution time $c_{A^{lcm_l}}$. At lcm k, it has an overlap instance A^{lcm_k} with $c_{A^{lcm_k}}$. The total execution time given to A in $[k \times lcm, l \times lcm)$ has to be $\frac{(l-k)\times lcm}{p_A} \times c_A$. If $c_{A^{lcm_l}} < c_{A^{lcm_k}}$ (subset condition), a repeated execution of the schedule will assign too much execution time to A: $c_{A^{lcm_k}} + (\frac{(l-k)\times lcm}{p_A} - 1) \times c_A + (c_A - c_{A^{lcm_l}}) = \frac{(l-k)\times lcm}{p_A} \times c_A + c_{A^{lcm_k}} - c_{A^{lcm_l}}$ Therefore, the scheduled execution time of either A^{lcm_k} or A^{lcm_l} has to be decreased by $c_{A^{lcm_k}} - c_{A^{lcm_l}}$. See figure 7 for an example.

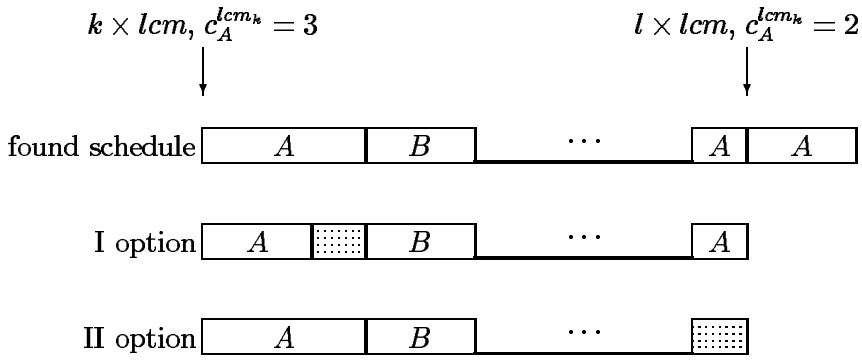

Figure 7: Adjusting the Execution Times

3.5 Pseudo-code for the Complete Algorithm

Let $overlaps[i]$, $i = 1, \ldots, k$ denote the set of overlaps at $i \times lcm$. $stack[i]$, $i \geq 0$ is a stack to store the task set at lcm boundaries.

meta algorithm

init) $current\_lcm := 0$.
$stack[0] := original\ taskset$
$taskset := original\ taskset$.

start) **do**
$current\_lcm := current\_lcm + 1$.
apply basic scheduling algorithm.
if solution found **then exit**.
until $k \times lcm$ reached.
$overlaps := \{all\ subtasks\}$.
foreach completed subtask t:
$overlaps := overlaps \setminus \{t\}$.
foreach $i = 1, \ldots, current\_lcm - 1$
if ($overlaps[i]$ *subset* $overlaps[k]$)
then feasible schedule between i and current_lcm found ($\rightarrow$ **exit**).
done.
$stack[current\_lcm] := taskset$
$taskset := taskset\ \cup\ overlaps[current\_lcm]$.
foreach subtask $T_i\ \epsilon$ overlaps[current_lcm]:
adjust timing parameters of T_i
done.
if $current\_lcm > maxlcm$ **exit**
Continue at start.

end meta algorithm

backtrack
if lcm boundary not crossed
perform normal backtracking (e.g., undo last scheduling decision
and prepare to take a different path
else
$taskset := stack[current\_lcm]$
$current\_lcm := current\_lcm - 1$
endif
end backtrack

4 Evaluation of the Algorithm

The results of the experiments presented in this section show that the idea of proceeding with the search to find schedules even at higher multiples of *lcm* and of permitting schedules with lengths that are multiples of *lcms* helps produce feasible schedules.

To test the algorithm, it was run under various parameter settings, tightness of deadlines and periods. We choose the *Success Ratio* as the performance metric. If the algorithm found feasible schedules for F task sets out of a total of T, the success ratio (SR) is said to be (F/T). Each point in the given plots was produced by (around) 300 different periodic task sets.

The system is assumed to consist of 10 sites and 5 communication channels. The task sets were generated by a graph generation package [17]. It can be used to construct acyclic, directed graphs of arbitrary connectivity. The complexity and other characteristics of the generated task (graphs) can be specified by various parameters. The computation time of each subtask is uniformly distributed between 50 and 100 time units. The communication cost attached to an arc in the precedence graph is ($comm\_ratio$ (CR) $\times$ C) where C is the average computation time of a subtask. The trends seen for different CR values are similar. Hence we show results only for experiments with a CR value of 0.4. Some of the subtasks were duplicated, others were not. The graphs had average sizes of around 200.

