Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems

Gerhard Fohler*
Computer Science Department
University of Massachusetts at Amherst, MA 01003-1355
fohler@cs.umass.edu

September 13, 1995

Abstract

In this paper we present algorithms for the joint scheduling of periodic and aperiodic tasks in statically scheduled distributed real-time systems. Periodic tasks are precedence constrained, distributed, and communicating over the nodes of the systems. Both soft and hard aperiodic tasks are handled.

After a static schedule has been created in a first step, the algorithms determine the amount and distribution of unused resources and leeway in it. These are then used to incorporate aperiodic tasks into the schedule by shifting the periodic tasks' execution, without violating their feasibility. Run-time mechanisms are simple and require only little memory. Processors and communication nodes can be utilized fully. The algorithm performs an optimal online guarantee algorithm for hard aperiodic tasks of O(N).

An extensive simulation study exhibits very high guarantee ratios for various load and deadline scenarios, which underlines the efficiency of our method.

1 Introduction

Processes controlled by real-time computer systems typically experience phases of continuity as well as of changes. Variations in the environment can be reflected in the real-time system as modifications to the entire task set, called mode changes then. Smaller changes may result in the occurrence of tasks with arrival times unknown at design time, named aperiodic tasks. Research in real-time scheduling has been focused on either continuous or changing phases. Algorithms for continuous phases were dealing with guaranteeing before the run-time of the system that fixed sets of periodic tasks with fixed properties would meet their timing requirements. Such algorithms have been presented for both dynamic e.g., [15], [21] and static scheduling, e.g., [18]. The focus of other algorithms was on-line scheduling of aperiodic tasks, without consideration of off-line guarantees, e.g., [23].

As attention was directed towards scheduling of both continuous and changing phases, algorithms to handle mode changes have been presented for dynamic, [20], [26], and static scheduling, [8]. Uniprocessor schedulability tests have been presented for sporadic tasks, i.e., tasks with a bound on their occurrence frequency, [2] and mixed task sets of non preemptive sporadic and periodic tasks, [11]. Joint scheduling of periodic and aperiodic tasks with unknown and unlimited arrival times has been addressed in the context of dynamic scheduling for single processors. Schedulability tests of $O(N^2)$ were presented in [10] The issue of using resources unused by the periodic tasks on single processors has been studied in the context of earliest deadline first scheduling, [4]. [3] gives online acceptance tests of $O(N)$. Efficient server algorithms for earliest deadline first have been presented in [22].

Server algorithms for fixed priority scheduling have been presented in [14]. Later, the slack stealing algorithm was introduced, [13] and extended to deal with hard aperiodic tasks, [24, 25] and more general scheduling problems, [5, 1]. As priorities are set according to the rate monotonic algorithm, [15], the worst case processor utilization achievable is about 69%, although higher in the average case. These dy-

*This work was funded in part by the National Science Foundation under grant IRI-9208920.

[0]The work presented in this paper was carried out while the author was with Institut für Technische Informatik, Technische Universität Wien, Austria and during visits at the University of Massachusetts.

* This paper has been accepted for the 16th Real-Time Systems Symposium, Pisa, Italy, Dec 1995

namic algorithms consider tasks which are restricted to be independent or synchronize via simple constructs on single processors.

Handling soft aperiodic tasks in statically scheduled systems was addressed in [19]. Static scheduling allows periodic tasks to have arbitrary precedence constraints and to be distributed and communicating over several processing nodes. Without further provision, however, it requires the transformation of aperiodic tasks into pseudo periodic ones, [16], which can be prohibitively expensive, or they can only be run, when no periodic tasks are ready, which does not allow hard deadlines for aperiodics.

In this paper, we present mechanisms for the joint online scheduling of periodic and aperiodic tasks in statically scheduled systems. This allows us to combine benefits of static scheduling, in particular a distributed system and complex, precedence constrained tasks, and dynamic online scheduling, which allows flexible task execution to incorporate aperiodic tasks with both soft and hard deadlines. Processors and communication medium can be utilized fully. The online mechanisms are very simple and require only little memory. They can be used to select nodes for the execution of aperiodic tasks and the handling of complex, distributed aperiodic tasks. As our algorithms integrate fully with the mode change algorithm [8], they allow real-time systems to be statically scheduled and highly flexible towards major and minor changes in the environment, as detailed in [9].

The first step of our method is to construct a static schedule for the periodic tasks, which meets all timing, synchronization, and communication requirements. Next, the amount and distribution of unused resources and leeway, called spare capacities, is determined. So we know for each point in time, by how much the execution of the next periodic task can be shifted without missing any deadlines. This is equivalent to the amount of aperiodic execution that can be incorporated into the static schedule. It is straightforward to use this information at run-time to perform a very simple test whether it is feasible to execute a soft aperiodic task. As the amount and distribution of unused resources are known in advance as well, we provide an optimal on-line acceptance test for hard aperiodic tasks. Spare capacities are calculated off-line and incrementally maintained on-line, so the run-time mechanisms are very simple and require little memory.

