Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems

Jens Hildebrandt, Frank Golatowski, Dirk Timmermann
{hil, gol}@e-technik.uni-rostock.de

University of Rostock
Department of Electrical Engineering and Information Technology
Institute of Applied Microelectronics and Computer Science
Richard-Wagner-Str. 31
18119 Rostock-Warnemünde
Germany

Abstract

Scheduling time impact on system performance increases especially when using dynamic priority algorithms, because of the enlarged computational effort at runtime. This overhead can be reduced by using dedicated hardware that does the time consuming computations necessary for scheduling. This can be a coprocessor capable of implementing dynamic scheduling algorithms which are, until now, rarely used because of their complex computations at schedule time. One of these algorithms is Least-Laxity-First (LLF). This is an optimal scheduling methodology that allows detection of time constraint violations ahead of reaching a tasks deadline, but has the disadvantage of showing poor runtime behavior in some special situations ("thrashing").
In this paper, we present a universal deterministic scheduling coprocessor that implements the newly developed Enhanced Least-Laxity-First-algorithm (ELLF) which eliminates this disadvantage of LLF. Computation time of this device is rather a matter of time resolution than of the number of tasks .

1. Introduction

Operating system overhead degrades the performance of a computer system as it reduces the utilization available for user processes below the theoretical possible value. This is especially important for real-time systems where tasks must complete prior to a given deadline. Every computation time unit consumed by operating system for scheduling, context switches etc. narrows the time frames for the completion of tasks.
Another equally important aspect of real-time systems is predictability. Schedulability analysis needs to know worst-case execution times of tasks as well as of operating system functions in order to verify if a given set of tasks can be executed without missing any deadlines. Furthermore, these execution times should vary as little as possible to allow a prediction of temporal system behavior with a small degree of uncertainty.
Both goals, short execution times and predictability of operating system functions, can be achieved by implementing certain parts of the operating system in hardware. There are several projects addressing this area [2, 5, 6] with different concepts. Our focus is on the scheduler as this part has direct influence on the order of task execution and because of the very frequent activation on operating system overhead as well as on predictability.
When using dynamic priority scheduling algorithms, i.e. algorithms that compute task priorities at run time, like Earliest-Deadline-First (EDF) or Least-Laxity-First (LLF), the advantage of allowing high utilization is accompanied by a high computational effort at schedule time and poor overload performance[1, 8]. A way to partly overcome this dilemma is to implement a scheduler using one of these algorithms in a coprocessor.
In this paper we present a universal coprocessor for Enhanced Least-Laxity-First (ELLF) scheduling for single-processor systems which offers a deterministic behavior thus qualifying itself for hard real-time systems. The algorithm that is used, an improvement of Least-Laxity-First scheduling method, was developed in the context of the work described in this paper. This algorithm preserves all advantages of LLF while improving the run time behavior by reducing the number of context switches. This is of high importance as LLF in certain situations causes a big number of unnecessary context switches that can dramatically increase operating system overhead.
The coprocessor design described in this paper implements the ELLF algorithm. It represents a passive scheduling coprocessor, i.e. the device determines the task to be executed next only after an external start signal. It

does not perform context switches neither does it monitor the tasks states. These are duties of the operating system as the coprocessor has no access to processor registers. Switching from one task to the one selected by coprocessor and signaling changing task states to the coprocessor have to be done under software control. The prototype of the coprocessor described in this document supports up to 32 tasks with a parameter resolution of 16 bit. This limitation is of technological rather than conceptual nature due to the maximum size and speed of available target hardware. The coprocessor is universal in that it can be used with almost any combination of hardware and operating system.

This paper comprises an overview over related work in Section 2, an explanation of the ELLF scheduling algorithm in Section 3 followed by a detailed description of the coprocessor and its components in Section 4. An overview over the current state of work is given in Section 5, and we give conclusions in Section 6.

2. Related Work

In this section we present related work in hardware-supported scheduling.

RTU [2] is a hardware implementation of a real-time kernel. The RTU includes all time critical functions of the operating system and is designed to work in both single- and multiprocessor systems. The main advantages of this design are increased speed and predictability of real-time kernel functions. Beside other functions like interrupt handler, IPC, semaphores, flags, periodic start of tasks with deadline control etc. , one component of RTU is a scheduler. This component uses the rate-monotonic static scheduling algorithm. In contrast to dynamic scheduling algorithms used in our work, static methods like rate-monotonic have problems handling aperiodic tasks and achieve lower utilization. The prototype discussed in [2] can handle 64 tasks with 8 priority levels.

