# An Operating System for a Time-Predictable Computing Node

Guenter Khyo, Peter Puschner, and Martin Delvai

Vienna University of Technology
Institute of Computer Enginering
A1040 Vienna, Austria
peter@vmars.tuwien.ac.at,
http://ti.tuwien.ac.at

**Abstract.** The increasing complexity of speed-up mechanisms found in modern computer architectures makes it difficult to predict the timing of the software that runs on this hardware, especially when the software itself has many different execution paths. To fight this combined hardware-software complexity that makes an accurate timing analysis infeasible, we have conceived a very simple software structure for real-time tasks: We do not allow that decisions about the control flow are made at runtime, i.e., all decisions are resolved in an off-line analysis before runtime.

In this paper we show that simple control structures generated before runtime can as well be used within the operating system of an embedded real-time system. In this way we make not only task timing but also the timing of the operating system and thus the timing of the entire real-time computer system fully deterministic, thus time-predictable. We explain the principles and mechanisms we use to achieve this predictability and show the results of an experiment that demonstrates the feasibility of our concepts.

**Keywords:** Real-Time Operating Systems, Time-Triggered Architecture, Determinism, Temporal Predictability.

## 1 Introduction

Real-time systems need to provide timing guarantees in order to provide safe operation. In practice, the problem in achieving these guarantees is that hardware and software systems have become overly complex. It is therefore difficult to design time-predictable systems and provide evidence that a constructed system really meets the demanded timing properties. The reason for this complex behavior, which is hard to analyze, is mainly the fact that (a) dynamic decisions are taken at runtime, and that (b) the effects of a possibly long execution history have to be considered when argueing about the system state at a particular time instant.

The goal of our work is to develop an architecture that provides predictable and repeatable timing. The focus of this paper is on the operating system. We

U. Brinkschulte, T. Givargis, and S. Russo (Eds.): SEUS 2008, LNCS 5287, pp. 150–161, 2008.

describe the architecture and operation of an operating system that provides completely deterministic, reproducable timing and can thus be the basis for constructing entire applications with predictable timing.

The operating system shall serve as a basis for building real-time systems for which the timing of every operation can be predicted with an accuracy of a single CPU clock cycle. We will show how we can achieve this goal by using an adequate task-execution model – the single-path model – and a time-triggered, table driven control approach in the operating system, together with a realization of all kernel and operating-system routines in single-path code.

The interface of the proposed architecture is a time-triggered state message interface that blocks all asynchronous external control signals (Section 2). The propose software architecture builds on a simple task model and time-triggered, table-driven scheduling (Section 3.1) as well as on a deterministic code execution scheme (single-path code) in applications and in the operating-system code (Section 4). Within the paper we will focus on the description of the operating system that manages the resources of the system in a pre-planned manner. We will demonstrate that it is possible to get along with an OS implementation that uses a combination of single-path code and a static parameterization, i.e., the parameters of all calls to the operating system can be evaluated before runtime. This design yields a timing that can be fully predicted in a pre-runtime timing analysis (Section 5). The predictability will finally be demonstrated by a simple evaluation of a prototype implementation of the operating system (Section 6).

## 2 The Safety-Critical Subsystem Interface

In this section we describe the interface of the proposed architecture. As a prerequisite for building a time-predictable (sub)system, the interface of this system has to be predictable as well.

The following description uses the model and terminology of the DECOS integrated architecture [1]. This does, however, not mean that our work is only useful in the context of DECOS. On the contrary, the architecture can be adopted to any architecture that provides a time-triggered state message interface.

### 2.1 The Connector Unit

A DECOS component consists of two separated subsystems, the safety-critical subsystem for executing all safety-critical tasks and the non safety-critical subsystem for performing all other, non-critical services. Both types of subsystems are connected to the rest of the distributed computer system via so-called connector units. The connector units realize the architectural services of the distributed architecture, comprising the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures [1].

