Journal of Universal Computer Science, vol. 6, no. 10 (2000), 1034-1051
submitted: 14/4/00, accepted: 5/9/00, appeared: 28/10/00 © *Springer Pub. Co.*

MSparc: Multithreading in Real-Time Architectures

Alexander Metzner
Carl von Ossietzky University Oldenburg, Germany
metzner@informatik.uni-oldenburg.de

Jürgen Niehaus
Carl von Ossietzky University Oldenburg, Germany
niehaus@informatik.uni-oldenburg.de

Abstract: This paper presents the use of multithreaded processors in real-time architectures. In particular we will handle real-time applications with hard timing constraints. In our approach, events (e.g. timer interrupts, signals from the environment, etc) are distinguished into three classes according to the reaction times that have to be met. Since two of these classes are well known in real-time systems, we will focus on the new class, for which the special features of a multithreaded processor together with a real-time scheduler realized in hardware are employed. Doing so enables us to realize the handling of events from this new class in software while still meeting the demands on reaction time. Additionally, the predictability of the application and the ease of implementing them are increased. The processor, named MSparc, which we developed to support these features, is based on block multithreading and is outlined in this paper, too. We then present an architecture, designed for rapid prototyping of embedded systems, to show the feasibility of this approach. Finally, a case study shows the potential of multithreading for embedded systems.

Category: C.1 Processor Architecture
Key Words: MSparc, multithreading, real-time systems, rapid prototyping

1 Introduction

Embedded systems can be found in most of today's technical products. A typical application field are safety critical functions with hard real-time constraints. In order to make the design of such systems easier and more effective, we suggest the use of multithreaded processors as the base processor of such embedded systems.

In conventional embedded systems the answer to incoming events is produced in two different ways: Hard timing constraints with very short answer times can only be guaranteed by implementing the necessary computation directly in hardware. All other computations are allocated on a processor. A real-time scheduler guarantees that the deadlines of the application threads are met[7]. Today's requirements on computational power in combination with dynamic scheduling strategies lead to the use of standard RISC processors in embedded systems as well as in traditional computer systems. With more complex instruction set architectures the disadvantage of longer context switch times wrt. microcontrollers with the same clock frequency has to be accepted. Saving all registers (i.e. the working set of a thread) of a SPARC for instance, costs many more cycles than with conventional microcontroller-based architectures, due to many registers and

– compared to processor speed – slow memory access time. To close this gap, we suggest to use multithreaded processors. This approach combines powerful computation due to the use of a RISC processor with fast context switches provided by multithreading and an external scheduler realized in hardware.

One of the main problems during design of an embedded system is the predictability of application threads, since the time of execution affects the ability to guarantee deadlines. During the development of a real-time application the partitioning and scheduling of threads has to ensure that all threads allocated on processors satisfy their timing constraints. Typical techniques to predict the execution time of threads are dynamic simulation with a processor simulator like MSPINT[9] or static cache simulation[14]. These methods always suffer from thread switches and interrupts, which change the cache behavior of the system in an unpredictable way, i.e. it is a challenging task to estimate the timing behavior of threads wrt. cache behavior. One way to solve this problem is to switch off the caches, another way is to partition the cache, as suggested in [13]. Our approach using multithreaded processors follows the latter in a similar way.

Multithreaded processors[1] are well known in the context of multiprocessor systems, like the Alewife[11]. The main goal of processors with multithreading is to hide latencies caused by memory hierarchies or network accesses and to provide a high processor utilization. Another approach to use multithreading in embedded real-time architectures[6] essentially employ this technique to replace interrupt service routines on a multithreaded Java microcontroller by so-called interrupt service threads, which provide fast context switches due to the multithreading architecture. By using the special hardware that provides multithreaded execution of threads, processor utilization is increased and therefore overall execution time of the application is decreased. Similar to this approach, our main goal is to improve task switching/scheduling operations by using the hardware supported switch abilities of a multithreaded architecture with a RISC processor and scheduling mechanisms realized in hardware. Additionally, task execution times can be improved by using simultaneous execution of different threads in multiple functional units [3, 4]. This is also possible in our approach, but it is out of the scope of this paper and we did not implement it in our first prototype.

This paper focuses on two important benefits provided by using multithreaded processors: faster reaction times (shorter execution times) of real-time threads and a better predictability of the worst case execution time.

The rest of the paper is organized in the following way: Section 2 combines the principles of multithreading and real-time scheduling and outlines the MSPARC processor. In section 3 we present the event classification and the architectural concepts to increase predictability and speed of the threads, and section 4 describes an example architecture and a case study. Finally, section 5 concludes this paper.

2 Multithreading and real-time scheduling

In this section we combine the principles of multithreading and real-time scheduling. First the chosen multithreading approach is described, then the similarities to real-time scheduling are outlined.

2.1 Multithreading

Multithreaded processors[1] may contain a fixed number of threads at the same time on chip. We only consider multithreaded processors, which employ a single datapath, therefore the execution of these threads is sequential. The threads are allocated in *contexts*, which typically include all important processor registers (program counter, special state registers and operand registers). The processor is able to switch between these contexts in a few processor cycles, without reloading the entire task control block of a thread.

There are two main approaches to multithreaded execution: *fine* multithreading and *block* multithreading. Fine multithreading (or interleaved multithreading) processors execute an instruction of a different thread at each cycle, thus the threads are executed instruction by instruction and every cycle a context switch is necessary, as shown on the left of figure 1. Examples for such types of processors can be found in [2, 5]. Another approach (e.g. [3]) assigns free execution slots in architectures containing multiple functional units to different threads, to attain simultaneous execution of these threads.

