Improving Processor Utilization with a Task Classification Model based Application Specific Hard Real–Time Architecture*

Georg Färber Franz Fischer Thomas Kolloch Annette Muth

Laboratory for Process Control and Real–Time Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München
D–80290 München, Germany
Phone: +49–89–2 89–2 35 50, Fax: +49–89–2 89–2 35 55
{Georg.Faerber,Franz.Fischer,Thomas.Kolloch,Annette.Muth}@lpr.e-technik.tu-muenchen.de

Abstract

Modern microprocessors with caches and pipelines show increasing performance, but at the price of a decreasing predictability of execution times. The design of hard real–time systems however has to be based on worst case considerations. Consequently, real–time systems are generally oversized and fail to profit of developments in the standard processor field. This paper presents an approach where real–time systems are analyzed and built according to a task classification model. Each class of tasks corresponds to a type of processor best suited in terms of performance and deterministic execution times. The resulting target architecture framework is a tightly coupled heterogeneous multiprocessor system based on templates using off-the-shelf components. The described real–time system design process includes a schedulability analysis method that supports the partitioning and allocation process and provides the necessary real–time guarantees. The result is a event–driven hard real–time system with improved processor utilization that will provably meet all its deadlines. A rapid prototyping platform implementing this concept is presented as well as application examples.

Keywords: task classification model, hard real–time, schedulability analysis, processor utilization, caches

*This work was supported with funds of the *Deutsche Forschungsgemeinschaft* under reference number Fa 109/11-1 within the priority program "Rapid Prototyping for Embedded Hard Real–Time Systems."

1. Introduction

The significant technological advances provide an enormous growth of computing power within the new generations of microprocessors. Up to 500 MIPS are available in one chip, and there are projections to 2 GIPS until the end of the decade. However, memory access times could not follow the increasing processor performance: Even with today's wide data busses — the DECchip 21264 uses 128 bit for cache and a 64 bit system bus as opposed to the 16 bit bus of the MC 68000 — and the fastest static RAMs, memory bandwidth is by a factor of at least 5 too low for modern microprocessors. The only way to approach the peak performance is using a memory subsystem with a hierarchy of caches:

- small 1st level caches on chip (typically $2 \cdot 8 - 32$ KBytes), that can be accessed in one pipeline cycle, i. e. $2 - 3$ ns,
- a larger 2nd level cache of $0.5 - 4$ MBytes SRAM with access times of $8 - 10$ ns and
- main memory with access times of typ. 50 to 80 ns.

With hit rates of $98 - 99\%$ the microprocessors can perform quite near to the optimum if the cache architecture matches the problem structure. Unfortunately for event driven real–time systems with required response times in the micro– or millisecond range one can not rely on a good cache behavior. Exceptions and interrupts cause context switches in a non deterministic way and thereby one task's working set may be disturbed by another [9]. To guarantee response times even in the worst case, a zero hit rate has to be assumed and hence only about $2 - 10\%$ of the processor's peak performance are available for timing considerations.

Improving the utilization of new generation microprocessors for real–time applications is essential.

Several approaches have been investigated to overcome that problem: Cache partitioning techniques [8, 7] and latency hiding by multi threading architectures [11]. While the first may impose limits on the number of tasks, the complexity of the second seems not adequate for real–time systems.

The approach presented here matches the real–time application's task mix to well suited architectural templates. The following section shows different classes of real–time tasks, elucidated by examples. Section 3 describes an architectural framework that provides several types of processing units. The real–time system design process including schedulability analysis is outlined in Section 4. The paper closes with an overview of a target architecture we built mainly from off–the–shelf components according to the framework and applications we implemented to validate the system's properties.

