Scalable Performance Scheduling for Hardware-Software Cosynthesis*

Thomas Benner, Rolf Ernst and Achim Österling
Institut für Datenverarbeitungsanlagen
Hans-Sommer-Str. 66
D 38106 Braunschweig, Germany

Abstract

The paper presents a static process scheduling approach as a front-end to hardware-software cosynthesis of small embedded systems which allows global system optimization. Unlike earlier approaches, scheduling is executed before hardware definition assuming scalable system performance. Scheduling supports process communication and external timing requirements. We explain the algorithm and give results using an example.

1 Introduction

Process scheduling is a well known problem in embedded system design. In current approaches, the hardware-software architecture is widely known at compile time. With the advent of hardware-software co-synthesis, hardware-software partitioning can be moved to a very late design stage (late binding) because changing system descriptions and generating and modifying hardware-software architectures has become much easier. This is very much like the automation of physical layout has reduced the cost of netlist changes and logic synthesis has simplified RT-level modifications. As a consequence, process scheduling could move ahead of hardware architecture definition. We will first explain why process scheduling before hardware definition is useful and will then show that this approach leads to a new kind of scheduling problem and finally give a scheduling approach and first results.

Many of the more complex embedded system applications consist of a mixture of processes with very different requirements. In the example of an automotive motor management, ignition control requires a simple process, which must be repeated every 10 us, while fuel injection and emission control are computation intensive processes with iteration rates in the range of many milliseconds. This is why we often find a mixture of hardwired logic, ASPPs (application specific programmable processors) and general purpose processor cores. Today, this architecture is typically defined before the system is implemented on this architecture, in most cases using preemptive scheduling. Global system optimization is difficult in this context and whatever changes occur during system design will hardly be able to influence the hardware architecture. Co-synthesis systems, such as VULCAN [5] or COSYMA [4], start with unified process system description and then derive a hardware architecture. In the current state, COSYMA takes a single process in a superset of C, C^x, together with timing constraints on this C^x process. COSYMA first tries to implement the process on a given processor. If the timing constraints are not met, COSYMA automatically creates a coprocessor for process acceleration. Co-synthesis can also be controlled by a required speedup factor for a given processor. Processor-coprocessor architectures are popular in embedded system design such as in the motor management example [9]. If we, now, would be able to map the set of processes of an embedded system to a single process, as in fig. 1, then we could use the co-synthesis system to create a new processor-coprocessor architecture, independent of the original process structure. This allows global system optimization. To give an example, it might be more efficient to speed up one or more small processes with a high iteration rate, or, instead, to speed up one or more computation intensive processes with loops. Partitioning would also share the coprocessor for several processes, such that the original process structure might differ from the optimized hardware architecture. This is a difficult and time consuming task if done manually.

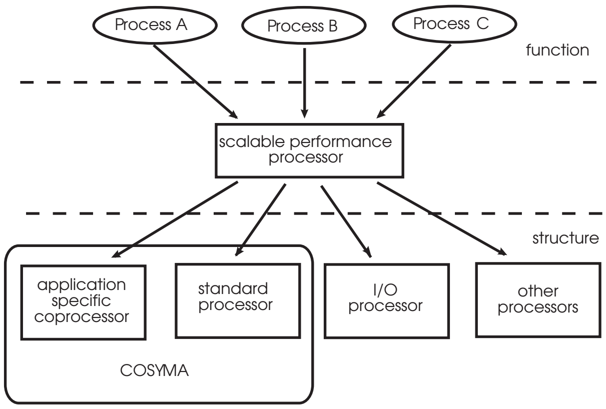

Figure 1: Mapping of the processes

In the next section, we will explain the requirements to scheduling which result from this approach, in section 3 we will inspect related work, section 4 will describe our approach, in section 5 we give results which we obtained with a complete system example, and finally, we draw some conclusions.

*This project is sponsored by the European Union under ESPRIT No. 8135.

