Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor

Florian Kluge, Chenglong Yu, Jörg Mische, Sascha Uhrig, Theo Ungerer
Department of Compute Science
University of Augsburg
86159 Augsburg, Germany
{kluge, mische, uhrig, ungerer}@informatik.uni-augsburg.de

Abstract

The AUTOSAR specification provides a common standard for software development in the automotive domain. Its functional definition is based on the concept of single-threaded processors. Recent trends in embedded processors provide new possibilities for more powerful processors using parallel execution techniques like multithreading and multi-cores. We discuss the implementation of the AUTOSAR operating system interface on a modern simultaneous multithreaded (SMT) processor. Several problems in resource management arise when AUTOSAR tasks are executed concurrently on a multithreaded processor. Especially deadlocks, which should be averted through the priority ceiling protocol, can reoccur. We solve this problems by extending AUTOSAR OS by the Task Filtering Method *to avoid deadlocks in multithreaded processors. Other synchronisation problems arising through the parallel execution of tasks are solved through the use of lock-free data structures. In the end, we propose some extensions to the AUTOSAR specification so it can be used in software development for SMT processors. We develop some additional requirements on such SMT processors to enable the use of the* Task Filtering Method*. Our work gives also perspectives for software development on upcoming multi-core processors in the automotive domain.*

1. Introduction

Like in other domains, current trends in embedded systems target new hardware architectures, like simultaneous multi-threading (SMT) or multi-core processors. These processors utilise task-level parallelism. In comparison, hitherto single-threaded processors just allowed a quasi-parallel execution of applications by time-slicing the available processing time.

In the automotive domain, OSEK [17] and its successor AUTOSAR [2] define common interfaces for software development. The design of these standards is geared towards single-threaded hardware platforms. The use of such traditional programming models on upcoming hardware architecture inevitably leads to problems. The interfaces and programming models need to be enhanced in concert with the advancement of the hardware architectures.

Particular attention has to be paid to the fact that many embedded systems operate under hard real-time conditions. This implies some restrictions for operating system and application development. The software must be designed in a way that timing guarantees can be given, i.e. a worst case execution time (WCET) analysis must be possible. Parallel execution of tasks requires specific scheduling and resource management techniques to preserve WCET analysability of the tasks and to avoid blocking of higher priority tasks by lower priority tasks.

In this paper, we address the problems that arise when applying the concepts of current automotive software standards to upcoming processor architectures. Especially we investigate the problems of synchronisation and resource management in an SMT processor under real-time requirements. We introduce *task filtering* as a solution for the resource management problem in multithreaded processors. With our solution we are able the preserve the behaviour of AUTOSAR high priority tasks on an SMT processor and simultaneously gain some advantage through earlier execution of lower priority tasks.

The appearance of multi-core architectures in the automotive domain will pose similar problems for software developers. Applications will also be executed utilising a task-level parallelism on such processors. So our results can be conferred on these architectures as well.

In section 2 we give a description of AUTOSAR OS and the hardware architecture we are dealing with. Section 3 presents the mapping of the AUTOSAR OS scheduler to

an SMT processor and shows which problems arise in such an environment. In section 4 we present related work, followed by our own solution in section 5. In Section 6 we discuss our solution in more detail. Some evaluation results are presented in section 7. Section 8 concludes this paper.

2. Basics

2.1. AUTOSAR Scheduling and Resource Management

The AUTOSAR operating system [3] is mostly based on OSEK OS [18]. It provides common concepts like tasks, interrupts and resource management. These are manipulated using system services as an interface between the high-level application and the operating system.

In the following we describe the parts of these standards as far as they are relevant for our work. Naturally, the standards themselves are much more comprehensive.

2.1.1. Task Scheduling

The OS scheduler manages *tasks* as basic units. A task acts as a container for specific functionalities of an *application*. An application can be subdivided into several tasks. OSEK OS distinguishes two types of tasks. *Basic Tasks* can enter the states *suspended, ready* and *running*. *Extended Tasks* additionally have a *waiting* state. Furthermore, tasks can be defined as preemptive or non-preemptive. Once running, the latter can only be rescheduled in a very restricted set of cases.

OSEK OS defines an event-driven priority scheduling. Tasks are assigned static priorities, which cannot be changed by the user at runtime. Only in the particular case of resource accesses (see OSEK Priority Ceiling Protocol in 2.1.2) the operating system can temporarily assign a higher priority to a task.