The rest of the paper is organized as follows: Section 2 lists the task and system model as well as terminology used throughout the paper. We present our methods for single nodes first in sections 3 and 4. The off-line preparations are described in section 3, followed by the online mechanisms in section 4. It explains the guarantee algorithm, online scheduling, and maintenance of spare capacities. Section 5 deals with extending our method to distributed systems. The analysis in section 6 gives results from a simulation study. A summary concludes the paper in section 7.

Due to space limitations, we do not give proofs here. The interested reader is referred to [9].

2 System Description and Task Model

We consider a *distributed* system, i.e., one that consists of several *processing nodes* and *communication nodes*. Our time model is *discrete* [12]: An external observer counts the ticks of the globally synchronized clock with granularity of *slot-length* and assigns numbers from 0 to ∞ to them. We denote by "in slot i" the time between the start and end of slot i, $slot_i$, i.e., the time-interval $[slotlength \times i,\ slotlength \times (i+1))$. Slots have uniform length and are synchronized globally, i.e., they start and end at the same time for all nodes in the system.

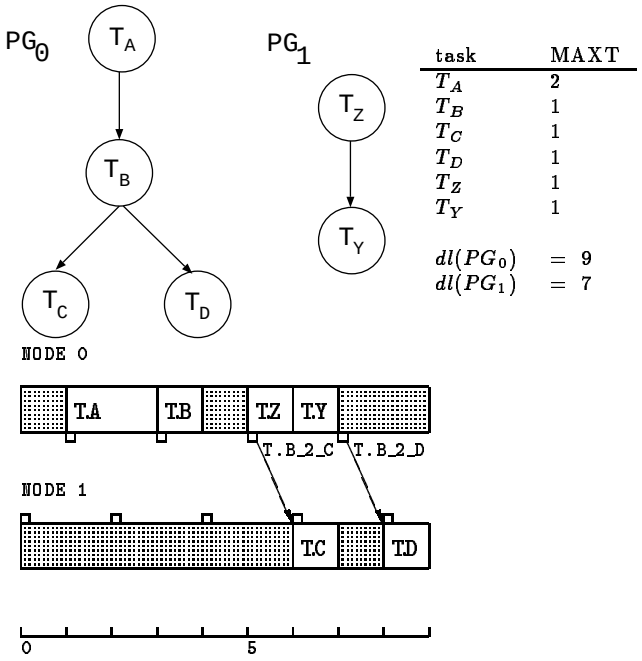
A *schedule* is a sequence of slots $slot_i$ $i = 0, \ldots,$ N-1. N stands for the number of slots in the schedule. For static schedules, N is typically equal to the least common multiple (lcm) of all involved periods. At run-time, a static schedule is executed repeatedly, i.e., at slot t of the run-time of the system, slot t mod N is executed[1]. Communication nodes, i.e., the communication medium, are slotted and pre scheduled as well. Protocols with bounded transmission times, e.g., TDMA or token ring are applicable. At run-time, a *dispatcher* is invoked at regular time intervals of uniform length (slot-length) and assigns a task to the processor each time or leaves it idle.

Tasks are fully preemptive and communicate with the rest of the system via data read at the beginning and written at the end. The *maximum execution time (MAXT)* of a task is a calculated upper bound on the execution time of a task [17], given in slots. This number of slots is used by the scheduler, regardless of a - possibly smaller - actual execution time. Task synchronization is expressed exclusively by *precedence constraints*. Precedence constraints and tasks, can be viewed as a directed acyclic graph. Tasks are represented as nodes, precedence constraints as edges. Tasks that have no predecessors are called *entry* tasks, tasks without successors *exit* tasks. The *period* of a

[1] Our algorithms deals with shifting these static execution slots, as detailed in the following sections.

precedence graph PG is the time interval separating two successive instances of PG in the schedule. *Deadline intervals* are defined for the maximum execution of each individual precedence graph. Entry tasks of instance i of a precedence graph PG become ready for scheduling at time $i \times period(PG)$. Exit tasks of instance i of a precedence graph PG have to be completed by time $i \times period(PG) + deadline(PG)$. All other tasks "inside" a precedence graph are constrained only by the precedence order.

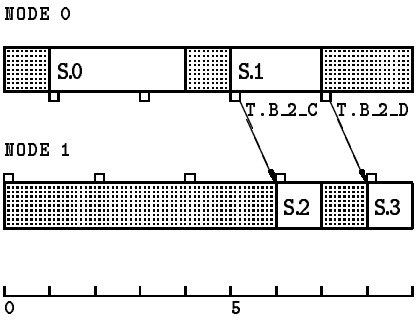
Tasks or precedence graphs may be attributed *static*, indicating their periodic execution and treatment by the static scheduler. Tasks not treated by the static scheduler can be of the following types (along the classification of [16]): *Sporadic*, i.e., they have known minimum inter-arrival time and execution time, and a given deadline. They are called *aperiodic*, if no minimum inter-arrival time is given. If a deadline is assigned, the task is called *hard* aperiodic. The dispatcher checks at each invocation for aperiodic tasks that have arrived during the last slot. We assume that aperiodic tasks are ready to run at the time of their of arrival.



