Parallel, Hardware-Supported Interrupt Handling in an Event-Triggered Real-Time Operating System

Fabian Scheler, Wanja Hofer, Benjamin Oechslein,
Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann
Friedrich–Alexander University Erlangen–Nuremberg, Germany
{scheler,hofer,oechslein,wosch,lohmann}@cs.fau.de

ABSTRACT

A common problem in event-triggered real-time systems is caused by low-priority tasks that are implemented as interrupt handlers interrupting and disturbing high-priority tasks that are implemented as threads. This problem is termed rate-monotonic priority inversion, and current software-based solutions are restricted in terms of more sophisticated scheduler features as demanded for instance by the AUTOSAR embedded–operating-system specification.

We propose a hardware-based approach that makes use of a coprocessor to eliminate the potential priority inversion. By evaluating a prototypical implementation, we show that our approach both overcomes the restrictions of software approaches and introduces only a slight processing overhead in exchange for increased predictability.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time systems and embedded systems; D.4.1 [**Operating Systems**]: Process Management—*Threads*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Design, Performance

Keywords

CiAO, Real-Time Systems, Priority-Driven, Rate-Monotonic Priority Inversion, TriCore, Interrupt Handling

1. INTRODUCTION

Besides functional correctness, timeliness is the most important property of real-time systems, since results that are delivered too late may have the same impact as incorrect ones. Thus, it is necessary that the timely completion of all tasks associated with hard deadlines can be guaranteed in advance by performing a schedulability analysis. The more predictable the overall system behavior is, the more likely it is that a schedulability analysis yields a precise and safe result.

CASES'09, October 11–16, 2009, Grenoble, France.

1.1 Rate-Monotonic Priority Inversion

Unfortunately, current event-triggered systems maintain a bifid priority space: The hardware reigns over the priorities associated with interrupts, and the operating system governs the priorities of threads. Leyva-del-Foyo et al. showed that real-time systems can suffer significantly from such a bifid priority space [5]: Interrupts that are associated with soft or low-priority real-time tasks may interrupt hard real-time tasks with higher priorities. This can influence the response times of the high-priority tasks at such a rate that they may miss their deadlines. This disturbance caused by low-priority tasks implemented as interrupt handlers is known as *rate-monotonic priority inversion*. Often only relatively weak assumptions can be made about the occurrence rates of such soft real-time tasks, which also leads to a less predictable system behavior. Hence, the results of a schedulability analysis will also be less precise and, thus, more pessimistic.

To tackle this problem, Leyva-del-Foyo et al. suggested to implement all tasks as threads, and to trigger them from very short interrupt handlers by posting a semaphore, for instance [5, 6]. Furthermore, an interrupt-leveling mechanism implemented in software prevents any disturbance by lower-priority interrupt handlers by masking all corresponding interrupt sources. However, this solution is not suitable for systems that demand more sophisticated scheduler features like being able to store multiple activations of a task or allowing for multiple tasks with the same priority. For instance, the AUTOSAR-OS standard [2], whose implementations are widely used in automotive microcontrollers, prescribes such abilities.

1.2 Parallel, Hardware-Supported Interrupt Handling

Our approach is *hardware-based* and prevents unwanted disturbance without masking any interrupts. Instead, we redirect interrupt requests to a coprocessor and handle them in parallel to the normal program execution. Our solution improves on the one proposed by Leyva-del-Foyo et al. by eliminating its weaknesses:

- *Multiple task activations* can be stored. With the software-based solution, at most one additional activation of the same task can be buffered by the hardware.
- Multiple tasks can *share the same priority* while the order of their activations is preserved. This is not possible with the software-based approach.
- We do not use semaphores to inform a task about event occurrence. Explicit waiting for a semaphore in the software-based solution prevents the implementation of *stack-sharing techniques*, which are very important in embedded systems to save costly RAM.

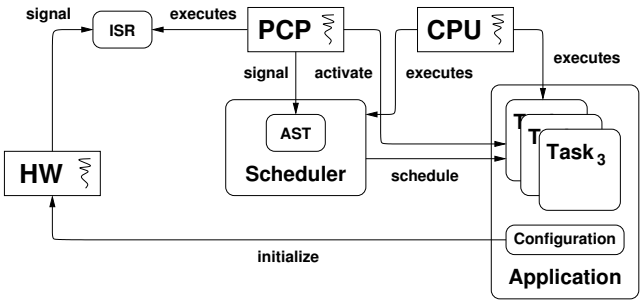


Figure 1: Parallel–Interrupt-Handling Design

Section 2 provides a detailed description of our approach and clarifies its improvements over the software-based approach. We implemented a prototype of our approach in the CiAO operating system for the TriCore microcontroller (see Section 3) and evaluated its properties (see Section 4). Work that is related to ours is given in Section 5. Section 6 discusses aspects related to our approach—like the problem of interrupt overload in the context of our solution, or its general applicability.

2. DESIGN

A design overview of our approach is sketched in Figure 1; it requires an additional hardware component that works in parallel to the main CPU. Furthermore, that component must also be able to service interrupt requests issued by other peripheral devices, to access scheduler data structures, and to send a signal to the CPU—via an inter-processor interrupt, for instance. Since our implementation targets the TriCore microcontroller, this hardware component is the peripheral control processor (PCP), but any coprocessor with similar properties is suitable for our purposes.

