



# Best practice for timing optimization

Optimization on RTOS-level and code-level

Embedded Software Engineering Congress 2018

# Contents

- Summary
- Introduction
- Timing analysis techniques
- Performance optimization
  - On RTOS level
  - On code level
  - Memory usage
- Conclusion



# Summary



# Summary on performance optimization

- There are few simple rules for achieving good performance.
  - Consider and – if possible – follow them.
  - Most of the optimization potential cannot be easily exploited.
    - detailed analysis and detailed knowledge required



- Rule number one : optimization always top down
  - Looking at a single ECU, start at the RTOS level
  - When done, move down to the code level



(\*) cf. single-core to multi-core C2C compiler, automatic debugger, etc.

# Introduction

Who is GLIWA?



# Who is GLIWA embedded systems?

- Timing analysis expertise since 2003
  - **hundreds** of mass-production projects
  - located near Munich in Weilheim i.OB., Germany
  - Ca. 40 employees with many embedded timing experts
  - Average annual growth over the past 8 years: **>25%**
- T1.stack: Stack Analysis combining static and dynamic methods
- T1.accessPredictor: “offline-MPU” and more



## Who is Peter Gliwa?

---



- CEO and owner of GLIWA embedded systems
  - Owner of GLIWA Inc. and GLIWA engineering
  - Actively coaching/consulting international automotive OEMs and Tier-1s
  - AUTOSAR work-package leader of AUTOSAR work-package “ARTI”
  - 1998 – 2003: RTOS development/product-management at ETAS
  - 1995 – 2003: BOSCH
  - Degree in Electronic Engineering
-

# Timing analysis techniques





## Two dimensions: *level* and *development phase*

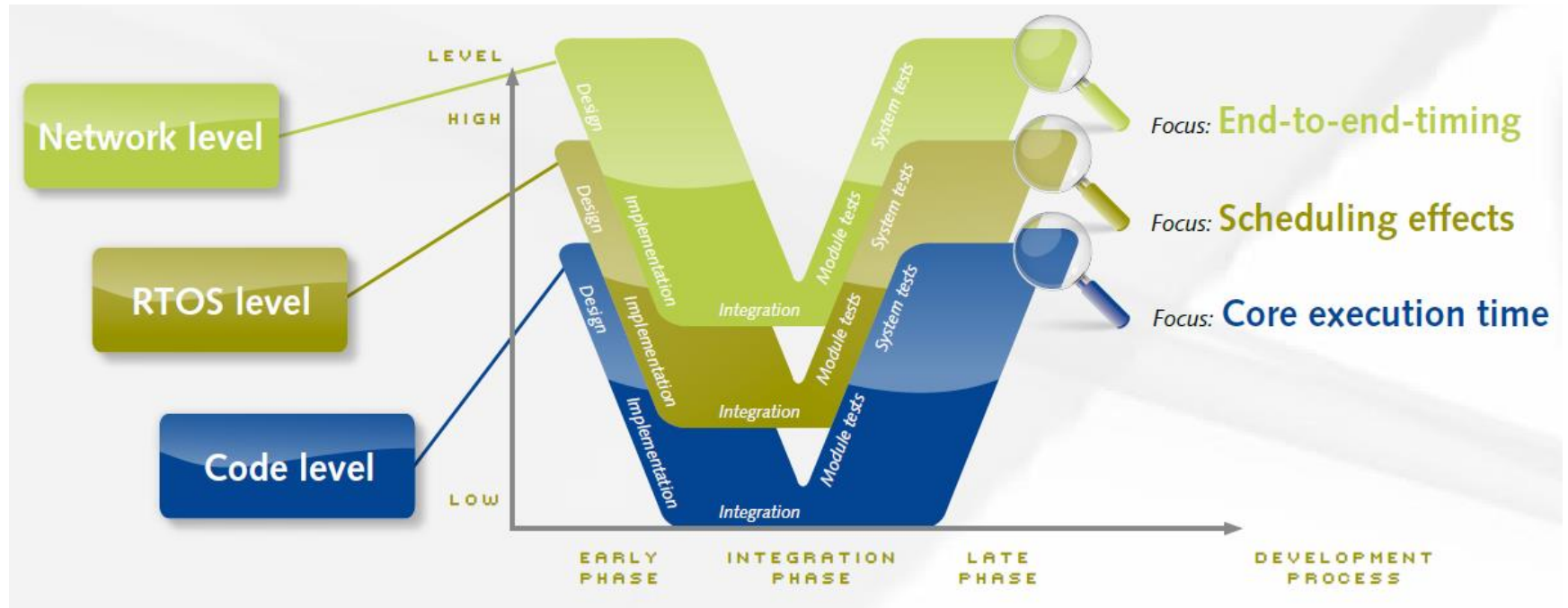
- **Network level**
  - inter ECU communication
  - end-to-end-timing
  - typically OEM business
- **RTOS level (also: scheduling level)**
  - one scheduling entity
  - scheduling effects
  - typically tier-1 business
- **code level**
  - fragment of code (e.g. function)
  - Scheduling not regarded.
  - core execution time most important result

