

From Single-Core to Multi-Core Platforms - Systematic Migration of Hard Real-Time Software in AUTOSAR

Peter Gliwa
Gliwa GmbH
Munich, Germany
peter@gliwa.com

Jens Harnisch, Ursula Kelling
Infineon Technologies AG
Munich, Germany
Jens.Harnisch@infineon.com,
Ursula.Kelling@infineon.com

Christoph Ficek
Symtavision GmbH
Braunschweig, Germany
ficek@symtavision.com

1. Introduction

The performance requirements of embedded applications in the automotive domain are increasing, for instance to enable a more precise engine control and reduce gasoline consumption or simply to reduce costs by integrating distributed controllers into one ECU. On the other hand growing EMC problems, thermal issues and high design costs for systems with high clock rates lead to multi-core approaches, to reach the desired features at reasonable costs. The necessary migration of software from single-core to multi-core systems raises several questions, for instance:

- How shall the user partition and map software, using the existing hardware resources efficiently?
- What are actually the metrics for a successful transition from a single-core application to a multi-core application?
- How to analyze runtime behavior, and how is it affected, are deadlines still fulfilled by the software?
- Which overhead is introduced to synchronize the control flow, to exchange data and to protect shared data?

The traditional approach is to modify software in small steps, and continuously check results. However, when porting software to multi-core architectures small steps may not always be feasible:

The controller architecture is not only changed in terms of additional cores, but also in terms of supported clock rates, memory architecture, communication architecture or instructions of the cores. Run time may change considerably.

The new architecture may require new compiler versions and operating systems, again possibly influencing the run time behavior.

A major issue in this complex scenario is the communication overhead between software parts on

different cores. These overheads can reduce the expected performance significantly. Mastering this complex scenario leads to a model based approach, where hardware and software components are abstracted and represented with their timing behavior. The description of a corresponding model based framework and the results of applying it onto a typical automotive real time profile, based on AUTOSAR, are the focus of this paper.

2. Investigated hardware architectures

This paper makes some assumptions about the hardware architecture, though the methodology may probably be used for other architectures as well. Those assumptions are:

- The cores have local fast memories, which are globally accessible.
- The cores have access to shared memories (both RAM and Flash) as well.
- Bus architectures are duplicated, hence, the cores can communicate to memories in parallel, without conflicts, unless the cores are communicating with the same memory block (slave).
- In case the cores are communicating with the same memory block, there will be some hardware arbitration necessary. However, the time needed for the hardware arbitration is not considered explicitly in the modeling approach described in this paper.

3. Communication overheads in AUTOSAR

AUTOSAR 4 introduces a mechanism for multi-core systems. One important feature is the *Inter OS-Application Communication* (IOC) which allows the communication between the different cores. However, this also introduces overheads to the system. A risk at this point is that the expected performance boost of a multi-core hardware is eaten up by these overheads.

This paper concentrates on the exchange of data; data has a specific size and it is placed in a specific memory of the hardware. Depending on the hardware architecture, the memories may have different access costs. Local memories, directly coupled to a core, are usually particularly fast, but also rather limited in size. In contrast, the access from one core to the local memory of another core (remote access) can be expensive. Access to shared memories may be the slowest, due to bus latencies and additionally because the memory itself is slower. Of course, programmer would prefer one large shared memory, without the need to consider memory access times. However, for embedded multi-core systems the need to use the different memories with care may even increase.

When mapping runnables to the cores, the following needs to be considered:

- Amount of data to be exchanged between runnables running on different cores
- Overhead introduced by hardware and operating system for data exchange
- System scheduling overhead

4. A generic communication benchmark

This paper suggests to incorporate the results of a specific communication benchmark. The probably most known benchmark for embedded multicore architectures has been developed by the EEMBC organization. This benchmark is measuring the system performance (hardware architecture, operating system, and compiler) in complex scenarios, and is well suited to compare architectures, concerning scalability, memory and IO performance, dedicated communication and scheduling support. The benchmark used in this paper is much simpler; it is only measuring the costs for communication in a multi-core system, with or without involvement of an operating system. The intention of our benchmark is to help the user to find a good mapping of his software specifically on Infineon multicore microcontrollers.