The Spring scheduling coprocessor (SSCoP) [6] is designed to work together with the Spring-Kernel [9] in real-time systems with more than one processor and /or shared resources. As there is no optimal scheduling algorithm for such systems, SSCoP uses heuristic functions. The device tries to build a feasible schedule for a set of nonpreemptive tasks. If it is able to do this, it guarantees that every task in this schedule will satisfy its timing constraints. New tasks are added to the set if and only if a feasible schedule can be found for the junction of the existing task set and the new tasks. In this case the new tasks are guaranteed, too. The SSCoP can use static as well as dynamic scheduling methods. Because of the dedication of the SSCoP to the Spring-Kernel, it can be difficult to use with other operating systems or in embedded systems with low resources.

In [5] Kim and Shin describe a scalable hardware scheduler for ATM networks. This device, using the Earliest-Deadline-First algorithm, sorts incoming data packages from a number of channels into an output queue in the order of deadlines that were assigned to the packages according to their transmission bandwidth. Further, the scheduler can handle non-real-time traffic, too. This scheduler is very specialized and can not easily be used for other applications, as our solution can.

Another related work, although there is no hardware implementation mentioned, is [7]. In this paper Oh and Yang propose a Modified Least-Laxity-First algorithm (MLLF) that is aimed at the same problem as ELLF, the reduction of the number of context switches at runtime. Although very similar from its main idea and results to ELLF, MLLF uses a more complex algorithm to determine the order of execution of tasks in situations where two or more tasks have the least laxity.

3. Enhanced least-Laxity-First Scheduling

This section describes the scheduling algorithm used in the scheduling coprocessor. In the following three subsections, we will first give a short explanation of Least-Laxity-First scheduling as this is the method our algorithm is based on. Then, the working principle of the Enhanced Least-Laxity-First scheduling algorithm (ELLF) will be described and finally, we will show that our algorithm is a real enhancement of Least-Laxity-First scheduling.

3.1 Least-Laxity-First Algorithm

The Least-Laxity-First algorithm (LLF) is a dynamic scheduling method, i.e. it makes the decision about which task to execute next at schedule time. For every task ready to run at the given moment the difference S between the time until deadline D and the remaining computation time C is computed. This difference, called slack or laxity, can be seen as an inverted priority value. The task with the smallest S-value is the one to be executed next. Whenever a task other than the currently running one has the smallest slack a context switch will occur.

Least-Laxity algorithm is an optimal scheduling method. That means, if a given set of tasks is schedulable, then it can be scheduled by Least-Laxity-First. [4] Another great advantage of the Least-Laxity-First algorithm is the fact that except schedulability testing no further analysis, e.g. for assigning fixed priorities to the tasks, has to be done at development time. Furthermore, a task going to miss its deadline is recognized at the same moment when its slack turns to zero with the task currently not being executed. At that time the deadline is not yet reached and emergency measures can be taken to cover the miss of a deadline.

These advantages are accompanied with the disadvantage of an enormous computational effort at schedule time.

Furthermore, Least-Laxity-First shows poor performance in situations in which more than one task have the smallest slack. In this situation, every basic time unit a context switch to another one of these tasks takes place until all of them are finished (Fig. 1). This behavior is called "thrashing" and creates a great amount of unnecessary context switches, as the tasks would meet their deadlines if they were executed without preemption, one after another. The increased number of context switches means a loss of computation time, since the time needed for such an operation is not neglectible, and makes offline schedulability analysis more complicated.

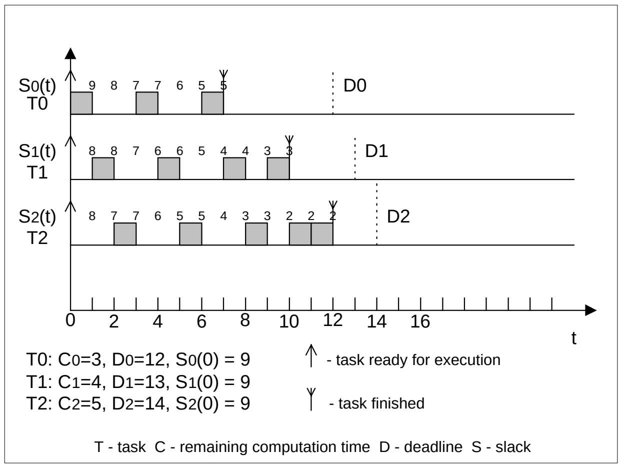
S0(t)
T0
S1(t)
T1
S2(t)
T2
D0
D1
D2
t
T0: C0=3, D0=12, S0(0) = 9
T1: C1=4, D1=13, S1(0) = 9
T2: C2=5, D2=14, S2(0) = 9
- task ready for execution
- task finished
T - task C - remaining computation time D - deadline S - slack