We prevent rate-monotonic priority inversion by redirecting all hardware interrupt requests to the PCP and by executing them on the PCP. Furthermore, we configure the interrupt-control system to map the task priorities to interrupt priorities in order to create the unified priority space that is necessary to avoid rate-monotonic priority inversion. Interrupt-control systems supporting this are common in state-of-the-art microcontrollers used for embedded systems, the TriCore being among them. The mapping itself is application-dependent and only needed to initialize the interrupt system properly.

The interrupt handler on the PCP coprocessor has the responsibility to activate the task that is associated with the given interrupt source. To accomplish this activation, the PCP needs to access the data structures of the scheduler. If and only if the activated task has the highest priority of all ready tasks in the system, the PCP informs the CPU that a different thread has to be dispatched. This is done by signaling an *asynchronous system trap* (AST) to the CPU. The AST is implemented as an interrupt handler *on the CPU* and simply triggers rescheduling on the CPU. In contrast to the ISRs that are executed on the PCP, the AST is only triggered and thus only interrupts the CPU if a task of higher priority becomes ready. Hence, it is not a source for rate-monotonic priority inversion itself.

2.1 Design of the PCP Interrupt Handler

The control-flow structure of the interrupt handler executed for every interrupt request signaled to the PCP is given in Figure 2.

First, the interrupt source causing the current request has to be acknowledged. Following that, the task associated with the interrupt source is looked up from the application configuration and is activated. Eventually, the priority of the currently running task is compared to the one of the recently activated task and the CPU is informed about the rescheduling via an inter-processor interrupt *if necessary*.

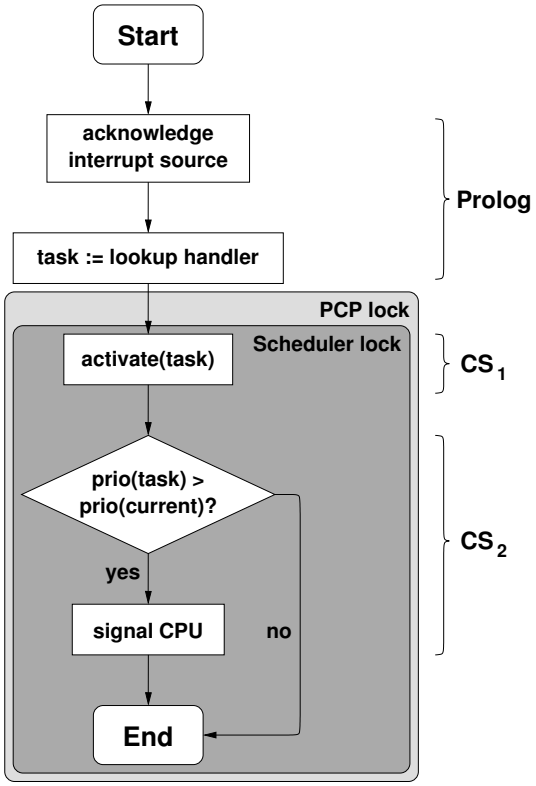


Figure 2: Structure of the PCP Interrupt Handler

To keep the latency of a task activation for high-priority tasks as small as possible, the ISR on the PCP can be interrupted by interrupt requests of higher priority. Both this and the fact that the ISR is executed on the PCP *in parallel* to the tasks on the CPU implies the necessity to synchronize the access to commonly used data structures. Synchronization is achieved by guaranteeing mutual exclusion within two critical sections CS_1 and CS_2 (see Figure 2). These critical sections are secured by two nested locks. The outer PCP lock prevents that the ISR is interrupted by high-priority interrupt requests by locking the interrupts on the PCP. The inner lock ensures mutual exclusion against the scheduler via Peterson's algorithm, a spinlock-based mutual-exclusion algorithm [9]. The PCP lock has to be taken before the scheduler lock; otherwise, a deadlock occurs if an ISR tries to take the scheduler lock that is already locked by an interrupted ISR.

We decided to guard the critical sections CS_1 and CS_2 contiguously and not separately. Since we use Peterson's algorithm to synchronize the kernel and the interrupt handler, the interrupt handler can be blocked by the kernel every time it tries to acquire the scheduler lock. The blocking time introduced at that point would be larger than both of the two critical sections CS_1 and CS_2 together (see also Section 4.3).

The prolog of the ISR does not have to be synchronized since we target statically-configured systems such as AUTOSAR OS [2] or OSEK OS [17]; that is, the association of interrupt sources does not change at runtime.

However, the activation of the task associated to the interrupt request (critical section CS_1) obviously demands for synchronization. It has to be guarded against interruptions by other interrupt requests, since these also activate tasks and, thus, manipulate the same data structures. The activation also has to be guarded against interleaving accesses from the CPU, as a task executing on the CPU can alter the scheduler data structures in a synchronous manner—when a task terminates, for instance.