The benchmark measures:

- Costs for accessing (read and write) different data types (1, 2 and 4 byte) and different data sizes (multiples of the basic data types) in different memories (local and global) from the cores (without OS involvement)

- Costs for using read and write functions of the AUTOSAR RTE, again for different data sizes
- Costs for triggering events

The costs are measured in a scenario without conflicts on bus architectures and without conflicts for accesses from different masters to a single slave. This is certainly a strong assumption. On the other hand, the Infineon multicore microcontroller architecture comes with duplicated communication and memory components; hence, many conflicts can be avoided by a reasonable mapping of software. Of course not all conflicts at run time can be avoided, sometimes arbitration at the slaves (in particular memories) may be necessary. The user can of course apply those increased communication costs, by manually modifying the results of the communication benchmark, before importing them into SymT/AS.

The user can always decide which compiler or AUTOSAR version to use. Gliwa GmbH, Infineon and Symtavision defined the communication benchmark in a joined project. Gliwa GmbH developed the setup for actually running the communication benchmark and measuring reliable timing data.

5. Communication overhead analysis

This methodology concentrates on the application layer of AUTOSAR systems and the RTE communication methods. Details of the BSW are abstracted and not directly modeled currently. Section 3 described three factors, which need to be considered when mapping runnables to cores:

- Amount of data to be exchanged between runnables running on different cores
- Overhead introduced by hardware and operating system for data exchange
- System scheduling overhead

To cover these factors, three metrics are introduced. The metrics extend each other. This helps to make first simple steps in the development, even with a small set of data, and ends up in a “full-blown” analysis covering a complete scheduling analysis.

5.1. Data rate metric

The information about the dependencies between the runnables and the data is used in a pure data rate

metric. This shows the amount of data, which is exchanged between different runnables on the same or on different cores.

This metric uses the activation pattern of the runnables, the data access count during the execution and the variable size to calculate the necessary data rate per variable and runnable. The variables can already be mapped to memories or not, the data rates do not depend on this.

Data Rate Result					
Variable	Access Type	Variable Accessor	Accessing Core	Data rate	
1 Var11_INT_20 DataRate	READ	R6	Core2	6400.0 bit/s	
2 Var11_INT_20 DataRate	WRITE	INT_1	Core1	64000.0 bit/s	
3 Var10_100_100 DataRate	READ	R10	Core2	80.0 bit/s	
4 Var10_100_100 DataRate	WRITE	R11	Core2	80.0 bit/s	
5 Var9_50_100 DataRate	WRITE	R9	Core2	160.0 bit/s	
6 Var9_50_100 DataRate	READ	R11	Core2	80.0 bit/s	
7 Var8_10_50 DataRate	READ	R8	Core2	1280.0 bit/s	

Figure 1: Data rate results (screenshot SymTA/S)

Example results are shown in Figure 1, a screenshot from SymTA/S. In line one of the table, the variable *Var11_INT_20* is accessed by the runnable *R6*, which is mapped on *Core2*. The access type is a *READ* and the produced data rate for this access is *6400bit/s*. The same variable is also accessed by *INT_1* (line two in the screenshot), which is placed on *Core1*. The data rate for this access is *64000bit/s*. Therefore, the variable should be placed close to *Core1*.

These data rates give a first hint how the communication will behave in the system. From here, first decisions about the variable to memory mapping can be made. Data should always be put into the memory of the core, which is accessing the data the most.

5.2. Com-overheads with costs metric

The data rate metric is extended by information about the variable to memory mapping and the cost catalogue to quantify each access. The outcome is the overhead time necessary for each access of data, which is influenced by the hardware and the operating system. The overhead times are calculated to (processing) loads per core for each variable. This makes different mappings well comparable with only the load value.

Variable Load Result					
Buffer	Memory	Mapped on Core	Accessing Core	Load	
1 Var11_INT_20 Core1 load	Var11_INT_20	DML_Core2	Core1	0.0852	
2 Var11_INT_20 Core2 load	Var11_INT_20	DML_Core2	Core2	0.0062475	
3 Var10_100_100 Core2 load	Var10_100_100	DML_Core2	Core2	0.0012155	
4 Var9_50_100 Core2 load	Var9_50_100	DML_Core2	Core2	0.00...175	
5 Var8_10_50 Core2 load	Var8_10_50	DML_Core2	Core2	0.0080495	
6 Var7_10_10 Core2 load	Var7_10_10	DML_Core2	Core2	0.012155	
7 Var6_10_100 Core2 load	Var6_10_100	DML_Core2	Core2	0.005627	

Figure 2: Variable load result per core (screenshot from SymTA/S)

Figure 2 shows the results in a table. All variables (row one “*Buffer*”) and the memories (row two) and cores (row 3) to which they are mapped are shown. Row 4 and 5 show how much load on the accessing core is produced to access this variable. For example in line one, the variable *Var11_INT_20* mapped on a memory *DML_Core2*, which is a local memory of *Core2*, is accessed by *Core1*. This access produces a load of 0.0852 (8.52%) on *Core1*. This means that ~8% of processor time on *Core1* is only used to access this variable.