With block multithreaded processors the switch mechanism is usually activated by a demand from the system environment, typically the memory hierarchy[1]. Because the duration between two switch demands depends on special actions (e.g. remote memory accesses) and can be some ten to some hundred processor cycles, the threads are executed block by block. On the right of figure 1 the principle of block multithreading is outlined. Examples for block multithreaded processors can be found in [10, 11]. Since switches are rare wrt. the switch frequency of fine multithreading, the requirements for the duration of the switch mechanism are not as hard as in the case of fine multithreading. Additionally, since the environment in this case can determine the context which is to be activated, complex priority schemes can be applied to the selection of this context. Intuitively block multithreading is much nearer to thread execution in scheduled real-time application. However, the main hardware extensions are similar, both concepts need the duplication of important registers and additional control logic.

2.2 Real-time scheduling and multithreaded processors

In real-time systems incoming events (usually new sensor values) must be handled by threads. The event is usually passed to a scheduler which determines the thread that is to be activated[7, 12] or it is handled by an interrupt service routine (ISR), if no operating system is used. ISRs are employed for fast reactions on incoming events and therefore use static scheduling algorithms like rate monotonic scheduling, accepting the disadvantage of lower processor utilization (see [7]).

By comparing the execution of threads in traditional systems with the execution of contexts on a multithreaded processor, one can see some similarities, especially with the block multithreading approach: Both approaches are based on the block by block execution of (parts of) threads. In embedded real-time systems, scheduling is often done according to complex schemes, e.g. dynamic

[1] Nevertheless switches can be initiated by software in an operating system with corresponding higher costs.

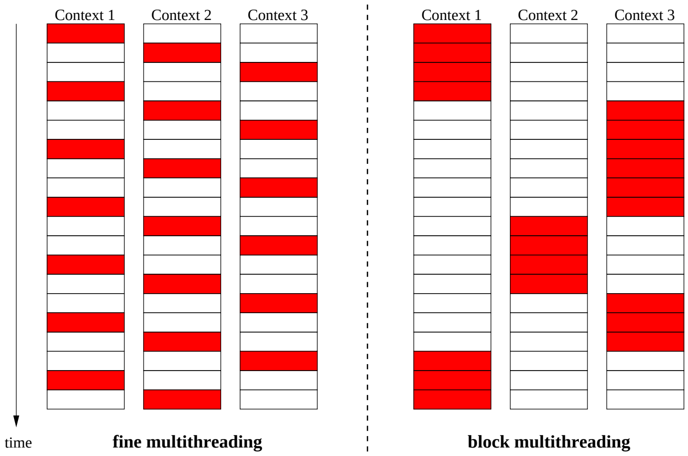

Figure 1: Fine and block multithreading can be distinguished by the manner of thread execution. The similarity of block multithreading to typical task execution in real-time architectures is apparent.

priority based ones like Earliest Deadline First [7]. Fine multithreaded processors often cannot take these schemes into account, because the selection of the thread the next instruction is fetched from has to be done in a single processor cycle. Even when operating system support is given (i.e. the operating system scheduler selects a set of active threads between which the processor switches), there are basically two possibilities: Either the active set contains a single thread, in which case the performance of fine multithreaded processors often degenerates [1], or the set contains more than one thread, in which case the scheduling scheme is more difficult to work out and deadlines are more difficult to guarantee.

For this reasons, we have taken a block multithreading approach. Using multithreading, the scheduling can avoid long switch latencies, because of the short switch mechanism provided by the processor. To support an arbitrary scheduling approach, there are two main ways: First, one context of the processor contains the scheduler and initiates context switches via software instructions or, second, a scheduler can be implemented in hardware which becomes the master of the switch demands. The first case is somewhat similar to ISRs or scheduling by an operating system, but with the advantage of a faster scheduling due to the fast switch mechanism, which is supported by multithreading (a task control block need not be saved/restored).

To speed up the reaction on incoming events in our approach we suggest to use the second way. The main idea is to move the scheduling algorithm from software to hardware, e.g. as programmable logic, and utilize the mechanism of block multithreading: Typically, the environment (memory hierarchy) in such multithreaded systems can force the processor to switch to a new context. This is exactly what a scheduler wants to do, hence we can employ this mechanism. Additionally, schedulers must have control of the dedicated thread which is to be executed. This fact necessitates some extensions of the switch mechanism, as we will see later.

Using multithreaded processors in real-time architectures in this manner has some effects on the way we see events, as we will show in a later section.

2.3 MSparc

Our implementation of a multithreaded processor (called MSparc[18, 19]) is based on the SPARC standard[20]. We adapted the SPARC specification in such a way, that no application task running in user mode on the MSparc is able to notice the multithreaded features. Only operating system functions have this knowledge, therefore context management is done in privileged processor mode. Hence, compilation can be done by using state-of-the-art compilers for SPARC architectures. A task created in this way can be associated to one of the four hardware contexts, that are provided by the MSparc.

To appear to user tasks as a standard SPARC we duplicate the whole register set for each context. Additionally, the caches are partitioned statically (dynamic partitioning is less predictable in real-time systems, as we will see later).

Context management, executed by the operating system, is done via a new state register (Context State Register [CSR]). Figure 2 shows the basic principle: The operating system (running in one of the hardware contexts) can change the data view by changing the associated value in the CSR. Hence, the OS task runs in its context, but can access the registers (status and data) of another context, determined by the CSR value. This enables us to initialize contexts and to up- and downloads threads from/into a context[2]. To provide data communication between different data views, we add additional global data registers (ASRs), visible to all contexts.

After initializing some contexts and marking them as active, the switches

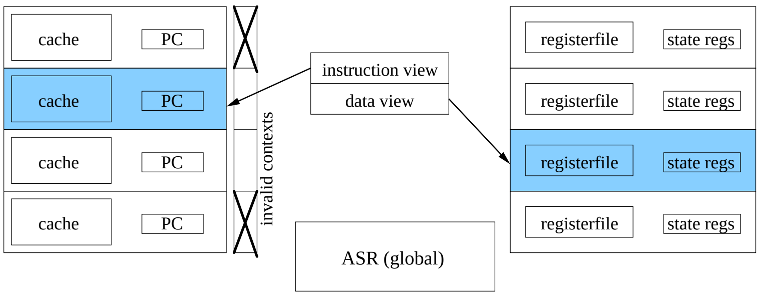

Figure 2: Context management is done by different views on data and instructions in the MSparc. Communication between different data views is realized through the global ASR registers.

coming from the memory hierarchy are handled without the operating system: If a switch demand is set by the environment, the MSparc stops the currently running task and immediately starts the next one (determined by an field in the

[2] Especially, this mechanism supports the ability of switching contexts by software.