***Level***

- **Early phase**
  - timing requirements
  - Timing design
  - Hardware selection
  - OS-config, mapping to cores
- **Integration phase**
  - Debug
  - Optimize
- **Late phase**
  - Verify timing against requirements (→ tests)
  - Document actual timing
  - Permanently supervise timing on ECU

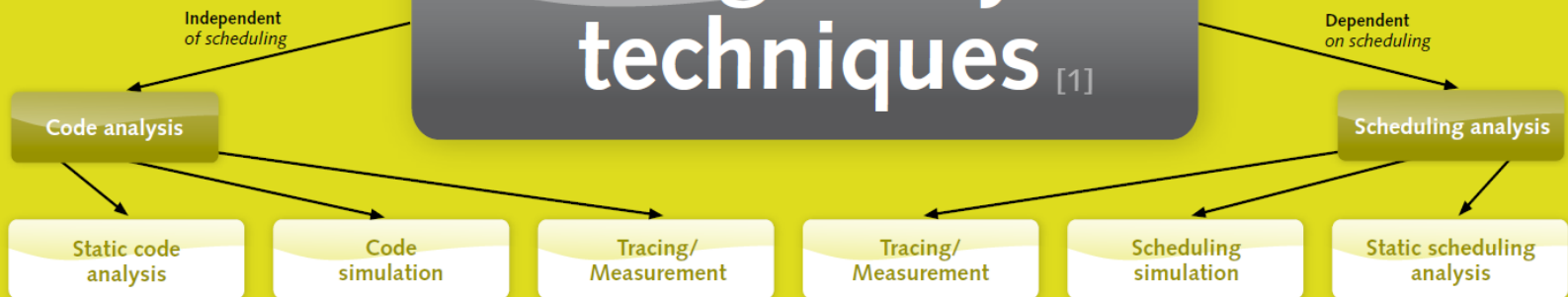
***Development phase***

# Two dimensions: *level* and *development phase*



# Overview of timing analysis techniques

## Timing analysis techniques [1]



```

ldc r24, 0x0068
ldc r25, 0x0069
add r24, r20
adc r25, r21
sts 0x0071, r25
sts 0x0070, r24
    
```

```

TASK (myTask)
{
  CalcV();
  WrPort();
  TerminateTask();
}
    
```



Fine grained (low level)

Scope/granularity

coarse grained (high level)

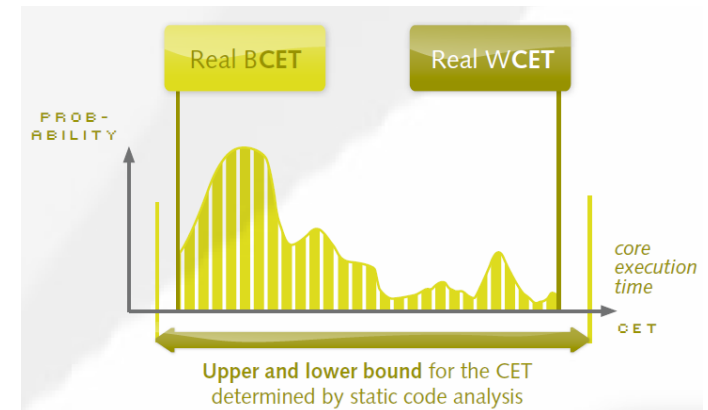
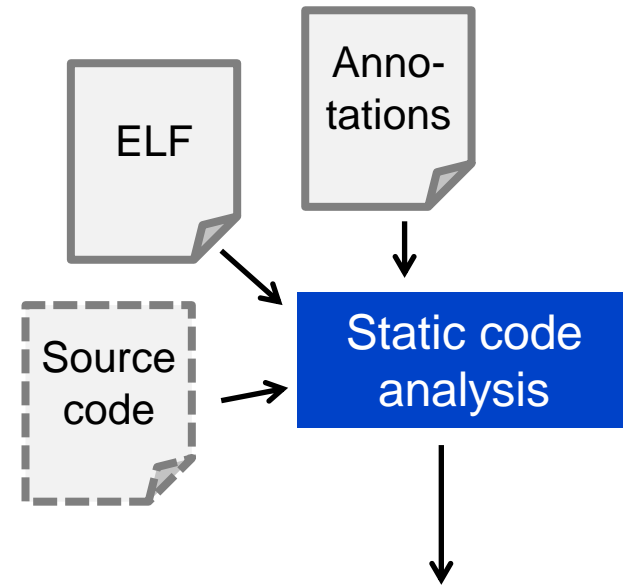