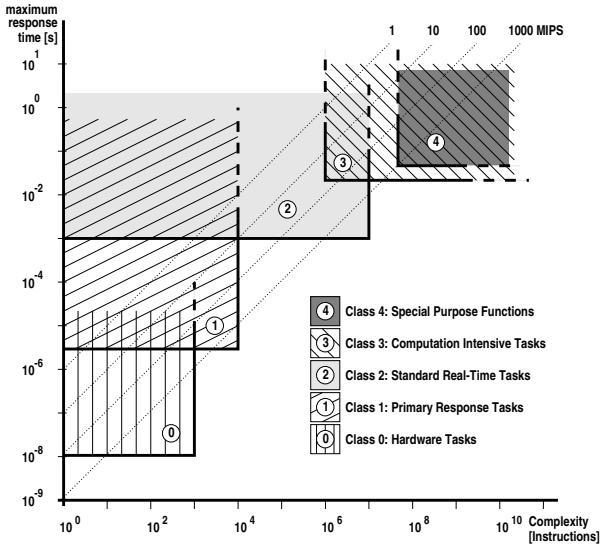

Figure 1. Classification of application tasks

2. Classification of Real–Time Tasks

In order to define suitable system architectures in the context of real–time systems, the following criteria describing individual tasks need to be considered:

***Maximum response time (deadline)*:** The time that may pass before the real–time system reacts to an event. Determined by the time constants in the technical process, it typically ranges between microseconds and hundreds of milliseconds.

***Maximum amount of computation (complexity)*:** The — for worst case considerations maximum — number of dynamically executed instructions of the task. The number depends on data associated with the event that activates the task, and it translates into the maximum processing time.

***Memory requirements*:** Size of code of the individual tasks, as well as the amount and the locality of their data.

Using the two parameters "complexity" (X-axis) and "maximum response time" (Y–axis), Figure 1 shows a plot of different classes of tasks (class 0 – 4). The $45°$–lines show the response times that can be achieved for a given amount of computation, depending on the processor performance.

Class 0: These tasks react to events with deadlines shorter than 1 μs, performing functions of low complexity. Typically implemented as hardware state machines, they can act as event filters for the real–time system.

Class 1: Interrupt handlers or primary response tasks. The very short response times (a few μs or more) and the small code (less than 10,000 dynamic instructions) indicate that cache less systems with 10 – 100 MIPS processors would be best suited to execute these tasks.

Class 2: The standard real–time tasks are described by response times of ≥ 1 ms and code lengths in the 5 to 50 KByte range. Again, processors of 10 – 100 MIPS performance are appropriate to perform these tasks. Caches make no sense for response times in the 1 ms range.

Class 3: Very computation intensive tasks, i. e. optimization tasks, of at least several million instructions, that show less critical response times of 20 ms and above. In this case, the 100 to 1000 MIPS performance of modern processors is necessary. Caches can be used, since hit rates may be assumed to be high with uninterrupted execution times of ≥ 20 ms.

Class 4: Special purpose functions, i. e. image processing, that can be executed best by special purpose hardware components: only a limited number of well defined algorithms is executed with response times between 20 ms (i. e. frame rate frequency) and 400 ms, in this example taking at least 25 million instructions. However, in the future the increasing power of standard processors may take over more and more work from special hardware modules.

For a given application normally a mix of these task classes has to be executed. In a typical autonomous mobile robot system there are for example:

- high speed sensor data acquisition, e. g. with a panoramic laser range finder (class 0),
- tasks to control the movement of the system and the manipulators (class 1 or 2),
- tasks to do short term planning (e. g. course planning) with time constants in the 20 – 200 ms range (class 2 or 3),
- tasks for long term planning including knowledge based system processing (time constants of many seconds), that are very computation intensive (class 3),
- tasks for sensor data processing (including image processing), time constant of 20 – 40 ms (class 3 or 4),

and many other tasks including communication to the outside world or building and maintaining the internal representation of the robot's environment.

The basic hypothesis here is that there is not one homogeneous architecture that is able to support all classes of tasks: More than one architectural template is necessary to provide an optimal runtime environment for such a real–time system. For each task the two parameters of Figure 1 have to be determined, as well as information on memory requirements. Finally they are allocated to the appropriate processing units.