Ready tasks are grouped by their priority in FIFO queues with the oldest task (the one activated first) at the front of each queue (see fig. 1). If rescheduling takes place, the scheduler regards all *ready* and *running* tasks. The oldest task with the highest priority is scheduled next. Thereby, the currently *running* task is treated as if it was at the front of its priority's FIFO queue. If a task has been preempted by a previous rescheduling, it is treated as the oldest task in the FIFO queue of its priority. A task that has left its *waiting* state is treated as the newest task of its priority and thus is put at the end of the *ready* queue.

Basic tasks have synchronisation points at their beginning and end. Extended tasks comprise additional synchronisation points when they are *waiting* for an event.

Rescheduling of a non-preemptive task can only take place if the task terminates itself successfully, explicitly calls the `schedule` system service, or waits for an event. Preemptive tasks additionally can be rescheduled through a task activation, by setting an event for another task, on releasing a resource, and on return from interrupt level to the task level.

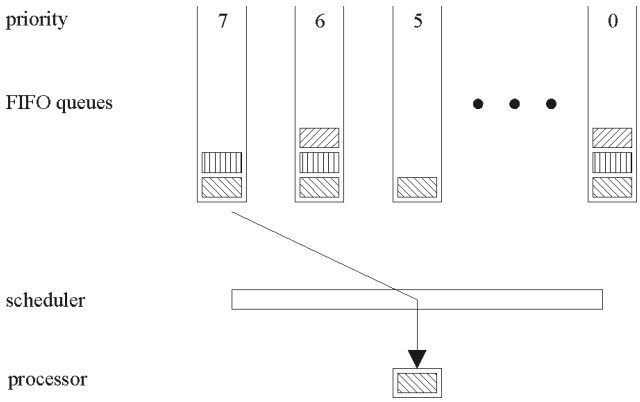


Figure 1. Scheduling model of AUTOSAR: The scheduler picks (arrow) the oldest and highest priority task for execution

2.1.2. Resource Management

The OSEK resource management coordinates the tasks' access to resources. Its main objectives are as follows:

1. two tasks cannot hold the same resource concurrently (mutual exclusion),
2. priority inversions and
3. deadlocks do not occur,
4. a resource access will never put a task into the *waiting* state.

To provide these properties, OSEK OS requires the implementation of the *OSEK Priority Ceiling Protocol* (OSEK PCP). It is based on the Priority Ceiling Protocol introduced by Sha et al. [21]. Thus, a predictable timing behaviour can be guaranteed at least for the highest priority task.

Each resource R is statically assigned a so-called *ceiling priority* CP_R. This ceiling priority is at least as high as the highest priority of any task accessing R. Simultaneously, it must be lower than the lowest priority of tasks that do not access the resource, but have a higher priority than the highest priority task accessing the resource. Each time a task is accessing a resource and its current priority is lower than the resource's ceiling priority, the task's priority is raised to the ceiling priority of this resource. Vice versa, if the task releases the resource, its previous priority is restored [18].

Thus, the OSEK PCP ensures that the attempt to get a resource will never result in blocking a task. This is because the task holding a resource will run at the ceiling priority and thus be executed before any other task that might access this resource. Deadlocks and priority inversion are anticipated efficiently by this approach (see fig. 3(a)). Mutual exclusion for accesses to a specific resource is also ensured by the OSEK PCP, as the task holding a resource cannot be preempted by any other task that might access this resource.

The task scheduler is treated as a special resource in OSEK. It can be accessed by all tasks in a system. Following the OSEK PCP definition, its ceiling priority thus will be the highest priority available in the system. Hence, if a task performs a rescheduling it will always run at highest priority.

AUTOSAR extends the OSEK scheduling interface by the definition of *schedule tables*, among others. These encapsulate a set of expiry points at which specific actions will be executed. The expiry points and the associated actions are statically configured.

2.2. SMT Processor Model

Simultaneous Multithreading (SMT) was introduced by Tullsen et al. [22, 6]. The basic idea of SMT is to issue instructions of different threads within one issue cycle into a superscalar pipeline (fig. 2). Thereby, a higher utilisation of the pipeline is achieved, as well as a higher throughput of multithreaded applications. In figure 2 we compare an SMT processor to a 2-way superscalar dual core processor. As can be seen, the SMT processor can reach a higher utilisation in the highest priority thread. Remaining issue slots can be used for other threads as well.