Code level

RTOS level

Network level

- Main result: **safe** upper bound for the **WCET** for a given code fragment, e.g. a function
- **Annotations required** for many indirect calls and loop bounds
- Dramatic overestimation for multi-core  
→ theoretical WCET irrelevant



- Code simulators simulate the execution of given binary code for a certain processor.
- Wide range available:
  - from simple instruction set simulators to
  - complex simulators considering also pipeline- and cache-effects
- Code simulators rarely used for timing analysis.

- Observation of the real (executing) system
- For dedicated events, time stamps together with event information are placed in a trace buffer (for later analysis/reconstruction).
- Wide range of granularity:
  - from fine grained like for flow traces (instruction trace) to
  - schedule traces showing tasks/interrupts only
- Measurement/tracing through instrumentation (i.e. software modification) or using special hardware (on-chip/off-chip)



# Measurement vs. Tracing

- Timing measurement
  - produces timing parameters (“numbers”) but no traces
  
- Scheduling Tracing
  - produces traces which can be viewed and from which timing parameters can be derived



# Static scheduling analysis

$$RT_i = CET_i + JIT_i + \sum_{j \in hp(i)} CET_j \left\lceil \frac{RT_i}{T_j} \right\rceil \leq DL_i$$



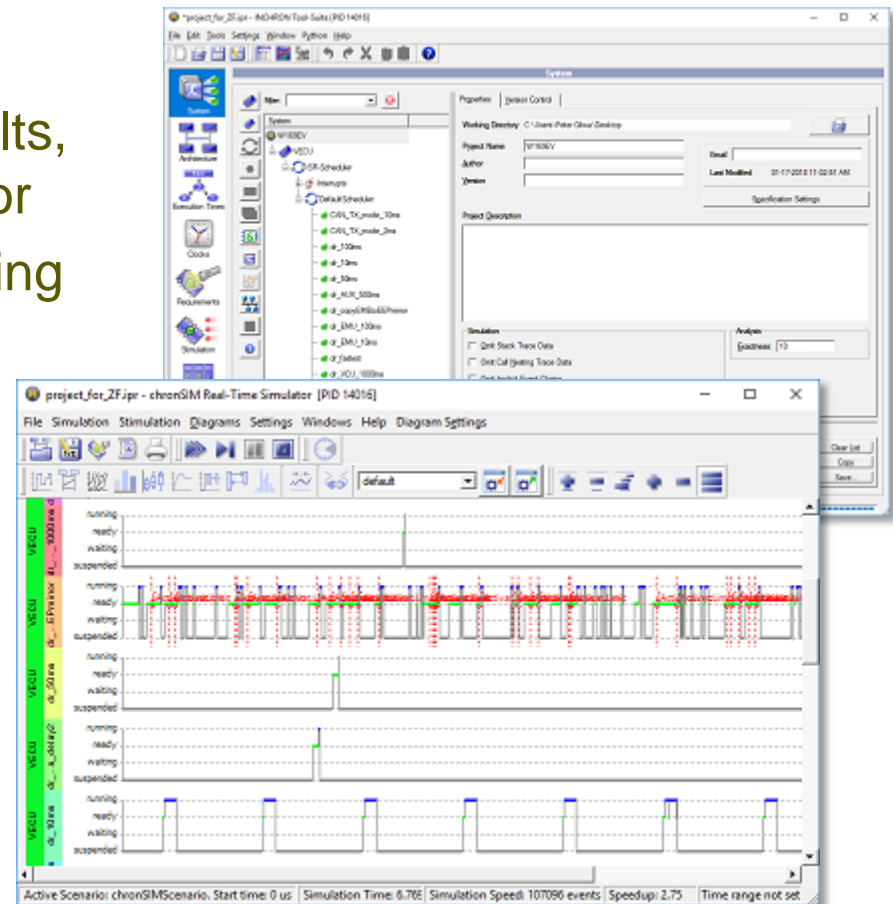
- Input: scheduling model and min/max execution times
- Calculates (no simulation!) the worst case scheduling situation for a given timing parameter, e.g. the WCRT of task A.
- No code or hardware required.
- The execution times fed into the analysis can be either budgets, estimations, or outputs from other tools, e.g. statically analyzed BCET/WCET or traced/measured data.



