AUTOSAR OS on a Message-Passing Multicore Processor

Florian Kluge, Mike Gerdes, Theo Ungerer
Department of Computer Science, University of Augsburg, Germany
{kluge,gerdes,ungerer}@informatik.uni-augsburg.de

Abstract—The multicore era in automotive computing is just starting. Meanwhile, processor technology in high-performance domains is advancing from shared memory to message-passing multicore processors. In this paper we present an approach on implementing AUTOSAR OS on such a message-passing multicore processor. We show where AUTOSAR semantic can be preserved, and if not, point out solutions.

I. INTRODUCTION

The number of cores on a processor chip has been increasing over the last years. In the beginning, this has mostly affected general purpose and high performance computing. Over the last years, multicore technology has been entering also the domain of embedded real-time systems, e.g. in automotive systems. Today, multicore processors are already available for automotive body electronics [1] and safety-critical applications [2]. On the software side, also considerable work is being performed to make multicore processors work in automotive systems [3], [4]. While the multicore era for automotive is still at the beginning, a wider spread of multicore computing in automotive is to be expected.

Simultaneously, processor technology is advancing. In 2010 Intel has presented the Single-Chip Cloud Computer (SCC) which integrates 48 cores on one die [5]. In 2011 Tilera announced the TILE-Gx 3000 architecture with up to 100 cores [6]. The processing cores in these processors are no longer connected by a shared bus like in today’s multicores. Instead, they use a Network-on-Chip to exchange messages. The Intel SCC for example still provides a shared address space for application developers, but in hardware this is implemented based on message-passing. We think that in the long run such message-passing processors will be available also for embedded, and even hard real-time applications.

Several works on general purpose operating systems for multicore processors have already shown the performance impacts of shared memory and caches on the performance of multicore systems. Therefore, they propose to control sharing of data [7], or even relinquish shared memory at all and only use explicit messages for communication between tasks and/or operating system (OS) components [8], [9]. Concerning automotive hard-real-time applications, the mentioned performance impacts of resource sharing will have even stronger consequences. Hard real-time applications must be timing predictable, that is one must be able to calculate a worst-case execution time (WCET) bound that is never exceeded. The implicit sharing of resources can introduce much pessimism into a static WCET analysis, which will result in higher WCET estimations, while a measurement-based analysis is unlikely to observe all possible interferences between threads [10].

A solution for these problems is to remove dependencies that result from the sharing of resources. Primarily this means to use only explicit messages for communication between processes. As Baumann et al. [8] already pointed out, the theoretical foundations to analyse such a system already exist, e.g., Hoare’s communicating sequential processes. WCET analyses would also be eased and made more accurate, as the pessimism introduced through implicit sharing of resources would be removed. In this context, we think that basing an automotive computer system on such a modularised and distributed architecture will be a remarkable option.

Switching over from today’s shared memory programming model to a message-passing one cannot be accomplished in a single step. A vast amount of code is existing in automotive industry, parts of which might be reused for years. At least for some time, it must be possible to deploy this software within the new programming environment with as less changes as possible. In our work, we aim to port the AUTOSAR interface to a message-passing architecture, while preserving as much as possible of the original semantics.

In this paper, we examine the part of AUTOSAR OS [11] that it inherited from OSEK OS [12], but also take the multicore-specific parts of the specification into consideration. We present a general approach how these specifications can be implemented on a message-passing processor. We estimate the impacts of our approach on the WCET bounds of system services and point out solutions for where the original semantics of system services cannot be preserved.

The remainder of this paper is organised as follows: In section II we describe the software architecture we are proposing. In section III we present the concepts our implementation is based on. In section IV we discuss our approach under the aspects of WCET analysis, synchronisation and shared memory. Section V concludes this paper and gives an outlook on future work.

II. SOFTWARE ARCHITECTURE

The AUTOSAR OS [11] specification requires for multicores that most of the AUTOSAR Basic Software (BSW) is executed on only one core (called master core), but together with application code. Other cores can execute only a limited set of services. If an application running on such a core has to use a basic software service, the call might be delayed by the applications running on the master core. We want to
advance this concept by dedicating single cores to execute OS services exclusively. The basic concepts of our approach are inspired by the factored operating system (fos) [9], which is targeting cloud computing and does not account for real-time requirements. We employ this approach to achieve high locality of code and data also in real-time systems, and thus improve WCET estimations.