Figure 1 Least-Laxity-First with thrashing

3.2 Working Principle of Enhanced Least-Laxity-First Algorithm

To avoid thrashing, a modification of the original Least-Laxity-First algorithm was made. The aim of this improvement is to ensure that in a situation when some tasks would normally start to thrash, they are executed consecutively without preempting each other. This can not be done by simply making the whole system temporarily non-preemptive. With such a non-preemptive LLF-algorithm, tasks may miss their deadlines. This happens, if both the time a task gets active and the moment its slack turns zero arrive during the run time of the currently running task.
The algorithm used in the scheduling coprocessor solves this problem by excluding all except one of those tasks that would thrash (Fig. 2). The exclusion is valid until the task that was chosen for execution finishes or is preempted by another task. The excluded tasks are not suspended. They still take part in the comparison of slack values. With their continuously decreasing slack value these tasks build up a threshold. A task can preempt the currently running one only if it is not excluded and if it has a slack smaller than that of the excluded tasks. If the latter condition is not true, the task has to wait, even if its slack is smaller than that of the running task.
The initial choice, which task of a set of tasks having the smallest slack will be executed, is based upon deadlines. The one with the earliest deadline of this group is chosen. Thus, comparing deadlines of the tasks that were previously selected as having the smallest slack guarantees that no task misses its time constraints because it is excluded from execution by a task with a later deadline. If there are two or more tasks having the earliest deadline of all the tasks with the smallest slack then one of them is randomly chosen for execution.

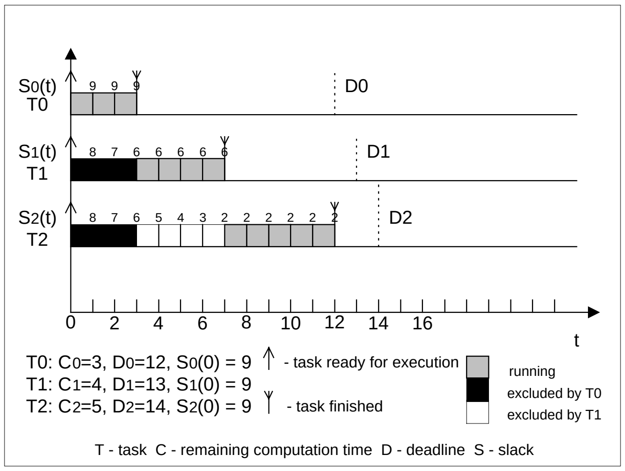
S0(t)
T0
S1(t)
T1
S2(t)
T2
D0
D1
D2
t
T0: C0=3, D0=12, S0(0) = 9
T1: C1=4, D1=13, S1(0) = 9
T2: C2=5, D2=14, S2(0) = 9
- task ready for execution
- task finished
running
excluded by T0
excluded by T1
T - task C - remaining computation time D - deadline S - slack

Figure 2 Enhanced least-Laxity-First

3.3 Advantages of Enhanced Least-Laxity-First Scheduling

The Enhanced Least-Laxity-First algorithm has all the advantages of the original Least-Laxity-First scheduling method. Under normal conditions, i.e. when there is only one task having the least laxity, both algorithms behave identically as the additional comparison of deadlines done in ELLF becomes trivial.
The improvement with regard to Least-Laxity-First becomes obvious in situations where two or more tasks have the smallest slack.
Given a set of tasks in which at the time t_0 a subset V of n tasks $T_1...T_n$ have the same, smallest slack, with T_1 having the earliest and T_n having the latest deadline of these n tasks. We assume that there will be no other task with a smaller slack until $T_1...T_n$ are finished and that the Least-Laxity-First algorithm will chose tasks for execution in the order of their numbering. Each task T_i has a deadline D_i and its remaining computation time C_i. The moment task T_i finishes, the response time, we will call R_i . R_i, D_i and C_i are given in multiples of the basic time unit, which is the activation period of the scheduler, and are measured relative to t_0. For the purpose of simplification we assume task switching times to be ideal, i.e. equal zero.