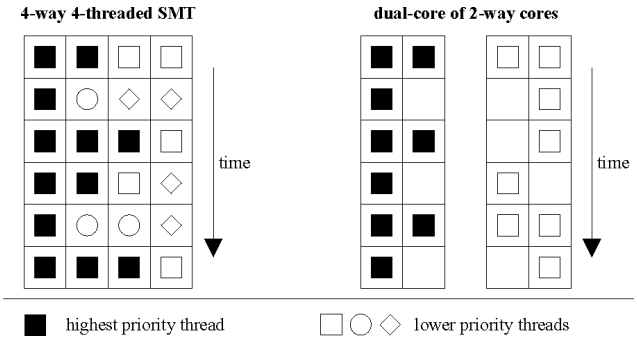


Figure 2. Instruction issue in an SMT and in a dual-core processor; the lower priority threads can issue instructions arbitrarily or due to their priority

For this work, we use an SMT architecture with the following properties: The processor is able to issue in each cycle up to i instructions from up to t threads (*thread slots*), i.e. an i-way t-threaded processor. Instructions are issued in-order. The thread slots are served due to priorities set by the operating system. If, due to dependencies or latencies, less than i instructions can be issued from the highest-priority thread, the issue stage tries to select further instructions from the next prior threads. Thus, time-predictability can be guaranteed for one hard real-time applications. WCET analysability of the application is preserved. The processor architecture must guarantee that the highest priority thread is never blocked by a lower priority thread.

3. Mapping AUTOSAR Scheduling onto an SMT Processor

3.1. Algorithmic Extension

The aim of this work is to preserve the real-time behaviour of the highest priority tasks in AUTOSAR OS. As an additional benefit we allow for an earlier execution of lower priority tasks as far as possible. For these tasks, like in AUTOSAR, no timing guarantees can be given.

The static priorities assigned to AUTOSAR tasks can easily be mapped onto the thread slot priorities of an SMT processor. If the processor provides less thread slots than tasks are in the *running* or *ready* state, the AUTOSAR system services related to task scheduling have to take care of filling the thread slots correctly. Therefore, scheduling for each slot is done separately. The scheduler selects the oldest and highest prior task as described in section 2.1.1, loads it into the thread slot and sets the slot's priority to the tasks priority. Thus, the highest priority task will always be executed preferentially.

This approach can lead to the execution of tasks of a specific priority although one or more tasks of a higher priority are still running. However, this will not change the system behaviour in any harmful way. There should not be any unmet dependencies between these tasks, if they are in the *ready* state at the same time. If one task depends on some results of another task, it must be activated by this task just when these dependencies are fulfilled.

3.2. Problems

However, the task-level parallelism rises new problems especially for the resource management. We will give an overview of these problems in the following.

Mutual Exclusion and the *Waiting* State One requirement for the resource management in AUTOSAR is that a resource access must never put a task into the *waiting*

state (see sect. 2.1.2). This is achieved by the OSEK Priority Ceiling Protocol, where a task accessing a resource is boosted above the maximum priority of all tasks that may access this resource for the time of the resource access. Thus, the task holding a resource is executed by all means before any other task possibly accesses this resource (cf. fig. 3(a)).

On an SMT processor, it can happen that one task is holding a resource and another task still running would need to access the same resource. This would have to result in the blocking of the second task because of the required mutual exclusion of accesses to the same resource. However, the AUTOSAR task model does not comprise such a state. Putting the task into the *waiting* state is not possible. On the one hand, the definition of the resource management forbids this explicitly. On the other hand, basic tasks do not even have a *waiting* state but are nevertheless allowed to access resources.

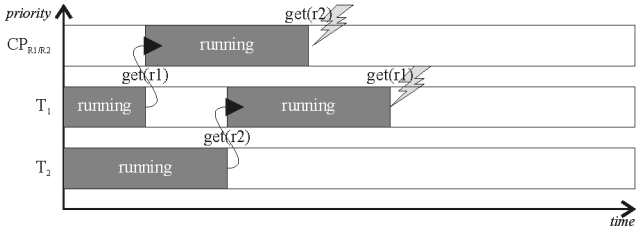
Also transitions into other states are not possible. Putting the blocked task into the *ready* state would result in the task being scheduled again immediately, due to the definition of this state. The *suspended* state can only be assumed through the termination of the task and thus also does not present a solution. Furthermore, if the blocked task is defined as non-preemptive, such a state transition would violate the non-preemptive scheduling policy.