task	MAXT
T_A	2
T_B	1
T_C	1
T_D	1
T_Z	1
T_Y	1

$dl(PG_0) = 9$
$dl(PG_1) = 7$

Figure 1: Example Precedence Graphs and Schedule



scheduling block		est	MAXT	dl
S_0	$(T_A T_B)$	0	3	5
S_1	$(T_Y T_Z)$	0	2	7
S_2	(T_C)	6	1	8
S_3	(T_D)	8	1	9

Figure 2: Schedule for Example with Scheduling Blocks

3 Off-Line Preparations

In a first step, we use a static scheduling algorithm to create scheduling tables for the periodic tasks. It allocates the tasks to nodes and resolves the precedence constraints by ordering the task executions accordingly. Figure 1 gives an example. These execution sequences consist of subgraphs of the precedence graphs, that have been separated by the allocation of tasks to nodes. Each of these subgraphs is limited either by the sending and receiving of inter-node messages or entry and exit tasks which define the start-times and deadlines. We combine the tasks of these subgraphs to *scheduling blocks*, as depicted in figure 2 for the example of figure 1. These are independent, as all dependencies have been resolved during off-line scheduling. This grouping reduces the number of items, while scheduling blocks can be treated in the same way as tasks. Due to space limitations, we cannot discuss this issue in detail here, the interested reader is referred to [9]. We will not distinguish between tasks and scheduling blocks in the following.

3.1 Start-times and Deadlines

The scheduling tables list fixed start- and end times of task executions, that are less flexible than possible. The only assignments fixed by specification, however, are starts of entry and completion of exit tasks, and, as we assume message transmission times to be fixed for now[2], tasks sending or receiving inter-node messages. The points in time of execution of all other tasks may vary within the precedence order. Based on that knowledge, earliest start-times and latest finish-times for all tasks can be calculated per node. As we want to determine the maximum leeways of task ex-

[2] We will discuss shifting of messages in section 5.

ecutions, we calculate the deadlines to be as late as possible. Let $end(PG)$ denote the end and $start(PG)$ the start of precedence graph PG according to the schedule[3]. The start of an inter-node message transmission M is denoted $start(M)$, the time it is available at all receiving nodes $end(M)$. These are the only fixed start-times and deadlines, all others are calculated recursively with respect to logical successors, i.e., in the precedence graph. The *deadline* of task T, $dl(T)$, of precedence graph PG in a schedule is: If T is exit task: $dl(T) = dl(PG)$, if T sends an inter-node message M: $dl(T) = start(M)$. The *earliest start-time* of task T, $est(T)$, of precedence graph PG in a schedule is calculated in a similar way: If T is entry task: $est(T) = start(PG)$, if T receives an inter-node message M: $est(T) = end(M)$.

In our approach, earliest start-times serve only to prevent early execution of entry tasks or receivers of inter-node messages. The precedence order is already kept in the execution sequence constructed by the static scheduler.

3.2 Execution Intervals

The deadlines of tasks are then sorted for each node and the schedule is divided into a set of *disjoint execution intervals* for each node. Spare capacities are defined for these intervals.

Each deadline calculated for a task defines the end of an interval I_i, $end(I_i)$. Several tasks with the same deadline constitute one interval. The earliest start-time of an interval I_i, $est(I_i)$, is the minimum of the earliest start-times of its constituting tasks. The start of an interval I_i, $start(I_i)$, is the maximum of its earliest start-time or the end of the previous interval. An interval I_i is called *independent*, if there is no interval $I_e, e < i$ with $end(I_e) > est(I_i)$ and no interval $I_l, i < l$ with $end(I_i) > est(I_l)$. The length of an interval I_i is denoted $|I_i|$. Figure 3 illustrates the intervals for the example of figure 2.

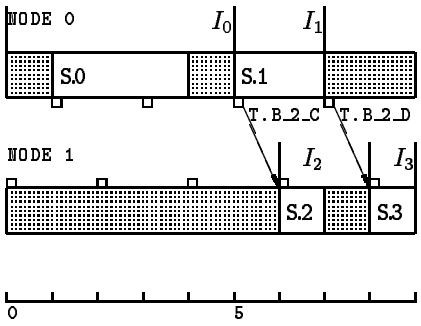
3.3 Spare Capacities

Calculating spare capacity of an independent interval is straightforward. The executions of such an interval can be shifted arbitrarily within the interval boundaries. So the *spare capacity (sc)* of such an interval I_i is given by

$$sc(I_i) = |I_i| - \sum_{T \in I_i} MAXT(T) \qquad (1)$$

[3]For simplicity of explanation, we consider only a single instance of the precedence graph. Consequently, we do not give instance numbers.

We illustrate this by the example depicted in figures 1. The resulting schedule along with earliest start-times, deadlines and spare capacities is shown in figure 3. The spare capacity in interval I_0 can accommodate



int	start	end	$\|I\|$	MAXT	sc	node
I_0	0	5	5	3	2	0
I_1	5	7	2	2	0	0
I_2	6	8	2	1	1	1
I_3	8	9	1	1	0	1

Figure 3: Example of Simple Spare Capacities with Independent Intervals

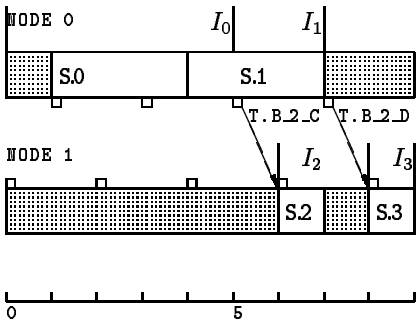
dynamic activities requiring two slots, that of I_1 none.