3. Heterogeneous Multiprocessor Architecture Framework

Modern real–time systems often are distributed systems, where several *nodes* are connected (loose coupling) by a communication network, e. g. a field bus. In our approach one node in turn is built as a heterogeneous, tightly coupled multiprocessor system consisting of different types of *processing units* to match the needs of the different classes of real–time tasks. In this paper only the architecture of **one** single node is considered.

The architectural framework provides a hardware integration platform for different processor systems as well as operating system support for the application tasks and the necessary library functions.

The processing units' basic difference is the loss of predictable performance caused by interrupts and context switches:

Real–Time Units (RTUs) are optimized for small tasks with short response times. They use a limited amount of high speed memory to enhance predictability (Section 3.1).

High Performance Units (HPUs) are based on standard computer architectures to benefit from the technological advances regarding processing performance. The impact of interrupts and context switches on predictability is limited by software means (Section 3.2).

Special Purpose Units (SPUs) are based on processing elements optimized for special classes of tasks. Examples include DSP–based SPUs for digital signal and image processing algorithms or FPGA–based units for processing fast input and output tasks (Section 3.3).

Though a uniform operating system would be desirable from the application developers point of view, this is in general impossible for the heterogeneous system outlined above. Hence, local real–time operating systems[1] control the execution of the application's tasks. As a key integration factor however, a uniform and low overhead mechanism for communication between tasks on different processing units will be provided, which is based on a multi master bus (Section 3.4).

3.1. Real Time Unit (RTU)

This unit executes tasks with small amounts of computation, short deadlines and limited code (class 1 and 2). The basic architecture corresponds fully to the architecture of classical process control computers like the PDP 11. Because of the technological advances, they have about 100 times the PDP 11 performance. They consist of:

- A RISC processor without on-chip cache like (MIPS R3000 or SPARC) and about 50 MIPS performance.
- Fast memory instead of caches with cache–like speed and a size of about 1 – 2 MBytes (it is assumed that the code and the data of all tasks fit into the memory).
- A multi master bus adapter and I/O interface.

Typically, all these components fit on one single board which is directly inserted into the node's bus connector.

The RTU runs a small multi–threading real–time kernel supporting lightweight processes, resulting in very small kernel code ($\leq$ 50 KBytes) and in very short context switch times (1 – 5 microseconds for 100 – 200 instructions). Disabling hardware interrupts for certain times (e. g. $50\ \mu$s) avoids too many context switches.

Since there is no cache memory, the interrupt frequency is controllable and DMA transfers including accesses from other processors are predictable, the determination of worst case execution times on the RTU is straightforward. As it will be shown in section 4, it can be determined whether the system will meet all deadlines for a given task load. Typically, the interrupt service routines and about 10 to 50 tasks are allocated to one RTU. If the application requires it, more than one RTU has to be configured into the system.

[1] or simple run–time systems in the case of DSP–based SPUs

3.2. High Performance Unit (HPU)

This type of processor subsystem is intended to execute tasks (class 3) with medium to high amounts of computation (more than 1 million dynamic instructions) and with medium to high response times (more than 20 ms). Typically, code and data require megabytes of main memory.

For these tasks a real–time system should always make use of latest technology, i. e. standard architectures and components being developed for the competitive market of workstations and personal computers. But "standard" means also cache architectures: 1st– or 2nd–level caches are part of the boards and even the chips. To avoid the worst case scenarios mentioned above, these processors should be used in an operating mode that allows continuous operation of tasks without interruptions:

- All code and data are in main memory, no disk access during real–time operation.
- Preemptive multitasking is done in a priority or deadline based way with interrupts blocked for given intervals (time slices) T_S (e. g. $T_S = 20$ ms). That means, interrupts and context switches take place not more frequent than $1/T_S$, except a task blocks explicitly e. g. waiting for a message to receive. Hence the cache behaviour becomes predictable and allows the application tasks to execute near the processors maximum performance.
- With the methods shown in Section 4 it can be proven that the real–time conditions can be met also in worst case situations.

