# Rapid Prototyping with Reconfigurable Hardware for Embedded Hard Real-Time Systems*

Frank Golatowski, Jens Hildebrandt, Dirk Timmermann
{gol, hil}@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**: This paper describes a rapid prototyping system for the design and development of hard real time systems. The main focus of this rapid prototyping system is on scheduling algorithms. Issues of their implementation including hardware solutions will be considered. The contributions of this work include: 1. General scheduling analysis using an extendable scheduling analyzer 2. Description of a reconfigurable scheduling coprocessor 3. First performance evaluation and comparison of two prototype variants. One scheduling algorithm implemented in hardware represents a solution for the threshing problem of least laxity scheduling.*

## 1. Introduction

Scheduling analysis is often used to ensure that all tasks running on a system will meet their deadlines. Much work has been reported in this field [1].

Most of it deals with fixed priority systems (FPS). Dynamic priority systems (DPS) suffer from their disadvantages in terms of overload behavior and computational effort [2, 3]. Thus DPS are rarely used in commercial operating systems despite their ability to gain a much better utilization than FPS.

In real real-time systems several factors have to be considered like scheduling or interrupt latencies. They cause the utilization available for real-time applications to be below the theoretical possible value. Especially on-line scheduling algorithms require an immense computation effort at schedule time.

One way to solve this problem is an implementation of the scheduling algorithm in dedicated hardware. The SPRING-scheduling coprocessor [7] and the Real-Time Kernel Coprocessor RTU [6] are examples of this approach.

In this paper we present our work towards a rapid prototyping system for reconfigurable hardware that makes the development of coprocessors for embedded real-time systems easier and faster. The main focus of this work is on reducing the big impact of scheduling on system overhead especially when using dynamic scheduling algorithms.

We prefer a hardware/software co-design in which we implement only those functions or subfunctions of the kernel in a coprocessor which provide a considerable speed-up through execution in hardware while the rest of the operating system is software. This means that the execution time of a certain hardware function plus the extra time needed for communication between the operating system and the coprocessor has to be much smaller than the software-only execution time so that it is worth the cost of extra hardware. Therefore an analysis of a given real-time system and of the task set to be run has to be done as well as an analysis of the coprocessor that could be used.

This paper is organized as follows. First we describe an extendable scheduling analyzer as a part of the rapid prototyping process. Thereafter we report the design of a reconfigurable scheduling coprocessor and first results of a performance evaluation.

## 2. Using an extendable Scheduling Analyzer

Our WindowsNT based scheduling analyzer includes well known scheduling analysis and algorithms (dynamic and fixed priority) for uniprocessor systems and resource protocols. Until now the following scheduling algorithms and appropriate scheduling analysis are integrated (Figure 1): preemptive priority based (PRIO), rate-monotonic (RMS), deadline-monotonic (DMS), earliest deadline first (EDF), earliest deadline late (EDL), preemptive and non-preemptive least-laxity first scheduling (LL). The scheduling analyzer is extendable with user defined (USER) scheduling analysis by writing a new Windows-DLL (dynamic link library). This DLL consists of an interface and the appropriate empty wrapper functions.

* This work is supported with funds of the Deutsche Forschungsgemeinschaft within the priority program "Rapid Prototyping for Integrated Control Systems with Hard Real-Time Constraints"