Deadlocks Another point is the avoidance of deadlocks through the OSEK PCP. Here again, this only works on a single-threaded hardware (see fig. 3(a)). Let task T_1 access resources r_1 and r_2 nested, and task T_2 access the same resources in the reverse order. When T_1 has locked r_1, T_2 might still be running on an SMT processor. Now if T_2 locks r_2 in the meantime, both tasks will come to a point where they are trying to lock the resource held by each other (see fig. 3(b)).

As can be seen, the hitherto AUTOSAR standard using the OSEK PCP is not able to cope with these synchronisation problems occurring in multithreaded processors.

Access to Shared Data Structures Through the parallel execution of tasks it may happen, unlike in a single-threaded processor, that two tasks need to perform a rescheduling at the same time. Therefore, they need to access and especially alter the scheduler's FIFO queues containing the ready tasks. In a single-threaded processor this can never happen, because a task performing a rescheduling will automatically run with the highest priority. Hence, the access to the scheduler would need to be synchronised in some way for multithreaded task execution. Using mutual exclusion will not solve this problem, because in fact it would introduce a new *blocked* state for tasks and result in an unpredictable timing behaviour.

(a) Single-threaded: only one task running. Deadlocks are prevented by priority ceiling



(b) Multithreaded: two tasks sharing the processor can lead to a deadlock!

Figure 3. Resource accessed in a single- and multithreaded processor

4. Related Work

4.1. Resource Management for Multiprocessing Systems

Considerable work on synchronisation has been done for the area of multiprocessors. Rajkumar et al. [20] extended the priority ceiling protocol [21] for multiprocessor use (Multiprocessor Priority Ceiling Protocol, MPCP). The MPCP distinguishes between application and synchronisation processors. Tasks are statically assigned to an application processor. Local critical sections (LCS) are executed on the application processor according to the PCP. Synchronisation processors are responsible for the execution of a global critical section (GCS). However, nested locks of resources of different types are not possible, because the access to the global critical section can lead to long blocking periods.

The Multiprocessor Stack Resource Policy (MSRP) by Gai et al. [7] extends the SRP algorithm for single-threaded multiprocessors [4]. SRP is based on the priority ceiling protocol. Similar to the MPCP, resources are divided into LCS and GCS. However, MSRP executes GCSs locally on the respective processor. Nested resource occupation here is also forbidden, as it can lead to deadlocks.

The former two concepts were developed for their use in multiprocessor systems. Lo [15] proposed a solution for thread scheduling and resource management on a multithreaded processor. Processor performance is distributed on several logical processors (LP). While a thread T_1 on LP_1

holds a resource R, and a thread T_2 on LP_2 is blocked because of its access on the same resource R, the processing time of LP_2 is bestowed to LP_1. After T_1 has released R, this processing time is given back to LP_2. This approach is called *LP-Time Inheritance and Returning*.

The presented concepts use a static assignment of tasks to real processors or logical processor. If tasks are blocked by other tasks, their processing time is transferred temporarily. On our proposed SMT architecture this is not possible, because it exploits the parallelism on the instruction level. If tasks are blocked, the related processing time (issuing of one or more instructions in a clock cycle) is lost. Furthermore, AUTOSAR does not distinguish between local and global resources. Forbidding the use of nested locks like in [20, 7] presents a problem, as these are explicitly allowed by the OSEK OS specification. A dynamic stack management like proposed by Baker [4] also is not necessary, because each task is assigned its private stack statically to gain a higher performance for the targeted automotive applications.

4.2. Thread Synchronisation

An inherent problem in multithreaded environments is the synchronisation of concurrent threads that need to access shared data. Lamport [13] devised a solution called "Bakery Algorithm" extending Dijkstra's work [5] in this domain. Another solution was presented by Peterson [19]. However, both approaches use busy waiting to enter a critical section that is currently blocked by another thread. Such busy waiting will waste processing cycles that could be utilised otherwise by the blocking thread to finish the critical section earlier. Therefore, typical solutions for mutual exclusion suspend the execution of the blocked thread until the lock is free. As we have shown above, the AUTOSAR task model cannot cope with this.

The avoidance of mutual exclusion and blocking has been a research topic for many years. The first lock-free algorithm was presented by Lamport [14]. Further work by Herlihy [9] has shown the necessity for universal synchronisation primitives. These can be provided by a processor as an atomic COMPARE&SWAP instruction (CAS) or a LOAD LINK/STORE CONDITIONAL instruction pair (LL/SC). According to Michael [16], the CAS instruction can be emulated using the LL/SC pair. Valois [24] has presented an implementation of lock-free queues using the CAS primitive. Anderson et al. [1] have shown the benefits for real-time computing using lock-free synchronisation instead of mutual exclusion. These techniques can present a solution to some of the problems stated in section 3.2.