Let us have a closer look at the idle slot 4[4]. I_1 is tight, i.e., neither can any further activity be added at run-time, nor could a specified increase in execution time of its tasks be accommodated. The schedule as a whole, however, is not tight, i.e., more or longer tasks of PG_1 or an aperiodic task can be fitted in. Let us assume the modification to our example that S_1 has a MAXT of three slots instead of two. We then construct the schedule and calculate, the intervals, and the spare capacities as described. The resulting schedule is given in figure 4: Deadlines and intervals stay the same, so I_1 is too short by one slot. The additional execution slot of S_1 was fitted into the idle slot 4 of I_0. In other words, I_1 "borrowed" one slot from I_0 to fit in S_1. I_0 can give this one slot from its two idle ones, having only one left then. This usage of other intervals' capacities is indicated by the spare capacity value of -1. Apparently, the leeway of I_0, i.e., the spare capacity, will be smaller by the amount "lent". Taking into account the "borrowing" mechanism, we can now extend equation 1:

$$sc(I_i) = |I_i| - \sum_{T \in I_i} MAXT(T) + min(sc(I_{i+1}), 0) \qquad (2)$$

The spare capacity of an interval is decreased by the

[4]This example was constructed for illustration purposes. The scheduler would not create the idle slots at 0 and 4, without S_0 and S_1 waiting for inter-component messages.



int	start	end	$\|I\|$	MAXT	sc	node
I_0	0	5	5	3	2	0
I_1	5	7	2	3	-1	0
I_2	6	8	2	1	1	1
I_3	8	9	1	1	0	1

Figure 4: Example of Simple Spare Capacities with Dependent Intervals

amount "lent" to subsequent intervals. Aperiodic tasks may be started, when the spare capacity is non negative.

4 On-Line Mechanisms

During system operation, the on-line scheduler is invoked after each slot. First, it checks whether aperiodic tasks have arrived during the last slot. Soft aperiodic tasks can be executed when the current spare capacity is greater than zero. If hard aperiodic tasks have arrived, the guarantee algorithm is invoked. Next, one task out of the ready aperiodic or periodic tasks is selected for execution or the processor left idle. This decision is then used to update the spare capacities. Finally the scheduling decision is executed in the next slot.

4.1 Guarantee Algorithm

At run-time of the system we have a set of tasks T_i, assigned to intervals I_i with deadlines $dl(I_i)$ and spare capacities $sc(I_i)$. The current time is denoted t. We denote the *current spare capacity* of the current interval I_c at t with $sc(I_c)_t$. $sc(I_c)_t \leq sc(I_c)$. Now assume a hard aperiodic task T_A arrives with known parameters $est(T_A) = t$, $MAXT(T_A)$, and $dl(T_A)$. If several such tasks arrive, they are sorted in earliest deadline order. Next the guarantee routine is applied to the first task, if successful to the second, and so on. This may prevent the finding of an optimal solution, but reduces the run-time efforts spent for the guarantee[5]. The trivial condition $dl(T_A) \geq MAXT(T_A)$ is checked before. The guarantee algorithm shall now decide, whether T_A can be accommodated to meet its deadline without causing any other guaranteed deadline to be missed.

We identify three parts of the total spare capacities available:
$sc(I_c)_t$, the remaining spare capacity of the current interval,
$\sum sc(I_i),\ c < i \leq l,\ end(I_l) \leq dl(T_A) \wedge end(I_{l+1}) > dl(T_A)$, the spare capacities of all *full* intervals between t and $dl(T_A)$, and
$min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, the spare capacity of the last interval, or the execution need of T_A before its deadline in this interval, whichever is smaller

If the sum of all three is larger than $MAXT(T_A)$, T_A can be accommodated, and therefore guaranteed. To prevent other hard aperiodic tasks from using spare capacities needed by T_A, we recalculate the spare capacities and intervals from right to left taking into account that the resources for T_A are not available for other tasks. This applies the latest deadline last strategy, which has been shown to provide *"maximum quantity of processor idle time available for processing sporadic tasks"* by Chetto and Chetto [3].

Unless $dl(T_A) = dl(I_l)$, the interval where $dl(T_A)$ lies in, I_{l+1}, needs to be split. A simple decreasing of $sc(I_{l+1})$ might result in the following: Assume that T_A has been guaranteed as described, and that $sc(I_{l+1})_{new} = sc(I_{l+1})_{old} - (dl(T_A) - start(I_{l+1}))$. Then another hard aperiodic task T'_A arrives with $dl(T_A) = dl(T'_A)$ and $MAXT(T'_A) < sc(I_{l+1})_{new}$. The algorithm would guarantee T'_A, because T'_A requests less execution time than the spare capacity in that interval. It needs these spare capacities, however, before its deadline, but these are already needed for T_A. This will result in either T_A or T'_A missing its deadline.