The second critical section (CS_2) has to be synchronized to prevent rate-monotonic priority inversion on the CPU. Provided the current ISR is interrupted exactly after checking if scheduling is necessary and before informing the CPU about the positive outcome of this check, then an interrupt is signaled by both the interrupting high-priority ISR and the lower-priority ISR that was interrupted. This would cause an interrupt on the CPU that does not result in dispatching a task, as rescheduling and dispatching the high-priority task has already taken place. The additional scheduler lock is necessary because a task running on the CPU could manipulate the system in such a way that makes rescheduling unnecessary—by activating and dispatching a highest-priority task, for instance. Again, this scenario would result in an unnecessary interrupt on the CPU, which could be interpreted as rate-monotonic priority inversion.

Hence, the design of the PCP interrupt handler completely eliminates rate-monotonic priority inversion while keeping the interrupt latency as low as possible.

2.2 Advantages over Software-Based Designs

In contrast to software-based approaches to avoid rate-monotonic priority inversion (such as the one presented in [5]), the parallel–interrupt-handling approach does not require to mask lower-priority interrupt sources. This is because the task leveling in our approach is implemented in hardware using the interrupt system—which is designed for that purpose—and not artificially in software.

Our approach therefore introduces some significant conceptual advantages.

2.2.1 Multiple Task Activation

Masking and unmasking interrupt sources to prevent rate-monotonic priority inversion like performed in [5] results in an inflexible handling of tasks that are activated more than once, as illustrated by the example in Figure 3 (a). The graph shows two tasks T_1 and T_2, with T_1 having a higher priority than T_2. At time t_1, task T_1 is activated for the first time and, hence, preempts the previously running task T_2. During its execution, T_1 is activated two more times at t_2 and t_3. The first activation at time t_2 can be buffered by the interrupt system (provided a level-triggered interrupt system is used), but the second activation at t_3 will definitely be lost. Thus, when T_1 terminates at t_4, it will only be scheduled one more time.

Our approach does not suffer from this limitation, as illustrated by the same example in Figure 3 (b). Since no interrupt sources are masked, the interrupt requests at t_2 and t_3 can be serviced by the PCP and task T_1 is activated twice, leading to two more executions of T_1 after t_4. Hence, arbitrary multiple-activation scenarios as required by the AUTOSAR-OS specification, for instance, can be handled as long as the scheduler supports them.

2.2.2 Multiple Tasks per Priority

The software-based approach also has limited capabilities to handle multiple tasks sharing the same priority, as indicated by the example in Figure 4 (a). At t_2 and t_3, two tasks T_2 and T_3 sharing the same low priority are activated; the requests are buffered in hardware, because the interrupt sources are masked at this point. It then depends solely on the hardware which of these tasks is handled first when T_1 terminates at t_4.

Our approach, however, preserves the activation order of tasks, as can be seen in Figure 4 (b). Since interrupt sources are not masked, the task activations can be carried out in the order of their occurrence and the corresponding threads can be enqueued in the scheduler accordingly. Thus, different tasks sharing the same priority can be handled without losing the order of their activations as required by AUTOSAR OS, for instance.

2.2.3 Stack Sharing

The software-based design does not only limit scheduling features, but it also hampers the employment of stack-sharing techniques, which are crucial in embedded systems to save expensive RAM. The software leveling requires all event handlers to be scheduler-managed threads, which are then informed about the event occurrence via a single-sided synchronization facility like a semaphore posted by the ISR executed by the hardware. Thus, every thread has to wait for a corresponding semaphore. This use of blocking, single-sided synchronization makes the implementation of effective stack-sharing techniques very cumbersome and inefficient [12].

Our approach does not rely on blocking operating-system primitives, allowing for unrestricted stack optimization and, therefore, optimization of the system's RAM consumption.

3. IMPLEMENTATION

We prototypically implemented our approach on the TriCore microcontroller platform by extending our CiAO research operating system. In the following subsections, we describe the operating-system environment, the relevant peculiarities of the hardware platform, and the details of the interrupt-handler implementation.

3.1 The CiAO System

CiAO[1] is a configurable family of operating systems that targets embedded and deeply embedded systems and supports the TriCore platform detailed below [14]. The particular variant that we used for the extension and evaluation is oriented at the OSEK-OS [17] and AUTOSAR-OS [2] specifications. It therefore implements an event-triggered operating system with a multi-level queue scheduler without time-slicing and statically assigned priority levels for the threads.

In the standard configuration, the CiAO kernel for the TriCore platform is synchronized with interrupt service routines by raising the interrupt priority level to the one of the ISR with the highest priority upon entering the kernel, and resetting it to zero upon exiting it. This way, ISRs triggered during that time are deferred by the hardware until after the critical section. With the extension described in this paper, the CPU running the tasks has only one interrupt left that can interrupt it: the asynchronous system trap (AST) triggered by the PCP. Hence, the kernel is kept synchronized by setting the interrupt-priority to the one of the AST during the system calls.

Since CiAO is implemented using techniques from aspect-oriented programming (AOP [10])—namely the AOP language and weaver AspectC++ [19]—, the adaptation of the kernel described in this paper is implemented in a single, encapsulated aspect module. This aspect includes the code to synchronize the kernel with both the AST (as described above) and the PCP (using Peterson's algorithm, see Section 2.1). By making use of the AOP feature of *quantification*, this heavily cross-cutting adaptation (each system

[1] CiAO is Aspect-Oriented.