5. Task Filtering

To fulfil the requirements on a resource management like it is defined by AUTOSAR, we propose a mechanism of filtering tasks. For this purpose we extend the OSEK PCP and scheduling and thus adapt it for its use in multithreaded processors. Initially, we have to define some containers used by our algorithm and clarify some terms used in the following.

Definition 1 *(Containers)*

- *The* Execution Set E *contains all tasks that are currently scheduled into the SMT processor's thread slots.*
- *The* Active Set A *contains all active tasks (states* ready *and* running*), i.e. the tasks from all FIFO queues of the scheduler (cf. section 2.1.1).*
- *A task's* resource set R_T *comprises all resources that a task T will access during its execution.*
- *A resource's* task set A_R *references the active tasks that might access resource R ($T \in A_R \Leftrightarrow R \in R_T$).*
- *The* Task Filter TF *contains tasks, that are ready and should be executed according to their priority, but cannot be scheduled yet due to resource conflicts (see definition 2).*

Definition 2 *(Terms)*

- *Two tasks T_1, T_2 have a* resource conflict*, if they might use at least one resource R concurrently at runtime (i.e. $\{T_1, T_2\} \subset A_R$ resp. $R \in R_{T_1} \cap R_{T_2}$)*
- *A task T is ready concerning the resource management* (resource-ready)*, if*
  - *it does not need any resource, or*
  - *$\forall R \in R_T$: T has the highest priority in A_R*

Based on these premises, we are now able to decide which tasks shall be allowed to run:

Definition 3 *(Task Filtering)*
An active task $T \in A$ can be put into the execution set E, i.e. can be scheduled for execution, if it is resource-ready. All other tasks are in their FIFO queues.
If a task T_1 has a resource conflict with a running lower-priority task T_2, which cannot be preempted, T_1 is buffered in the task filter, until T_2 can be rescheduled.

We will give a short example, how the previous definitions will work. Let $A = \{T_1, T_2, T_3\}$ be the active tasks with descending priorities ($P_{T_1} > P_{T_2} > P_{T_3}$). Resources are used as follows:

- $R_{T_1} = \{R_1, R_2\}$
- $R_{T_2} = \{R_1\}$
- $R_{T_3} = \{\}$

Thus, the resources' task sets are

- $A_{R_1} = \{T_1, T_2\}$
- $A_{R_2} = \{T_1\}$

As can be seen, T_1 and T_2 have a resource conflict on R_1 (definition 2). Nevertheless, as T_1 has the higher priority in A_{R_1}, it is regarded as resource-ready and transferred into the execution set E. T_3 does not use any resources, so it can also be executed directly. Thus we get as execution set:

$$E = \{T_1, T_3\}$$

In the meantime, T_2 is buffered in the task filter TF until T_1 finishes operation.

With this approach, deadlocks are completely avoided. The filtering of the tasks ensures, that only such tasks are executed that do not have any resource conflict among each other, i.e. the execution set E is free of resource conflicts. For the same reason, no direct blocking can occur.

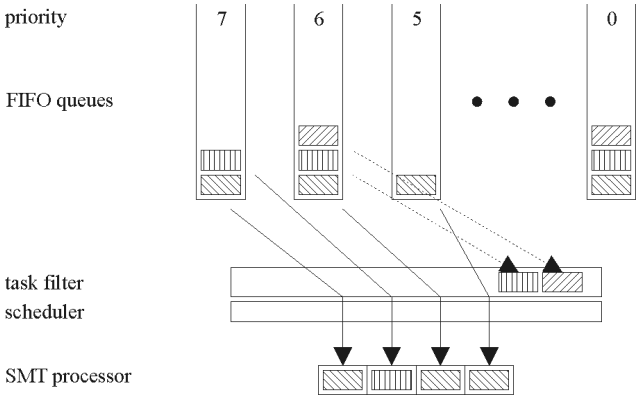


Figure 4. Extended scheduling model of AUTOSAR for an SMT processor. The task filter ensures that there are no resource conflicts.

Figure 4 shows a schematic representation of the scheduling mechanism. Between the FIFO queues and the scheduler, another layer was integrated. Here in the task filter all tasks are stored that currently have a resource conflict with other tasks running. Also, the figure shows that multiple tasks are scheduled simultaneously in the exemplary four thread slots.