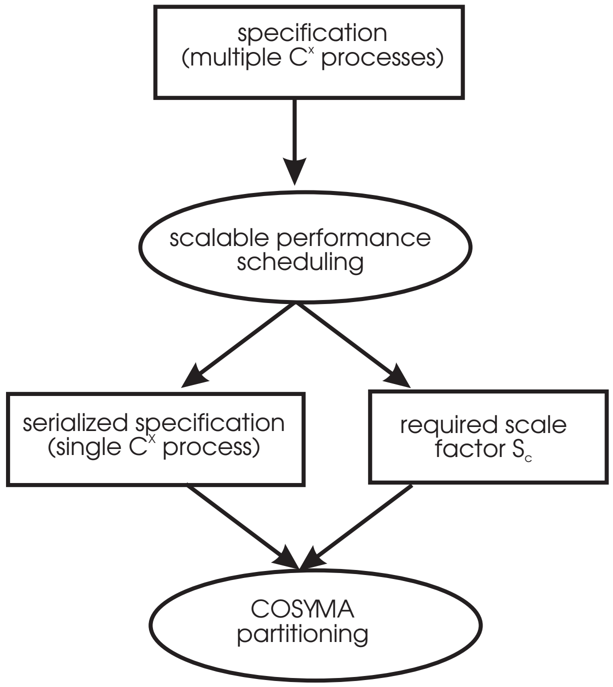

Figure 2: Coupling the scheduler with COSYMA.

2 Scheduling requirements

The major difference of the scheduling problem, as compared to other process scheduling problems is that the scheduler works on an architecture with scalable performance. Given a process system with time constraints, the scheduler cannot simply anticipate the execution time of the individual processes or process segments, because acceleration is determined by the co-synthesis system. Co-synthesis can, however, regard I/O time constraints.

To alleviate the scheduling problem, we assume that hard I/O-timing constraints are buffered by a peripheral device and that processes with cycle times of less than a microsecond are moved to hardwired logic upfront. In a manual design, such decisions would be reasonable, as well.

The *scalable performance* is defined *relative to the performance of a given processor* which shall be used as core processor in the hardware-software system. To express the required performance, a performance scaling factor is defined,

$$S_c = \frac{required\ system\ performance}{given\ processor\ performance},$$

such that S_c can be used as *speedup factor for co-synthesis* (fig. 2).

Communication with buffering increases the solution space of scheduling and co-synthesis and is therefore included. Buffering does not change rate constraints and it must regard I/O-constraints, but it allows to reschedule process initiation times as well as communication among processes. We define: An I/O-constraint (i.e. an event relating to I/O-communication) is scalable when buffering is permitted for at least one scheduling period. All other I/O-constraints are unscalable. Inter-process communication constraints are always scalable.

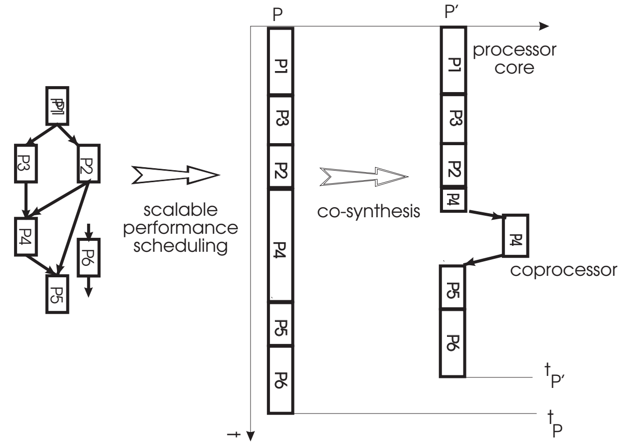

Figure 3: Speedup of a macro period.

S_c is a global speedup factor that does not require that COSYMA is able to speed up the execution time of *each individual* process. The partitioning is performed on the serialized process, P, enabling a true global optimization that is not restricted by the boundaries of the original processes. Assuming that the communication between the processes can be buffered for at most one macro scheduling period it does not matter in which part of the code the speedup can be reached. In figure 3, six processes with two different rates are serialized in one single macro process. After that, COSYMA would choose a part of the macro process for a hardware implementation that offers the greatest capacity for speedup. In fig. 3, we assume that this is a part of P_4. As a result, the execution time is shortened to t'_p. This way, the *average* iteration rate of each individual process has been reduced to the required rate, but the execution within a macro period is not, which is why buffering is necessary now.

3 Related work

Scheduling has been an important area of research in real-time computer systems and in signal processing. Many of the publications are focused on scheduling of periodic processes. Liu and Layland [8] have shown that *Rate Monotonic Scheduling* (RMS) produces an optimal schedule, but it does not consider process communication. Therefore, several extensions have been suggested [13], [10], [11]. All these approaches are fixed priority assignment algorithms meaning that the order of the processes is determined at run time depending on the priorities which are assigned a priori.

In order to serialize a system of communicating processes as a single process, we need a static order of the processes which is decided at design time. A (mostly) static scheduling approach is given by [7] and [12].

