AN INTRODUCTION TO AUTOMOTIVE

Multi-core

EMBEDDED SOFTWARE TIMING

01 INTRODUCTION

A fable of parallel processing:

- Imagine you want to have a kitchen built in one day (~ 8 hours).
- You ask a craftsman to do it but he says: "It will take me **16 hours**."
- So you might hire a **second one** in order to get the job done in time.
- BUT: while one craftsman connects the electric items (and therefore takes the fuses out), the other one cannot use his power tools **and is blocked**.
- **They also spend a lot of the time talking to each other**.
- They finish after 11 hours (completely stressed out) and you agree to **plan next time**.

This poster sheds a light on automotive multi-core embedded software timing aspects. Proper multi-core know-how helps to avoid software projects running into situations as described above.

02 WHY MULTI-CORE?

Multi-core processors have been used for decades in domains other than automotive. Every PC and every smartphone comes with at least a dual-core processor. The main reason for using more than one core within the processor is the ever increasing need for more computation power. [1] Moore's law – stated 1965 – says: "The number of transistors in a dense integrated circuit doubles approximately every two years."

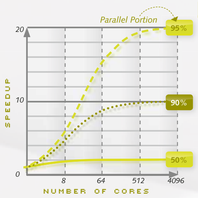
Good reasons for more computing power include:

- More and more advanced vehicle features (zero-emission, autonomous driving, car-to-X communication, etc.)
- Stricter safety requirements (diverse computing, memory protection, on-target supervision, etc.)
- Increasing use of standards and generated code limits the scope of optimization

Building faster (higher clock-speed f) single-core processors becomes too expensive at some point due to the following reasons.

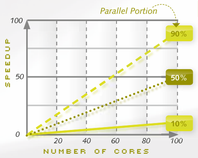
- Power consumption: P ~ f³ (limiting-case)
- EMC (Electromagnetic compatibility) problems
- Power dissipation → "Melting dashboard"

03 MULTI-CORE THEORY



Amdahl's law:
"The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program." [2]
In other words:
It takes a woman nine month to carry a child to term. Nine women are not going to do it in a month.

Amdahl's law applies when there is a significant portion of code which cannot be parallelized.

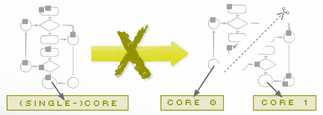


Gustafson's law:
"Programmers tend to set the size of problems to use the available equipment to solve problems within a practical fixed time. Therefore, if faster (more parallel) equipment is available, larger problems can be solved in the same time." [3]

Gustafson's law applies when a given problem can be replaced by a bigger problem solving the old problem plus other problems.

Amdahl's law and Gustafson's law seem to contradict. Which one applies to automotive projects?

- Automotive multi-core projects are mostly successors of existing single-core projects.
  - Single-core projects typically come with a great share of sequential code.
  - Thus, Amdahl's law is more appropriate.



- Automotive multi-core projects come with a defined set of features.
  - The "problem" has a fixed size. In other words: we are not going to add just any code in order to increase the throughput of the cores.
  - Thus, Gustafson's law does not apply.

-> **Amdahl's law matches the automotive situation better**, limiting the speed increase that we can realistically expect with multi-core processors.

04 MULTI-CORE HARDWARE ARCHITECTURES

Heterogeneous multi-core processors have different cores of different types.
Examples:

- Infineon TC1797 (TC1.3.1 and PCP)
- Freescale MPC5xx with TPU
- Freescale S12X with XGATE
- Infineon TC277 (several different cores, see next section)

Homogeneous multi-core processors have a number of cores of the same type.
Examples:

- Freescale MPC5xxx
- Infineon TC277 (two TC1.6P cores, see next section)

Lock-step multi-core processors execute the same **single-core software** on two separate cores at the same time, for safety reasons. The results of the two cores get continuously compared by the hardware. When a mismatch (=error) occurs, the processor can switch to a safe state.
Chip designers spend a lot of effort to avoid common mode failures: slight execution delay between the cores, separate clock-trees, rotated and flipped 2nd CPU, potential guard ring around each CPU, etc. [4]
Example: Texas Instruments TMS570, Infineon AURIX™ (TC1.6.1 core with checker core, see next section)