Therefore I_{l+1} is split into two intervals, after T_A has been guaranteed: $I_{l+1,left} : [start(I_{l+1}), dl(T_A))$ and $I_{l+1,right} : [dl(T_A), end(I_{l+1}))$. Then the spare capacities of all intervals $I_i,\ c \leq i \leq I_{l+1,right}$ are recalculated.

In order to perform the acceptance test, we have to go through the intervals up to the aperiodic's deadline. If it is accepted we have to possibly create a new one and then update these intervals. Thus, the complexity of our guarantee algorithm is $O(N)$, N being the number of intervals. N can be large when periods are not harmonically related. Unless the aperiodic's

[5] We might even think of limiting the number of task arrivals at a time.

deadline is significantly larger than that of the other tasks, the number of intervals we have to go through, i.e., up to the aperiodic's deadline, will be unaffected by the maximum N. It is shown in [9], that this acceptance test has equivalent results – but with simpler run-time handling – as to the ones presented in [10] and [3], which are optimal for single processors. Due to space limitations we do not give proofs here. The interested reader is referred to [9].

4.2 On-Line Scheduling

On-line scheduling is performed locally for each node. We have the set of tasks T_i, consisting of static, guaranteed, and soft aperiodic tasks, with known $est(T_i)$, $MAXT(T_i)$, and $dl(T_i)$, and the set of intervals I_i. The scheduler is given the ready set, $R(t) = \{T | est(T) \leq t\}$ and the set of intervals and spare capacities. It may choose one task out of the ready set or leave the CPU idle. We identify the following cases:
$R(t) = \{\}$*:* There are no tasks ready to be scheduled, the CPU remains idle.
$R(t) \neq \{\} \wedge sc(I)_t > 0 \wedge \exists T$, T *soft aperiodic:* T is executed.
$R(t) \neq \{\} \wedge sc(I) = 0$*:* A static or guaranteed task *has to* be executed. Zero spare capacity indicates, that adding further activities will result in a deadline violation of the task set.
$R(t) \neq \{\} \wedge sc(I)_t > 0 \wedge \nexists T$, T *soft aperiodic:* A static or guaranteed task is executed. The task with the earliest deadline in $R(t)$ is selected. Ties are broken arbitrarily, e.g., by index number.

4.3 Maintenance of Spare Capacities

The decision of the scheduler is now used to update the spare capacities depending on which type of task was selected for execution:
Aperiodic execution: One slot of the spare capacities is used to execute a slot of a dynamic task. Obviously, the spare capacity of the current interval has to be decremented by one.
No execution: One slot of spare capacity is used without dynamic processing. The spare capacity has to be decremented by one, too.
Static execution: Executing a static task T only swaps spare capacities. Depending on the interval to which the executed task belongs to, the current interval I_i or a subsequent one I_j, $j > i$ is affected.
$T \in I_i$*:* The amount of spare capacity is not changed for any interval.
$T \in I_j$*:* One slot of spare capacities is swapped between I_i and I_j. $sc(I_i)$ is decremented by one, $sc(I_j)$ is incremented by one. This is, however, not generally complete. There might be one or more intervals I_k, $i < k < j$ in between I_i and I_j. This will happen if the start-time constraint for I_i and the intervals I_k has not been fulfilled yet, while that of I_j has. If I_j has negative spare capacity, it borrowed spare capacities from I_{j-1}. If one slot of execution of I_j has been executed in I_i, this slot need not be executed in I_j, which will need to "borrow" one slot less from I_{j-1}. Then, the same applies to I_{j-1} also: If I_{j-1} had spare capacity, it now has to borrow one slot less from I_{j-2}. So, the spare capacities of the consecutive, with decreasing indices, "lending" intervals I_l, $i \leq l < j$, $sc(I_l) < 0$ are increased.

We will now illustrate the run-time maintenance of spare capacities with the example of figure 4 and show how negative spare capacities are forced to become non-negative by our mechanisms. Let us assume that no dynamic activities are requesting execution at the beginning of the schedule, i.e., before slot 0. Then, S_0 can execute in slots 0 - 2, and S_1 in slots 3 - 5. Two slots of S_1 execute in I_0: One has been given by the borrowing mechanism, the other is the one unused by aperiodics. So S_1 needs two slots less of its interval I_1. They are added to the spare capacity: $sc(I_1) = -1 + 2 = 1$. One slot is available for aperiodic tasks in I_1. Both "surplus" slots in I_0, one more than pre calculated, have been used by S_1, so $sc(I_0) = 1 - 1 = 0$, which reflects the situation correctly.

Let us assume one slot of I_0 was used for dynamic activities. Then no idle slots are available for further dynamic execution and so S_0 has to execute in slots 1 - 3, and S_1 is forced to start execution in slot 4, which is in I_0. So S_1 will not need this slot in I_1 anymore and the spare capacity of I_1 is increased: $sc(I_1) = -1 + 1 = 0$. So when I_1 starts, its spare capacity is non-negative, as required.

5 Global Shifting

The local approach deals with incorporating aperiodic tasks on single nodes only. In this section, we extend it to handle aperiodic activities in the distributed system.