Chou and Boriello present in [1] a static, non-preemptive scheduling algorithm for reactive real time systems, which is implemented as a part of the Chinook framework. By serializing a process dependency graph both interprocess and intraprocess constraints can be regarded. In the Chinook framework partitioning and scheduling is an iterative process starting with partitioning [2]. This approach is especially efficient for interface synthesis, but less suited to data dominated parts with (possibly data dependent) loops because it requires loop unrolling. The same holds for Gupta's approach [6]. Both approaches do not consider communication buffering.

All these scheduling approaches depend on a fixed target architecture with known performance. In this paper we present a completely different solution, which allows a serialization of parallel processes with only partial knowledge of the final target architecture.

4 Scalable performance scheduling

For scheduling we define different classes of processes [3]. The major differences are the constraint types and the timing requirements. The scheduling of the process classes is separate and hierarchical. We omit details here, in order to focus on the most challenging class of periodic tasks with a wide variation of process execution times and iteration rates.

At the beginning of the scheduling process (fig. 4), a process dependency graph (PDG) is derived from the C^x description. The processes are split into basic blocks in order to get short time segments, which allow to schedule processes with a shorter cycle time in the order of a few microseconds. Each basic block also ends at a blocking communication or at a label which corresponds to an intertask constraint. Each node in the PDG represents a code segment and is attributed by the index of the code segment as well as its execution time on the target processor, which is estimated by prescheduling and simulation.

In a prescheduling step, the processes are first serialized such that the order of the processes does not violate the data and communication dependencies but without regarding the external time or rate constraints. Being executed with external stimuli data, the ordered segments show the correct function and timing when executed on a single target processor simulator. The prescheduling step saves an extra simulator. The sequential process provided by prescheduling is functionally correct, because all dependencies were considered, but the timing constraints are not regarded. Therefore a different scheduling is required, now.

For both, the prescheduling and the final scheduling we adapt the algorithm of Chou [1] based on the traversal search through a graph. The algorithm originally produces *one* valid schedule if possible, but this is not necessarily the optimal one. A suboptimal schedule results in a worse processor utilization caused by idle times. This is not a problem in the case of a fixed target architecture, because, there, processor utilization is not the primary concern. Since the target architecture generated by COSYMA depends on the scheduling, a bad processor utilization results in an overdimensioned target architecture. That is the reason why a schedule close to the optimum is required, here

Neglecting the dependencies of the processes, the "non scaled" utilization U_{ns} is the sum of the loads of each process on the processor, which is the ratio of the execution time T_i to the cycle time C_i of process i:

$$U_{ns} = \sum_{i=1}^{n} \frac{T_i}{C_i}.$$

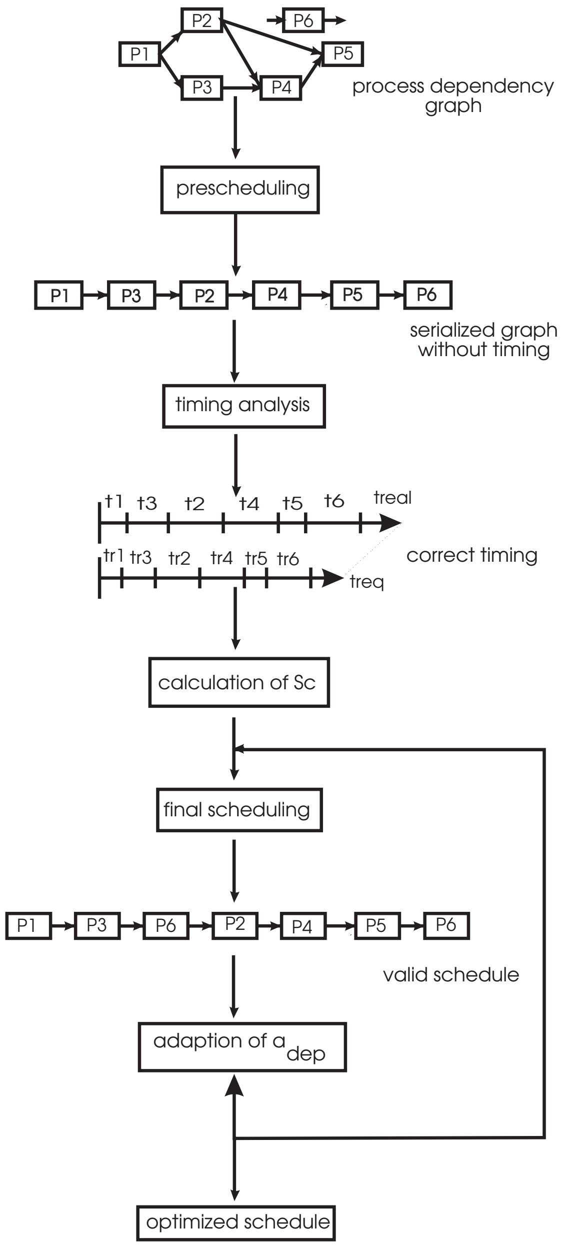

Figure 4: The design flow.