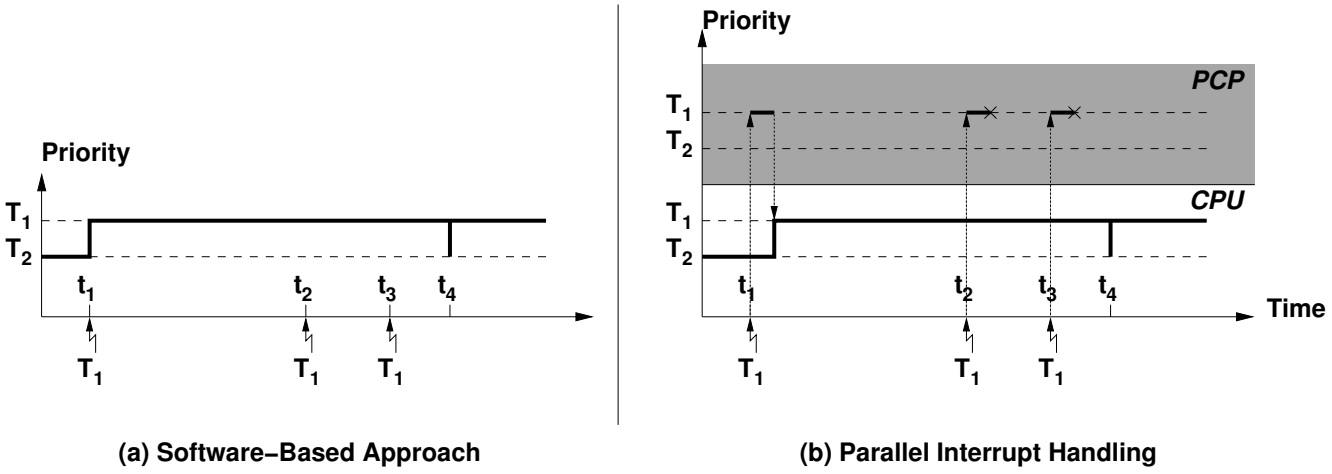


Figure 3: Example for Runtime Behavior of Multiple Task Activations

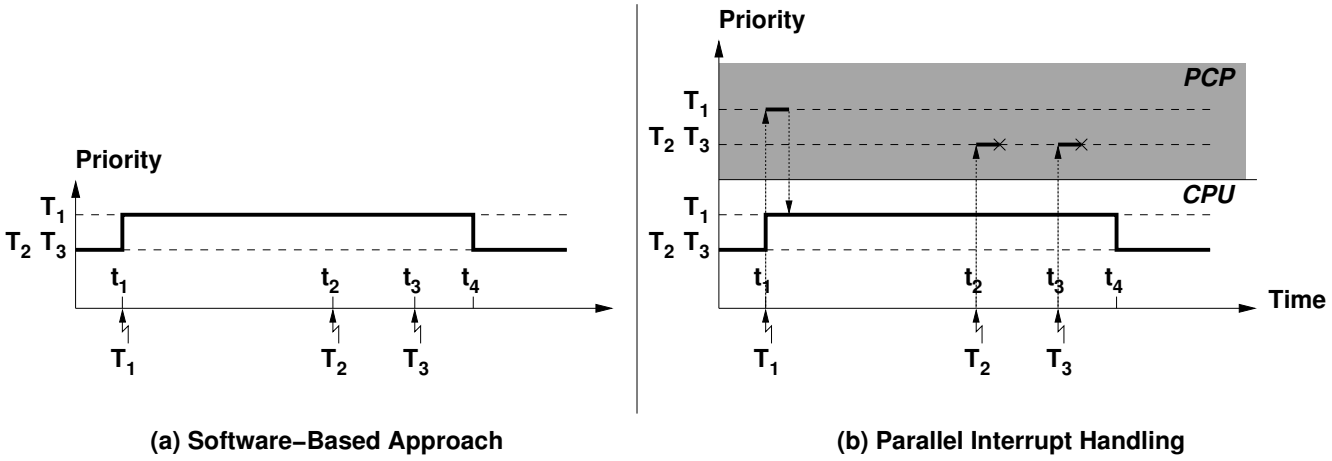


Figure 4: Example for Runtime Behavior of Shared Task Priorities

call has to be instrumented with the synchronization code), is kept in a concise and therefore maintainable piece of code.

3.2 The TriCore Platform

Our prototype is implemented on top of a TriCore TC1796 microcontroller, which, as previously mentioned, has all the features needed to realize our concept. The TriCore's peripheral control processor (PCP) provides a thread of execution in parallel to the main processor and can be used for interrupt processing. Inside the TriCore architecture, so-called service request nodes (SRNs) represent interrupt sources. Each SRN can be assigned to a service provider (either the CPU or the PCP) and be given an interrupt level. Thus, these SRNs can be configured in such a way that they implement an interrupt-leveling scheme directly in hardware as described in Section 2. Data exchange between the PCP and the main CPU is enabled by memory that is shared through a special bus; the pieces of memory are mapped differently in the two processing units, however.