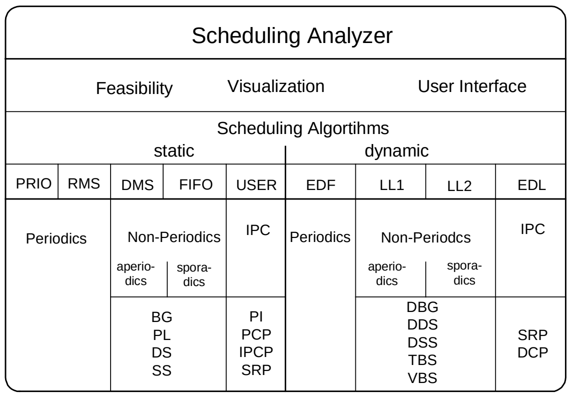

Figure 1: Components of an extendable scheduling analyzer

The specification is done according to [5] by specifying process and resource tables. Both periodic and non-periodic processes (sporadics, aperiodics) are treated. The usage of servers (background (BG), polling (PL), deferrable (DS), sporadic (SS) for static priority systems and the dynamic server variants (dynamic background (DBG), deadline deferrable (DDS), dynamic sporadic (DSS) ) or total bandwidth servers (TBS, VBS) for dynamic priority systems to handle non-regular events is possible. In addition interprocess communication protocols like priority inversion (PI), priority ceiling protocol (PCP), instant priority ceiling protocol (IPCP), stack resource protocol (SRP) and dynamic ceiling protocol (DCP) are supported. As an example we have integrated analysis shown in [10] in an own USER-DLL. The scheduling analyzer is an interactive tool with a powerful window based frontend, specification editor and integrated help system. Also factors affecting the real behavior like jitter, different priority levels, scheduling overhead, system timer may be taken into account.

The scheduling analyzer supports a database interface. This database contains real numbers measured in real systems for example execution times of system calls, fine-grained system times and information of the system (used RTOS and platform). The database has a standard interface (ODBC, SQL) and can also be integrated into an intranet.

In the context of this work we use the scheduling analyzer to decide whether or not it is necessary or possible to use dynamic priority systems. We will concentrate on least laxity first scheduling here.

## 3. Reconfigurable Scheduling Coprocessor

A main part of the rapid prototyping system is a reconfigurable scheduling coprocessor. It will help minimizing system overhead in hard real time systems using dynamic scheduling algorithm like earliest-deadline-first (EDF) or least-laxity-first (LLF). This coprocessor makes it possible to use scheduling algorithms that are to time consuming if implemented in software.

Our aim is not to build the entire scheduler in hardware but only those parts which promise to achieve a significant speedup when implemented in a coprocessor. In general these will be computations that can be done by a dedicated hardware device in a fully parallel manner and with higher speed.

We are developing a scheduling coprocessor design that does the entire online computation for a dynamic scheduling algorithm and gives back the identification number of the task that has to be executed next. All other functionality of the scheduler like context switch, event registration or handling of resource requests is implemented in software. This is a *passive* scheduling coprocessor. It starts computation only after being triggered by the host and has to be provided with information about changing task states or task parameters.

The design of the coprocessor consists of a set of identical functional blocks. Each of them represents a single task and contains several parameters and status information. Inside these blocks a value is computed. Depending on this result the next task to be executed is determined. Note: the meaning of these values is determined by the scheduling algorithm used and thus differs in different coprocessor implementations. The comparison of all these values is done in a separate module that produces the ID number of the task block that has the smallest/biggest computed value, depending on the scheduling algorithm used.

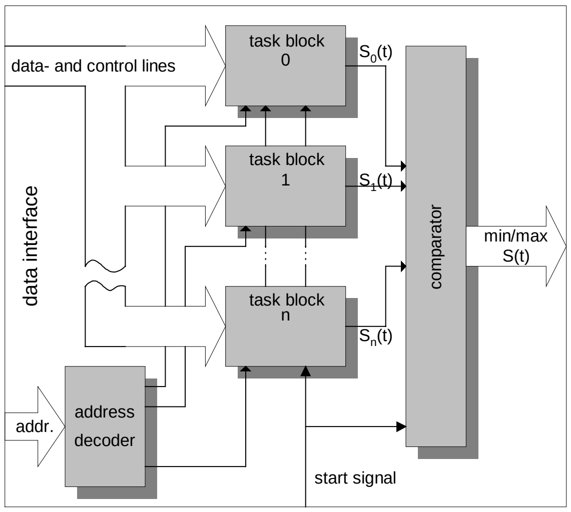

Figure 2: Fundamental design of the Scheduling coprocessor

This ID number appears at the output of the coprocessor and signalizes the host processor that the next task to be executed is the one that is represented by that number.