The utilization is a measure of the ratio of the busy times to the idle times of a processor. The best of all valid schedules is the one which is closest to the value 1. A valid schedule can only be found, if this value is smaller or equal one. Liu and Layland proved, that a set of n processes are always schedulable for $U <= U(n) = n * (2^{\frac{1}{n}} - 1)$. For the interval between U(n) and 1 they gave also criteria which guarantees a valid schedule [8].

In a hardware-software cosynthesis environment, a typical application can not be implemented on a single core processor. The utilization of the processor becomes greater than 1. Therefore we introduced the scaling factor S_c, in order to bring the processor utilization in the valid range.

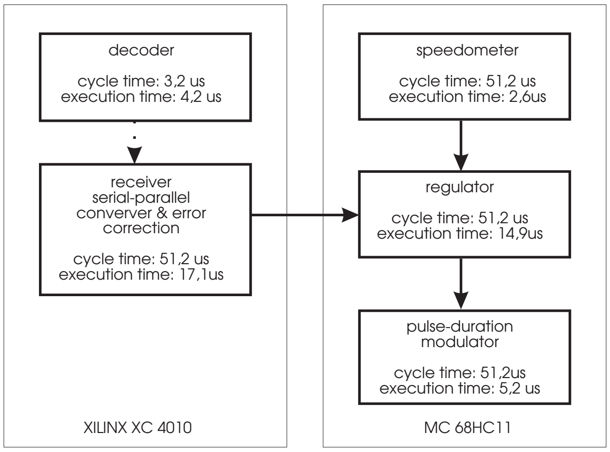

Figure 5: Block diagram of the train control.

The execution time of each process is normalized by S_c. In addition to the processor utilization, S_c depends on the task dependencies, the constraints, the communication and the overhead for context switching. We merge all these components to a factor a_{dep} to determine the scaling factor:

$$S_c = a_{dep} * U_{ns} = a_{dep} * \sum_{i=1}^{n} \frac{T_i}{C_i}.$$

Because a_{dep} depends on the number of context switches and the order of the processes which are unknown before scheduling, it cannot easily be estimated, but is determined heuristically. Scheduling iterates several times and adapt a_{dep} and S_c after each iteration. For the adaption we use simple binary search[1], starting with a heuristic value of 5. If this is successful, the interval between one and five is traversed, otherwise the one between five an ten. By choosing the width between the first invalid and the last valid schedule, the quality of the solution can be controlled.

As a result of the scheduling, we get a serialized PDG and an S_c which is close to the optimum. The cosynthesis system now tries to reach the required speedup S_c by generating an application specific coprocessor.

5 An example

A model train control serves as an example. The train is controlled by a personal computer, from which 35 bit values are transferred over the rails. The model train has a rather powerful motion speed regulation that moves the train close to original large trains. Figure 5 shows the block diagram. A decoder scans the pulse duration on the rails with a rate of 313 kHz. Due to serious noise, a low pass filter and a 3 bit burst error correction code are used. The corrected and decoded value is passed to a regulator, that receives the actual velocity from a speedometer. Finally a motor controller generates pulses for the motor electronics.

The train control serves as a demonstrator for a student VHDL course. The current prototype consists of a XILINX XC 4010 and a microcontroller MC 68HC11. The bit decoder and the receiver including the error correction are mapped on the FPGA, whereas the regulator, the speedometer and the motor control are implemented on a microcontroller. With the scheduling experiment, we want to evaluate if the system could be implemented on a 33 MHz SPARC processor with little additional hardware.