5.1 Shifting Inter Node Messages

The *global shifting mechanism*, allows inter-node messages to be shifted as well. This enables earlier execution on the receiving node. As for tasks on

each node of the system, static scheduling defines an execution sequence of messages for the communication medium. Consequently, messages can be shifted within that sequence in the same manner as tasks on processing nodes. The restricted accessibility of the communication medium, as imposed by protocols as TDMA or token ring, has to be taken into account, though. The methods for deriving start-times and deadlines, the construction of intervals, and the calculation of spare capacities remain unchanged and is applied to the communication nodes as well.

The difference for global shifting is that inter-node messages can be sent earlier, i.e., the earliest start-times of tasks receiving these messages will decrease at run-time. This may extend intervals and in turn increase spare capacities. Let M_{IM} denote an inter-node message that is scheduled to be received at time $end(M_{IM}^{sched})$. It is received earlier at run-time, at time $end(M_{IM}^{new})$, $end(M_{IM}^{new}) < end(M_{IM}^{sched})$. Furthermore, $I_{i_{IM}}$ stands for the interval that the task receiving M_{IM} is assigned to. The early transmission of M_{IM} may have one of the following effects:
Spare capacities of $I_{i_{IM}}$ increase: This is the case when $end(I_{i_{IM}-1}) < start(I_{i_{IM}})$. Then $I_{i_{IM}-1}$ and $I_{i_{IM}}$ do not overlap, they are independent. Let us assume M_{IM} was sent δ slots earlier. Then $end(M_{IM}^{new}) = end(M_{IM}^{sched}) - \delta$, $\delta > 0$ and the spare capacities of $I_{i_{IM}}$ are given as $sc(I_{i_{IM}}^{new}) = sc(I_{i_{IM}}^{sched}) + \delta$.

Spare capacities of other intervals increase: The early receiving of M_{IM} enables tasks assigned to $I_{i_{IM}}$ to be executed early using unused spare capacities in earlier intervals, increasing the spare capacities of the intervals they were scheduled to execute in. Let I_{early} be the interval with $early < i_{IM}$ and $end(I_{early}) < end(M_{IM}^{sched})$ and $end(I_{early}) \geq end(M_{IM}^{new})$. There exist intervals I_j with $early \leq j \leq i_{IM}$. Let *unused* denote the slots of unused spare capacities in these intervals. Then tasks $T_k \in I_{i_{IM}}$ can execute in *unused*, and thus increment spare capacities in the intervals they were scheduled to execute in.

No spare capacities increase: The decreasing of start-times does not necessarily allow tasks to start early, e.g., if the schedule is tight. In that case no spare capacities are increased, and the global mechanism performs in the same way as the local one.

5.2 Node Selection

The local guarantee algorithm determines, whether an aperiodic task that has arrived at a particular node can be executed on that node feasibly. If the task is not accepted, it will be discarded. In a distributed system, however, other nodes might be capable of accepting this task [23]. We can use the local guarantee algorithm to perform a system wide acceptance test. By subsequently performing the local guarantee algorithm for a task on all nodes, we can decide whether the task can be accepted anywhere in the system. This requires that aperiodic tasks arrive at a selected node, which in turn initiates the local tests for each aperiodic in sequence. While this method is simple, it is not optimal if more than one aperiodic task arrives to the system at the same time: Tasks are guaranteed and assigned to nodes in sequence, one by one. So a different task order for the acceptance test may result in a different set of guaranteed tasks.

5.3 Distributed Complex Aperiodic Tasks

We use static scheduling to prepare the complex periodic tasks for on-line scheduling. It transforms these tasks into simple independent, start-time-deadline constrained tasks on single processors. To handle distributed complex aperiodic tasks, we propose a similar step. They can be transformed into single independent, deadline constrained aperiodic tasks on single processors using the method described in [6]. During schedule preparation, each of these aperiodic tasks is broken down into a set of aperiodic tasks assigned to certain nodes. Upon the arrival of a complex aperiodic task at run-time, this set of tasks is known for each node. So we perform the local acceptance test on all nodes for all these tasks. Only if all tasks comprising the complex aperiodic task can be guaranteed at all nodes, it is accepted.

6 Analysis

We have implemented the described mechanisms and have run simulations for various scenarios. The system simulated consists of 4 processing nodes, connected via a TDMA driven communication medium. Some 1600 task-sets were generated and scheduled by the MARS scheduler [7]. We randomly generated aperiodic task loads for each of these, so that the combined load of static and aperiodic tasks was set to 20%, 40%, 60%, 80%, and 100%. Aperiodic tasks were simple and preassigned to processing nodes. The deadline for the aperiodic tasks was set to their MAXT and two times MAXT. We studied the guarantee ratio of the - randomly arriving - aperiodic tasks, and the performance of both local and global scheme. Each point represents a sample size of 700 - 1000 task sets. 0.95 confidence intervals were smaller than 5%.