Figure 5 gives an example of task filtering. The scheduler starts with the highest priority task T_1. As no resources are used yet, T_1 can be scheduled. T_2 does not use any resources, so it can also be executed. Now, T_3 has a conflict with T_1 on resource R_5. Thus, T_3 is retained in the task filter for later execution. In the given example, finally six tasks are ready for execution, whereas T_3 and T_7 will be kept in the task filter until the conflicts are resolved.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
R_1	X							
R_2			X			X		
R_3	X							
R_4				X				
R_5	X		X					
R_6					X		X	
R_7			X				X	X
R_8					X			
SC	**o**	**o**		**o**	**o**	**o**		**o**

Figure 5. Example. $P_{T_1} \geq P_{T_2} \geq \ldots \geq P_{T_8}$; task's resource usage is marked with an 'X'; the tasks marked with an 'o' can be executed simultaneously.

6. Discussion

In the following section we discuss the concept of task filtering in more detail. We will show that it does not change the external behaviour of an operating system implemented this way and thus the OS will stay AUTOSAR compatible. We also discuss the treatment of the scheduler as a special resource.

6.1. Preservation of Task Behaviour

As mentioned, the problems described in 3.2 are eliminated through the task filtering by avoiding resource conflicts between concurrently executing tasks. But it is also necessary to check the influences of task filtering on the execution order of tasks.

The AUTOSAR scheduling can be subsumed under these objectives:

1. higher priority tasks are executed before lower priority tasks: this means that the runtime behaviour of tasks is not influenced unpredictably by tasks that have a lower priority (except in the case of shared resources, see 4.);

2. older tasks are executed before younger tasks of the same priority: tasks are executed in their arrival order;

3. waiting for an event may result in the degradation of the task to the back of its priorities queue; the task "gets younger";

4. accesses to resources can boost a task's priority temporarily.

Objectives 1 and 2 are, basically, ensured through the definition of the scheduling method (see 3.1). By the interaction with the task filter it can happen, that a task T_2 will not be cleared for execution because it has a resource conflict with an older task T_1 of the same priority. In this case, the scheduler might put a *ready* task T_3 of lower priority into the execution set. When T_1 terminates, we can distinguish two cases:

- T_2 and T_3 have **no** resource conflict: T_2 will be scheduled immediately after T_1 terminates (if no task with higher priority is ready);
- T_2 and T_3 have a resource conflict on resource R. Here the scheduling policy of tasks (non-/preemptable) and objective 4 must be considered:
  - T_3 is non-preemptable: T_2 will be scheduled when T_3 reaches a synchronisation point (see 2.1.1). This possible blocking is predictable.
  - T_3 is preemptable and not holding R or any other resource with higher ceiling priority: T_3 is preempted and put into the *ready* state, T_2 is scheduled. The delay is predictable.
  - T_3 is preemptable and holding R: T_3 is running at CP_R, rescheduling will take place when T_3 releases R and does not hold a resource of higher ceiling priority. Else rescheduling is triggered when the priority of T_3 sinks below P_{T_2} after release of other resources. This delay is also predictable, but must be given special consideration in the design process.

  The given mechanisms will work analogous if T_2 and T_3 have a conflict in more than one resource.

Objective 3 results in a reordering of a FIFO queue through the event mechanism. However, this does not interact with the scheduling.

As we have shown, the externally observable behaviour of the scheduler will stay the same as when running AUTOSAR on a single-threaded processor, or even will be improved through the faster execution of lower priority tasks. Alone, for timing analysis of tasks it might be necessary to include some higher delays. This mostly affects the scheduling system service, which can also be called by other system services if rescheduling needs to take place. Here the task filtering will be performed resulting in a longer runtime. For this overhead, a worst-case upper bound is set by the maximum number of resources and active tasks according to the specification. A full timing and flow analysis after system integration could narrow this overhead.

The task filtering technique is based on the assumption, that in such an integrated system tasks exist with disjoint sets of resources they are using. If there are many tasks using the same resource, or the system does only provide one resource like in the OSEK conformance class 1, it can happen that all tasks will be executed as if running on a single-threaded processor.

6.2. Scheduling

In the original OSEK specification the scheduler is treated as a special resource and even the only one for systems of OSEK conformance class 1. Naturally, it can be used by all tasks. With the presented task filtering method this would not make sense because it would prevent any parallel execution of tasks.