We configured the system so that all external interrupts are signaled to the PCP instead of the main processor. The program running on the PCP and reacting directly to the interrupts is built according to our design explained in Section 2.1; it is detailed in the following subsection. The only interrupt that is still signaled to the CPU is used to trigger the AST. This mechanism is employed by the PCP to cause rescheduling on the main processor if a thread with a higher priority than the one of the currently executing thread is activated.

To attach the threads maintained by the scheduler to the interrupt sources and ISRs managed by the PCP, we use a table stored in shared memory to map threads to interrupt sources; this table is initialized upon system start-up.

3.3 The PCP–Interrupt-Handler Implementation

The implementation of the PCP interrupt handler is split into three different parts as sketched in Figure 2.

Since interrupt-source acknowledgment is performed automatically by the hardware on the TriCore platform, the prolog only consists of operations that are necessary to be able to subsequently alter the data structures inside the operating system. These encompass querying static data like the priority of the thread that belongs to the currently handled interrupt request or the addresses of core scheduler data structures like the ready list for threads. Since these are read-only operations on static data, this part is interruptible by both higher-priority PCP handlers and the main program flow running on the TriCore CPU.

During the first critical section, CS_1, the thread corresponding to the current interrupt's source is inserted into the ready queue of the scheduler. This part uses the variables queried during the prolog.

In the second critical section, CS_2, the PCP handler has to send a signal to the CPU if rescheduling becomes necessary. To determine that this is the case, it has to check if the newly activated thread has a higher priority than the one of the currently executing thread. In addition to that, it has to check that there is no higher-level thread in the ready list, because if a higher-level thread is already enqueued in the ready list, it will get scheduled in the near future—rising the priority level of the system to its level and rendering the AST useless. If neither of these criteria match and our thread in fact has the highest priority, the AST signal is sent to the main processor to trigger rescheduling. By performing the decision to send an AST and the actual notification in an atomic way, it is ensured that every AST sent really leads to a rescheduling on the main processor as no new threads can be activated until after the AST is triggered. This property thus completely prevents rate-monotonic priority inversion.

The critical sections CS_1 and CS_2 are guarded as described in Section 2.1. Interruptions by higher-priority PCP programs are avoided by simply disabling interrupts on the PCP until the program flow is past the critical section. To avoid concurrent modifications of the scheduler's data structures by the CPU, a spinlock implementing Peterson's algorithm is used.

4. EXPERIMENTAL EVALUATION

We conducted a detailed evaluation of our prototype that serves two purposes: On the one hand, we show that our prototype eliminates *unwanted* disturbance by low-priority and soft real-time ISRs, and on the other hand, we quantify the overhead that is caused by our implementation of the unified priority space.

4.1 Methodology

For our experiments, we use a TriBoard TC1796 by Infineon, which features the TriCore TC1796 microcontroller. It is clocked at 50 MHz, resulting in an instruction cycle of $20ns$. We furthermore use a Lauterbach TRACE32 hardware debugger and tracer to precisely determine all execution times on the CPU. The tracing facility, however, does not cover the PCP. Thus, for the measurements on the PCP, we toggle I/O pins and use a digital storage oscilloscope to determine the duration of the slopes at the I/O pins. We use a function generator to generate square-wave signals at various rates to mimic external events at the I/O pins of the TC1796. For our measurements, we exclusively use the internal, no-wait-state RAM of the TC1796 for both code and data.

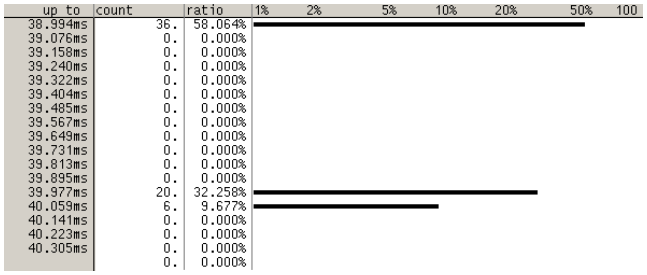
4.2 Behavior

The first part of our evaluation shows that our prototype indeed prevents unwanted disturbance on the CPU in the form of rate-monotonic priority inversion. Our synthetic sample task set is depicted in Table 1; it consists of one hard–real-time task T_1 and one soft–real-time task T_2 that can potentially disturb T_1. In one scenario, we implemented the soft–real-time task as an ISR in a traditional bifid priority space; in a second scenario, we implemented it as a thread using our parallel–interrupt-handling prototype.

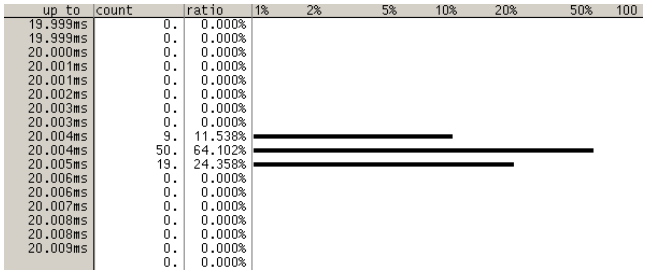
We measured the response time of the real-time task for both the classic interrupt-handling approach with a bifid priority space inside the operating system and for our implementation of a unified priority space for interrupts and tasks. We ensured that our prototype works as expected by examining the execution traces generated by the TRACE32 debugger, and, secondly, by measuring the response time of the task T_1, also with the help of the debugger.