Figure 5 depicts the overall simulation results. The

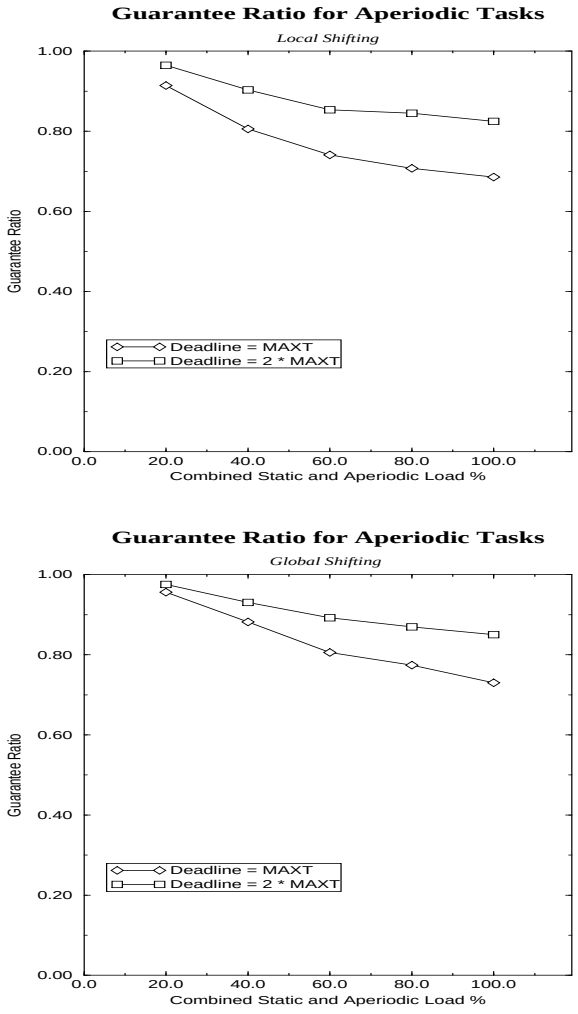


Figure 5: Guarantee Ratio for Aperiodic Tasks

graphs show the efficiency of the slot shifting mechanisms, as guarantee ratios are very high. As expected, the guarantee ratio for aperiodic tasks with larger deadlines is higher than for smaller deadlines. Even under high load, guarantee ratios stay high. As expected, global shifting is superior to local shifting. The difference decreases for low loads. Low loads provide large amounts of spare capacities and leeways. So even the local mechanism will be able to guarantee a high number of aperiodic tasks.

We have further studied the influence of the tightness of timing constraints of the statically scheduled tasks on the guarantee ratio. We used the following scheme to test the algorithm under different deadline and periodicity constraints: Two parameters, *period_laxity*(pl) and *deadline_laxity_factor* (df) were used to set deadlines and periods of precedence graphs, respectively, as follows: $period(PG) = total\_cost(PG) \times pl$ and $dl(PG) = period(PG) \times df$ where *total_cost* is the sum of the computation as well as communication within a precedence graph. The results presented here have the following settings: pl of 1.2 and 1.5, and df of 0.6, 0.8 and 1.0; load was generated for 40% and 80%. The resulting guarantee ratios are depicted in figure 6. Note that each plot consists of two graphs, one for each setting of pl.

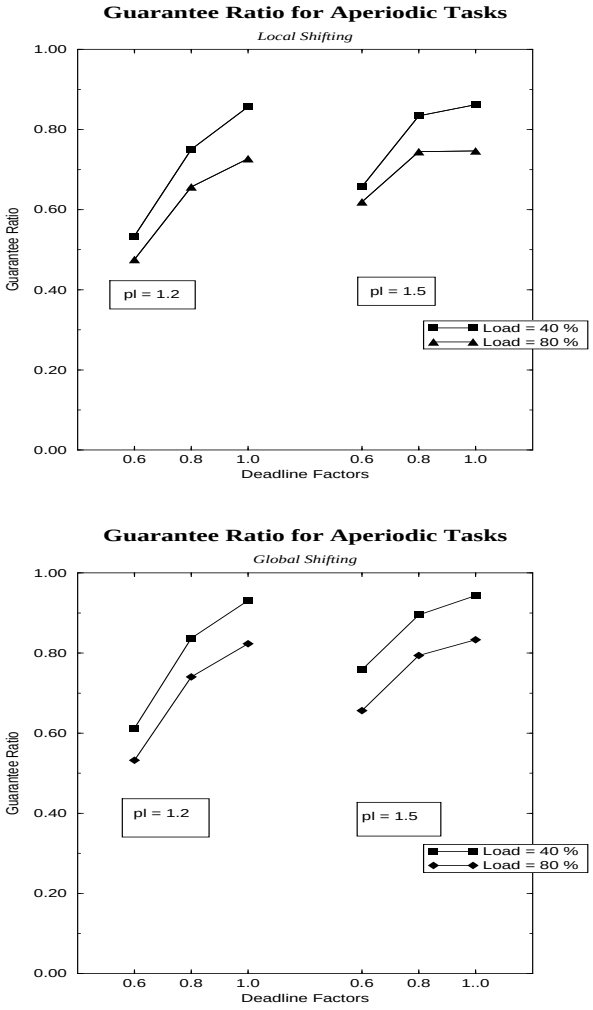


Figure 6: Guarantee Ratio for Aperiodic Tasks under Different Timing Constraints of Statically Scheduled Tasks

The plots of the individual settings follow the same tendencies as of the overall guarantee ratio discussed above. As deadlines increase, guarantee ratios do so, too, since more leeway in the deadlines allows for more

shifting. The guarantee ratios of both approaches benefit from the increased leeways of schedules with larger periods.