CSR, which is calculated during a switch using a round robin algorithm). The switch delay is one cycle. Since the thread in the new context starts with an (almost) empty pipeline[3], total context switch latency is five cycles in the worst case.

As mentioned in the previous section, multithreading in real-time architectures requires additional features, caused by the usage of a scheduler realized in hardware: memory hierarchy is not the only origin of switch demands and the processor acts as a slave in systems designed using this methodology. In order to provide this feature, we developed the so called embedded control mode (ECM)[22]. The MSparc in ECM is controlled by an external scheduler, which can start/resume and stop tasks at will. In figure 3 the basic principle is shown. The external hardware scheduler sends the MSparc via the ECM-interface a set

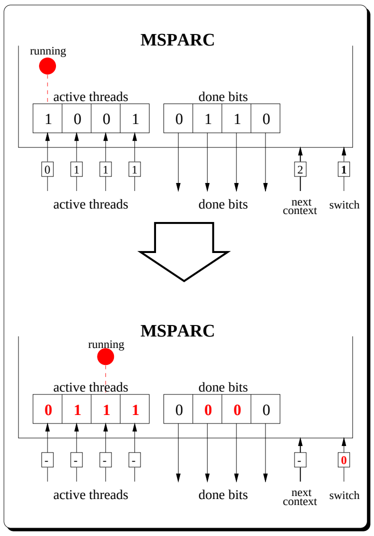

Figure 3: In ECM the MSparc's context switch mechanism is controlled by the environment. In this example, context 0 is running (indicated by the point), context 3 is enabled in the upper picture and a switch demand to context 2 occurs (indicated by the values of the switch and the next-context inputs). Simultaneously contexts 1 and 3 are enabled (indicated by the active-thread input). In the lower picture the changes in the next cycle are shown: DONE-Bits (signaling the termination) of threads marked as active are reseted and context 2 is actually running.

of *active contexts* and a specific context, which has to be started immediately. *Active contexts* are those contexts, which contain currently running threads. They will be automatically activated in a round robin manner, if a switch occurs, whose origin is *not* the external scheduler (e.g. termination of the actually

[3] The current implementation needs an empty pipeline. In further implementations this can be avoided. Nevertheless, when a new thread is activated the first time, the pipeline will be empty. Hence for worst case analysis this is an important factor and must be taken into account.

running thread). Additionally, all contexts are able to indicate the termination of their computation to the environment by using the ECM interface (in figure 3 the DONE-bits). If such a termination message occurs, the MSparc automatically initiates a switch and uses the set of active threads to choose the context to switch to. The running thread identification is sent to the scheduler via the ECM-interface, too (not shown in figure 3). If all tasks have terminated, the processor will halt until the environment sends a new switch demand.

With the above described interface, the scheduler has full control over the MSparcs execution of threads. Details of the supported interface between environment and MSparc can be found in [22] and [17].

A first version of the MSparc was implemented with the MIETEC $0.5\mu m$ standard cell process within the EUROPRACTICE program and is in fabrication process at the moment (expected to be delivered in September 2000). Key data are: Five stage pipeline, four contexts, for each context 8 register windows (544 registers at all) and 2 KByte on-chip instruction cache. The process data are: 78 mm^2 die area, about 26,000 standard cells and five types of macro cells with over 100 Kbit of memory. The processor in this version works at 30MHz.

This implementation is a feasibility study and we have no intention to compete with today's state-of-the-art processor technology. But we can show, that the extension of a standard RISC processor with multithreading is feasible with a penalty of less than 5% on the clock cycle. Hence, scaling our implementation with today's clock frequency for instance, leads to a new generation of processors in embedded systems, which – in combination with external hardware schedulers – are able to schedule threads in some ns, not in many μs as today's real-time operating systems. Therefore the impact of multithreading on threads and scheduling will be discussed in the next chapters.

3 Event classification

In typical embedded systems events occur asynchronously. They require (mostly) fast and predictable reactions, which are restricted by timing constraints. Typically, events can be classified in two groups: *control dominated events* and *computation dominated events*. Threads that handle events of the first class do not require complex computation but merely cause updates of actuators or the system state. Additionally, the timing constraints usually span only short timing intervals. Since software solutions are often not able to guarantee these timing constraints, the appropriate actions are usually implemented in hardware. The latter class of events requires complex computation and is mapped to software tasks on the systems processor. Hence, their timing constraints must be less stringent than the constraints of control dominated events. However, due to the limited number of processors with regard to the potentially great number of tasks, a real-time scheduling is necessary to guarantee the deadlines (see section 2).

If multithreaded processors are used instead of "normal" ones, two main effects can be noticed: First, if the scheduler is located in a specific context, the reaction time can decrease if the next scheduled task is in one of the hardware contexts. The main advantage of a system configuration like this is the missing necessity of storing and reloading the task's control block, including in worst case all registers of the processor. Instead, the multithreaded processor only needs to switch to the hardware context the task is mapped to. Otherwise, if the task is currently not allocated to a context, the reaction time does not differ from the time such an action would take on a standard processor (due to the reloading of the task control block). Second, if the scheduler is realized external to the processor, as outlined in section 2, we can create a new class of events, *fast computation dominated events*. In systems with a “normal” processor such events can not be handled by software tasks due to the short time period they require. Therefore in such systems they are implemented in hardware. With our new class of events we can move such tasks to software. In order to do this, two assumptions must hold: no other system component can initiate context switches, and the scheduled task is allocated in a context and no other task is mapped to the same context, thus no software scheduler in this context is necessary. Reserving one context in each processor for multiple tasks (with the ability of real-time scheduling) but putting this context under control of the hardware real-time scheduler, allows us to provide all of the three event classes. Figure 4 shows the three classes.