AUTOSAR OS defines OS Applications as containers for tasks and associated OS objects like counters, schedule tables etc. [11]. Figure 1 shows two such applications, each consisting of three tasks; the OS objects are combined in the box labelled OS. We deploy these applications onto a message-passing multicore chip by mapping each task statically to a separate core. All OS objects of an application are mapped to another core acting as OS-Server. Thus, each AUTOSAR OS application is represented by a cluster of cores. Furthermore, we allocate cores as I/O-servers. These I/O-servers are shared by all applications running on the processor. Locality is achieved in our approach in several places:

1) Each core executes only one task.
2) OS and BSW components are executed on dedicated cores.
3) All I/O operations are performed on separate cores. This means that only limited portions of code must be loaded to the cores respectively reside in the local memories. Insofar, we extend the concepts of AUTOSAR OS, which already achieves some locality by putting most of the AUTOSAR software stack on one core. However, we aim also at decoupling I/O operations, and to remove inter-application dependencies by segmenting the AUTOSAR OS and BSW between applications. Communication between the cores strictly follows the message-passing paradigm.

Figure 2 depicts the generic software architecture. A basic abstraction from the hardware is provided by a Hardware Abstraction Layer on each node. This could be an extension of the AUTOSAR MicroController Abstraction Layer (MCAL). On task nodes, a small BSW wrapper provides the AUTOSAR functional interface. The wrapper translates the function calls into messages that are sent to the relevant servers. OS/BSW-Servers execute the AUTOSAR software stack except for external I/O, which is handled by the I/O servers.

III. IMPLEMENTATION CONCEPTS

Task node, OS/BSW and I/O server are implemented as control loops. They are designed in a way such that data times of tasks calling OS/BSW or I/O services are bounded. We present their internal structure in more detail in the following section. Finally, we discuss the limitations of our approach.

A. Task Node

The basic structure of a task node is depicted in algorithm 1. After system start, a task node waits for an activation message from its BSW server. If such a message arrives, it starts executing its task that was statically assigned during system integration. When the task has finished its work, it calls the TerminateTask service (line 4). Thereafter, the task node is ready for another activation.

Algorithm 1 Task node structure
1: while true do
2:    Wait for activation message from BSW
3:    Task Execution
4: TerminateTask \(\triangleright \) part of task!
5: end while

Services provided by the OS/BSW and I/O servers are implemented as wrapper functions on a task node. These functions provide the interface specified by AUTOSAR. Internally, they send service request messages to the relevant servers and wait for reply. In the meantime, the calling task is blocked.

B. I/O Server

The control loop of an I/O server for processing tasks’ I/O requests is kept quite simple. The server waits in a loop for a message from any task. On receipt, the server processes this request and performs the I/O operation. Finally, it sends a reply message to the task and waits for the next request. Incoming requests can be processed either in FIFO order, or they can be reordered following client priorities. Then, however, it must be ensured that waiting times of clients are still bounded, e.g. by using an appropriate synchronisation protocol like the Multiprocessor Priority Ceiling Protocol [13].

The I/O server is also responsible for handling interrupts originating from its associated I/O devices. So, at least parts of Interrupt Service Routines (ISR) will be executed on the I/O server. Further interrupt processing must be performed by tasks running on other nodes to keep the I/O server’s worst-case latency low.
C. OS/BSW Server

Each OS/BSW server manages several task nodes that are assigned statically. The server holds all data that is relevant for managing the tasks, e.g. states and events of tasks. Algorithm 2 depicts the general functionality of the OS/BSW server control loop. After receiving a service request, the server performs some local processing. Generally, this involves error checking and ensuring that the operation does not fail. For the OS services we have investigated so far, this work does not require any long-latency interaction with other nodes, as all OS and task management data is kept on the server. Thus, the server can immediately send a return value to the task node (line 4). Just then, the server interacts with other nodes and process the request to completion.

We demonstrate the functionality of the OS/BSW server using the ActivateTask system service as an example. The service implementation is depicted in algorithm 3. If the server receives an ActivateTask(t) request, it checks the activation conditions for task t. This work does not require any further interaction, as the server fully keeps track about the tasks’ states. Thus, it can directly send a reply to the caller. The following work does not involve the caller anymore. The server checks, whether task t is currently suspended. If so, it sends an activation message to the core task t resides on. Else, if task t is already running, the activation is queued to be sent later, after the current instance of task t has terminated.

Algorithm 2 OS/BSW Server main loop

1: while true do
2:   Receive request
3:   Local message processing for reply
4:   Send reply ▷ if necessary/possible
5:   Interaction/process to completion ▷ if necessary
6: end while

D. Constrictions

The semantics of the event management are completely preserved. The same holds for the services that are used for alarms. Task management required some changes. The Schedule system service no longer has any meaning, as a task holds its core exclusively. For the GetTaskState service, it is important to note that the return value may be inaccurate. The task, whose state is queried, might have terminated while the GetTaskState service was processed by the OS/BSW server. Finally, the GetTaskID service can be implemented without a server request.

For the resource management services GetResource and ReleaseResource, it is not possible to preserve the semantics. As discussed in [14], when dealing with real parallelism, the priority ceiling protocol defined for these services will not work properly. AUTOSAR OS [11] accommodates this fact by explicitly forbidding resource sharing across cores. Inter-core synchronisation is performed by using spinlocks in shared memory. In our work we abandon shared memory, so the spinlock approach will not work either.