Within this paper our focus is on the operating system of the safety-critical subsystem of a component (see Figure 1). This is where time predictability is

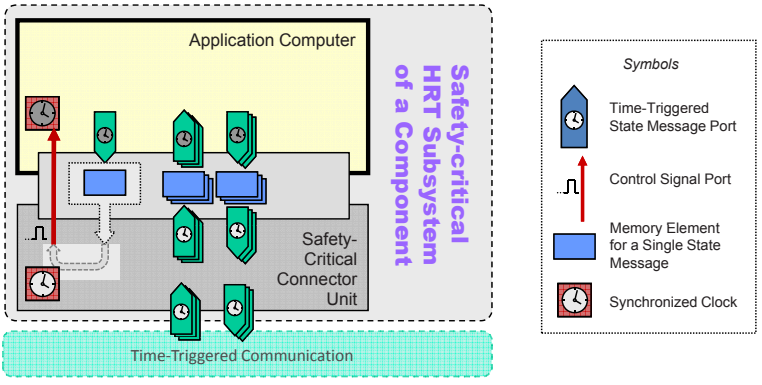


**Fig. 1.** Interfacing between the safety-critical hard real-time subsystem of a component and the time-triggered communication channel

needed. The application computer of this subsystem communicates with its environment solely via the safety-critical connector unit. The connector unit provides the following services in support of the time-predictable software architecture of the application computer.

- The connector unit implements a temporal firewall interface [2] for all data elements exchanged between the application computer and the communication subsystem. The read and write operations of the communication subsystem access the memory elements of the temporal firewalls only at predefined times, according to the a-priory known time-triggered communication schedule (in Figure 1 small rectangles represent the memory elements and the arrows marked with light clocks show the accesses of the communication subsystem to the firewalls).
- The time windows during which the communication system accesses the memory elements of the connector unit are known for each temporal firewall.
- The communication system provides a time-signal service to the application computer. A dedicated memory element in the connector unit can be written to set the timer (Figure 1, left). When the global system time reaches the timer value, the connector unit sends an interrupt signal over the signal port to the application processor.

## 3 A Time-Predictable Application Computer

The timing of the actions performed by a computer system depends on both, the software running on the computer and the properties of the hardware executing the software [3]. We therefore list the hardware and software features that in combination allow us to make an application computer time-predictable.

### 3.1 Hardware Architecture

A central idea of our approach is to obtain time predictability by constructing software that has an invariable control flow. By applying this restrictive software model we can allow for the use of hardware features that are otherwise considered as being "unpredictable" (e.g., instruction caches) and yet build systems whose timing is invariable. So the idea is to keep hardware restrictions and modifications within limits (e.g., we restrict caches to direct-mapped caches but do not demand special hardware modifications as, for example, needed for the SMART cache [4]). To support our execution model, the following hardware properties have to be fulfilled:

- The execution times of instructions do not depend on operand values.
- The CPU supports a conditional move instruction or a set of predicated instructions that have invariable execution times.
- Instruction caches are either direct mapped or set-associative with LRU replacement strategy.
- Memory access times for data are invariable for all data items. (This is the strongest limitation. We will try to relax it in future work).
- The CPU has a programmable instruction counter that can generate an interrupt when a given number of instructions has been completed.

### 3.2 The Software Architecture

To construct a time-predictable computer system we need to be very strict about the software structure. The proposed software architecture does not allow for any decisions in the control flow whose outcome has not already been determined before the start of the system. This property is true for both the application tasks and the operating system. Even task preemptions are implemented in a way that does not allow for any timing variation between different task invocations.

**Task Model.** The structure of all tasks follows the simple-task model (S-task model) found in [5]. Tasks never have to wait for the completion of an input/output operation and do never block. There are no statements for explicit input/output or synchronization within a task. It is assumed that the static schedule of application tasks and kernel routines ensures that all inputs for a task are available when the task starts and that outputs are ready in the output variables when the task completes. The actual data transfers for input and output are under control of the operating system and are scheduled before respectively after the task execution.

An important and unique property of our task model is that all tasks have only a single possible execution path. By translating the code of all real-time tasks into single-path code we ensure that all tasks follow the only possible, predetermined control flow during execution and have invariable timing. For more details about the single-path translation see Section 4.

**Operating System Structure.** If not properly designed, the activities of the operating system can create a lot of indeterminism in the timing of a computer

system. We have therefore been very restrictive in the design of the operating system and its mechanisms (see Section 5).

Predictability in the code execution of the operating system is achieved by two mechanisms. First, single-path coding is used wherever possible. Second, all data that are relevant for run-time decisions of the operating system are computed at compile time. These data include the pre-determined times for I/O, task communication, task activation, and task switching. They are stored in static decision tables that the operating system interprets at runtime.