7 Summary

In this paper we presented algorithms for the joint scheduling of precedence constrained distributed periodic and hard and soft aperiodic tasks in statically scheduled systems. The systems under consideration are assumed to be consisting of a set of nodes, connected by an inter node communication medium with bounded transmission times such as TDMA. Our approach, *slot shifting*, makes use of the unused resources and leeways in static schedules, called spare capacities, to incorporate aperiodic tasks.

In a first step, static schedules are created for the distributed, precedence constrained periodic tasks. The created scheduling tables are then used to determine spare capacities. While the periodic tasks are complex and distributed, the spare capacities are assigned locally to each node in the system. We use the spare capacities to test, whether aperiodic tasks can be executed without violating the feasibility of periodic tasks. If an aperiodic task is executed, the execution of the periodic tasks is shifted. We presented an optimal on-line guarantee algorithm for hard aperiodic tasks of O(N), N the number of tasks on that node.

In order to reflect distribution, we have presented extensions to allow for the shifting of inter node message as well. Furthermore, we addressed guidelines for the selection of nodes to execute aperiodic tasks and to handle distributed complex aperiodic tasks using the slot shifting algorithm as basis.

As the major part of the calculations involved is performed off-line during schedule preparation, the on-line mechanisms are very simple and require only little memory. Nodes can be fully utilized. Implementation and analysis by extensive simulation of the slot shifting algorithm show its efficiency. The resulting guarantee ratios of aperiodic tasks were high for various load and deadlines scenarios. Our algorithms integrate fully with the mode change algorithm [8]. This allows real-time systems to be statically scheduled and highly flexible towards major and minor changes in the environment.

Using the slot shifting algorithms, future research will include issues of integrating sporadic tasks, i.e., aperiodic tasks with lower bounds on their inter-arrival times, using priorities to indicate importance, and ellaborating on node selection and complex aperiodic task handling.

Acknowledgements

The author wishes to thank Krithi Ramamritham for his interest in the work, fruitful discussions and helpful comments, Hermann Kopetz and the MARS research team for support, establishing a creative atmosphere, and countless discussions. The student Christian Erlacher implemented the work presented here and provided many useful comments. Finally, a very sincere thanks goes to Dr. Epiniki Politschnig of Neunkirchen hospital, whose outstanding competence, initiative, and persistence saved the author's life in a critical medical situation, allowing this paper to be written.

References

[1] N.C. Audsley, R.I. Davis, and A. Burns. Mechanisms for enhancing the flexibility and utility of hard real-time systems. In *Proc. 15th Real-Time Systems Symposium*, pages 12–21, San Juan, Puerto Rico, Dec. 1994.

[2] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. 11th Real-Time Systems Symposium*, pages 182–190, Dec. 1990.

[3] H. Chetto and M. Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Information Proceedings Letters*, 30:177–184, February 1989.

[4] H. Chetto and M. Chetto. Some results on the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[5] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proc. 14th Real-Time Systems Symposium*, pages 222–231, Dec. 1993.

[6] M. DiNatale and J.A. Stankovic. Dynamic end-to-end guarantees in distributed real-time systems. In *Proc. 15th Real-Time Systems Symposium*, pages 206–215, San Juan, Puerto Rico, Dec. 1994.

[7] 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.

[8] 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.

[9] G. Fohler. *Flexibility in Statically Scheduled Real-Time Systems.* PhD thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Wien, Österreich, April 1994.

[10] M.R. Garey, D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and dead lines. *IEEE Transactions on Software Engineering*, 10(2):256–269, May 1981.

[11] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proc. 12th Real-Time Systems Symposium*, pages 89–99, Dec. 1991.

[12] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *12th Int. Conf. on Distributed Computing Systems*, pages 460–467, Yokohama, Japan, June 1992.

[13] J.P. Lehoczky and Sandra Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proc. 13th Real-Time Systems Symposium*, pages 110–123, Dec. 1992.

[14] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. 8th Real-Time Systems Symposium*, pages 261–270, 1987.

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[16] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology, 1983. Report MIT/LCS/TR-297.

[17] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[18] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *10th Int. Conf. on Distributed Computing Systems*, pages 108–115, 1990.

[19] K. Ramamritham, G. Fohler, and J.-M. Adan. Issues in the static allocation and scheduling of complex periodic tasks. In *Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software*, NY, NY, USA, May 1993.

[20] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–265, December 1989.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, C-39(9):1175–1185, September 1990.

[22] M. Spuri and G.C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proc. 15th Real-Time Systems Symposium*, pages 2–11, San Juan, Puerto Rico, Dec. 1994.

[23] J. A. Stankovic, K. Ramamritham, and S.-C. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, 34(12), Dec. 1985.

[24] S. R. Thuel and J.P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proc. 14th Real-Time Systems Symposium*, pages 160–171, Dec. 1993.

[25] S. R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proc. 15th Real-Time Systems Symposium*, pages 22–33, San Juan, Puerto Rico, Dec. 1994.

[26] K.W. Tindell, A. Burns, and A.J. Welling. Mode changes in priority pre-emptively scheduled systems. In *Proc. 13th Real-Time Systems Symposium*, pages 100–109, Dec. 1992.