The multitasking mode outlined above can be integrated into a standard operating system by modifying the interrupt system and device drivers. Predictions of worst case execution times for these class 3 tasks are based on a minimum effective performance P_{min} of the processor subsystem for a time slice of duration T_S. P_{min} increases with T_S, which can be shown qualitatively using a simple cache and processor model:

Assuming that the average instruction time T_I after execution of R instructions decreases negative exponentially, the number of instructions executed within T_S can be calculated by summing up (integration) of $T_I(R)$ for R_S instructions executed in sequence:

$$\begin{aligned} \frac{T_I(R)}{T_C} &= 1 + (\frac{T_0}{T_C} - 1) \cdot e^{-\frac{R}{R_E}} \\ \frac{T_S(R_S)}{T_C} &= \int_0^{R_S} \frac{T_I}{T_C} dR \\ &= R_S + R_E \cdot (\frac{T_0}{T_C} - 1)(1 - e^{-\frac{R_S}{R_E}}) \end{aligned}$$

In these formulae the constant R_E depends on the cache architecture and locality of code and data.

Dividing R_S by T_S yields the minimum relative performance of a processor subsystem based on the assumptions above. Figure 2 shows a plot of P_{min}/P_{max} as a function of T_S and R_E for $T_0 = 20 \cdot T_C$).

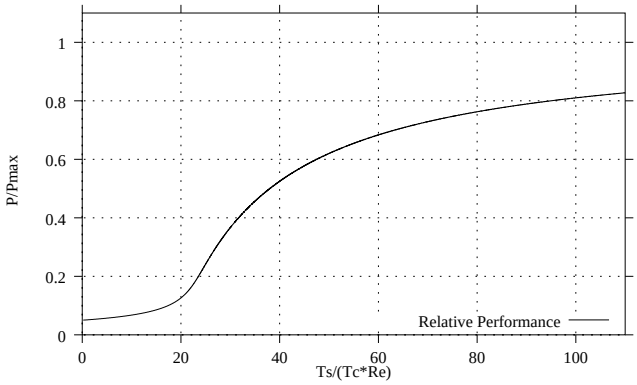

Figure 2. Relative performance as a function of uninterrupted execution time

Here 50% of the maximal processor performance are reached if $T_S = 40 \cdot R_E \cdot T_C$. With $R_E = 10000$ instructions, T_S must allow the execution of 200000 instructions (50% performance: $T_I = 2 \cdot T_C$). For a 100 MIPS–processor T_S would be 4 ms. If 80% of the specified performance are requested, T_S must be 10 ms (corresponding to 800000 instructions executed in one time slice T_S).

These figures show already that quite large amounts of computations have to be performed to make use of the performance of new processors. For a few 100 or 1000 dynamic instructions (class 1 or 2) the effective performance would be not adequate.

For complex tasks it is very difficult to define the worst case number of instructions that have to be executed dynamically. If the code for the tasks is available, the execution times can also be measured using the target system with different sets of data. The time slice approach presented here avoids additional uncertainties due to statistical events with a negative impact on cache performance.

3.3. Special Purpose Unit (SPU)

These units fit into the same framework and provide support for very demanding special algorithms. As the other units they access the global bus system for inter unit communication. Typical SPUs are digital signal processing (DSP) and image processing systems (class 4). Special hardware functions can be accommodated on SPUs: They may be realized with FPGAs or other programmable logic (class 0).

For SPUs on standardized busses (PCI) an increasing number of subsystem types are available off the shelf and can be integrated into future real–time systems.

3.4. Communication Issues

For the overall performance of distributed real–time systems low overhead and latency of task communication and synchronization mechanisms are crucial. This is also shown by the sensitivity of the schedulability analysis to these parameters. Additionally, a uniform communication layer is important to hide the details of the underlying hardware architecture, thereby simplifying the implementation of distributed real–time applications.

For the outlined multiprocessor architecture framework two different communication situations have to be considered: inter node and intra node communication.

The first involves transferring data over the — compared to the global intra node bus — slow communication network. Usually transmission times will dominate and using a dedicated (high priority) communication server task is appropriate.