The communication costs depend on whether a variable is read or written, by which mechanism is the variable accessed (RTE, IOC or pure access), in which memory type is it placed and the variable size. The cost catalogue provides this information for different data sizes. Larger data is split into several transfers. The overhead time considers the operating system effect and overheads for each possible communication.

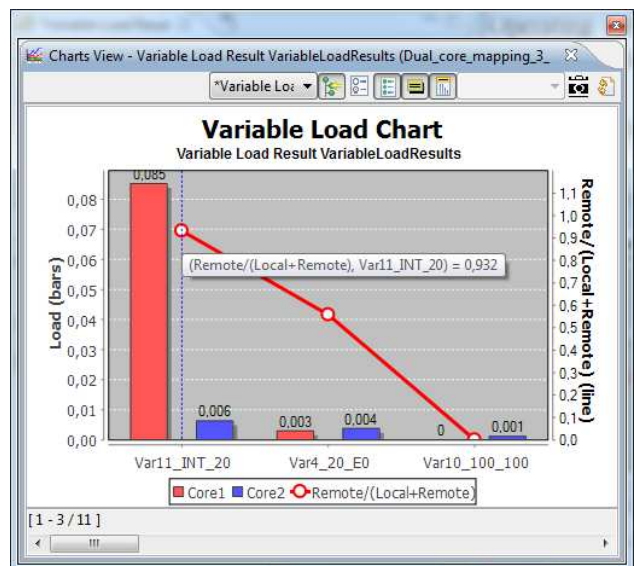


Figure 3: Variable load results in a chart (screenshot from SymTA/S)

Figure 3 shows the values from Figure 2 in a chart for a better overview. It shows how much load is produced on which core (the bars) by accessing a specific variable. (variable names along the x-axis). Furthermore, in the diagram the line shows a quotient of the load, indicating the quality of a mapping.

The quotient only exists for variables, which are mapped to a local memory of one core and is defined as:

$$Q_i = \frac{\text{remote load}}{\text{remote load} + \text{local load}} \in [0..1] \quad (1)$$

The *local load* is the overhead load produced on the core to which the variable is mapped. The *remote load* is the overhead load produced by all other cores. The highest and worst value is 1 and reached if the remote core only accesses a variable. This is the worst mapping situation. The value 0 is reached, if the variable is only accessed by the core to which it is mapped. This is the best situation. Clearly, variables with a quotient value of one should be remapped to another memory. For variables, which are mapped to shared memories, the quotient is not defined, because no local load exists.

In the shown example, the quotient for variable *VAR11_INT_20* is 0,932. This high value indicates, that the variable should be remapped, because the remote core is accessing the variable much more than the local core.

This metric and diagram makes different mapping situations comparable and helps to find a mapping with reduced overheads. Future work will develop a heuristic approach to find the best mapping automatically.

5.3. Scheduling analysis

The presented metrics helps to quantify the overheads in the system and makes it easier to make decisions about the runnable to core and variable to memory mapping. However, the overall system timing must be also proved to check if timing constraints like deadlines are violated, especially by the introduction of the overheads. For this, the overheads for each runnable are added to the WCET of the runnable and a full scheduling analysis is executed in SymTA/S and shows how loaded the cores are and if constraints like deadlines are missed.

6. Methodology

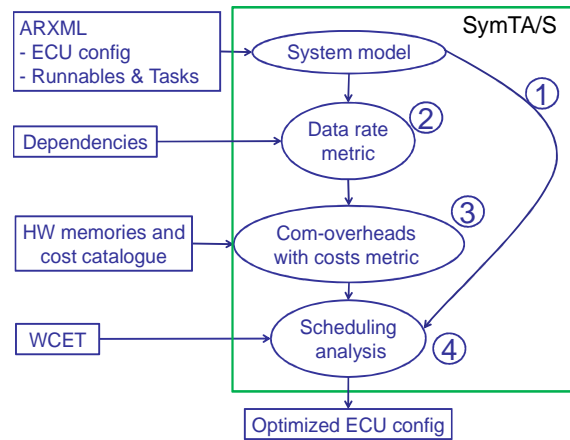


Figure 4: Methodology overview (ARXML means AUTOSAR XML)