We used the following scheme to test the algorithm under different deadline and periodicity constraints: Two parameters, $deadline\_laxity\_factor$ (df) and $period\_laxity\_factor$ (pl) were used to set overall deadlines, D, and periods, P, respectively, as follows: $P = total\_cost \times pl$ and $D = P \times df$ where $total\_cost$ is the sum of the computation as well as communication within a task. We experimented with pl values between 0.4 and 1.2 and df values between 1.0 and 2.5.

Note that $df > 1$ means that the deadlines of the task set are larger than the periods. A pipelining algorithm can exploit the fact that tasks can execute beyond their periods.

4.1 Results

We obtained a large number of results from our simulation study, but will confine ourselves here to those concerning the detection of pipelined schedules only.

Note that each plot is comprised of 3 plots, one for each value of period_laxity.

Overall Comparison — Success Ratio: Figure 8 gives the success ratios when pipelining is exploited (using the meta algorithm) and when it is not (that is, when the search for the schedule just stops at the end of the first *lcm*). The latter corresponds to using just the *basic* algorithm, one that looks for a schedule between 0 and *lcm*. Since the meta algorithm invokes the basic algorithm, it finds non pipelining schedules as well, i.e., when the first application of the basic algorithm in the first *lcm* finds a schedule.

Recall that deadlines are larger than periods when $df > 1$.

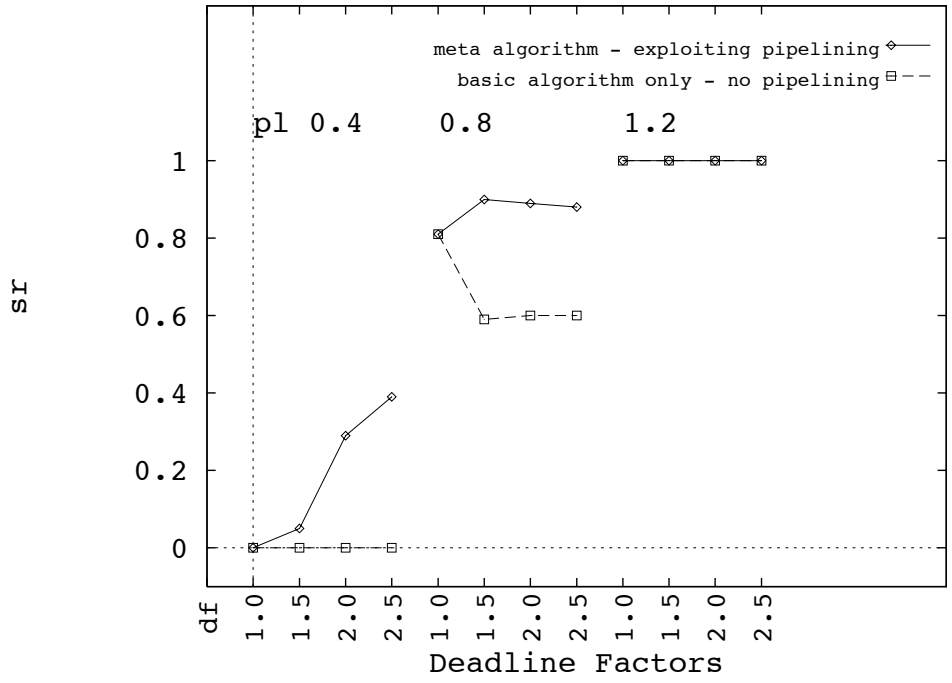

Figure 8: Comparison of Success Ratios: Meta Algorithm (exploiting pipelining) and Basic Algorithm (no pipelining)

The meta algorithm is clearly needed. For $pl = 0.4$, only the meta algorithm finds schedules and for $pl = 0.8$ it contributes about 1/3 of the schedules constructed. Just stopping after attempting to schedule between 0 and *lcm* may at times fail to produce a schedule when in fact the task set is feasible. Note, however, that a deadline larger than the period means that pipelining is allowed, but not always necessary. This is because even if the deadline of a task is larger than its period, it is still possible that it will finish execution before the start of the next instance, because the task's laxity may be high. This is shown by $pl = 1.2$: the laxities of the tasks are high compared to the computation and communication requirements of the tasks, so the basic algorithm can find all the schedules.

Schedule Boundaries: We investigated up to which multiple of the *lcm* the algorithm had to search to find a schedule, i.e., at what outmost boundary the schedules were found. Figure 9 plots the success ratio for different schedule boundaries.

Pipelining requires higher multiples of *lcm* to be searched, whereas non pipelining situations were found up to multiple 1, i.e., within the first *lcm*. The "most outmost" schedule was found between $7 \times lcm$ and $11 \times lcm$.