The C^x description consists of five different processes. The edges between the processes represent communication. The dotted line describes a non-blocking, whereas the others describe blocking communication. The first process is the decoder that has a cycle time of 3,2us and an execution time of 4,2us. This process alone could not be implemented on a SPARC, but a speedup of 1,3 would be required. All the other processes are executed with a cycle time of 51.2us. The processor utilization is the sum of the quotients of the execution time and cycle time of each process (non scaled) :

$$\begin{aligned} U_{ns} &= \sum_{i=1}^{n} \frac{T_i}{C_i} = \frac{4,2us}{3,2us} + \frac{17,1us}{51,2us} + \frac{2,6us}{51,2us} \\ &+ \frac{14,9us}{51,2us} + \frac{5,2us}{51,2us} = 2,09. \end{aligned}$$

Without communication a speedup of 2,09 would be necessary in order to run the example on a SPARC processor.

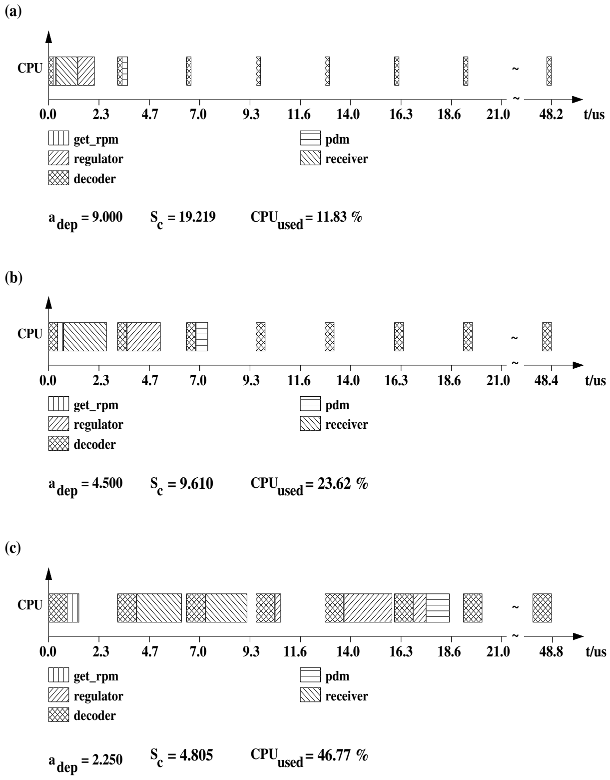

Figure 6: Adaptation of the scaling factor.

In order to consider the communication the utilization

[1] Binary search has been useful for other cosynthesis approaches, as well [14].

is multiplied by the heuristic factor a_{dep}. This factor is adapted in an iterative process starting with a heuristic value of 9,0. Figure 6 shows the Gantt diagrams of three valid schedules during the iterative process. A value of 9,0 (fig. 6a) for a_{dep} leads to a scaling factor of 19,2. The processes are split into basic blocks. One block in the diagram represents the activation of at least one basic block of a process. The *decoder* process is activated every 3,2us. All the other processes are active only at the beginning of the period. In the interval from 6us until 52us only the decoder task is active. This leads to a scaled processor utilization of $U_s = 11,83$ % (CPU_{used} in Fig.6).

In fig. 6b a_{dep} is reduced to 4,5. This adaption influences the order of the processes, the regulator activation is moved behind the second run of the decoder. The processor utilization increases to 23,6 %. As a result, the schedule becomes more compact.

One more valid schedule is found for a_{dep}=2,25. After the next iteration a_{dep} is set to 1,25 which does not result in a valid schedule. Neither for $a_{dep} = 1,69$ nor for $a_{dep} = 1,97$ a valid schedule is found. Since the difference of a_{dep} of the last valid schedule (2,225) and the current a_{dep} is less than the default minimal interval width (0,5) the scheduler stops returning the best $a_{dep} = 2,25$.

As a consequence a speedup of 4,8 is required in order to implement the design on the SPARC. Most of this speedup is required to implement the decoder as seen in the Gantt diagram. The PDG of the train control example consists of about 770 nodes. The whole scheduling algorithm takes only a few minutes on a SPARC 10, if the lowest interval width of the binary search is set to 0.5.

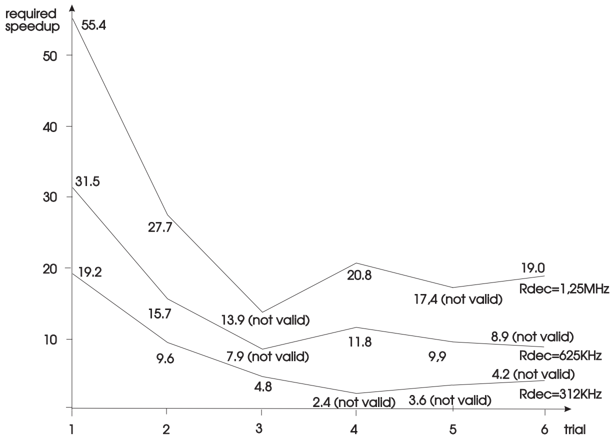

Figure 7: Convergence of the binary search algorithm.