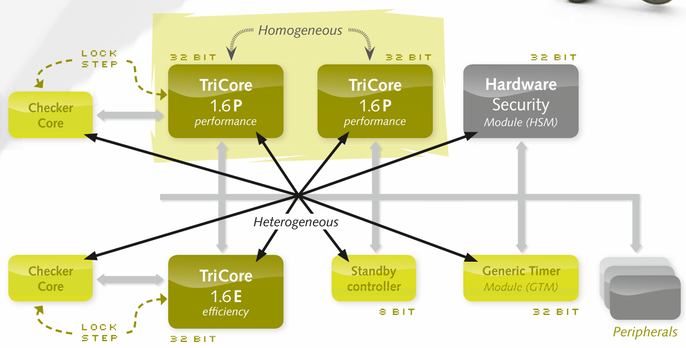
05 EXAMPLE INFINEON TC27X "AURIX™" [5]

- *Three main processing cores: two homogeneous (⟶) 1.6P "performance" cores and one 1.6E "efficiency" core. Since all three share the same instruction set, you could also regard them as three homogeneous TC1.6.1 cores.*
- *Two TC1.6.1 cores have an additional lock-step (⇢) core.*
- *There are several other heterogeneous (⟶) cores.*

Each TriCore has local program memory and local data memory that it can access with no delay. With significant delay (up to 5 CPU stall cycles), each TriCore can also access data/program memory of other cores, see also section "09 Cloning".

Accesses to peripherals "cost" up to 4 or 7 CPU stall cycles depending on the peripheral bus configuration.

The shared program flash and the shared data flash cause a maximum of (5 + number of wait-states) CPU stall cycles [6]. These numbers show that location of data and code has a significant impact on the timing.

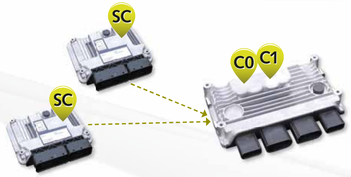


06 DIFFERENT KINDS OF PARALLELISM

The term "parallelism" refers to two or more fragments of a program being executed at the same time on several cores. Parallelism can take place at different levels.

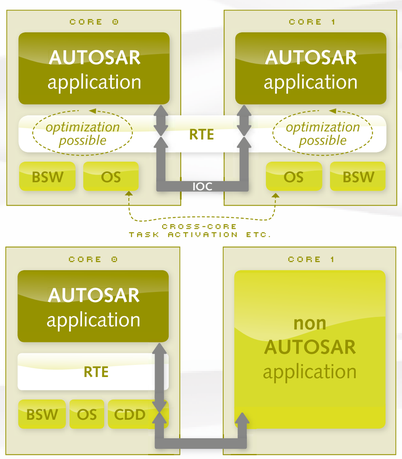
6.1 APPLICATION PARALLELISM

Each application runs on one core only. One core can still handle more than one application though. The applications come with low cohesion i.e. they are largely independent.



Example 1: To reduce costs, two single-core ECUs are merged into one dual-core (= multi-core) ECU. With **application parallelism**, the software of each single-core ECU gets its own dedicated core on the multi-core ECU.

Example 2: AUTOSAR concept [7]. Each core comes with its own set of TASKs and ISRs i.e. its own AUTOSAR application.
The AUTOSAR OS allows e.g. cross-core TASK activations and inter-core communication through the IOC (Inter-OS-Application Communicator). It explicitly requires data to be copied and thus might be inefficient for large data. Communication through the RTE can be optimized (e.g. direct accesses instead of working on copies) as long as it is intra-core communication.