The new class of events has the advantage of faster reactions and, as we will

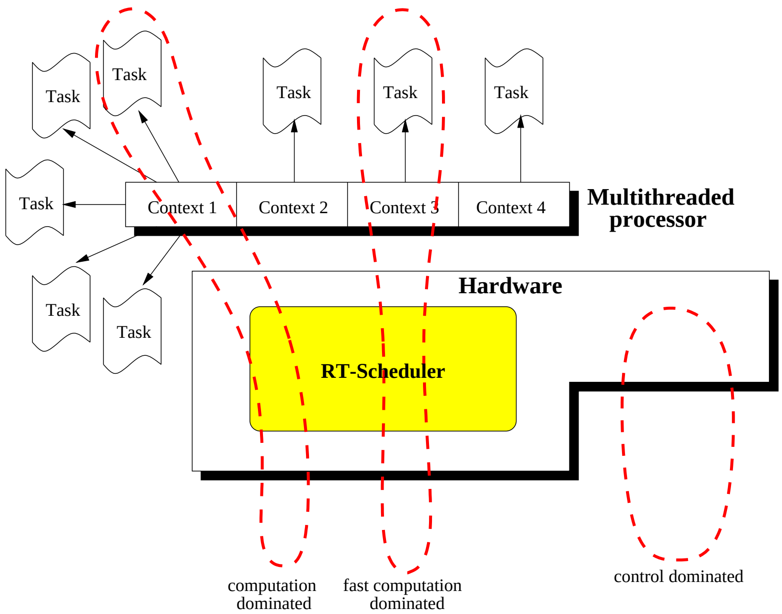

Figure 4: Our approach provides three classes of events. Control dominated events only use parts of the hardware, *fast* computation dominated events use one statically allocated context and computation dominated events use tasks which are located in parallel with other tasks on a context of the processor. The multithreaded processor shown in this example architecture supports four contexts.

see in the next section, is more predictable than other computation dominated events. The faster reaction is caused by three effects:

- Since the scheduler is implemented in hardware, it costs no processor time to determine the next task; the scheduler works in parallel to the processor.
- Furthermore, the mapping of only one task to one context effects, that a conflict on a context never occurs and neither the scheduling of other tasks to this context is necessary, nor do other tasks induce conflicts in the cache, which is mapped to this context.
- Additionally, in conventional systems, the event must be indicated via interrupts[8]. If another interrupt is active, the detection of this special event can last long. In particular, without special measures one is often not able to accurately determine the time of initiation of such interrupts in traditional systems. Using our approach, the event is immediately detected due to the use of switches instead of interrupts, even if the processor serves an interrupt request. In this case only the scheduler, which is implemented in hardware, determines to start a task.

These factors will be discussed in detail in the next two sections.

3.1 Estimating speedup

To estimate the speedup provided by our approach, the run-time of tasks with arbitrary size has to be calculated for both architectures, the conventional (with signaling task switches by interrupts and a software scheduler) and the multithreaded. Generally, the speedup depends on the type of applications (its run-time) and the cost of scheduling and detecting events. The speedup of benchmark $\mathcal{B}$ is determined by:

$$speedup(\mathcal{B}) = \frac{D_I(\mathcal{B})}{D_M(\mathcal{B})} = \frac{D(\mathcal{B}) + I}{D(\mathcal{B}) + 5},$$

where $D_I(\mathcal{B})$ is the run-time of benchmark $\mathcal{B}$ on a normal and $D_M(\mathcal{B})$ on a multithreaded architecture using the features described in the previous sections. $D(\mathcal{B})$ is the pure run-time of the benchmark without operating system and task switch costs. I stands for the cost of detecting and switching via RT-operating system[4], whereas this factor in the multithreaded architecture is fixed to 5 cycles (filling the MSPARC pipeline in the worst case, see section 2).

The formula contains two simplifications: First, in our MSPARC implementation we have seen, that the cost of multithreaded versus non multithreaded processors results in a small overhead on the clock cycle. That is not taken into account in our speedup formula. Second, assuming a rate of interruptions of the analyzed benchmark (which is typical in real-time systems due to higher prioritized threads), the pure run-time of $\mathcal{B}$ on the multithreaded and the standard processor is not the same due to different cache behavior: Interruptions on a standard processor will potentially remove cache contents according to the analyzed benchmark and therefore the speed on the multithreaded processor will be improved. For simplicity we assume, that both effects equalize each other as a first approximation. Further simulations are necessary to analyze these factors, because the effect of improving speed on multithreaded processors through cache behavior is application dependent and cannot be determined in general.

[4] including event detection, scheduling costs and also pipeline effects

Context switch times with state-of-the-art real-time operating systems vary and depend on several parameters, like the type of processor. Due to high constraints on the computational power in modern embedded systems, the use of standard high-performance microprocessor instead of micro-controllers will be necessary. To compare our approach with a standard system, we must know the context switch time of state-of-the-art real-time operating systems for SPARC processors. Typical values are between one hundred and a few thousand clock cycles (see [23]). Since these values are always quoted as *typical*, further simulations/measurements on running systems are necessary to find a worst case of context switch times. In [24] the authors measure a system based on the operating system RTEMS, which is quoted with a typical switch cost of above 150 cycles (see e.g. [26]). The results of this analysis show, that worst case switch costs are a few thousand. Worst case values for I typically range between 5000 and 7000 cycles (see [24] and [25]). These values also depend on the clock frequency: The average number of cycles for memory accesses in systems with high clock frequency is greater than in systems with lower frequency (even if caches are used) due to the slowness of today's memory devices. Figure 5 shows the estimated speedup on benchmarks with various sizes and three values for I. Tasks with run-times far beyond 10,000 cycles do not profit from multithreading, hence these tasks can be mapped to the third class of events, as described in the previous section.