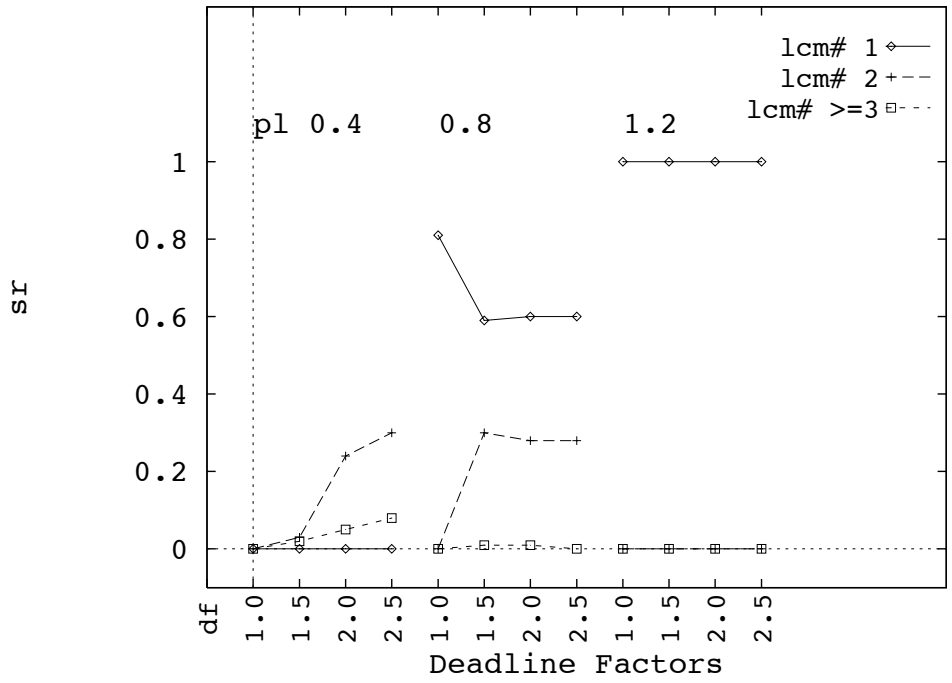

Figure 9: Success Ratio w.r.t Schedule Boundaries

Lengths of the Schedules: We examined the lengths of the schedules found. Figure 10 plots the success ratio of various schedule lengths. Pipelining requires longer schedule lengths. All non pipelining schedules, of course, have a length of one *lcm*. The longest schedule length found was $4 \times lcm$.

Before we conclude this section we note that for the cases tested here, we found that very little improvement resulted from allowing backtracks even though a very large increase in overheads occurs. This conforms to the observation made in [14] for non-pipelined tasks.

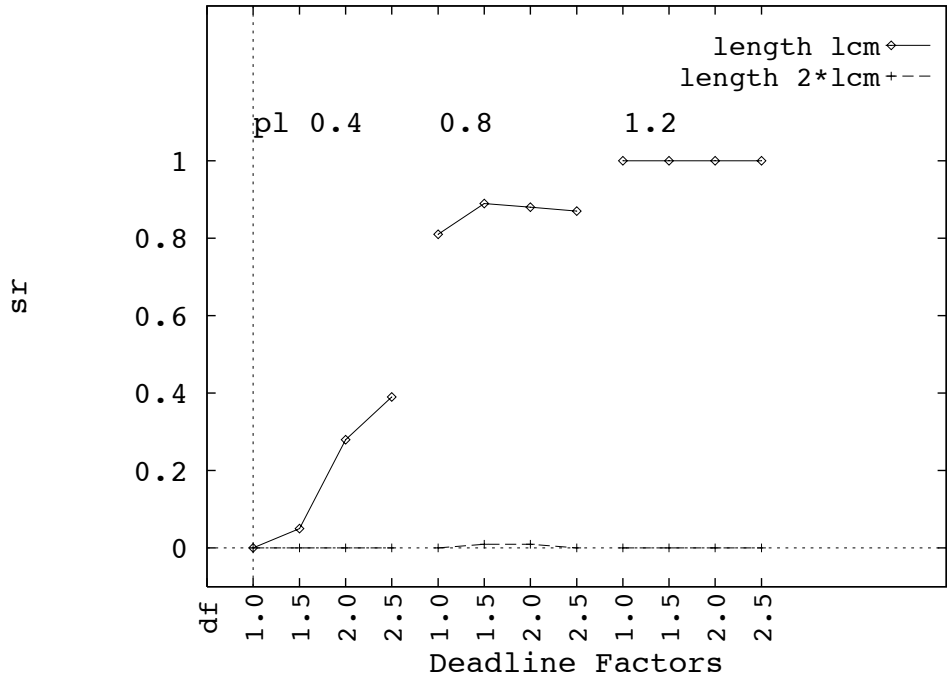

Figure 10: Success Ratio w.r.t. Schedule Lengths

5 Summary

In this paper we have been concerned with the problem of guaranteeing the execution of real-time tasks that require pipelined execution. Since previously proposed algorithms already handled tasks whose deadlines are less than or equal to their periods, we wanted to exploit them. Toward this end, we developed an approach that makes use of one such scheduling algorithm for scheduling within *lcm* and "takes over control" only at *lcm* boundaries.