Example 3: When migrating a single-core application to multi-core, one sensible approach is to have "AUTOSAR cores" and "non-AUTOSAR cores". This approach is used with early AUTOSAR standards that do not support multi-core. The AUTOSAR software communicates with the non-AUTOSAR software via complex device drivers (CDD). A non-AUTOSAR core could, for example, handle time-critical and/or very frequent interrupts, reducing the number of cache misses and pipeline stalls. There is no need for complicated function parallelism.

6.2 FUNCTION PARALLELISM

Function parallelism executes closely related fragments (with potentially high cohesion) of an application in parallel. In order to find/design suitable fragments, dependencies have to be analyzed/specified

- DFA (data-flow-analysis)
- Execution order constraints

Function parallelism is largely absent in Windows/Linux/Mac software, mobile devices etc. These use **application parallelism** mainly! There are very few examples of successful function parallelism and these include 3D rendering software, mainframe database software, computationally intensive scientific software at research institutes and universities, etc.

Fragmenting software so that it supports function parallelism is not easy and, when done poorly, can result in massive use of protection mechanisms like spinlocks, with a negative impact on the overall performance. As Amdahl's law shows, the benefit does not scale with the number of cores!

6.3 INSTRUCTION PARALLELISM

Processor cores have pipelines which process typically 4 to 7 instructions in parallel. However, just because you have a pipeline does not mean you exploit instruction parallelism, which relies on being able to fetch enough instructions to fill the pipeline.
The following techniques can reduce flow changes that stall the pipeline and so they support efficient instruction parallelism:

- inline function calls (use at least macros where inlining is not possible)
- fewer interrupts (use polling where applicable)
- instruction reordering (optimization performed by the compiler)

07 AUTOSAR AND MULTI-CORE

AUTOSAR originally was designed for single-core processors but has been extended with a number of multi-core features.

- Starting and shutting down other cores
- Cross-core task activation and task chaining (however task-migration is not supported and also not expected)
- Spinlocks ("cross-core semaphore", explained later)
- IOC (Inter-OS-Application Communicator)

AUTOSAR does not (yet) support

- AUTOSAR RTE optimization across cores (unnecessary resource locks can be optimized away on a single-core system but unnecessary spinlocks cannot be optimized away on multi-core systems)
- Inter-core data-passing by reference (currently copying data is mandatory which becomes an issue when dealing with large data)

AUTOSAR
GLIWA GmbH is an AUTOSAR development member

08 DATA-CONSISTENCY, SPINLOCKS

Whilst a single-core application can use interrupt locking to ensure data-consistency, this is not sufficient for multi-core systems sharing data between cores. A command "disable all interrupts" only affects the core executing the command. AUTOSAR introduces spinlocks for synchronization in multi-core systems.
Example: assume an application has two, frequent interrupts and it needs to know the total number of executions of both interrupts.

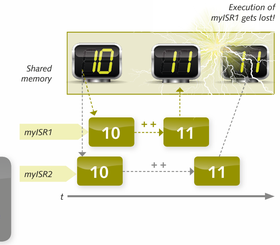
Without any protection, data-consistency cannot be guaranteed.

a) Both interrupts get executed on a **single core**.

```
ISR (myISR<x>)
{
  DisableAllInterrupts();
  counter++;
  EnableAllInterrupts();
}
```

b) Both interrupts get executed on **different cores.**

```
ISR (myISR<x>)
{
  GetSpinlock(spinlock);
  counter++;
  ReleaseSpinlock(spinlock);
}
```



The spinlock related AUTOSAR services are:

```
StatusType  ReleaseSpinlock (SpinlockIdType SpinlockId);
StatusType     GetSpinlock (SpinlockIdType SpinlockId);
StatusType TryToGetSpinlock (SpinlockIdType SpinlockId,
                             TryToGetSpinlockType* Success);
```

- ReleaseSpinlock releases a spinlock. Obtained spinlocks must be released in the correct order, the last obtained spinlock must be released first.
- GetSpinlock obtains a spinlock when no other core is using it. If another core is using it then GetSpinlock loops (spins) until the spinlock can be correctly obtained.
- TryToGetSpinlock is a non-blocking version of GetSpinlock. It always returns immediately with no spinning.