Tasks with very small run-time profit especially from multithreading. This

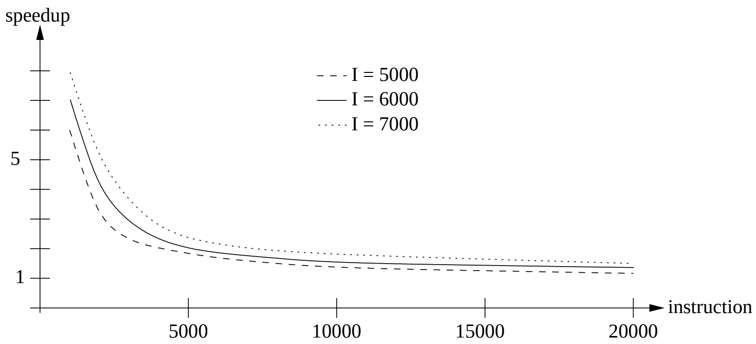

Figure 5: The speedup of applications on a MSPARC based real-time architecture depends on the run-time of the application and the scheduling costs I in a standard architecture.

leads to another important effect: a higher *interruptability*. Assume a task T_S with a given deadline. *Interruptability* of T_S is the rate of interruptions by smaller or faster tasks with higher priority, which still guarantees T_S's deadline. Figure 6 clarifies interruptability. In conventional architectures each interrupt will waste a large amount of time (determined by factor I) of the deadline of task T_S. In our approach the task switch costs are minimal and we can allow a much higher rate of interrupting T_S.

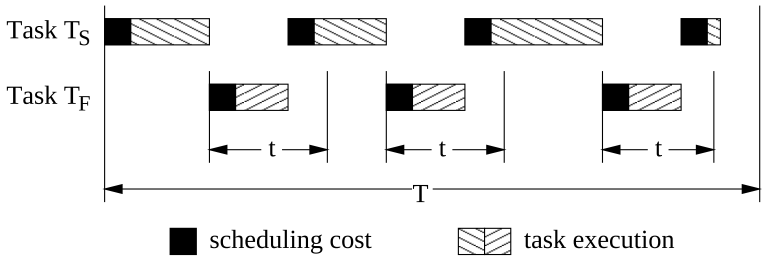

Figure 6: Task T_S with deadline T is interrupted by task T_F with deadline t. The higher the scheduling costs (black area), the less the number of interruptions which are possible without violating T_S's deadline T.

3.2 Increasing predictability

In embedded systems high demands for computational power result in the same problems as in traditional computer systems. The use of standard RISC processors for instance makes architectures with caches necessary, but caches always complicate the predictability of applications running on the system. Using multithreading can help to get better results in application prediction, as we will see in this section.

Predictability of real-time applications is important for guaranteeing, that the given real-time constraints are met in every case. Particularly, the timing behavior of caches is always difficult to predict, because calling scheduler code or serving interrupts can always induce conflicts in the cache. Additionally, the occurrence of events is often unpredictable, thus no hit ratio of the system caches can be estimated. Therefore, to estimate the duration of a real-time application, either the caches of a system will be switched off in the analysis, or the interruptability is restricted in such a way that parts of the application code cannot be interrupted. The latter case is somehow unrealistic, because tasks with a high priority typically should have the ability to be executed before low prioritized tasks terminate. The drawback of this first method is an inaccurate timing interval of the task's execution time, especially the upper bound can in reality be much nearer to the lower bound than shown in the analysis. Hence, this method will possibly reject an architecture, because the analysis shows that the performance is not adequate even though in reality the architecture's performance *is* adequate.

One promising approach is the partitioning of the caches[13]. Thus, the problems with interrupts and scheduling code as described above don't exist and the analysis of the timing behavior is more accurate. However, both previous discussed approaches are not able to avoid the influence of ignoring interrupts. An accurate estimation of the execution time from signaling the event to thread termination, especially the initiation of the task, therefore is not possible.

To estimate the execution time of a task wrt. the instruction cache behavior, an approach using static cache simulation is outlined in [14, 15, 16]. In order to use the results produced by methods like static or dynamic cache simulation (or similar methods), the task is required to run uninterrupted until its termination. Otherwise arbitrary program fragments could induce conflict situations in the instruction cache. If for example, one thread is interrupted by another thread, not only the code for the scheduler has to be executed which can invalidate parts of the thread's cache but also the newly started thread potentially replaces the cache's contents. Especially in cache systems with virtual addressing this is a quite probable effect. In conventional scheduled real-time systems uninterrupted execution of threads is unusual, because tasks can be suspended at every point of their execution. Additionally, the fact of ignoring events (via interrupts) complicates the predictability of the execution time of a task, because they cannot easily be taken into account by simulation methods. For data caches similar arguments hold.

Using multithreaded processors with a direct mapping of one task to one context can avoid the unrealistic requirement of uninterrupted execution on the one hand, and can prevent ignoring events on the other hand. One important point is the partition of the caches in fixed portions[5], i.e. each processor context has its own part of the cache. Additionally, we have no task migration between direct mapped tasks. Hence, we support a high level of locality in the caches. The predictability[6] of *fast computation dominated events* will be increased by two facts:

- If the cache is partitioned in a fixed portion for each context, no cache interference is able to cause conflict situations. This means, whenever a task is interrupted due to a switch to another task and eventually this task continues its execution after an additional switch, the cache behavior is the same as if no interruption would have taken place. Hence, from the task's point of view its execution is uninterrupted wrt. hit ratio and a stand-alone static or dynamic cache simulation of this task can be used to estimate the timing bounds to a realistic interval.

- Only interrupts may disturb the locality, but if on each processor one context is reserved for interrupt handling and the other contexts ignore interrupts, the locality is only destroyed in this context. Particularly, this context can include the tasks of the computation based events, as outlined in the previous section. On the other hand, by using multithreading to start/resume and stop tasks we can avoid the usage of interrupts at all, since we do not use ISRs to detect events.