The basic idea behind our approach is the following: Static schedules need not be restricted to be of length *lcm*; neither do they have to start with tasks that arrive at time 0 and end at *lcm*. Any schedule of length $n \times lcm$ $(n = 1, 2, 3...)$, and hence, an interval $[i \times lcm,\ (i+n) \times lcm)$ $(i = 0, 1, 2, 3...)$ will do. So the algorithm may start search at time 0, continue to *lcm*, and if no feasible schedule is found, search is continued with the not yet scheduled subtasks and all subtasks of the original task set. Each time a multiple of *lcm* is reached, this procedure is repeated. Once a feasible schedule (as precisely defined by the termination conditions) has been found, the portion of length $n \times lcm$, corresponding to the *shortest repeating interval*, is "cut out" and used at run-time as the static schedule.

As our experiments show, this simple approach is very successful in finding the schedules.

Since the meta algorithm applies a static scheduling algorithm to construct schedules, it can be used with scheduling under mode changes [5] and aperiodic task scheduling [4].

References

[1] J. A. Bannister and K. S. Trivedi. " Task allocation in fault-tolerant distributed systems". In *Acta Informatica*, 20, Springer-Verlag, 1983.

[2] J. Sun, R. Bettati, and J. W.-S. Liu. "An End-to-End Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems". In *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software.*

[3] H. Chetto, M. Silly, and T. Bouchentouf. "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints". *Real-Time Systems*, 2(3):181–194, Sept. 1990.

[4] G. Fohler. " Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Task s in Statically Scheduled Systems". In *Proc. 16th Real-Time Systems Symposium, Pisa, Italy, Dec 1995.*

[5] G. Fohler. Realizing Changes of Operational Modes with Pre Run-Time Scheduled Hard Real-Time Systems. In *Proc. of the Second International Workshop on Responsive Computer Systems*, Saitama, Japan, October 1992.

[6] G. Fohler. "Analyzing a pre run-time scheduling algorithm and precedence graphs". *Research Report* 13/92, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, September 1992.

[7] C.M. Krishna and K.G. Shin , "On Scheduling Tasks with a Quick Recovery from Failure", *IEEE Transactions on Computers*, Mjay 1986, pp 448-155.

[8] J.P. Lehoczky, Sha, L. and Strosnider, J. "Enhancing Aperiodic Responsiveness in a Hard Real-time Environment", *IEEE Real-Time Systems Symp.* 1987.

[9] J.P. Lehoczky. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines", *IEEE Real-Time Systems Symp.* 1990, pp. 201-212.

[10] M.G. Gonzalez Harbour, and J.P. Lehoczky. "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority", *IEEE Real-Time Systems Symp.* 1991, pp. 116-128.

[11] G. Le Lann, "The 802.3D Protocol: A Variation on the IEEE 802.3 Standard for Real-Time LANs". *Technical Report*, INRIA, July 1987.

[12] N. Malcolm and W. Zhao. "Hard Real-Time Communication in Multiple-Access Networks", *Real-Time Systems*, Vol 8, No 1, January 1995, pp. 35-78.

[13] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks With Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, August 1989, pp. 1110-1123.

[14] Ramamritham, K. "Allocation and Scheduling of Precedence-Related Periodic Tasks" *IEEE Transactions on Parallel and Distributed Systems*, Vol 6, No 4, April 1995, pp. 412-420.

[15] K.W. Tindell and A. Burns and A.J. Wellings. "Allocating Real-Time Tasks (An NP-Hard Problem Mad Easy)", *Real-Time Systems*, Vol 4, No 2, June 1992, pp.145-166.

[16] K.W. Tindell and A. Burns and A.J. Wellings. "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks", *Real-Time Systems*, Vol 6, No 2, March 1994, pp.133-152.

[17] A. Welzl. GRAPHGEN – Generation of Precedence Graphs". *MARS Praktikum*, Vienna, Austria, Dec. 1989.

[18] J. Xu and D. L. Parnas. "On Satisfying Timing Constraints in Hard Real-Time Systems". *IEEE Transactions on Software Engineering*, 19(1):70–84, Jan. 1993.