In the following, we assume that the rate of the the bit decoder ranges from 312 kHz to 1,25 MHz depending on the transfer rate between the PC and the train. Fig. 7 shows the trials of the binary search to find a valid schedule close to the optimum for three different rates. Each of them converges to the optimum quite quickly. For this example the binary search terminates whenever the distance of factor *a* between a valid and an invalid schedule is lower than 0.5.

6 Conclusion

We presented a scalable performance scheduling algorithm for small embedded systems. Scheduling goal is a minimum performance requirement for a given set of communicating processes with rate constraints. All periodic processes are mapped to a single process which is then accelerated to the required performance using hardware-software cosynthesis. This allows global optimization, because cosynthesis is not limited to the original process structure.

Unlike earlier process scheduling problems, hardware performance is not fixed at scheduling time. Hardware performance, however, influences process ordering in the schedule and, on the other hand, process ordering decides on the required performance. Iteration with binary search is used to adapt both of these parameters. Based on a relatively fine grain process segmentation, scheduling can efficiently handle processes with very different execution times and iteration rates. The results with an example show high processor utilization and acceptable computation times.

Scalable performance scheduling can be used for other problems, as well. Instead of controlling cosynthesis, the performance scaling factor could be used for core processor selection, if the relative performance of other processors is known. This, of course, is not useful for arbitrarily different processor architectures where scaling is process dependent. A designer could also use the Gantt diagram to identify inefficiencies when serializing the process system to a valid schedule. Identified inefficiencies can result from the definition of timing requirements, rates, process communication and required context switching.

The hardware architecture is still rather simple. As future work, we will extend the scheduling approach and cosynthesis to small, heterogeneous multiprocessors.

References

[1] P. Chou, G. Borriello, *Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems*, 31st Design Automation Conference, San Diego, CA, June 1994.

[2] P. Chou, E.A. Walkup, G. Borriello, *Scheduling for Reactive Real-Time Systems,* IEEE Micro, pp. 37-47, August 1994.

[3] R. Ernst, Th. Benner, *Communication, Constraints and User Directives in COSYMA*, Technical Report CSY-94-2, Institut für Datenverarbeitungsanlagen, Technical University of Braunschweig, 1994.

[4] R. Ernst, J. Henkel and Th. Benner, *Hardware-Software Co-Synthesis for Microcontrollers*, IEEE Design & Test of Computers, pp. 64–75, Dec. 1993.

[5] R.K. Gupta, G. De Micheli, *System-Level Synthesis Using Re-programmable Components,* Proc. of EDAC'92, pp. 2-7, 1992.

[6] R.K. Gupta, G. De Micheli, *Constrained Software Generation for Hardware-Software Systems,* Third Int'l Workshop on Hardware/Software Codesign, Grenoble, Sep. 22-24, 1994.

[7] E.A. Lee, G.G. Messerschmitt, *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*, IEEE, Trans. on Computers, Jan 1987.

[8] C. Liu, J. Layland, *Scheduling algorithm for multiprogramming in a hard-real-time environment*, Journal of the ACM, 20, pp. 46-61, 1973.

[9] Motorola Inc., MC68332 User's Manual, Phoenix, Arizona,1990.

[10] L. Sha, J.B. Goodenough, *Real-Time Scheduling Theory in Ada*, IEEE Computer, April 90, S.53-62.

[11] L. Sha, R. Rajkumar, J. P. Lehoczky, *Priority Inheritance Protocols: An Approcah to Real-Time Synchronization*, IEEE Trans. on Computers, Vol. 39, No. 9, Sept. 1990.

[12] G. C. Sih, E. A. Lee, *A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures*, IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 2, Febr. 1993.

[13] T. Teixeira, *Static priority interrupt scheduling*, Texas Conference on Computing Systems, pp. 5.13-5.18, Nov. 1978.

[14] F. Vahid, J. Gong, D.D. Gajski, A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning, Proc. of EURO-DAC 94, pp. 214-219.

[15] W. Ye, R. Ernst, Th. Benner, J. Henkel, *Fast Timing Analysis for Hardware-Software Co-Synthesis*, Proc. of ICCD 1993, Cambridge, pp. 452-457, 1993.