In the case of intra node communication, data transmission times are by approximately 2 orders of magnitude shorter. In that case processing time for copy operations, protocol handling, synchronization and context switches (including system calls) have to be minimized in order to keep the overall communication overhead low.

The basic idea implementing the communication layer functions e. g. inter unit message queues (with nonblocking send and blocking receive operations) is to use a shared memory area for a message buffer pool and the send and receive queues. In order to send a message, the application task allocates a message buffer, prepares the message and enqueues the message buffer in the receivers receive queue. The receiving task in turn then processes the message and afterwards deallocates the buffer, which then is available for allocation again.

This communication scheme assumes the following properties of the processing units and the global bus system:

- Most units can be master on the global bus in addition to their function as slaves (targets).
- On the global bus all units share a common physical address space.
- At least a portion of a processing unit's memory is accessible to other bus masters.
- A processing unit can generate an interrupt on a remote unit by accessing certain predefined addresses on the global bus.
- If the global bus or some units on it do not support an atomic "test–and–set" operation (which is usually the case), at least one unit should provide an efficient spinlock or semaphore mechanism to avoid excessive synchronization effort when accessing shared communication data structures.

The basic functionality of inter unit message queues has been implemented on the RTU and HPU for use by the CAN bus application described in Section 5.2. In the near future we will evaluate and optimize this implementation and include communication with Special Purpose Units.

4. Real–Time System Design Process

Especially for event driven real–time systems the design is often done in a somewhat "experimental manner", using an oversized system to decrease the probability for the violation of timing constraints. To yield a better utilization of the available processing units, even for hard real–time systems — i. e. systems where a deadline miss may result in loss of lives and money — a more engineering like approach has to be taken. This section outlines an idealized design process for real–time systems using the processing units introduced above. [2]

- First, the stimuli (events) from the technical process have to be identified and their functional and timing requirements specified.
- The real–time tasks that realize the required responses have to be designed and a first estimation of their worst case execution time (WCET) or amount of computation has to be determined.
- Next, the tasks can be classified and allocated to the different types of processing units (RTU, HPU).
- The required number and performance of the processing units can be quantified by the schedulability analysis developed by Gresser [4, 3]. In contrast to analysis methods for time driven (periodic) systems where task deadlines are guaranteed by the construction of the time driven schedule [6], this method takes into account the timely behaviour of the technical process which stimulates the event driven system's tasks.

  Event streams describe the maximum possible number of events of a certain type within an interval I and lead to an *Event Function* $E(I)$. Single tasks are characterized by their worst case execution times (WCET) and

[2] Only RTUs and HPUs are considered here for simplicity: SPUs are used if the real time application requires it (image processing, response times in the sub microsecond range). The aspect of Hardware–/Software-Codesign is out of the scope of this paper.

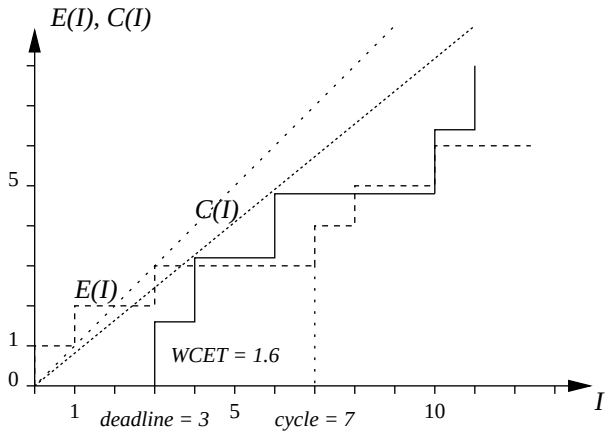

Figure 3. Example of Event Function $E(I)$ and resulting $C(I)$

the respective deadlines (maximum allowed response times) for the triggering events. The $C(I)$ Function is defined as maximum computation time requested and due within interval I. For a single task $C_i(I)$ can be calculated easily from $E(I)$ by shifting by the deadline and multiplication with the WCET (Figure 3).