3.3.1 Response Time Analysis of Thrashing Tasks We will show, that for ELLF response times R_i are less or equal to the response times w.r.t. LLF, i.e. :

$$R_{i\ ELLF} \leq R_{i\ LLF}$$

When scheduled by LLF, the tasks $T_1 ... T_n$ of V will thrash until all of them are finished.
For n = 2 we will find

$$R_1 = 2*C_1 - 1 \text{ and}$$
$$R_2 = C_1 + C_2$$

For n = 3 this will be

$$R_1 = 3*C_1 - 2$$
$$R_2 = C_1 + 2*C_2 - 1 \text{ and}$$
$$R_3 = C_1 + C_2 + C_3$$

Or, for any n:

$$R_{i\ LLF} = n*C_1 - (n-1) \qquad (i = 1)$$

$$R_{i\ LLF} = \sum_{j=1}^{i-1} C_j + (n-i+1)*C_i - (n-i) \qquad (1 < i \leq n)$$

For ELLF, tasks $T_1 ... T_n$ are executed in the order of their deadlines, which in this case is the order of numbering. Thus:

$$R_{i\ ELLF} = \sum_{j=1}^{i} C_j \qquad (i \leq n)$$

Building the difference between these two formulas we get:

$$R_{i\ LLF} - R_{i\ ELLF} = (n-i)*C_i - (n-i) \geq 0$$

This shows that response times with ELLF, compared to the response times of LLF, are equal for i = n and smaller for all other i < n in situations when two or more tasks have the smallest slack.

3.3.2 Number of Context Switches Another advantage of ELLF is the reduced number of context switches N_{cs} until T_n is finished. As for ELLF, there is one context switch at the beginning of each task T_i, when using LLF the number of context switches is equal the number of time units until the final context switch from T_{n-1} to T_n, i.e. R_{n-i} (in time units) plus one. Thus:

$$N_{csELLF} = n$$

$$N_{csLLF} = R_{n-1} + 1$$

This means, with computation times Ci > 1, the ELLF algorithm needs less task switches than LLF. This can be seen in the examples at figures 1 and 2 where the schedules are given for a set of thrashing tasks when using LLF (Fig. 1) and ELLF (Fig. 2), respectively.
The reduced number of context switches is important for real systems, where context switching times are not zero. Even if these times are very small , compared to the time base period, given the high number of context switches with LLF when thrashing occurs, these delays can sum up to an amount of one or more time units.

4. Architecture of the Scheduling Coprocessor

The ELLF scheduling coprocessor has a microprocessor compatible interface (Fig. 3) that can easily be adapted to different data width and transfer protocols. The coprocessor core is not affected by this.
The main components of the core are a number of identical functional blocks (task blocks), representing one task each, a comparator unit and several output modules equal in number to the number of task blocks. (Fig. 4)
For each task, the slack value is computed, and the current state is stored in the task blocks. The comparator unit determines the task that has the earliest deadline of all tasks with the smallest slack. The output modules translate the result of the comparator into task identification numbers which are given on the coprocessors output after a request by the operating system.

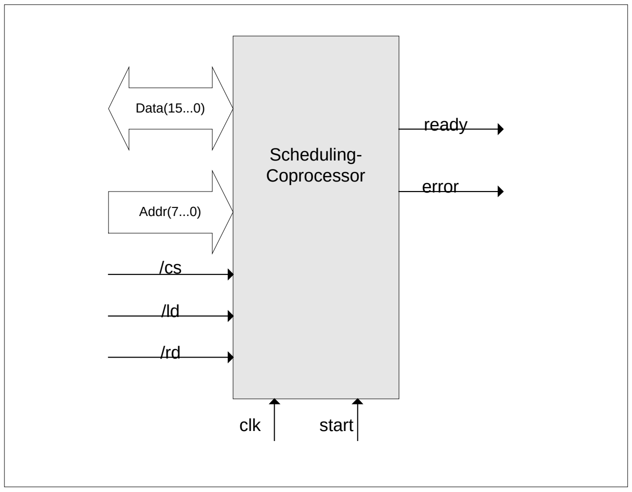

Figure 3 External interface of the coprocessor

The scheduling coprocessor has to be initialized before it can work. This means that for every task in the system the corresponding deadlines, computation times and initial states have to be loaded into a task module. There are four states a task can be in: “suspended”, “waiting”, “ready” and “running”. The modules and their internal registers are selected via address lines. The upper bits of these addresses are identical with the task identification numbers that are returned as the coprocessors result. The initialization parameters are transferred into the coprocessor via a bi-directional data port which is also used to output the scheduling result.
A scheduling operation is triggered by an external start signal. This could be a time-base clock or an operating system controlled signal. It has to be periodic as it is used as the internal time-base of the coprocessor. For

synchronization purposes it is recommended to generate the start signal under operating system control as there must not be any change in a tasks state during scheduling operations.
Two consecutive trigger pulses are generated from the start signal inside the coprocessor. The first one starts the computation of current slacks inside the task modules. The second one triggers the comparison of these values inside the comparator. When the comparator finishes its work, the availability of a result is signaled to the operating system via an output line that can be polled by the system or used to trigger an interrupt. The identification number of the task that has to be executed can then be fetched within a read access to the coprocessor.
If a task is going to miss its deadline, i.e. its slack has become zero and it is not running, this failure is reported to the system via an error-signal line. In response to this, the operating system can request the ID-number of the task that caused the error.