Task communcation and I/O is implemented by simple read and write operations to specific memory locations. As these memory accesses are pre-scheduled together with the application tasks, no synchronization and no waiting is necessary at run time.

The programmable time interrupt provided by the communication system is used to synchronize the operation of the application computer with the global time base. This way we ensure that the application computer performs all activities in synchrony with its environment, i.e., the rest of the system.

### 3.3 Tool Support

The software structure of our architecture is very specialized. Code generation for an application therefore needs to be supported by a number of tools:

- To generate single-path code, either a special compiler or a code conversion tool that converts branching code into single-path code is needed.
- A tool for worst-case execution-time analysis returns the execution times of the tasks and the operating system routines.
- An off-line scheduler generates the tables that control all operations of the application computer. The scheduler has to resolve all precedence and mutual exclusion constraints between task pairs as well as tasks and the communication system. It further has to plan all preemptions, thereby taking into account the effects of the preemptions on the system timing.

## 4 Deterministic Single-Path Task Execution

As all branches in the control flow of some code may cause variable timing, we translate the code of all tasks as well as the operating-system code into single-path code [6]. The code resulting from the single-path translation has only a single execution trace, hence the name single-path translation.

The strategy of the single-path translation is to remove input-data dependencies from the control flow. To achieve this, the translation replaces all input-data dependent branching operations by predicated code. It serializes the input-dependent alternatives of the code and uses predicates (instead of branches) and, if necessary, speculative execution to select the right code to be executed at runtime. All loops with an input-data dependent termination condition are transformed into loops with a constant number of iterations. The termination condition of the original loop is transformed into a condition that

occurs in the body of the new loop and makes the loop body execute conditionally, thus simulating the semantics of the original loop. More information about the conversion can be found in [7].

## 5 The Time-Predictable Operating System

The operating system has to be carefully designed in order to achieve predictability at the instruction level of the CPU. There must be no jitter in the execution times of the operating system routines.

### 5.1 Kernel Design

One of the key design decisions was to follow a microkernel-based design. From the perspective of timing analysis, microkernels have two significant advantages: First, microkernels are very small in code size and therefore, timing analysis gets much easier. More importantly, however, most of the activities of the operating system can be controlled at the task level, and thus, the static schedule and the progress of time determine the actions of the OS.

All components of the OS, i.e., the microkernel and the system tasks, are written in single-path code. The if-conversion is the most frequently applied rule to the code, followed by the rule for loop conversions.

### 5.2 Communication

Communication is an integral part of every operating system. In our OS, we use a simple model of interprocess communication which S-Tasks may use when they need to communicate with other tasks or the real-time environment.

**Interprocess Communication.** Figure 2 illustrates our model of interprocess communication. This model has been adopted from [8].

Every task has access to a local buffer in which messages are stored (1). These messages will be copied by a privileged task (the IPC task) into a global buffer called the message base upon its activation(2). All tasks have read access to the message base. The IPC task further has access to the message schedule. This schedule contains an entry for each message that has to be processed and copied to the message base. Messages are always broadcast within a node and written to a specific address in the message base. This address is determined offline and listed in the message schedule. If specified in the schedule, a message can also be sent to another node over the network. In this case, the IPC task accesses the temporal firewall and writes the message into the message buffer of the temporal firewall (see Section 2). The main advantage of this model is temporal transparency, as the point in time when a message is being processed is solely determined by the message schedule. This schedule is calculated offline.

**System Calls.** Most operating systems provide a set of system calls to enable tasks to communicate with the kernel. Because interprocess communication

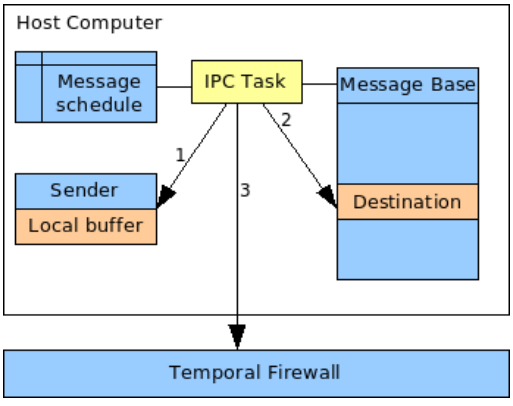


**Fig. 2.** Interprocess communication

depends on the corresponding IPC task (and thus, has to be scheduled), the scheduler makes use of system calls to communicate with the kernel. As explained in Section 5.3, the scheduler informs the dispatcher about the tasks that have to be executed according to the off-line calculated schedule. The scheduler also requests the reset of the global timer to an offline-calculated, specific value. All other tasks have to use the underlying IPC model and do not need to communicate with the kernel.