However, our approach removes partially the need for the resource concept, namely when it is used to synchronise accesses to shared I/O resources. Instead, I/O operations are performed by dedicated I/O servers which give timing guarantees (see III-B). We discuss the synchronisation of accesses to shared data structures in section IV-B.

IV. Discussion

A. Worst-Case Execution Time

We are especially interested in the influence of the client-server model on the WCET estimates of system calls. Let $W_T$ be the worst-case transport time of a message between a task node and its OS/BSW server, $W_N(c)$ the processing time on the task node, and $W_S(c)$ the server-side WCET of the system call c. Then, the basic WCET for a call c would sum up to $W(c) = 2W_T + W_N(c) + W_S(c)$. However, one OS/BSW server handles $n > 1$ task nodes. For the worst-case, we have to assume interferences through just previously issued system calls of the other task nodes. There will be at most $n-1$ calls pending from other tasks, as a system call blocks the tasks until the return message is received. Be $W_S(C) \geq W_S(c)$ the server-side WCET of the most complex system call, then

$$W(c) \leq 2W_T + W_N(c) + W_S(c) + (n - 1)W_S(C) \quad (1)$$

It is obvious that $W(c)$ is dominated by possible interferences from other tasks. The main objective is to reduce the pessimism introduced from these interferences. This, however, can only be done with knowledge about the concrete applications that are deployed to the processor. Just then, it will be possible to derive constraints about when and which application will be using the OS/BSW server. Concerning the I/O servers, equation 1 applies analogously if FIFO processing is used. Else, a WCET estimation must use the timing properties of the synchronisation protocol that is implemented by the I/O servers.

B. Synchronisation and Shared Memory

In a multitasking system like AUTOSAR it is sometimes necessary to synchronise accesses to shared resources, i.e. I/O devices and data structures in a shared memory. As stated above, the resource concept of AUTOSAR works only within a singlethreaded processor. The AUTOSAR OS therefore defines
spinlocks for inter-core synchronisation. However, these come with several drawbacks. They are deadlock-prone, and they can result in contention of the memory interface through spinning. Although spinlocks can be implemented such that waiting times can be bounded [15], [16], using them on a message-passing processor is not feasible.

Concerning I/O devices, we propose the use of dedicated I/O servers that handle all I/O requests on a processor. A similar approach can also be implemented for shared data structures, where these are managed by dedicated servers. When porting an application, accesses to shared data must be changed to special service requests.

However, we also see another possibility to overcome the stated problems. While at least at the moment the AUTOSAR OS specification explicitly requires shared memory, it also introduces the concept of message-passing between OS applications. Therefore it defines the Inter-OS-Application Communicator (IOC) as means for explicit communication. We take this as a hint that the developers of the standard are aware of the problems that shared memory poses for multicore real-time systems. So another way could be to extend the IOC for inter-task communication. This would require some re-thinking of the program code, as state must be kept consistent. Some clues on how to achieve such consistency are given by Baumann et al. [8]. Another possibility, targeting the shared memory problem, is discussed in the next section.

C. Efficiency

Today’s automotive ECUs sometimes comprise over hundred tasks. However, only few of these tasks are running concurrently. The OSEK OS specification explicitly limits the number of tasks that are not in suspended state to 16. Although we expect core numbers to increase, we assume that at least for some time the number of tasks will be higher than the number of cores available for task execution. Insofar, our approach still needs an extension towards multitasking on task nodes. Furthermore, this might alleviate the problem of shared memory. It would be possible to put tightly interacting tasks onto the same node where they would share the core’s local memory or cache. This however would introduce again the need for core-local schedulers and the associated problems.

V. Conclusions & Future Work

To evaluate the feasibility of our concepts, we have started an implementation of AUTOSAR OS for a message-passing multicore. We use the multicore simulator presented in [17] as execution platform. The simulated cores implement the Open RISC instruction set architecture [18]. So far, we ported basic components, which AUTOSAR OS inherited from OSEK OS, for a message-passing multicore. In this paper, we have shown the concepts that have guided our implementation. We have also shown the impacts of the processor architecture on the WCET of system services and the programming model of AUTOSAR OS.

In the future, we want to complete our implementation, and also overcome problems that come with the message-passing processor architecture. We plan to extend our approach according to the AUTOSAR OS specification by putting tightly coupled tasks on the same node. Thus, these tasks would again be able to use a local memory for fast sharing of data, and the node utilisation could be increased. Therefore, we plan to define guidelines about when to integrate tasks onto one core. Also, we aim to reduce the high pessimism in the WCET estimation, which we identify as a major drawback of our approach. We plan to achieve this by examining real automotive applications and deriving constraints about the usage of OS/BSW services.

REFERENCES

tion, Nov. 2011, version 5.0.0, Release 4.0 Rev 3.