While the resulting $C(I)$ for a number of *independent* tasks on a computing node is simply the sum of all the $C_i(I)$ functions, Gresser developed an algorithm to determine $C(I)$ for a network of communicating tasks, taking into account dependences of the triggering events, precedence constraints, inter node communication and mutual exclusion. For earliest deadline first scheduling he proved, that all tasks on one node meet their deadlines if the resulting $C(I)$ always runs under the bisector which specifies the available computing time in each interval. On the other hand, with

$$C(I) \leq C_{min} \cdot I \qquad \forall \quad I > 0$$

a minimum performance C_{min} for the processing units can be calculated. If there is no processor with the required performance, more than one processor of this type must be configured. As a result, there is a first guess for the numbers of RTUs and HPUs.

The task model used by Gresser is similar to that presented in [5], but Jeffay does not consider event dependences and limits the analysis to single processor systems.

- The allocation of the tasks to multiple RTUs and HPUs can be optimized using e. g. simulated annealing or genetic algorithms as described in [12].

After these steps the number of RTUs and HPUs is known as well as the allocation of all tasks to the processing units. Additionally, the schedulability analysis proved that the system will meet all its deadlines even in the worst case.

5. Target Architecture and Applications

5.1. REAR Hardware Architecture

Our target architecture REAR (Rapid Prototyping Environment for Advanced Real-Time Systems) was built according to the multiprocessor architecture framework presented in Section 3.

It is a configurable and scalable heterogeneous multiprocessor system consisting of processing nodes with state–of–the–art high performance microprocessors (CISC and RISC type) serving as HPU and RTU, and a SPU based on field programmable gate arrays. The nodes are tightly coupled by a global PCI–bus, which offers high throughput and low latency. Figure 4 gives an overview of the target system architecture, which is mostly built from off–the–shelf components.

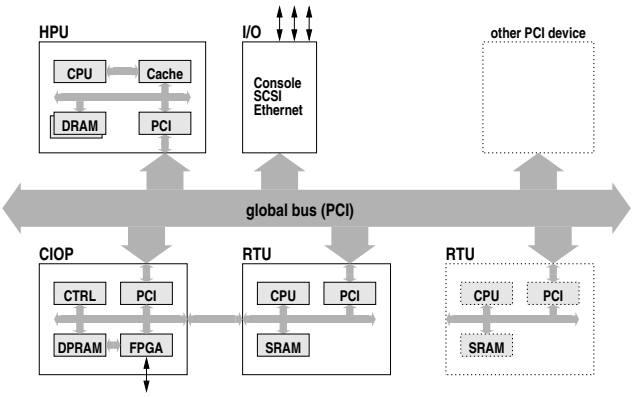

Figure 4. REAR hardware architecture

Our HPU is a PCI slot CPU with Intel Pentium processor, large L2–cache and main memory, satisfying very high demands for computing performance and for memory space. The RTU was built using a MIPS R4600 based single board computer with PCI interface. To narrow the memory bandwidth gap between the CPU and the DRAM, a fast SRAM module was added to the processor board. The SPU of our target architecture is called CIOP (Configurable I/O Processor), consisting of one Xilinx FPGA and additional dual ported RAM. It serves two dedicated functions: It acts as a separate application specific processing unit for tasks with deadlines too short to be met in software and it provides a flexible way of linking the prototyping architecture to the

embedding process. A more detailed overview of REAR is given in [2].

5.2. Applications

In this subsection we describe two sample applications, developed to test the REAR target architecture. As a first non–time critical example a **low level control of a robot arm** was designed to verify the basic functions of our CIOP (SPU), the I/O–interface and the HW/SW–interface (RTU/SPU and global bus/SPU). The flaw of this design was the expensive programming interface compared with the complexity of the whole task, taking up 72 % of the 576 configurable logic blocks (CLBs) and most of the routing resources available in the XC4013E . By extracting the non time critical functions, like the stepper motor speed control and even the phase generation to software, routing resources and therefore space for other I/O tasks in the CIOP could be regained.