### 5.3 Scheduling and Mode Switches

In a time-triggered RTOS, the schedule is determined offline. The scheduler is invoked at each global clock tick. After invocation, the scheduler interpets the scheduling table and tells the dispatcher which tasks have to be executed. To avoid blocking, these tasks are not invoked immediatly, instead the dispatcher "waits" until the scheduler has finished its current execution before executing all other tasks.

Any task may request a *mode switch*. We define the term mode switch as the transition from the current schedule to the requested schedule. A mode-switch request is encoded as a special message and written to a special location in the message base. Upon invocation, the scheduler and the IPC task read from this location and switch to the requested tables. The point in time when a mode-switch can be carried out is determined offline and implicitly defined by the scheduling and message tables.

### 5.4 Implementation of S-Tasks and Task Preemption

For each task, the programmer has to define two procedures. One procedure defines the initialization semantics of the task. This procedure is not constrained to timing bounds; it may take an arbitrary, non-defined amount of time for the

procedure to finish. Memory for all tasks is allocated during the initialization phase. When all tasks have been initialized, the operating system switches to real-time mode. In real-time mode, when a task is activated, its real-time procedure will be called. The real-time procedure must be written in single-path code, i.e., it must have constant timing. The operating system will detect a deviation of the specified timing of a task within an uncertainty of one clock cycle.

### 5.5 Temporal Characteristics

For the real-time application designer and the planning tool, knowledge of the execution times of the core components of the OS is essential. Table 1 lists the timing of the various components of the OS which are relevant to timing analysis.

**Table 1.** Timing of the OS

| Functionality | Component | Timing in CPU cycles |
|---|---|---|
| Scheduling | Scheduler | $666+Max(Tasks)\times 621$ |
| Context switch | Dispatcher | 253 |
| Task execution | Kernel/Scheduler | 1626 |
| IPC | IPC task | $349+M_i \times 199 + 10 \times M_i \times Size(M_i)$ |

Most parts of the operating system have constant timing. However, components like the scheduler and IPC task use bounded counting loops. Therefore, the entire timing behaviour of the OS is dependend on three parameters: (a) The maximum number of tasks that can be scheduled per timeslot, $Max(Tasks)$, (b) (for the moment) the maximum size of a message, $Size(M_i)$, and the number of messages that have to be copied to the message base, $M_i$, where $i$ refers to the corresponding index in the messaging table. All these parameters are determined offline and are stored in the scheduling and message tables.

## 6 Experiments

In order to show the feasibility of the presented approach we implemented a prototype operating system running on Spear2, a micro-controller developed at our department. In this section we first describe the hardware platform. Subsequently we explain two experiments performed on it. The first experiment shows that the timing of the OS is stable and free from execution-time jitter. The second experiment illustrates the impact of traditional code (i.e., branching code) on the timing of the OS.

### 6.1 Hardware Platform

Our platform consists of the soft processor core, Spear2 *(Scalable Processor for Embedded Applications in Real-time environments 2)* and associated tools. Spear2 constitutes a harvard architecture with separated data and instruction