The straight-forward implementation shown in Example (b) is rarely suitable for real applications and can cause significant, unintended delays when one core occupies a spinlock and then handles one or more interrupts. A better implementation is shown below and can be used as a design pattern for spinlock-usage.

```
TryToGetSpinlockType success;
DisableOSInterrupts( );
( void )TryToGetSpinlock( spinlock, &success );
while( TRYTOGETSPINLOCK_NOSUCCESS == success )
{
   EnableOSInterrupts( );
   /* Allow preemption. Optionally insert delays to reduce
   the number of memory conflicts caused by TryToGetSpinlock. */
   DisableOSInterrupts( );
   ( void )TryToGetSpinlock( spinlock, &success );
}
/* Region with spinlock obtained and interrupts disabled. */
... /* do what you need to do with spinlock obtained */
ReleaseSpinlock( );
EnableOSInterrupts( );
```

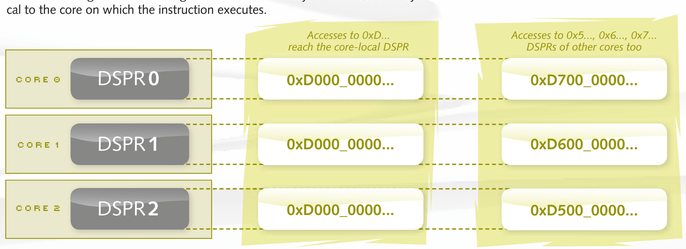
09 CLONING

Cloning is a very powerful concept. On the one hand it allows unmodified, single-core software to execute correctly on different cores at the same time. On the other hand, it provides an easy way to create efficient, dedicated, multi-core software guaranteed safe from certain kinds of data access conflict.

With cloning, all cores see their own, local memory at the same start address, e.g. 0xD0000000 for the DSPR (data scratch pad RAM) of the AURIX™ TriCores. These memories have the same addresses (overlaying) but can have different contents and are, in some sense, **clones**. Any load or store instruction using this address range accesses the memory local to the core on which the instruction executes.

Existing, single-core software with internal data can be executed simultaneously by each core as each core uses its own copy of internal data. No modification of the code is required, we simply locate the internal data in the cloned address range. Processors not supporting cloning have to allocate an array rather than a single variable and, at run-time, have to get the core identifier and access the corresponding array element, if they running the same code on different cores.

The AURIX™ additionally maps each DSPR address onto the linear shared address-space (mirroring) so that each core can also access the DSPR of other cores, although memory protection may be used to limit cross-core accesses.



10 WHY DOES MULTI-CORE SEEM TO BE SO DIFFICULT?

Multi-core is the standard in many other domains and the parallel paradigm is rather old, very well understood and not really complicated. So how can it be that so many automotive projects seem to struggle with multi-core?

Other domains mostly use **application parallelism** and in most cases, the software has always been organized in **threads**. Such applications can easily be ported from single-core to multi-core because parallelism has been made explicit in the threading architecture and the multi-core complexity can be devolved to the OS.

The application is not even aware of the number of cores it runs on and there is no attempt to guarantee improved performance on a multi-core processor.

Automotive engineers additionally want **function parallelism**, even if they are not aware of this and the resulting impact. Their "old", single-core application is not designed for multi-core and their code generators do not indicate any inherent parallelism in the code.

11 GOLDEN RULES

The golden rules for creating simple, easy-to-develop and efficient multi-core software are:

- Statically allocate code and data to cores so that you can analyze and optimize that allocation
- Localize code and data on one core to minimize cross-core accesses
  - Duplicate data and code where appropriate to achieve this goal
- De-couple code on different cores
  - Remember that atomically accessed data (typically up to 64 bits) can be accessed by one writer and 'n' readers with no synchronization
- Schedule
  - Reduce the number of conflicts by synchronizing the schedule across all (related) cores and using offsets so that tasks that access shared data run at mutually exclusive, or at least different, times
- Where synchronization is unavoidable, understand the relevant mechanisms
  - Spinlocks for very short delays
  - Spinlock with delay for longer delay
  - Task switch when even longer delays are expected