With this improved predictability, the estimation of the timing interval of fastest execution time to worst execution time is closer to the real execution duration and therefore makes the design process of an embedded system more effective. Thus, our approach not only allows faster reactions, but also guarantees real-time constraints of smaller time intervals due to better predictability and better locality of tasks. Finally, our system provides three classes of events with different answer times and, particularly, different predictability: the execution time of the "slower" events are less predictable than the execution time of the "faster" ones. The approximation of the run-time of tasks is work in progress at the moment.

How such a system looks like in real hardware is presented in the next section of this paper.

[5] similar to the suggestion in [13]

[6] Particularly, the use of results created by static or dynamic simulation to determine timing bounds

4 An example: the EVENTS architecture

4.1 Architecture overview

As an example of the described concepts we present an architecture, developed at our group in the EVENTS project[17]. The two main goals of the EVENTS project are, first, to develop a rapid prototyping system, that can be used as a 'universal prototype' for those embedded control applications, which require fast reaction to external asynchronous events and, second, to develop a software environment that enables users to automatically generate, synthesize and download code for this system automatically starting from graphical specification with real-time constraints. As specification languages we use Statemate[27] and symbolic timing diagrams with real-time constraints. Our focus in this paper is the hardware architecture.

The architecture currently developed in the EVENTS project is shown in fig. 7. It is a multiprocessor system (NUMA) with four nodes connected via a local network. We want to deal with HW/SW codesign applications, thus in this cluster the nodes are connected with programmable hardware, realized by a field of FPGAs, particularly the WEAVER-FPGA-Board developed at the University of Tübingen [21] . If the complexity of an application overstrains this cluster, more nodes can be added.

Each node contains a MSPARC-processor[18, 19] and some portion of local memory. The dual ported RAM (DPR) of the cluster is used to transfer data between processors and FPGAs. The FPGAs serve two purposes: They communicate with external devices and they are used to schedule the multithreaded processors. Unlike other architectures we view the FPGAs as the system's master and the processors as coprocessors to the FPGAs.

As a first step towards this architecture, we developed a single processor system without any caches. Therefore the clock frequency is restricted to 8 MHz. The other concepts are implemented: Dual ported RAM is used for data communication between software threads and hardware threads and the ECM interface is used directly by the FPGA field, which contains the systems scheduler.

To estimate the worst case execution time of tasks we use dynamic cache simulation with the MSPINT[9] as a first step. MSPINT is a processor simulator for the MSPARC processor developed at our working group, which can take executable files and arbitrary memory models to compute the real memory behavior and the execution time of processor instructions (cycle accurate). It is equipped with an open back end, so different environments, e.g. memory hierarchies or the FPGA field controlling the ECM interface, can be modeled and linked to the simulator kernel. Since we only consider Statemate designs, software and hardware parts of the application can be handled similarly. Hence, we are able to compile the FPGA parts into software code for simulations with the MSPINT system. This is done for the described first architecture and we will see the results in a case study presented in the next section. Extending the architecture with caches is a task we are currently working on.

In general, if the worst case parameters for an application task are given, the simulation will produce a worst case execution time for this run. In the next version our design framework will be able to schedule – if necessary – the task

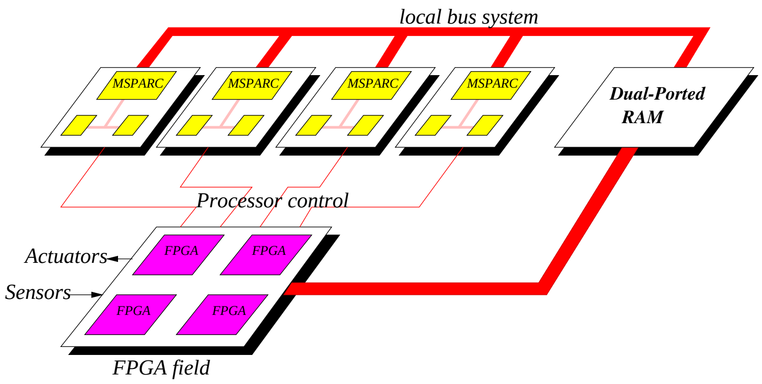

Figure 7: The hardware architecture of EVENTS.

in question to a free context via back-annotation.

Using this MSPINT environment, we have simulated first benchmarks on the EVENTS architecture to evaluate design decisions and to prove that multithreaded in real-time architectures causes the speedup predicted in section 3. In [29] we have presented a set of benchmark programs and their execution in our architecture in relation to an execution on non multithreaded architectures, based on the formula presented in chapter 3.1.

In [28] an ignition control system is implemented and simulated. The simulation results show that even if the processor runs with a low frequency, the architecture is able to handle realistic real-time applications with deadlines in the range below one msec. We summarize the results of [28].

4.2 Case study: Ignition Control System

The ignition control system is used to give ignition times to the four spark-plug of a car engine. It's main task is to set the ignition angle as close as possible to the knock region, while actual knocking has to be avoided as much as possible. The top level view – specified in Statemate – is shown in figure 8. The application has four inputs and two outputs. `BI` and `WI` are one bit signals coming from sensors monitoring crank- and camshaft motion. `BI` is raised on every complete turn of the camshaft and thus provides a reference point for the four-stroke operation. For each turn of the crankshaft, `WI` is raised sixty times. These inputs are used by the task `PB` to compute the actual position of cam- and crankshaft and by task `DZB` to compute the current rotational speed `DZ`. `PB` also computes the number of the spark-plug to be ignited next and controls a one bit signal `MESSEN` that causes the external component `KB` to begin or end integration of engine sound recorded by the knock sensor. By computing this integral, engine knock can be detected (see e.g. [30]), which `KB` signals via the one bit signal `KLOPFEN` to task `KEZ`. The 8-bit input `FRW` indicates the amount of air flowing through the intake manifold and thus, after being normalized by task `NUG`, serves as an indication