CAN controller and monitor A CAN bus controller and monitor system was implemented on REAR as an application which imposes a wide range of timing constraints and complexity on the implementation. CAN [1] is a serial field bus which was originally developed for communication in vehicles, but has reached by now widespread use in the field of production automation. The CAN bus runs a masterless, message oriented bus protocol with CSMA/CA (Carrier-Sense Multiple Access/Collision Avoidance) access mode. Bus access is granted to each participant by bitwise arbitration using individual message IDs. Several cooperating error detection mechanisms guarantee fast system wide error detection and error recovery. CSMA/CA bus access, in combination with message priorities, the short data block length (max. 8 Byte) and data rates up to 1 Mbit/s lead to very short message latencies.

In our example, the REAR prototyping environment is used to implement a CAN bus monitoring and diagnosis system. Two distinct functions need to be performed by the CAN monitor: First, it has to be a fully functional *CAN bus participant* [10]. In addition to that it needs to execute the data sampling and test signal and error generation functions necessary for *monitoring and analyzing the CAN bus*.

The individual tasks to be performed for the CAN bus participation (from now on called CAN bus controller) can be classified by an orthogonal set of attributes: The deadline of the task and complexity of the function to be performed. This is shown in Table 1 (see also Figure 1).

An analysis of the timing and complexity requirements resulting from the CAN bus protocol yields three distinct groups of tasks. At *message level*, the complexity of the tasks — message identification and message frame generation, CRC checksum generation, error protocol functions, data handling — is medium to high. The timing constraint here is identical with the length of one CAN message, $44 - 108$ μs (44 control and up to 64 data bits, at an assumed data rate of 1 Mbit/s).

Table 1. CAN controller functions

Function	Deadline	Complexity
Message Level:		
Message Identification	44–108 μs	medium
Message Frame Generation	44–108 μs	medium
CRC Checksum Generation	44–108 μs	high
Error Logic	44–108 μs	high
Data Handling	44–108 μs	medium
Bit Level:		
Message Transmission	1 μs	medium
CRC Error Detection	1–3 μs	medium
Bit Stuffing and Destuffing	1 μs	medium
Below Bit Level:		
Bit Timing	270 ns	low
Bitwise Arbitration	60 ns	very low

The controller tasks at *bit level* — transmission of the message bits, CRC checksum error detection, bit stuffing and destuffing— need to be finished in the worst case before the start of the next message bit. This results in a timing constraint of 1 μs. The complexity of these tasks is medium.

Bitwise arbitration — i. e. transmission is stopped before the next message bit if a station sending a message with higher priority ID is detected on the bus — and synchronization of the sample points while receiving the message bit stream (bit timing) are tasks with timing constraints *below bit level*. The complexity of these tasks is low to very low.

The monitoring and diagnostic functions of the CAN component are not mentioned explicitly in Table 1. Data sampling and test signal generation can be performed at message level or at bit level. Therefore, the timing constraints of the CAN controller functions are also valid for the monitoring and diagnostic functions.

In a first approach, the entire CAN controller was realized in hardware on the CIOP, while the CAN monitoring and diagnosis functions were implemented on RTU and HPU, using a basic implementation of message queues as outlined in Section 3.4 (Fig. 5). The automated design process for the HW–part involved the following CASE tool chain: The CAN controller was specified in Statemate which also generated the VHDL–Code. Synopsis was used for synthesis and Xilinx XAct for fitting the netlists into the target technology FPGA.

The hardest time constraints on the CAN controller are imposed by the bit timing and bitwise arbitration tasks, with deadlines of 60 ns and 270 ns. The execution times of the arbitration mechanism and the bit synchronization were found to be 1 and 2 clock cycles, respectively. On a FPGA

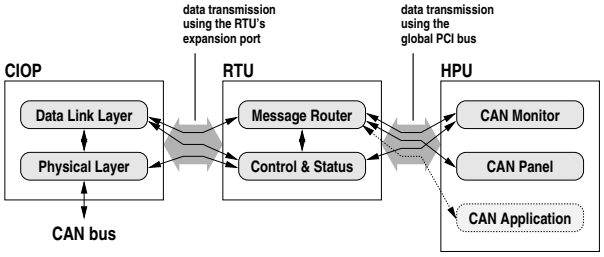

Figure 5. Task allocation and communication of the CAN application

board driven with 25 MHz the execution times then amount to 40 ns and 80 ns. It was therefore certain that the deadlines of these two functions would be met in this implementation.