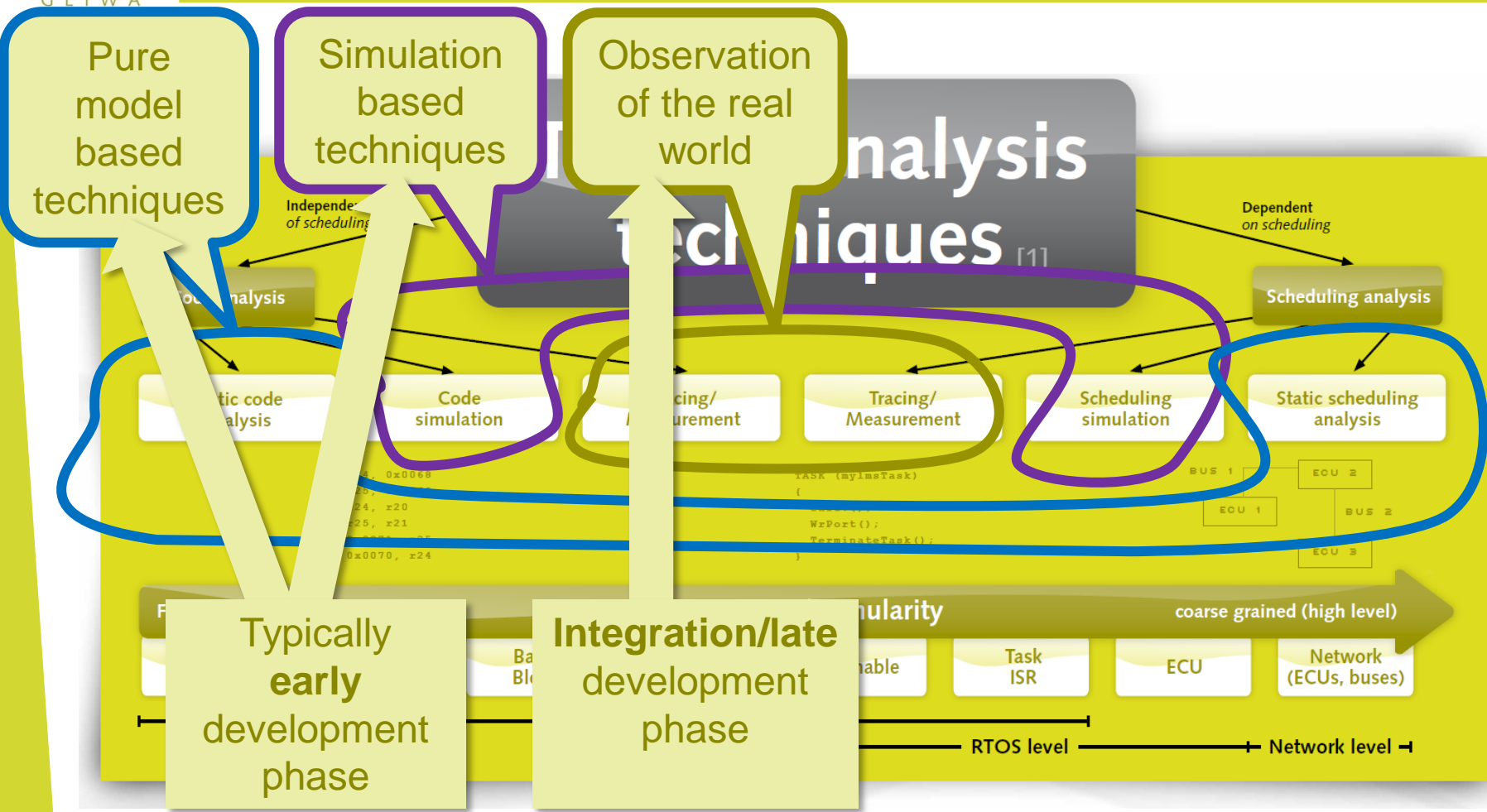
# Static scheduling simulation

Scheduling  
simulation

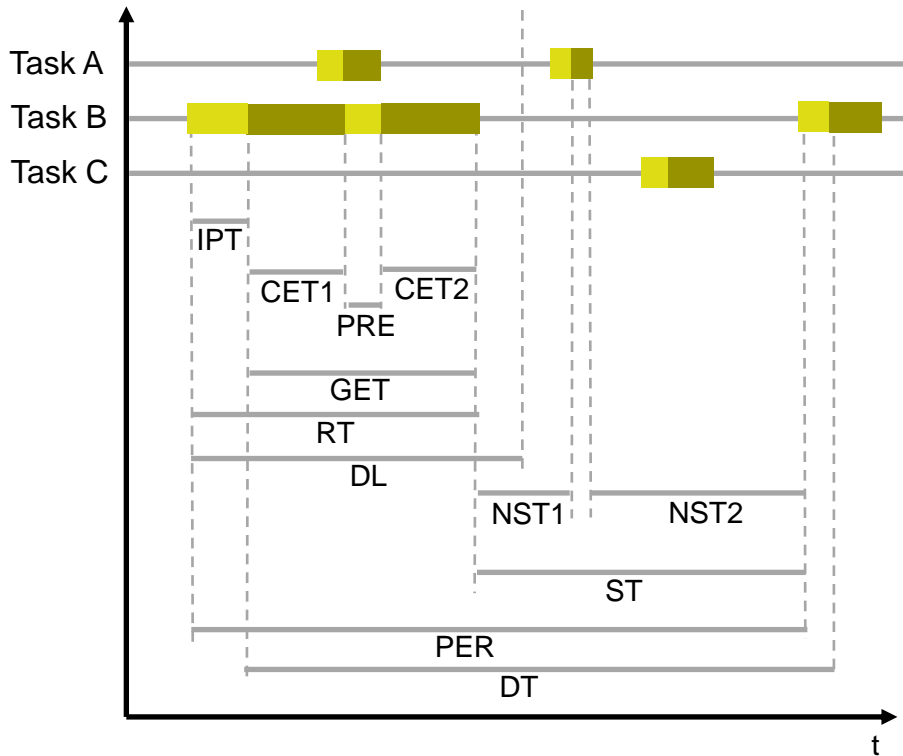
- Similar functionality as the scheduling analysis
- Instead of *calculating* the results, they *simulate* run time behavior
- Main output: the observed timing information and generated traces



# Overview of timing analysis techniques



# Timing parameters



$$\text{CET} = \text{CET1} + \text{CET2}$$

$$\text{NST} = \text{NST 1} + \text{NST2}$$

Abr.	Explanation (EN)	Erklärung (DE)
<b>IPT</b>	initial pending time	Initialwartezeit
<b>CET</b>	core execution time	Nettolaufzeit
<b>GET</b>	gross execution time	Bruttolaufzeit
<b>RT</b>	response time	Antwortzeit
<b>DT</b>	delta time	Deltazeit
<b>PER</b>	period	Periode
<b>ST</b>	slack time	Restzeit
<b>PRE</b>	preemption	Unterbrechungszeit
<b>JIT</b>	jitter	Jitter
<b>CPU</b>	cpu load	CPU Auslastung
<b>DL</b>	Deadline	Deadline
<b>NST</b>	Net slack time	Nettorestzeit

# Timing Poster – get your copy!

### WHAT IS EMBEDDED TIMING AND TIMING ANALYSIS?

The code running down in the human dimensions when embedded timing is done. The system consists of sensors, actuators and control units. The system consists of sensors, actuators and control units. The system consists of sensors, actuators and control units.

### TIMING ANALYSIS: LEVELS AND DEVELOPMENT PHASES

Any timing related activity, problem or use case can be placed in a diagram with two dimensions: the level and the development phase.

### USE-CASES AND TIMING PROBLEMS

USE CASE OR PROBLEM	POSSIBLE SOLUTIONS
System data requirements	• The scheduling problem is a scheduling problem for a scheduling problem. Hence, because of the scheduling problem, the scheduling problem is a scheduling problem.
System data requirements	• The scheduling problem is a scheduling problem for a scheduling problem. Hence, because of the scheduling problem, the scheduling problem is a scheduling problem.
System data requirements	• The scheduling problem is a scheduling problem for a scheduling problem. Hence, because of the scheduling problem, the scheduling problem is a scheduling problem.