12 SUMMARY

As of today (2015) neither the AUTOSAR standard nor the code-generators exploit the multi-core potential to a high degree. The first and most important step towards successful multi-core projects is a sound understanding of multi-core aspects. With this, developers will learn two things:
There is no "silver bullet" that allows legacy designs to suddenly exploit parallel processing.
Exploiting the great potential of multi-core performance requires parallelism to be designed in from the ground up and support from a range of tools to predict and validate timing effects.With proper understanding of multi-core aspects and the right tools, it is possible and very worthwhile to pursue multi-core designs.
As with the introduction of other fundamental technologies (compilers, code-generators), the period of transition requires extra know-how and brings some discomfort.
Before long, we can expect ubiquitous multi-core support, including the AUTOSAR standards and code-generators. Complex, single-core projects will be the exception and will be regarded fondly as antiques.

13 GLOSSARY

ABBR.	EXPANSION
AMP|Asymmetric multiprocessing	*A multi-core system with a separate operating system per core.*
AURIX™	*Infineon family of multi-core processors based on up to three TriCore CPUs.*
Cohesion	*Degree to interdependency (data, code and control flow) within a given software component.*
Coupling	*Degree to interdependency (data, code and control flow) between different software components.*
IOC	*Inter-OS-Application Communicator. Part of the AUTOSAR OS responsible for managing communications from one OS-Application to another and, by implication, from one core to another.*
ISR	*Interrupt Service Routine. A short piece software that executes sequentially to handle an interrupt.*
Multi-core	*Having more than one core in a processor. With no explicit qualifiers it generally implies homogeneous multi-core.*
Non-blocking	*An implementation of some kind of communication that is guaranteed not to block.*
OS	*Operating System. An ambiguous term used either to mean just a (multitasking) kernel or the combination of a kernel and low-level support software, such as device drivers. The AUTOSAR OS is just a kernel.*
OS-Application	*AUTOSAR term for a collection of application software. More than one OS-Application can run on one core but an OS-Application cannot span more than one core.*
Pipeline	*Set of processing stages for handling a sequence of data items. As soon as the first data item is passed from the first state to the second stage, the first stage can start to process the second item, introducing true parallelism.*
Spinlock	*Mechanism for achieving mutual exclusion. AUTOSAR uses spinlocks for mutual exclusion across multiple cores in the same processor.*
Symmetric Multiprocessor System (SMP)	*A multi-core system operating under a single operating system with two or more homogeneous cores.*
TASK	*Collection of software that executes sequentially and often, but not necessarily periodically.*

14 REFERENCES

[1] Dr. Karsten Schmidt, Rolf Schneider, André Kohn, Sven Schönberg et al.: *Efficient Virtualization for Functional Integration on Modern Microcontrollers in Safety-Relevant Domains, SAE 2014, Detroit, Jan. ,14*
[2] WIKIPEDIA: *en.wikipedia.org/wiki/Moore%27s_law*
[3] WIKIPEDIA: *en.wikipedia.org/wiki/Gustafson%27s_law*
[4] Texas Instruments: *Overview for Hercules TMS570 MCUs; www.ti.com*
[5] Dr. Michael Deubzer, Peter Gliwa, Jens Harnisch, Julian Kienberger, Stefan Schmidhuber: *Multicore Engineering Tools and Methods, ESE Congress, Sindelfingen, Dec.'14*
[6] Infineon Technologies: *AURIX™ TC27xT data sheet: www.infineon.com*
[7] AUTOSAR: Guide to BSW Distribution (previously: Guide to Multi-Core Systems); *www.autosar.org*

T1 TIMING **1.** CLASS
T1 – state of the art timing suite

GLIWA embedded systems
www.gliwa.com
GLIWA GmbH embedded systems | Winterstr. 9a | D-82362 Weilheim