Two versions of the CAN controller were realized and tested on the CIOP: The first one was implemented on a Xilinx 4013E FPGA. The entire CAN controller was by far too large for this component, so a simple controller with rudimentary functionality (transmission and reception of complete message frames, no message frame handling, no error handling, no CRC check) was implemented. The almost by factor two larger Xilinx 4025E, however, could accommodate the entire CAN controller. Both controllers were tested successfully on a CAN bus at the maximum bit rate of 1 Mbit/s.

In the near future, the CAN controller will be used as an example application for a further exploration of the HW–SW–boundary. Based on the timing/complexity analysis in Table 1, parts of the CAN controller will be implemented on the RTU, while the tasks with very short deadlines will remain in the CIOP.

6. Summary and future work

Taking into account the different characteristics of real–time tasks, we introduced a task classification model, which enables the design of less oversized target systems. This model leads to an architectural framework, whose processing unit templates allow the simple prediction of the WCETs. An adapted design process, extended by the schedulability analysis, results in an improved processor utilization of the target architecture components. Using the presented architectural framework, we built a rapid prototyping environment (REAR target architecture) and developed sample applications to validate the correctness of our classification concept in different scenarios.

In the near future we will refine and evaluate the simple cache model and the corresponding time slice operation mode on the HPU. Next, the design of a more sophisticated application example is planned, which requires the integration of a “Class 4 SPU” in the REAR architecture.

References

[1] K. Etschberger et al. *CAN Controller–Area–Network, Grundlagen, Protokolle, Bausteine, Anwendungen*. Hanser Verlag, 1994.

[2] F. Fischer, T. Kolloch, A. Muth, and G. Färber. A configurable target architecture for rapid prototyping high performance control systems. In H. R. Arabnia et al., editors, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997.

[3] K. Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. Number 268 in Fortschrittsberichte VDI, Reihe 10. VDI–Verlag, Düsseldorf, 1993. Dissertation am Lehrstuhl für Prozessrechner, Technische Universität München.

[4] K. Gresser. An event model for deadline verification of hard real–time systems. In *Proc. Fifth Euromicro Workshop on Real Time Systems*, pages 118–123, Oulu, Finland, June 1993. IEEE.

[5] K. Jeffay. Scheduling sporadic tasks with shared resources in real–time systems. In *Proceedings of the IEEE Real–Time Systems Symposium*, pages 89–99, Phoenix, AZ, Dec. 1992.

[6] H. Kopetz. Scheduling in distributed real time systems. In *Proceedings of the Advanced Seminar on Real-Time Local Area Networks*, pages 105–126, INRIA, Rocquencourt, France, 1986.

[7] J. Liedtke, H. Härtig, and M. Hohmuth. OS–controlled cache predictability for real–time systems. In *Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, June 9–11 1997.

[8] F. Müller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995.

[9] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Proc. 4th Intern. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, Apr. 1991. ACM.

[10] Philips Semiconductors, Eindhoven, The Netherlands. *PCA82C200, Stand–alone CAN Controller, Product Specification*, 1992.

[11] B. Smith. The hep supercomputer and its applications. In J. S. Kowalik, editor, *Parallel MIMD Computation*, pages 41–55. The MIT Press, 1985.

[12] H. Thielen. Automated design of distributed computer control systems with predictable timing behaviour. In J. A. de la Puente and M. G. Rodd, editors, *Proc. 12th IFAC Workshop on Distributed Computer Control Systems*, pages 47–52, Toledo, Spain, Sept. 1994. IFAC.