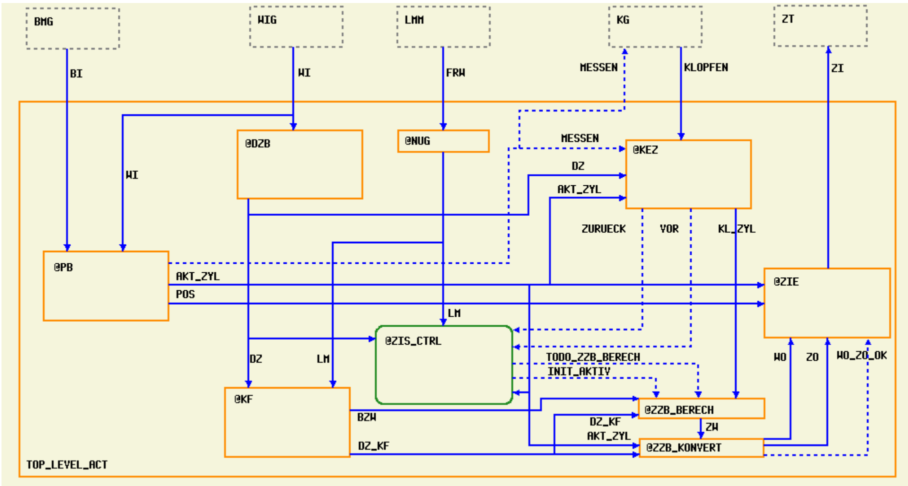

Figure 8: Top level view of the ignition control system

of the current engine load. Engine load and rotational speed are used by task `KF` to compute a base ignition angle by using a look up table and interpolating between values. This base angle is modified by task `ZZB_BERECH`. For each cylinder and each ignition cycle in which no engine knock occurs, `ZZB_BERECH` increases the base angle, thus moving ignition time closer to the knock region. Should engine knock occur, `ZZB_BERECH` decreases the angle used for the current cylinder yielding earlier ignition. This mechanism lets ignition time adapt to knock properties that may change over time and are different for each cylinder, at the "expense" of an occasional knocking.

The computed ignition time is sent to task `ZZB_KONVERT` which converts this value into a position offset and a time offset. It sends these values to task `ZIE`, which first waits until the indicated position is reached (by comparing the position offset to the current position computed by `PB`), then waits for the time offset and thereafter signals ignition to one of the four cylinders via the 4-bit signal `ZI`. Finally, task `ZIS_CTRL` is responsible for scheduling of the tasks.

The ignition control system has been synthesized for our target architecture using the GRACE-environment, which is also developed in the EVENTS project. Tasks `KF`, `ZZB_BERECH` and `ZZB_KONVERT` were compiled into software code and assigned to one MSPARC context each. The other tasks are implemented in hardware and thus FPGA-code is generated for them. Communication code and a run-time environment containing basic operating system functionality are automatically added to the system and the Statemate constructs for controlling tasks (used by `ZIS_CTRL`) are automatically mapped to the context switch mechanism of the MSPARC.

4.3 Preliminary results

The case study was simulated using the simulation environment described in section 4.1. As our single processor system will be running at a clock frequency

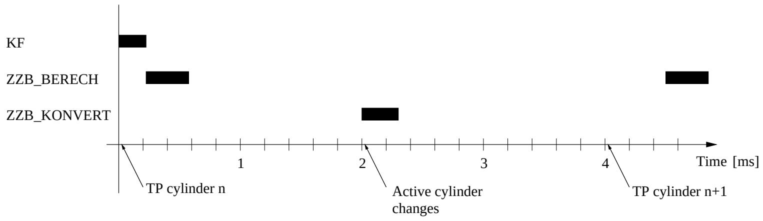

Figure 9: Software task execution at 7200 rotations per minute

of 8 MHz, this speed was set for the simulation, too. A maximum rotational speed of 7000 rotations per minute was required by the specification of the ignition control system. At this speed, an ignition occurs every 4.28 msec and for the ignition to occur with an accuracy of 0.5° the allowed jitter is 12 μsec. We modeled a simple environment that monitors the generated ignition impulses and compares them to the expected values. Additionally the environment signals engine knocking as appropriate.

Simulated rotational speed was constantly increased during the simulation. The ignition control system behaved correctly up to 7200 rotations per minute, after which there occurred knocking two times in a row for the same cylinder. Closer inspection revealed this "error" to be caused by an implicit assumption made while modeling the case study, namely, that rotational speed would never exceed the 7000 rotations per minute set as a maximum by the specification. Although the observed behavior already was sufficient for the specification, we removed this assumption from the code and measured the maximum attainable rotational speed, which turned out to be roughly 28000 rotations per minute. This indicates that even with as low a clock frequency as 8 MHz the architecture is more than sufficient for this kind of application.

Software tasks execution times were measured and yielded 223 μsec, 352 μsec and 335 μsec on average for tasks `KF`, `ZZB_BERECH` and `ZZB_KONVERT` respectively, with a maximum deviation of 5 μsec for all tasks. These numbers reflect the time from the context switch signal of the controller to the reception of the "DONE" signal by the FPGA. Since context switch time is negligible due to the multithreaded MSparc, the actual execution times of the tasks are almost identical. Figure 9 shows task execution times between the ignition of two consecuting cylinders at 7200 rotations per minute. As can be seen, CPU capacity is below 25%, even at an 8 MHz clock frequency. Closer analysis shows the potential of multithreading: Task `ZZB_BERECH`, which is triggered by the "DONE" signal of task `KF` on the left hand side of figure 9, is actually running 7 processor cycles (875 nsec) after the "DONE" signal is received. On the right hand side of the figure, `ZZB_BERECH` is triggered by the knock signal of hardware task `KEZ` and again runs after 7 cycles. The `AKT_ZYL` signal changed by hardware task `PB` triggers `ZZB_KONVERT`, which is running after 8 cycles. The additional cycles above the 5 cycles guaranteed by the MSparc for a context switch are caused by task `ZIS_CTRL` which has to analyze triggers and signal the appropriate context switch to the processor. Nonetheless, those times are still far below anything that could be reached with a conventional microcontroller.