The classification of the scheduler as a resource stems from the scheduler manipulating globally shared data structures, i.e. the FIFO queues containing the *ready* tasks. However, the inconsistency of these structures must not necessarily be prevented by mutual exclusion, like this is done through the OSEK specification. The FIFO queues could also be manipulated using non-blocking synchronisation techniques [24, 1]. Thus, the task model of OSEK and AUTOSAR will be preserved and the timing behaviour of the highest priority tasks will not change in any harmful way. This would only require the availability of hardware-implemented primitives, i.e. a COMPARE&SWAP instruction or the LOAD LINK/STORE CONDITIONAL pair (see section 4.2).

In some cases it may happen that one task activates another task that has a higher priority than all currently running tasks, but has a resource conflict with at least one running task. However, to conserve the timing behaviour this task should be scheduled as soon as possible. Therefore it is necessary to preempt the blocking task (if possible; see 6.1 above) and remove it from the processor, i.e. put it into the *ready* state. Just then the newly arrived task can be executed. From the hardware point of view this will need the possibility to raise an interrupt for another thread slot so that the task running there will carry out a rescheduling. After all conflicting tasks have been removed, the new task can start running. Then the OS has to try filling the free thread slots again.

7. Evaluation

To prove the practicability of the task filtering method, we performed a prototypical implementation of an AUTOSAR OS scheduler. This implementation was done on

the *CarCore* processor [23]. Its architecture is shown in figure 6. CarCore is a two-way four-threaded SMT processor designed for embedded real-time applications. It is binary compatible with the Infineon TriCore architecture [10, 11], which is widely used in the automotive domain. The processor comprises two pipelines, one for integer and one for address operations. The first stages Instruction Fetch and Schedule are shared between the two pipelines. A separate Instruction Window is assigned to each thread slot buffering instructions after fetching. Instructions are issued in-order. Two instructions of one thread can be issued in parallel, if an integer instruction is directly followed by an address instruction. Else, the other pipeline is filled by an instruction from another thread. For this purpose, the processor can manage up to four threads. The four thread slots are controlled by the Thread Manager. The Thread Manager sets the priorities for each thread slot in the Schedule stage, and also is responsible for swapping the executed threads from and to memory. Suspended threads are stored in a dedicated part of memory as so-called *Thread Control Blocks* (TCB).

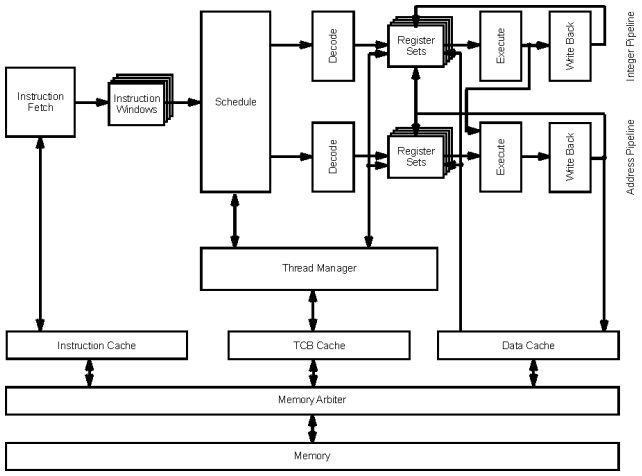


Figure 6. Architecture of the CarCore processor

In some earlier research [12] we found out that task switching in a single-threaded Infineon TriCore takes about 1400 clock cycles. The most expensive part here is the determination of the next task, which amounts to up to 900 clock cycles. 300 more cycles are needed to swap and adjust the OS management data. The remaining 200 cycles are spent loading the memory protection registers.

In the CarCore, context switching overhead from one thread slot to another is zero cycles. Swapping out a thread from one thread slot and loading another thread, i.e. saving all register values of one thread and restoring the register set of another takes about 50 clock cycles. Loading of the memory protection registers takes another less then 10 cycles. This work is mostly done by dedicated hardware and must only be initiated by the OS. While one thread slot is being swapped, the other slots are still running.

The task filtering brings in only a minor overhead. As the number of resources is limited by AUTOSAR to up to 16, the tasks' resource sets can be implemented using bit sets. The filtering thus will be implemented using bitwise boolean operations which are universally provided by processors. In our evaluations, selecting a new AUTOSAR task to be run within a thread slot took up to 1100 cycles including the task filtering. Thereto we have to add about 300 cycles for the adjustment of OS management data and 60 cycles for the context switch, i.e. the swapping of the register set and the memory protection registers. So, in total rescheduling of a thread slot will still take less then 1500 clock cycles. This is slightly more than on the single-threaded TriCore, but allows a multithreaded execution of AUTOSAR tasks.