### TIMING ANALYSIS TECHNIQUES

ANALYSIS TECHNIQUE	INPUT DATA	OUTPUT (RESULTS OR PROBLEMS)	MAIN OUTPUT
Formal Analysis (Mathematical)	Formal model	Formal model	Formal model
Code simulation	Code	Code	Code
Timing/Measurement	Hardware	Hardware	Hardware
Scheduling simulation	Scheduling	Scheduling	Scheduling
Scheduling analysis	Scheduling	Scheduling	Scheduling

### Timing analysis techniques

Timing analysis techniques include Code analysis, Static code analysis, Code simulation, Tracking/Measurement, Scheduling simulation, Scheduling analysis, and Static scheduling analysis.

### FROM SWCs TO RUNNABLES TO TASKS TO EXECUTION

The flow goes from SWCs to Runnables to Tasks to Execution. This process involves various stages of development and testing.

### GLOSSARY

ACR.	DEFINITION	PROPERTY
RTOS	Real-time operating system	Task A
RTOS	Real-time operating system	Task A
RTOS	Real-time operating system	Task A

### CODE SIMULATION

Code simulation simulates the execution of given binary code for a certain program. A wide variety of code simulation exists, ranging from simple to complex. Code simulation can be used to verify the correctness of the code before it is embedded into a hardware system.

### CPU-LOAD / BUS-LOAD

CPU-load and bus-load are the most important parameters of timing. They represent the amount of work that the system has to do. CPU-load is the amount of work that the system has to do. Bus-load is the amount of work that the system has to do.

### STRONG ABBREVIATION MEANING

Abbreviation	Meaning
RTOS	Real-time operating system
RTOS	Real-time operating system
RTOS	Real-time operating system

### REFERENCES

References include various technical documents and books related to embedded timing and real-time systems.

### SAFETY & AVAILABILITY

The safety and availability of a system are competing requirements. A system can never be both safe and available. The safety and availability of a system are competing requirements.

### STANDARDS

Standards for embedded timing include various international and industry standards. These standards define the requirements for embedded timing systems.

### MULTICORE

Multicore development presents significant additional challenges. The challenges include various technical and organizational issues.

# Performance optimization



**RTOS level**

## (Incomplete) collection of optimization aspects

---

- Rule number one :  
optimization always top down
  - Looking at a single ECU, start at the RTOS level
  - When done, move down to the code level
- In the following we will collect some
  - RTOS level optimization approaches
  - Code level optimization approaches