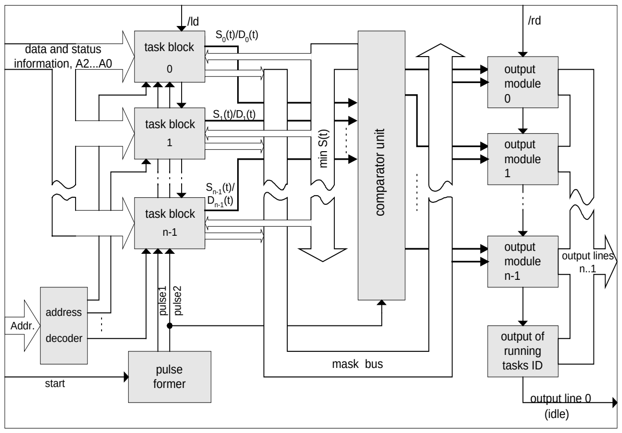

Figure 4 Internal structure of the coprocessor

The exclusion of tasks from execution being a main element of the ELLF algorithm is implemented by two internal busses. One of them is driven by the comparator. Its width is equal the number of tasks supported by the coprocessor. Its content is the result of the comparison of slack values that is done inside the comparator. In this vector all bits are set except those that correspond to tasks having the smallest slack. This vector is propagated to all task blocks which need this information to exclude other tasks from execution. The other bus is driven by the task block of the currently running task. The lines of the mask bus are connected to the corresponding output modules. The modules do not drive the coprocessors output if their mask input bit is „zero". Thus, the identification number of a task can not be transmitted to the operating system even if this task was determined by the comparator as being the one to be executed . Instead, if no other output module is driving the coprocessor output lines, the ID of the currently running task is given on the port as a default value, so no context switch will take place. In case no task is running, the least significant bit of the output is set to „one" thus indicating the idle state of the system.
On the following pages are given detailed descriptions of the coprocessors main components.

4.1 Task Blocks

A task block (Fig. 5) represents a task of a real-time system. It stores the tasks initial deadline D and worst case computation time C and computes the currently remaining time until deadline D(t), the remaining computation time C(t) and the current slack S(t) at schedule time. For this purpose, a task block contains two pre-loadable downward-counters for D(t) and C(t) and a subtractor that builds the difference S(t) = D(t) – C(t). As this result is transmitted to the coprocessors comparator unit in serial mode, a shift register is included in the task block that does the serialization of S(t).
The computation and the transmission of slack are triggered by two external signals, *pulse1* and *pulse2*, which are generated by a pulse former unit inside the coprocessor from the incoming start-signal. These signals are active consecutively. First, *pulse1* decreases the counters and then the result S(t) of the subtractor is clocked into the shift register by *pulse2*.
The register starts shifting out its contents as soon as shifting is enabled by the comparator unit via the *shift_enable*-signal. A second input signal of the shift register causes it to load the currently remaining time until deadline D(t) instead of S(t). This value is needed by the comparator unit in the second stage of its work and is shifted on the same output as S(t).
Changes of D(t) and C(t) depend on the current state of the task. D(t) starts decreasing as soon as a task is activated, i.e. when it leaves state "suspended". C(t) decreases only for a task that is currently executed, i.e. it is in state "running". The state information as well as additional internal information of the task block is contained in a state register.
Access to the task blocks is possible via 16 data– and 4 state-lines. A selection signal that is generated by the address decoder of the coprocessor from the five upper address lines enables write accesses to a task block. The lower three address lines are connected directly to the task blocks and are used for internal register selection. The upper of these lines, when set, signals that the task block is going to be initialized. With that line, called *A2*, activated, a write access to the D(t)- or C(t)-Counter affects the appropriate D- and C-Registers, too. Without that line being active, an access is directed only to one of the counters according to the state of address line *A1*. The state register is written if during a write access the lowest address line *A0* is active.
All changes of a tasks state must be submitted to the appropriate task block of the coprocessor. There is an automatic reload of the counters from the D- and C-registers when a task enters the "suspend"-state after it

has finished. Therefore, there is no need to initialize the task blocks of periodic tasks again.
The C(t)-counter can be set to any value at any time except when a schedule operation is running. This feature is useful when a task going to miss its deadline is replaced by an "emergency"-handler with a shorter remaining computation time while the deadline remains valid.