Task	Period	Deadline	Execution Time
T_1	50 ms	50 ms	20 ms
T_2	2 ms	– (soft)	1 ms

Table 1: Sample Task Set for the Evaluation



(a) Traditional Interrupt Handling



(b) Parallel Interrupt Handling

Figure 5: Response Times of the Task T_1

The distribution of the obtained response times for a traditional interrupt-handling implementation and for our prototype are depicted in Figures 5 (a) and (b), respectively. It can clearly be seen that the response times are significantly higher in the traditional implementation and also show a noticeable jitter. These symptoms are caused by the disturbance of T_1 by the soft–real-time ISR T_2. The jitter would have been even bigger if we really had generated external events at a variable rate. For reasons of feasibility, however, we triggered the ISR with a constant period of 2 ms, as shown in Table 1.

In contrast to that, the parallel interrupt handling implemented in our prototype shows a much better performance. The low response times that are almost identical to the execution time of T_1 and the really tiny jitter (caused by measurement inaccuracy) indicate that T_1 is no longer disturbed by the soft–real-time task. We were also able to prove the absence of this disturbance in the aforementioned execution traces.

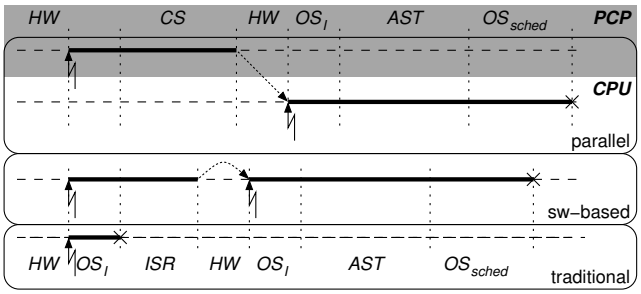
4.3 Overhead

In order to assess the costs of our approach, we quantified the overhead of our solution in contrast to traditional interrupt handling and software-based solutions by comparing the best-case and the worst-case latencies when reacting to external events. We mimiced the implementation of software-based solutions by a thread waiting for a semaphore; the semaphore is posted by an ISR that is triggered by an external event. The observed latencies are shown in Table 2.

The constituents of these latencies are depicted and explained in Figure 6. It should be noted that the hardware latency HW can vary slightly at all occurrences. However, we are not able to determine these latencies precisely, so the hardware latencies

	Best Case (bc)	**Worst Case** (wc)
Traditional	$0.900\mu s$	$6.660\mu s$
Software-Based	$9.062\mu s$	$14.822\mu s$
Parallel	$9.325\mu s$	$23.235\mu s$

Table 2: Event-Handler Latencies



HW	Hardware interrupt latency
OS_I	OS interrupt context management
OS_{sched}	OS scheduling and dispatching
ISR	Interrupt service routine
AST	Asynchronous system trap
CS	Critical sections CS_1 and CS_2

Figure 6: Constitution of the Event-Handler Latencies

are mentioned for illustrative reasons only. The best-case latency of the traditional interrupt handling only consists of interrupt context management performed by the operating system, and, thus, is much shorter than the latency of the software-based solution and our parallel interrupt handling. The software-based solution and our parallel–interrupt-handling implementation differ only in the interrupt handler that finally triggers the AST on the CPU. In the software-based implementation, this interrupt handler is executed on the CPU, whereas it is executed on the PCP for our parallel interrupt handling, leading to an average overhead of $263ns$ in the best-case event-handling latency.

The worst-case latencies depicted in Table 2 result from additional blocking an event handler can suffer. For the reason of a fair comparison, we assume that all implementations have to synchronize against the OS kernel to protect OS-internal data structures. The event handler in the traditional interrupt handling and the software-based approach can be blocked by at most one system call into the OS kernel. In the software-based approach, the AST itself cannot be blocked by the kernel because it is executed on the same processor; thus, this blocking time is determined by the longest system call. In the CiAO OS, the longest system call takes $OS_{max} = 5.760\mu s$, leading to the worst-case latencies $wc = bc + OS_{max}$ as given in Table 2. In the parallel interrupt handling, both the interrupt handler on the PCP and the AST might have to wait for the scheduler lock to become available. Thus, the event handler might be blocked by the kernel twice. As the interrupt handler on the PCP additionally has to synchronize against overlapping execution of other interrupt requests on the PCP (see Section 2.1), interrupts are locked throughout the critical sections CS_1 and CS_2, imposing an additional blocking time that is equivalent to the execution time of these critical sections $CS = 2.390\mu s$. All in all, this results in a worst-case latency of $wc = bc + 2 \cdot OS_{max} + CS = 23.235\mu s$.

If we had guarded CS_1 and CS_2 separately (see also Section 2.1), the interrupt handler on the PCP could additionally be blocked at the entrance of CS_2, yielding a worst-case latency of $wc = bc + 3 \cdot OS_{max} + CS = 28.995\mu s$. Thus, contiguously locking CS_1 and CS_2 improves the worst-case latency by $5.760\mu s$ while only minimally increasing the blocking time on the CPU (less than $2.390\mu s$).