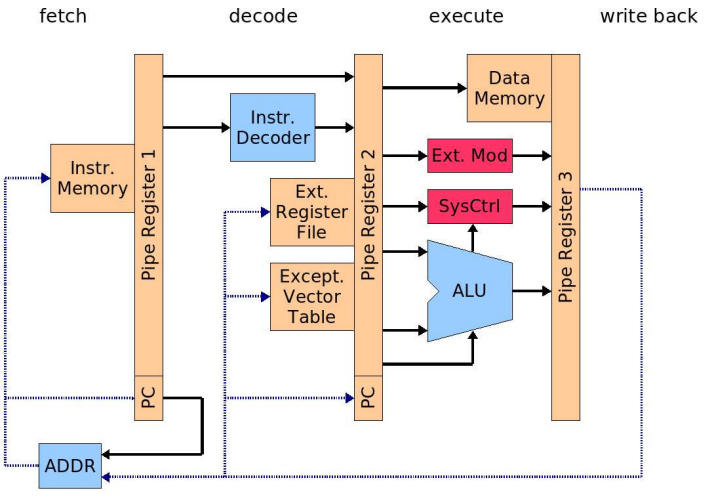


**Fig. 3.** Spear2 processor core

memories. While the instruction memory is always 16 bit wide, the data memory can be configured (16 or 32 bit). The register file comprises 16 registers, whereas two registers are reserved for storing the return addresses for subroutines (JSR and ISR). The instruction set consists of 119 instructions, most of them are predicated. All instructions are executed within one single clock cycle. Currently, there is no instruction caching.

However, tasks must not be preempted when they execute a jump instruction and trigger a pipeline flush because this could cause timing glitches.

Spear2 can be customized by mapping so called extension modules into the data memory. These modules can be accessed by ordinary load/store instructions. An extension module itself is an application specific hardware module which has a well defined interface: It consists of 32 eight-bit wide registers. The two first registers provide module status information while registers three and four are reserved for passing configuration parameters to the module. The remaining 28 registers can be used for data exchange. The position of the extension module register inside the data memory is defined by the so called *base address*. For instance, the instruction/clock cycle counter required to realize our operating system was implemented within an extension module.

A further interface – an AMBA interface – for the Spear2 processor core is under development. In this way Spear2 can be equipped with wide range of AMBA IP cores, among others a cache control. In the future, this will allow us to extend our investigations to more complex hardware structures.

Software is developed in C, using a C compiler based on gcc (*spear32-gcc*). The compiler creates either binaries for execution by SPEAR or a text file containing

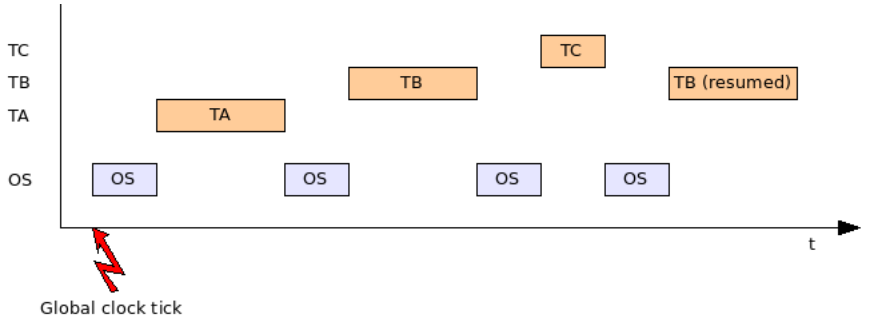


**Fig. 4.** Simple schedule of three S-Tasks

assembler code. The assembler code can be fed into a simulator that serves as debugging environment.

### 6.2 A Simple Application

Figure 1 depicts a simple schedule consisting of three S-tasks. This schedule is executed periodically, i.e., after task $T_B$ finishes, task $T_A$will be executed again, then task $T_B$, and so on.

The following roles are assigned to the tasks: (a) $T_A$ iteratively generates all permutations of a fixed-size array consisting of the numbers 1 to 5. The computed permutation is transferred to the message base by the IPC task. As task $T_B$ sorts the array, it is interrupted by a high priority, dummy task $T_C$. After the completion of task $T_C$, $T_B$ resumes and sorts the remaining part of the array.

### 6.3 Timing Measurements and Test Results

The goal of the experiments was to verify that for every execution cycle of the schedule, the timing behaviour is stable and identical to all other cycles. We used a logic analyzer to keep track of the program counter and the progress of time. We set the timing parameters as follows: $Size(M_i) = 8$ and $M_i$=1 for the IPC task and $Max(Tasks) = 8$ for the scheduler (for testing purposes we set the values of the parameters higher than necessary). For every task ($T_A$, $T_C$ and $T_B$, Scheduler and Kernel) we measured the execution time in CPU cycles.

We took approximately 280 measurements using different input data sets and all samples showed exactly the same temporal behaviour. Table 2 lists the results we obtained with the logic analyzer.