This fundamental design of a scheduling coprocessor (Figure 2) is highly reconfigurable as it allows the implementation of nearly every scheduling algorithm that is based on a comparison of task parameters. A first prototype of that coprocessor implements the LLF-algorithm. This algorithm decides which task to execute next by computing the slack or laxity, i.e. the difference between the time left until the task deadline *D(t)* and its remaining computation time *C(t)*. The task with the smallest slack *S(t)* is executed first. The main advantage of this algorithm is that a task going to miss its deadline is recognized prior to that deadline as the slack becomes a negative value. Thus measures to handle possible errors and to prevent any damage can be taken before the deadline is reached.

At initialization time the worst case computation time *C* and the deadline *D* of each task are stored in its task block via the data interface of the coprocessor. Furthermore there is a flag register inside the task block that memorizes the status of the task which can be *suspended, waiting, ready* or *running*. The prototype device is a synchronous design driven by a global system clock. The computation of the task to run next is triggered by a time base clock. The period of this clock determines the minimum time interval between two task switches. The *C*- and *D*- parameters are measured in multiples of that time enabling the computation of current *D(t)* and *C(t)* values at scheduling time in simple decrement-by-one operations. These operations are executed depending on the tasks status as the remaining time until deadline gets smaller every time base clock tick for all tasks being not suspended while the computation time left changes only for running tasks.

The comparison of the slack values of all tasks currently being ready to run is done bit by bit in a serial comparator (Figure 3).

This approach has the advantage that the time required for comparing all the values is limited to a fixed number of system clock periods. This number equals the number of bits in the parameter *S(t)*. Furthermore, since the time needed to compute slack values is constant the whole time from triggering the coprocessor until the result is valid on its output is constant too, independent of the number of tasks in the real time system. This guarantees the coprocessor to be deterministic.

Tests and simulations of this prototype revealed that the design performs well but suffers from threshing, a main disadvantage of LLF-algorithm. Threshing implies a situation in which two or more tasks are executed alternating for one time base period each after having all the smallest slack at one certain moment. This can cause a task in a feasible system to miss its deadline due to a loss of time in the great number of context switches during a threshing situation.

In order to solve this problem while preserving the advantages of the LLF-algorithm a new scheduling method was developed and implemented in the coprocessor. It is based on least-laxity–first-scheduling but eliminates threshing. In this algorithm all tasks that have the same – smallest – slack like the one that was just chosen to run are locked out from being executed. They are still taking part in the comparison of slack values until the currently running task ends or until there is a task ready to run with a slack even smaller than that of the locked out tasks.

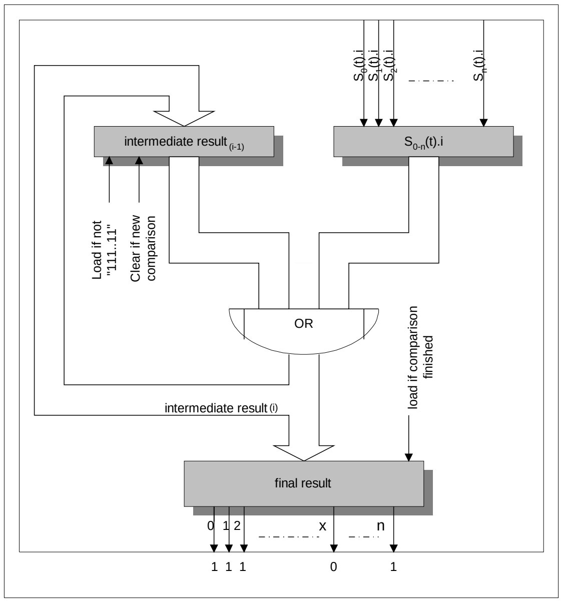

Figure 3 Serial comparator. Task with smallest S-value is marked with ‘0’ at the output