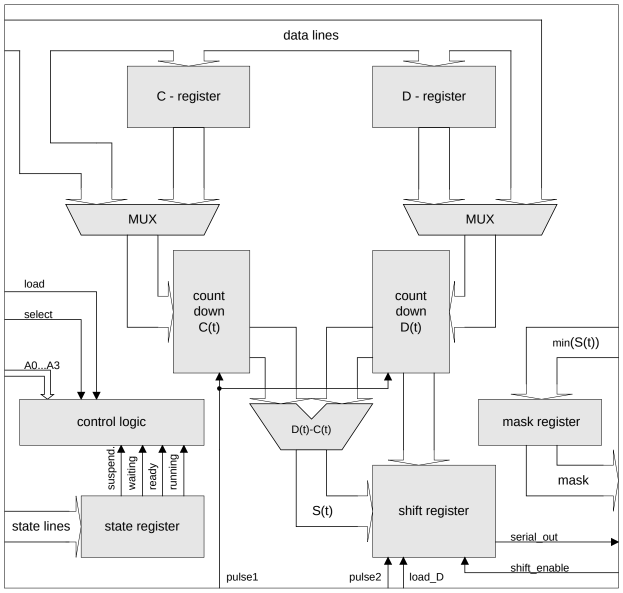

Figure 5 Internal structure of a task block

When the difference S(t) of a task block becomes zero and the task does not enter the "running"-state this task will miss its deadline. This is signaled by activating the error-signal of the task block which is propagated to the corresponding coprocessor-output. Simultaneously, the serial output of the task block is pulled high, thus seeming to the comparator like having the most slack of all tasks. As long as no measures are taken by the operating system to recover the error, the task will not take part in scheduling.
The exclusion of other tasks to prevent thrashing is accomplished by putting a mask bit-vector on the mask bus. When the "running"-Bit in the state register gets active, i.e. the task is going to be executed, the contents of the min(S(t))–Bus is latched into the mask register. Simultaneously the output drivers of that latch are enabled so that its contents is available on the bus. Inside the mask register the bit corresponding to the ID-number of the task block is always set to „one" , no matter what data appears on the registers input. The registers output drivers are disabled as soon as the "running"-bit in the state register is cleared. This disables any previous exclusion of tasks.

4.2 Comparator Unit

The comparator unit (Fig. 6) selects the smallest of all slack values S(t) which are serially shifted in via the corresponding input lines. After that, it reads in the D(t)-values of the tasks in the same manner and performs a comparison of them for those tasks that were previously determined as having the smallest slack. In both cases the result is a bit vector in which those bits are set to "zero" that correspond to tasks having the smallest of the compared values.
While the final result of the second comparison is directed to the output modules of the coprocessor, the result of the S(t)-comparison is given on an internal bus that is used by the task blocks for generating a mask vector.