On the other hand, a thread running on the CPU can be blocked by the PCP holding the spinlock during the critical sections CS_1 and CS_2. The maximum blocking time that can be suffered here sums up to $CS = 2.390\mu s$ and can occur at every system call issued by that thread. The relative performance impact of this blocking grows with the performance gap between the PCP and the CPU, of course. As a countermeasure, we kept the portion of code that is executed on the PCP as small as possible. Furthermore, the critical sections CS_1 and CS_2 could be guarded separately. That way, an improved blocking time on the CPU is traded for an increased maximum latency when reacting to external events.

Hence, in the best case, our solution imposes almost no overhead compared to a software-based solution, but it is much more flexible with respect to more sophisticated scheduler features as demanded by common OS specifications and the possibility for stack sharing, as discussed in Section 2.2. Due to increased synchronization demand between the PCP and the CPU, our parallel interrupt handling performs worse than a software-based solution in the worst case, but this still is an affordable price to pay if more sophisticated features are needed in combination with a predictable system behavior. Moreover, one should keep in mind that the worst-case–latency scenario of the parallel interrupt handling is rather unlikely to happen. It requires two closely following system calls, whereas in the software-based solution, one system call is sufficient to reach the worst-case latency.

5. RELATED WORK

The predictability issues introduced into real-time systems by interrupts are the target of a specific area of real-time–systems research. Many solutions try to ignore *unwanted* interrupt requests [2, 18]. For each interrupt source, parameters like minimum inter-arrival times, maximum arrival rates, or maximum burst lengths have to be specified; these parameters are then monitored during normal operation. Interrupt requests that do not fit within the given boundaries are treated as *unwanted* interrupts and are ignored. Such methods are definitely useful to prevent and bound overload situations caused by interrupts, but they do not eliminate the problems that are introduced by a bifid priority space. High-priority hard real-time tasks can still suffer disturbance by soft real-time tasks implemented as ISRs, for instance. Our solution itself is still susceptible to interrupt overload on the PCP to a certain degree. However, it is possible to additionally guard the *PCP* against such overload scenarios by implementing the techniques mentioned above *there*. The application itself, executed on the main processor, is automatically guarded against overload situations like these by the unified priority space. Since the scheduler controls every activity on the main processor, measures like the sporadic server [20] can be used to bound the time allocated to handlers of asynchronous events.

Asymmetric multiprocessing concepts have been used in the operating-systems community before, of course. Specialized coprocessors have also been used to support the operating system with certain tasks. The main target of those approaches is to maximize throughput and efficiency of the operating system, however. Popular applications are latency hiding in message-passing systems [3] or network-protocol offloading [11], for instance. An operating-system kernel that offers a solution related to ours is the Spring kernel by Stankovic and Ramamritham [21]. It uses an asymmetric-multiprocessing approach where so-called front-end processors in

a separated I/O subsystem are used to offload interrupt processing. That way, rate-monotonic priority inversion can effectively be avoided. The Spring kernel, however, is designed to execute tasks in a non-preemptive manner only. Thus, it is not suitable for many real-time systems—like implementations of AUTOSAR OS, for instance.

There are other approaches that try to aid the operating-system scheduler by using hardware abstractions; however, almost all of them rely on *customized hardware*. These approaches—including cs2 [15], FASTCHART [13], Silicon TRON [16], HW-RTOS [4], and Atalanta [22, 1]—move operating-system functionality to the hardware level by synthesizing special circuits on FPGA boards and offering that functionality on a co-processor–like basis. Some of these solutions may also prevent rate-monotonic priority inversion as a side effect, but none of them explicitly address this issue. In contrast, our approach does not implement dedicated scheduler functions in hardware; instead, we use *commodity off-the-shelf hardware* to implement a unified priority space (see also Section 6.3), thereby avoiding rate-monotonic priority inversion.

6. DISCUSSION

In this section, we discuss the implications of the related problem of interrupt-overload situations and how it affects our approach, the possibility to directly execute threads and ISRs on the PCP, as well as the general applicability of our solution.

6.1 Interrupt Overload

As already mentioned in the preceding section, our approach is still susceptible to interrupt-overload scenarios on the PCP—a high-priority interrupt could fully load the PCP. However, software-based approaches like the one presented in [5, 6] suffer from the same problem. In those approaches, a small ISR is needed to notify the task related to the particular event—this ISR could completely utilize the CPU.

A possible remedy for both approaches is to extend the sporadic-server algorithm to also affect the interrupt sources. Whenever the server's execution budget is exhausted, not only the scheduler has to delay further activations of this sporadic server, but also the interrupt sources triggering this sporadic server have to be reconfigured. Depending on the sporadic-server algorithm actually used, these interrupt sources have to be disabled or given a lower priority while the server has no execution budget. Thus, this task cannot consume more computing time via ISRs than the sporadic server is actually allotted, which effectively prevents interrupt overload.

6.2 Executing Threads and ISRs on the PCP

The PCP provides a control flow completely independent of the CPU, which gives rise to the idea to also execute threads or ISRs on the PCP. There are, however, two reasons that keep us from doing that.