For each task $T$, the time it takes the kernel to activate $T$ is exactly 1626 CPU cycles. The only exception is the scheduler which is always activated after 1341 cycles. The activation times for all tasks are identical for every execution cycle. Note that after the (partial) execution of each task, 43 cycles have to be added. These 43 cycles are consumed by the kernel which checks the timing of the corresponding task.

**Table 2.** Test results of all samples

| Task | Time of activation | Execution time | Termination |
|---|---|---|---|
| Kernel | 0 | 1341 | 1341 |
| Scheduler | 1341 | 5634 | 6975 |
| Kernel | 6975 | 1626(+43) | 8644 |
| $T_A$ | 8644 | 611 | 9255 |
| Kernel | 9255 | 1626(+43) | 10924 |
| IPC | 10924 | 628 | 11552 |
| Kernel | 11552 | 1626(+43) | 13221 |
| $T_B$ | 13221 | 1000 | 14221 |
| Kernel | 14221 | 1626(+43) | 15890 |
| $T_C$ | 15980 | 37 | 15927 |
| Kernel | 15927 | 1626(+43) | 17596 |
| $T_B(resumed)$ | 17596 | 1771 | 19369 |

Because the OS and the tasks are written in single-path code, we implicitly tested all possible execution scenarios (with the exception of those that involve the change of the timing parameters, i.e., the modification of loop bounds). Therefore, we can conclude that the timing is indeed stable and predictable.

**The Impact of Traditional Code on the Timing Behaviour.** An interesting question is how severe the implications of traditional code are on the timing behaviour of the OS. After all, in a time-triggered system, decision are taken offline. So one might hastily conclude that the execution-time jitter is neglectable. Therefore, we modified the kernel by transforming some parts of the interrupt service routine and the dispatcher back to the original (branching) code. By performing the same type of measurements again the kernel showed a variation of 52 clock cycles, which equals a jitter of 3.5%. Clearly, if we had modified more parts of the OS, the results would have been even more dramatic.

## 7 Summary and Conclusion

In this paper we showed that the realization of a time-deterministic operating system is feasible. The software of the operating system uses single-path code together with a parameterization of OS tasks that is based on a pre-runtime analysis to avoid that run-time decisions influence the run-time behavior of the operating system. The operating system synchronizes its time base with the environment by means with a programmable clock interrupt that the design system of an application guarantees to arrive at a time instant when no other system or task activity is in progress.

Experiments on a prototype implementation have demonstrated the feasibility of our approach: The task execution times and the execution times of the kernel routines and operating system tasks are invariable and can be predicted with the accuracy of a single CPU clock cycle.

**Acknowledgments.** The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 214373.

## References

1. Obermaisser, R., Peti, P., Kopetz, H.: Virtual Networks in an Integrated Time-Triggered Architecture. In: Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, Feburary 2005, pp. 241–253 (2005)
2. Kopetz, H., Nossal, R.: Temporal Firewalls in Large Distributed Real-Time Systems. In: Proc. 6th IEEE Workshop on Future Trends of Distributed Computing Systems, October 1997, pp. 310–315 (1997)
3. Puschner, P., Burns, A.: A review of worst-case execution-time analysis. Journal of Real-Time Systems 18(2/3), 115–128 (2000)
4. Kirk, D.B.: Smart (strategic memory allocation for real-time) cache design. In: Proc. 10th Real-Time Systems Symposium, Santa Monica, CA, USA, December 1989, pp. 229–237 (1989)
5. Kopetz, H.: Real-Time Systems. Kluwer Academic Publishers, Dordrecht (1997)
6. Puschner, P., Burns, A.: Writing temporally predictable code. In: Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, January 2002, pp. 85–91 (2002)
7. Puschner, P.: Transforming execution-time boundable code into temporally predictable code. In: Kleinjohann, B., Kim, K.K., Kleinjohann, L., Rettberg, A. (eds.) Design and Analysis of Distributed Embedded Systems. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002), pp. 163–172. Kluwer Academic Publishers, Dordrecht (2002)
8. Reisinger, J.: Konzeption und Analyse eines zeitgesteuerten Betriebssystems für Echtzeitanwendungen. Ph.D thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Wien, Österreich (July 1993)