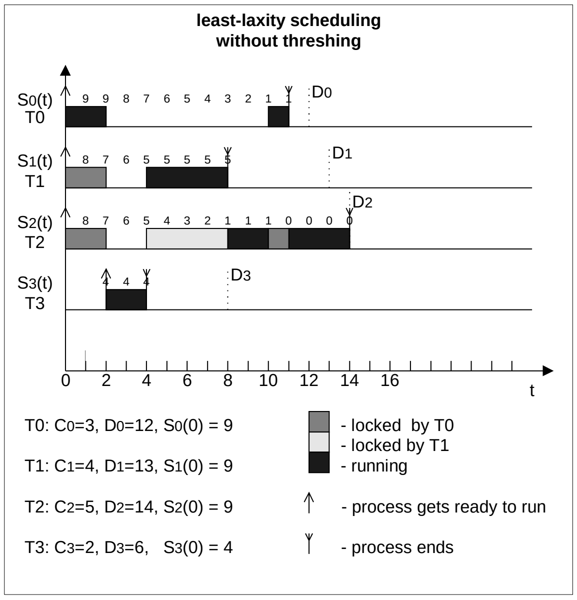

Figure 4 Example for LLF without threshing

The latter condition allows tasks that get ready to run only after the currently being executed task entered the state *running* to claim the processor. In non-preemptive systems these tasks would have to wait at least until the running task quits thus probably missing their deadlines. In this new algorithm the tasks that would thresh in normal LLF-scheduling are executed one after another. The order of execution is determined by the tasks deadlines. Of all the tasks that have the smallest slack the one is chosen that has the earliest deadline. That task locks out the other tasks until it ends or until it is preempted. (Figure 4).

## 4. Performance Evaluation

The scheduling coprocessor design is described in VHDL thus guaranteeing technology independence. Two prototypes have been synthesized in field programmable gate arrays (FPGA). One of them uses normal LLF-scheduling while in the other one our new scheduling algorithm is implemented.

The latter needs nearly twice as much time as the least-laxity-first scheduling coprocessor for determination of the next task to run. This is caused by an extra comparison of deadlines that has to be done.

At current stage of development the coprocessor using the enhanced LLF-algorithm needs 1µs to deliver a result for a set of 32 tasks with a parameter resolution of 16 bit. As mentioned above the coprocessor using standard least-laxity-first algorithm is much faster. It finishes its computations in 0.53 µs.

The next steps in the development of the coprocessor are aimed on further improvement of design speed and size as well as on integration of coprocessor supported scheduling in standard real-time operating systems.

## 5. Conclusion

This paper has described a rapid prototyping system currently under development. The reconfigurable scheduling coprocessor developed in this context implements dynamic priority scheduling algorithms. This allows even computation intensive algorithms to be used in real systems. Next steps will concentrate on robust algorithms like [2, 8]. The final goal is to develop a framework that simplifies creation of scheduling coprocessors and shortens design cycle time.

## Bibliography

[1] N. Audsley et al., "Fixed Priority Pre-emptive Scheduling: An Historical Perspective," *Journal of Real-Time Systems*, vol. 8, pp. 173-198, 1995

[2] S. Baruah, J. R. Haritsa, "Scheduling for Overload in Real-Time Systems," *IEEE Transactions on Computers*, vol. 46, No. 9, pp. 1034-1039, 1996

[3] 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

[4] Z. Deng, J. W.-S. Liu and J. Sun, "Dynamic Scheduling of Hard Real-Time Applications in Open System Environment," *17th IEEE Real-Time Systems Symposium*, WIP-Proc., pp. 47-50, 1996

[5] M. H. Klein et al., A Practitioner’s Handbook for Real-Time Analysis. Carnegie Mellon University, Software Engineering Institute, Kluwer Academic Publishers, Boston, Dordrecht, London, 1993

[6] Johan Furunäs 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

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

[8] M. Spuri, G. Buttazzo, F. Sensini, "Robust Aperiodic Scheduling under Dynamic Priority Systems," *Proc. 16th IEEE Real-Time Systems Symposium*, Pisa, Italy, pp. 210-219, 1995

[9] J. A. Stankovic, "Real-Time and Embedded Systems," In: Allen B. Tucker (Ed.), CRC Computer Science and Engineering Handbook. CRC Press, pp. 1709-1724, 1997

[10] K. Tindell, "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks," Technical Report YCS-91-189, Department of CS, University of York, 1994

[11] B. K. Kim, K. G. Shin, "Scaleable Hardware EDF Scheduler for ATM Network," *Proc. 18th IEEE Real-Time Systems Symposium*, San Francisco, CA, 1997