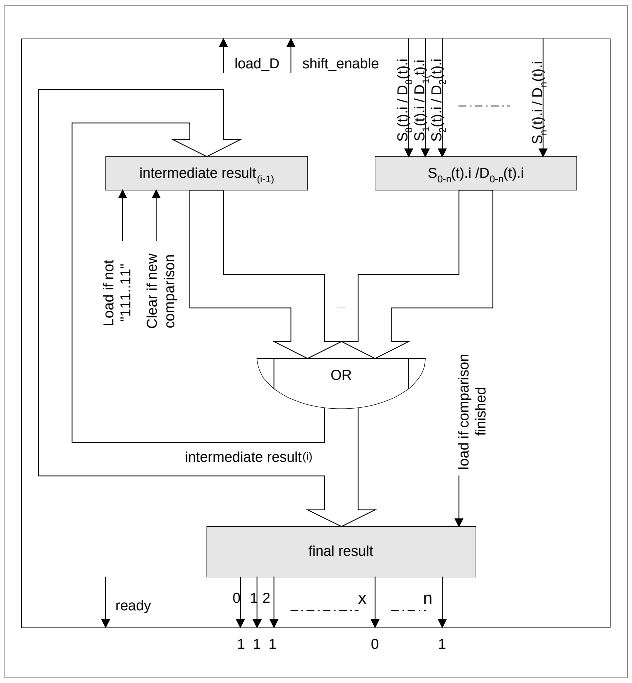

Figure 6 Structure of the comparator unit

The comparisons are done bitwise, starting from the most significant bits. The bits are linked in an OR-operation with the bits of the previous intermediate result. If in the resulting vector not all bits are "one" , it is stored in a register for the intermediate result thus becoming an operator for the next bit stages OR-operation. The intermediate register is cleared at the beginning of an S(t)-comparison and loaded with the result of that operation before the comparison of D(t)-values. The latter measure ensures that D(t) is only compared for those tasks that have the smallest slack S(t) as for all other tasks the corresponding bits in the result are explicitly set to "one".
If for all bit stages in the S(t)-comparison the result of the OR-operation is a vector containing only "ones" the system is in idle state as there is no task ready to run. In this case this "one"-vector is given on the comparators output causing all output modules to be deactivated and the coprocessors output to show a "one" in the least significant bit.
When the comparator unit has finished its work it activates the ready-output of the scheduling coprocessor until there is a read-access to it.
The duration of a complete comparison operation depends on the width of the S(t)- and D(t)-values rather than on

the number of tasks. The time needed to determine the task that has to run next is twice the bit-width in clock cycle periods, as there have to be shifted through the comparator two parameters, one for each comparison.

4.3 Output modules

For each task block of the coprocessor, there is a corresponding output module. (Fig. 7) The three-state outputs of these modules are connected together to the coprocessors output lines. If an output module is enabled, it drives the output lines with the ID-number of the task block it is associated with.
Whether an output module is enabled or not, depends on three signals. First, the corresponding task has to have the earliest deadline of all tasks that have the smallest slack, i.e. the task's bit in the output bit-vector of the comparator has to be "zero". Furthermore, the task must not be excluded from execution by a "zero"-bit in the mask vector and finally, there has to be an active signal at the modules *enable*-input. If at least one of the two former conditions is not true, an incoming *enable*-signal is propagated to a dedicated output which is connected with the *enable*-input of the next module. By this, the output modules form a daisy chain starting with the module that outputs the ID-number "0". The *enable*-input of this module is connected to the external input */rd* of the coprocessor. The daisy-chain ensures that only one module is enabled at a time. In cases with two or more tasks selected by the comparator which are not excluded from execution the module that carries the lowest ID-number is enabled first. If the operating system wants to know the numbers of all these tasks, it can get them by consecutive read accesses to the coprocessor. An output module, after it has been read once, starts behaving just like a module that was excluded or not selected by the comparator. During following read accesses it propagates the *enable*-signal to the next module in the chain. Thus, with every read-access the operating system gets another ID-number until there are no more output modules that could be enabled. This is signaled by the least significant bit of the coprocessors output which is internally connected to the *enable_out*-port of the last output module in the chain. If the operating system reads an odd number from the coprocessor, this means that the upper bits contain no valid ID-number and that there are no further IDs to be read.
The last module in the daisy chain is a special one. It does not carry the number of a particular task but the ID of the task that is currently running. Each time the operating system sets the "running"-bit in the state register of a task block, the address of this block, which is also the ID-number of the task, is stored in the last output module. This module is enabled only if all other modules propagate the *enable*-signal because they are either not selected by the comparator or excluded from execution or if they have been read earlier.

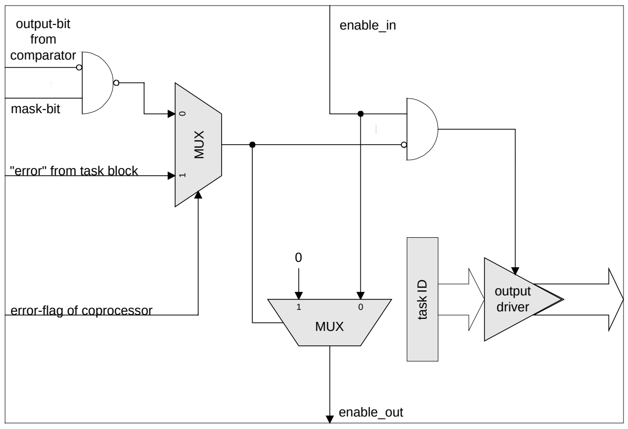

Figure 7 Output module

A second function of output modules is to put the ID-numbers of tasks on the output that are going to miss their deadlines. In cases when one or more task blocks have their error-outputs activated the operating system can connect these lines to the output modules by setting a special flag inside the coprocessor. If this is done, the output modules ignore the mask vector and the comparators output signals. Their output drivers are activated if the associated task block has its error-output set to "zero" and there is an active enable-signal on the input .