If it is necessary to preempt other running tasks before starting a higher priority task, rescheduling may take some more time. However, this time is bounded as we already have shown in section 6.1, but depends on the application.

8. Conclusion

AUTOSAR OS was designed to run on a single-threaded hardware. We have shown which problems arise when trying to port the current AUTOSAR OS specification (Version 3.1.1, [3]) onto an SMT processor. Resource accesses of tasks can result in blocking other tasks during execution, which cannot be coped with by AUTOSAR's task model. AUTOSAR OS requires the implementation of a priority ceiling protocol, however, deadlocks can still occur when running on a multithreaded processor. To avert these problems, we proposed the *task filtering method*, where only such tasks are executed concurrently that do not have any resource conflict. Task filtering will keep the predictability of the integrated system. However, in some cases a WCET analysis will need to allow for increased timing tolerances.

To allow the use of AUTOSAR on SMT processors, our work proposes the following extensions of the specification:

- *Scheduling* is extended by the *Task Filtering Method* to prevent the concurrent execution of tasks that might use the same resources; thus, an internal task state *resource-ready* is introduced;
- the hardware must provide a primitive for *Non-Blocking Synchronisation*, e.g. a COMPARE&SWAP instruction or a LOAD LINK/STORE CONDITIONAL instruction pair;
- the hardware must allow a thread slot to *raise interrupts* in other thread slots.

Today's benchmark suites usually are designed to evaluate the raw computing power of a processor. To the best of our knowledge, they do not address the problems of thread concurrency and thread synchronisation. One of our future goals is to design a benchmark that explicitly allows to evaluate the behaviour of a processor running concurrent threads that access shared resources.

Also, we want to extend the scheduling data structures by the lock-free algorithms [24] presented in section 4.2. Thus we will be able to evaluate the complete overhead of the task filtering technique. The lock-free algorithms must be implemented in such a way that the WCET analysability of the code is retained.

With this work we also have taken a step toward embedded multi-core processors. Meanwhile, the usage of multi-core architectures is an inevitable trend in processor architecture even in the embedded area. Developers have to cope with similar problems when porting their software to such new processors. The task-level parallelism of program execution again poses problems of synchronisation. In the MERASA[1] project [8] we are developing an embedded multi-core processor for hard real-time applications. The results presented in this paper will be transferred to our new multi-core platform.

References

[1] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.

[2] AUTOSAR AUTomotive Open System ARchitecture. http://www.autosar.org/.

[3] AUTOSAR GbR. *AUTOSAR Specification of Operating System*, 3.1.1 edition, Feb. 2009.

[4] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[6] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[7] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *IEEE Real-Time Systems Symposium*, pages 73–83. IEEE Computer Society, 2001.

[8] M. Gerdes, J. Wolf, J. Zhang, S. Uhrig, and T. Ungerer. Multi-Core Architectures for Hard Real-Time Applications. In *ACACES 2008 Poster Abstracts*, L'Aquila, Italy, July 2008. Academia Press, Ghent (Belgium).

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[1] **M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability

[10] Infineon Technologies AG. *TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1*, Jan. 2008.

[11] Infineon Technologies AG. *TriCore 1 Architecture Volume 1: Instruction Set V1.3 & V1.3.1*, Jan. 2008.

[12] F. Kluge, J. Mische, S. Uhrig, T. Ungerer, and R. Zalman. Use of Helper Threads for OS Support in the Multithreaded Embedded TriCore 2 Processor. In C. Lu, editor, *Proceedings Work-In-Progress-Session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 25–27, Apr. 2007.

[13] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[14] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[15] S.-W. Lo. Data sharing protocols for smt processors. In H. Haddad, editor, *SAC*, pages 891–895. ACM, 2006.

[16] M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2004. ACM Press.

[17] OSEK VDX Portal. http://www.osek-vdx.org.

[18] OSEK group. *OSEK/VDX Operating System*, 2.2.3 edition.

[19] G. L. Peterson. A new solution to lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. Syst.*, 5(1):56–65, 1983.

[20] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269. IEEE Computer Society, 1988.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[22] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, New York, NY, USA, 1998. ACM.

[23] S. Uhrig, S. Maier, and T. Ungerer. Toward a Processor Core for Real-time Capable Autonomic Systems. In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, pages 19–22, Dec. 2005.

[24] J. D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994.