Figure 4 shows the implemented methodology. Information on the software structure information is provided in an AUTOSAR XML (ARXML) file. For a full AUTOSAR multi-core system description, as of today AutoSAR 4.x needs to be used, as AutoSAR 3.2 only supports single core systems. However, this approach intends to help the migration from single-core to multi-core and so the starting point can also be an AUTOSAR 3.x single-core system. From the ARXML at least the information about the runnables, the runnables to task mapping and the activation patterns, e.g. periods, is needed. With this information, a system model is created.

The model-based approach allows fast changes in the system and can be used to verify different configurations and answer “what-if” questions.

After importing the data into SymTA/S, a first mapping of SWCs to the different cores needs to be made under functional aspects. For example, in some hardware architectures not every core may be allowed to access a certain peripheral.

With the system model and the WCETs, a scheduling analysis is directly possible (number 1 in Figure 4) and the state-of-the-art solution with SymTA/S.

Number 2 in Figure 4 is the data rate metric. The metric gives a first hint how big data rates in the system are and which memories (shared, local) should be assigned to which data.

Number 3 in Figure 4 is the Com-overhead with costs metric. With the help of this metric, the mapping of the variables to runnables can be checked. If the metric indicates a bad mapping, two options exist:

1. Remap variables to other memories

2. Remap runnables to other cores

Number 3 in Figure 4 is the scheduling analysis with the additional information about the overheads. Further optimization for better performance and better load balancing is possible by modifying the system model. If timing constraints are violated, the system needs to be changed. For example, a runnable needs to be remapped to another core, for better load balance. In most cases, iterations are necessary to find a well-structured and sufficiently optimized system. However, with the help of the model-based approach, it is possible to change and analyze the system quickly, hence the efforts are decreased.

7. Scheduling Analysis with SymTA/S

SymTA/S is a tool suite offered by Symtvision for scheduling analysis of complex embedded systems. SymTA/S supports OSEK and AUTOSAR schedulers as well as End-to-End analysis over busses like CAN or FlexRay.

8. Required data

The presented analysis methods need different sets of input data. The requirements are shown in Table 1.

	SW structure	Dependencies (R2R Com)	HW memories and Cost catalogue	WCET for runnables
Data rate metric	X	X		
Com-overheads with costs metric	X	X	X	
Scheduling analysis with overhead extension	X	X	X	X

Table 1: Metric requirements for input data R2R Com: Runnable to Runnable communication; WCET: Worst-case execution time

SW structure shall denote the information about the runnables in the system, the runnable to task mapping (RTE configuration) and information about the runnable and task activation, e.g. periods.

Dependencies (Runnable to runnable communication) are necessary for all methods. It must be known which runnable is reading or writing data and how often this is done during one runnable execution. Furthermore, the size of the data must be provided.

The information about the runnable dependencies can be determined in different ways. If a Software Component (SWC) is developed from scratch, then the information about the communication between runnables is mostly available from the development process. If this data is not available, the information can be generated by the compiler or by the usage of a code analyzer like CIL (C Intermediate language) [2]. A further possibility is to run an analysis at run time. There are advantages and disadvantages for each of the methods.

The hardware memory structure and the cost catalogue provide the information how “expensive” an access to a specific memory is. The cost catalogue is determined by the generic communication benchmark, described in section 4.

To prove the overall timing constraints of a system by the scheduling analysis, the WCETs (worst-case execution times) for all runnables must be provided. Those can be determined by static code analysis as provided by AbsInt [1], from time budgets in early design stages or from measurements on the real target hardware.

9. Conclusions and outlook

This paper suggests a methodology for mapping software to a multi-core system, having costs for communication overhead in mind. A central idea is to define a generic communication cost catalogue. Furthermore, the methodology assumes that it is not necessarily the objective to have as much idle time as possible on the cores, but simply to fulfill scheduling requirements.

One of the possible next steps is to work on heuristics for automatic adaption of some parameters, for instance variable mapping or priority setting.

References

- [1] AbsInt Angewandte Informatik, <http://www.absint.com/>
- [2] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, R. Nigel Horspool (Ed.). Springer-Verlag, London, UK, 213-228.