5. First Results

The design of the scheduling coprocessor described above was developed together with a scheduling analyzer as part of a HW/SW-co-design project which aimes at developing a rapid prototyping system with reconfigurable hardware for real-time systems with hard time constraints [3].
According to the aspect of reconfigurability, the coprocessor was implemented in an FPGA. Target devices are XILINX XC4000-family FPGAs. The description of the design is done in VHDL. Synthesis is done using SYNOPSYS- and XILINX-tools. Functional and timing verification is realized by back-annotating the completely routed design to a VHDL-file and a file containing timing information.
Several prototypes have been developed this way. In the final version, the coprocessor scheduling up to 32 tasks at a parameter width of 16 bit needs 850ns at 40MHz clock speed (34 clock periods) from the detection of an active start signal until a rising edge on the *ready*-output signals availability of a result to the operating system. This prototype is synthesized for a XC40250XV-FPGA and needs approximately 3200 configurable logic blocks (clb). As the time for a scheduling operation is determined by the resolution of S(t)- and D(t)-values but is independent from the number of tasks to schedule, the number of clock cycles N_{cc} needed for a scheduling operation does not

vary for a synthesized coprocessor design. It can be determined for a given bit-width b as:

$$N_{cc} = 2*b + 2 \qquad \text{(for ELLF)}$$

Where "2*b" originates from the two serial comparisons that are done inside the comparator. The two extra clock cycles are needed for computing the slack values and loading them into the shift register inside the task blocks.
A coprocessor prototype using LLF instead of ELLF, that was developed earlier but uses the same structural concept, needs less time for a schedule operation. There is no additional comparison of deadlines inside the comparator. Hence, for this device N_{cc} is determined as

$$N_{cc} = b + 2 \qquad \text{(for LLF)}$$

Consequently, for task sets that will never or rarely thrash, usage of LLF instead of ELLF is suggested.

6. Conclusion

The coprocessor described in this paper provides dynamic scheduling of real-time tasks with low operating system overhead. The algorithm implemented preserves advantages of Least-Laxity-First algorithm while eliminating its main disadvantage, thrashing. Because of its architecture the scheduling coprocessor is able to work deterministically. This makes it usable for hard real-time systems.
The ELLF algorithm implemented in this particular coprocessor is only one example of hardware supported scheduling. Currently variants of this device using other scheduling algorithms, e.g. Earliest-Deadline-First, are under development. The aim is to evaluate which algorithms are suitable for hardware implementation and to develop a set of various coprocessor-modules making it easy to customize a scheduling coprocessor for a given real-time system by exchanging only specific design elements. The partitioning of the design in functional blocks gives the possibility to easily adapt the design to other scheduling algorithms.
The coprocessor design currently being implemented in a FPGA can easily be transferred onto other technologies as it exists as a VHDL description. The benefit of such move could be a faster and/or cheaper device, as a FPGA provides reconfigurability at the prize of comparable low speed and high costs.

7. Acknowledgments

This work is supported by German Research Foundation (DFG) within the priority program "Rapid Prototyping for Integrated Control Systems with Hard Real-Time Constraints".
The authors would like to thank Peter Altenbernd (c-lab, Paderborn) for his comments and advices.

References

[1] H.Chetto, M.Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm", *IEEE Transactions on Software Engineering*, vol. 15, No.10, pp. 1261-1269 , 1989

[2] J. Adomat et al. , "Real-time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems", *Euromicro RTS'96 Workshop*, L'Aquila, Italy, pp. 164-168, 1996

[3] F.Golatowski, J. Hildebrandt, D. Timmermann, "Rapid Prototyping with Reconfigurable Hardware for Embedded Hard Real-Time Systems", *19th IEEE Real-Time Systems Symposium*, WIP-Proc., Madrid, Spain ,1998

[4] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", *Ph.D. Dissertation*, MIT, 1983

[5] B.K. Kim, K.G. Shin, "Scalable Hardware Earliest-Deadline-First Scheduler for ATM Switching Networks", *Proc. 18th IEEE Real-Time Systems Symposium*, San Francisco, CA, 1997

[6] D. Niehaus et al. , "The Spring Scheduling Co-Processor", *Proc. 14th IEEE Real-Time System Symposium*, Raleigh-Durham, North Carolina, pp. 106-111, 1993

[7] S.-H. Oh, S.-M. Yang, "A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks", *Proc. 5th International Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, 1998

[8] K. Ramamritham, J.A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems", *Proceedings of the IEEE* , vol. 82, No. 1, 1994

[9] J.A. Stankovic, K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", *Special Issue of OS Review*, 23(3), ACM