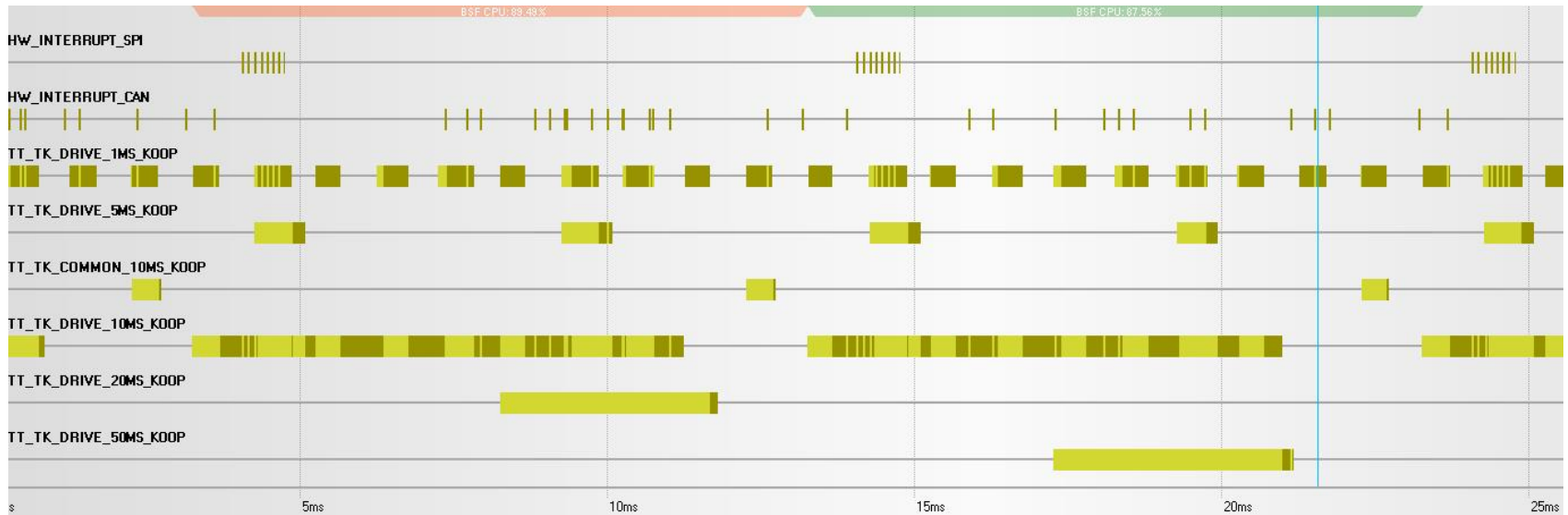


# RTOS level best practices

---

- Keep it simple!
  - Try to avoid ECC (extended conformance class)
    - unfortunately, most RTE generators advise you to use ECC
  - Do not use multiple task activations
- Use cooperative (“non preemptive/non preemptable”) scheduling
  - Reduce stack consumption → save RAM
  - Avoid protection mechanisms (data copies for data consistencies)
  - Reduce the risk of typical run-time problems
- Come up with a sound timing design
  - Allocate timing budgets
  - Use scheduling simulation/scheduling analysis for complex timing

# Positive example: BMW Active Steering



- Highly loaded (up to 93%)
- As a result of optimizations, a less powerful (and cheaper) processor than in the previous generation could be used
- Cooperative scheduling avoiding costly protection mechanisms