First, executing threads on the PCP would induce rate-monotonic priority inversion on the PCP. Thus, this is not an option, as we want to use the PCP to *prevent* rate-monotonic priority inversion. Even the separation of hard and soft real-time tasks and a corresponding allocation of these tasks on separated cores (i.e., the CPU and the PCP) would not eliminate rate-monotonic priority inversion completely. While interruptions of soft real-time tasks can be tolerated, rate-monotonic priority inversion also arises for ISRs associated with hard real-time tasks having different priorities. Thus, this measure alone is not a solution for the problem of rate-monotonic priority inversion, but could be used to minimize inter-processor synchronization between the CPU and the PCP.

Second, the PCP has much less computing power in comparison to the CPU. It is designed for rather simple tasks like controlling DMA transfers or interrupt preprocessing, and for exactly that purpose we use the PCP.

6.3 Applicability

We have implemented our approach in the CiAO operating system on the Infineon-TriCore microcontroller. The concept itself, however, is applicable to a broad range of hardware architectures and OS kernels. First, almost every multicore-processor or multiprocessor system is suitable for an implementation of this concept, provided the priorities of the interrupt sources can be configured as needed. Most interrupt subsystems of microcontrollers used in the automotive area, for instance, offer such features. Second, there are other microcontrollers available that provide similar coprocessors comparable to the PCP of the TriCore architecture. Examples include the S12X microcontrollers and their XGATE coprocessor [7], or the MPC5510 microcontroller family and their I/O coprocessor [8] by Freescale.

Our current implementation itself, of course, is bound to the hardware actually used. The effort to port our concept to other architectures, however, is small. First of all, it is transparently implemented within the OS and does not affect the application itself. Secondly, only a few small hardware-dependent parts have to be ported: the interrupt handler executed on the coprocessor, signaling the CPU, and the initialization of peripheral hardware components. In our prototype, these parts sum up to only 187 lines of code. This number mainly comprises the PCP interrupt handler, which is written in assembly due to the lack of a C/C++ compiler.

Concerning the applicability to other OS kernels, any event-driven RTOS kernel is suitable to use the concept of parallel interrupt handling in order to maintain a unified priority space. The only prerequisite is a special interrupt that triggers rescheduling in the kernel, which runs on the main CPU.

7. CONCLUSION

We have shown that rate-monotonic priority inversion can be avoided by making use of a coprocessor as available on commodity embedded hardware like the Infineon-TriCore platform. Our hardware-supported approach shows significant conceptual advantages over previously suggested software-based solutions; these advantages include the possibility for stack-sharing, the support for multiple task activations, and multiple tasks per priority—as demanded by the AUTOSAR–embedded-OS specification, for instance. Furthermore, by evaluating an implementation of our approach for the CiAO operating system, we have shown that the concept completely avoids the disturbance caused by interrupts from lower-priority events. Our solution only bears a slight overhead, which is a small price to pay for the increased level of predictability that is vital to any real-time system.

8. REFERENCES

[1] Bilge E. S. Akgul, Vincent J. Mooney III, Henrik Thane, and Pramote Kuacharoen. Hardware support for priority inheritance. In *24th IEEE Int. Symp. on Real-Time Systems (RTSS '03)*, page 246, Washington, DC, USA, 2003. IEEE.

[2] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[3] Ulrich Brüning, Wolfgang K. Giloi, and Wolfgang Schröder-Preikschat. Latency hiding in message-passing architectures. In *8th Int. Symp. on Parallel Processing (IPPS '94)*, pages 704–709, Washington, DC, USA, 1994. IEEE.

[4] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*, pages 324–329, New York, NY, USA, 2006. ACM.

[5] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE.

[6] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In *12th IEEE Int. Conf. on Emb. and RT Computing Systems and App. (RTCSA '06)*, pages 385–394, Washington, DC, USA, 2006. IEEE.

[7] Freescale Semiconductor. *XGATE Block Guide 02.09*, November 2004.

[8] Freescale Semiconductor. *MPC5510 Family Product Brief*, September 2007.

[9] Micha Hofri. Proof of a mutual exclusion algorithm. *SIGOPS Oper. Syst. Rev.*, 24(1):18–22, 1990.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

[11] Hyong-Youb Kim and Scott Rixner. Connection handoff policies for TCP offload network interfaces. In *7th Symp. on OS Design and Implementation (OSDI '06)*, pages 293–306, Berkeley, CA, USA, 2006. USENIX.

[12] David Lake. Stack usage in computer-related operating systems. *US Patent No. 722543*, May 2007.

[13] Lennart Lindh and Frank Stanischewski. FASTCHART – a fast time deterministic CPU and hardware based real-time-kernel. In *Proceedings of the 1991 Euromicro Workshop on Real-Time Systems*, pages 36–40, Jun 1991.

[14] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX TC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[15] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 869–875, New York, NY, USA, 2004. ACM.

[16] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *Proceedings of the 12th TRON Project International Symposium (TRON '95)*, pages 34–42, Nov 1995.

[17] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-03-26.

[18] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *2005 Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, pages 50–58, New York, NY, USA, 2005. ACM.

[19] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[20] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[21] John A. Stankovic and Krithi Ramamritham. The Spring kernel: A new paradigm for real-time operating systems. *ACM OSR*, 27(3):54–71, July 1989.

[22] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical report, Georgia Institute of Technology, 2002.