5 Conclusion

This paper has presented an approach that attains faster reactions on events with hard real-time constraints and also allows better predictability of the execution time. Hence, we can not only guarantee smaller time intervals than the conventional approach of computation dominated events, but also our estimated bounds of execution times of the new class of events is more realistic and can help to simplify the development of a real-time architecture. Cache simulation can be used to determine these timing bounds accurately. Furthermore we have outlined an example architecture, including the multithreaded processor MSPARC, that implements our concepts and is currently under construction. Modeling this architecture in our MSPINT simulation environment allows us to evaluate design decisions using example applications. Ongoing work is accomplishing the architecture and creating a tool set for run-time analysis in given system architectures to help automatize application development. Finally, we will have a system for all three event classes, that will provide a design framework with the ability of guaranteeing hard timing constraints.

References

[1] R. A. Iannucci, G. R, Gao, R. H. Halstead, J. Smith, editors: Multithreaded computer architecture : A summary of the state of the art. Kluwer Academic. 1994

[2] B. J. Smith: Architecture and application of the HEP multiprocessor computer system. SPIE 298, pages 191–202. 1981

[3] D. Tullsen, S. Eggers, H. Levy: Simultaneous multithreading: Maximizing on-chip parallelism. 22nd ACM International Symposium on Computer Architecture, pages 392–403. 1995

[4] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. 23rd ACM International Symposium on Computer Architecture, pages 191-202, May 1996

[5] R. H. Halstead, T. Fujita: MASA: A multithreaded processor architecture for parallel symbolic computing. 15th Annual International Symposium on Computer Architecture, pages 443-451. 1988

[6] U. Brinkschulte, C. Krakowski, J. Kreuzinger, T. Ungerer: A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling. PACT, New Beach, October 1999

[7] C. L. Liu, J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the Association for Computing Machines. Vol. 20. No. 1. pages 46–61. January 1973

[8] P. Laplante: Real Time Systems Design and Analysis. IEEE Computer Society Press. 1993

[9] C. Böckmann: Implementierung eines Simulators für die MSPARC. Carl von Ossietzky University Oldenburg. July 1997

[10] W. Grünewald, T. Ungerer: Towards extremely fast context switching in a block-multithreaded processor. 22nd EUROMICRO Conference: Hardware and Software Design Strategies. pages 592–599. 1996

[11] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowitz, K. Kurihara, B. Lim: The MIT Alewife Machine: A Large-Scale Distributed Memory Multiprocessor. Scalable Shared Multiprocessor. Kluwer Academic Publisher. 1991

[12] I. Stoica, H. Abdel-Wahab: Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. Technical Report TR-95-22. CS. Department. Old Dominion University. Norfolk. 1995
[13] D. B. Kirk: SMART (strategic memory allocation for real-time) cache design. IEEE Symposium on Real-Time Systems. pages 229–237. December 1989
[14] F. Mueller, D. B. Whalley, M. Harmon: Predicting Instruction Cache Behavior. ACM SIGPLAN Workshop of Language, Compiler and Tool Support for Real-Time Systems. June 1994
[15] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, M. G. Harmon: Timing Analysis for Data Caches and Set-Associative Caches. Proceedings of the IEEE Real-Time Technology and Applications Symposium. June 1997
[16] C. A. Healy, D. B. Whalley, M. G. Harmon: Integrating the Timing Analysis of Pipelining and Instruction Caching. Proceedings of the IEEE Real-Time Systems Symposium, December 1995
[17] K. Lüth, A. Metzner, T. Peikenkamp, J. Risau: The EVENTS Approach to Rapid Prototyping for Embedded Control Systems. Zielarchitekturen eingebetteter Systeme, 14. ITG/GI Fachtagung Architektur von Rechnersystemen. Rostock. 1997
[18] W. Damm, A. Mikschl: MSPARC : a multithreaded SPARC. L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert, editors, Euro-Par'96 Parallel Processing : Second International Euro-Par Conference, Vol II. LNCS 1124. Springer Verlag 1996
[19] A. Metzner, T. Bienmüller: Traps and fast context switches in a multithreaded SPARC. Technical Report, Carl von Ossietzky University Oldenburg. 1997
[20] SPARC International Inc.: The SPARC Architecture Manual. Version 8. Prentice Hall. Englewood Cliffs. 1992
[21] G. Koch,U. Kebschull, W. Rosenstiel: A prototyping environment in the hardware/software codesign in the COBRA project. Proc. 3rd International Workshop on Hardware/Software Codesign Codes/CASHE'94. Grenoble. 1994
[22] A. Metzner, T. Bienmüller: Embedded Control Mode der MSPARC: Spezifikation und Implementierung. Technical Report. Carl von Ossietzky University Oldenburg. 1997
[23] Embedded Systems Programming Buyer's Guide. http://www.embedded.com
[24] M. Baker-Harvey: ETI resource distributor: Guaranteed resource allocation and scheduling in multimedia systems. Proceedings of the Third Symposium on Operating Systems: Design and Implementation, pages 131–144, 1999
[25] K. Weiß, T. Steckstor, W. Rosenstiel, C. Nietsch: Performance analysis of real-time operating systems by emulation of an embedded system. Rapid System Prototyping. 1999
[26] RTEMS Documentation. http://www.rtems.army.mil/rg4/lithead.html
[27] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot: STATEMATE: A working environment for development of complex reactive systems. IEEE Transactions on Software Engineering 16, pages 403–414.
[28] J. Niehaus, K. Lüth, W. Damm: Multithreading in rapid prototyping target platforms. Technical Report. Carl von Ossietzky University Oldenburg, 2000. Accepted for AES2000.
[29] J. Niehaus, W. Damm, A. Metzner, A. Mikschl: Die EVENTS-Architektur. it+ti 2/2000, Oldenbourg Verlag. 2000
[30] A. Kirschbaum: Ein Rapid-Prototyping-Verfahren zum Entwurf von Kommunikationsarchitekturen in eingebetteten Systemen. PhD Thesis. Technische Universität Darmstadt