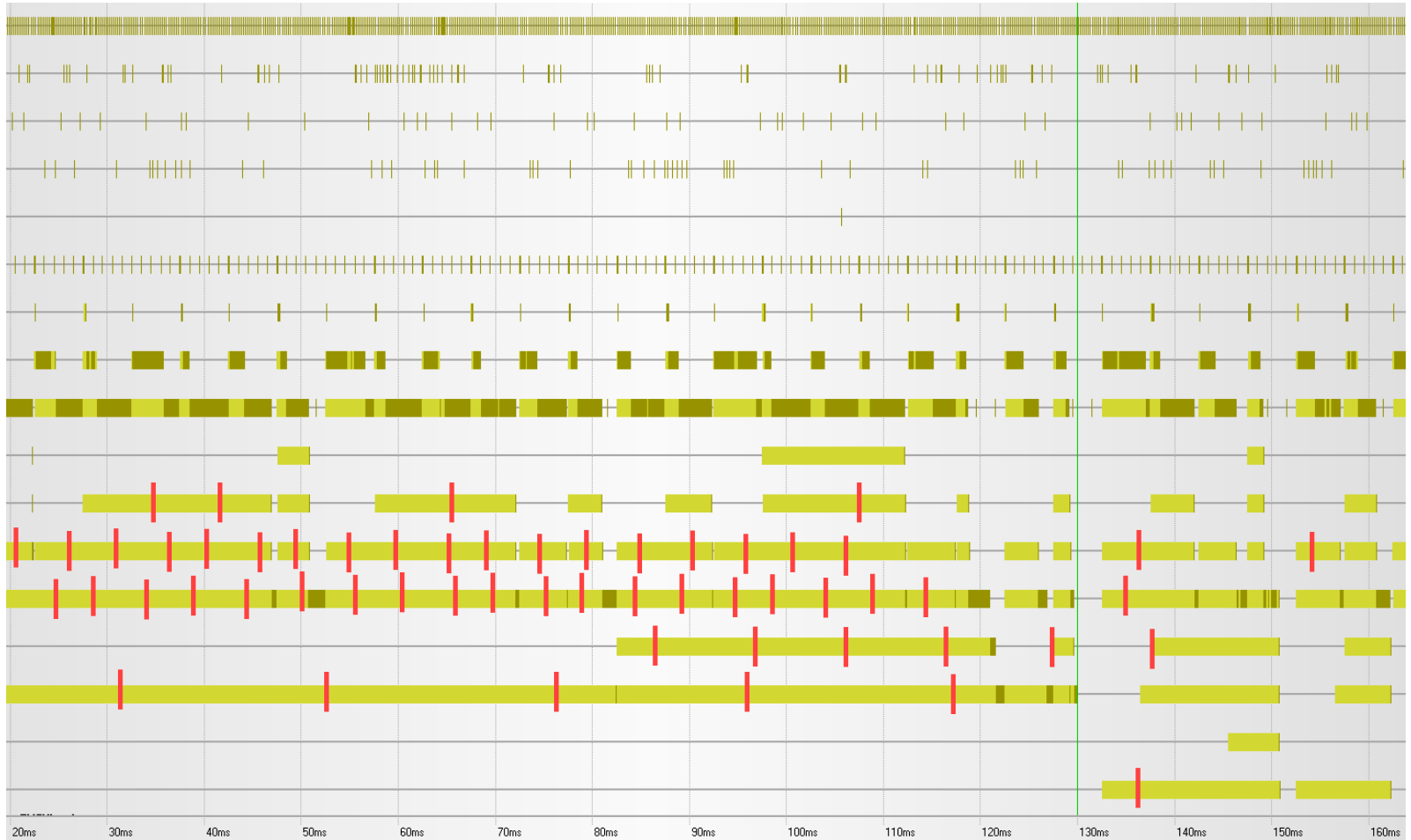


## RTOS level optimization approaches

---

- Move code to slower tasks
- Configure delays of periodic tasks so that the load spreads
- Understand the scheduling (and the hot-spots; see next slide)
- Multicore
  - Consider using one core for handling ISRs and “fast tasks”
  - The other core(s) do the “number crunching” exploiting the cache and the pipeline more efficiently
  - Avoid busy-spinning
    - Search/replace `__disable()` / `__enable()` with `GetSpinlock()` / `ReleaseSpinlock()` is a very bad idea
    - consider following the LET (“Logical Execution Time”) concept

# Overload situation the PL was not even aware of



# Performance optimization



**Code level**

# Code level optimization approaches

- Move frequently addressed symbols (code, data) to fast memory
- Use (and cross-check!) dedicated compiler optimizations
- Manual optimization
  - Inline functions
  - Alignment
    - Aligned data allows faster code
    - Code aligned to cache-lines can increase speed
  - Exploit specialized machine code
    - Example: saturation instruction avoids efficient wrap-around protection

In the following we will look at the optimization of the well-known `memcpy` function copying 1024 bytes.

# memcpy

```
/*----- The 'standard' memcpy routine -----*/
* Parameters:
*   *pDest - The destination to which data is copied across to
*   *pSrc  - The source of the data to be copied across. The addresses of
*           pSrc and pDest are passed as arguments. This avoids having
*           to pass the complete arrays in as arguments in order to
*           do manipulations. Note, they are void pointers to allow any type
*           of array to be passed.
*   nBytes - The number of bytes to copy from pSrc to pDest
*           Remember that a 'char' is 1 byte and an 'int' is 4 bytes (or a word)
*-----*/
void *memcpy_( void *pDest, void const *pSrc, unsigned short nBytes )
{
    /* Assign pSrc and pDest to 'char' Auto-variable pointers on the stack. This
       allows byte per byte transfer */
    char *pD = pDest;
    char const *pS = pSrc;

    /* Iterate through the number of bytes to copy across, decrementing nBytes
       until it reaches zero */
    while( nBytes-- )
    {
        /* Copy one byte from the source to the destination and then
           increment the index */
        *pD++ = *pS++; /* E.g. pD[i++] = pS[i++]; */
    }
    return pDest;
}
```

# Step 0: non optimized version (starting point)

Default Memory Locations				CET to Copy 1024 Bytes		CET to Copy 1 Byte	
Function Code	pDest	pSrc	nBytes	MAX	MIN	MAX	MIN
Cached Flash0	LMU RAM	Cached Flash0	LMU RAM	121us 030ns	114us 395ns	118,2ns	111,7ns



CET per Byte

## Assembly code

```

80006e6e <memcpy_>:
80006e6e: 40 42          mov.aa %a2,%a4
80006e70: a0 0f          mov.a %a15,0
80006e72: 01 f2 10 40    add.a %a4,%a2,%a15
80006e76: 01 f5 10 30    add.a %a3,%a5,%a15
80006e7a: 9f 04 03 80    jned %d4,0,80006e80 <memcpy_+0x12>
80006e7e: 00 90          ret
80006e80: 79 3f 00 00    ld.b %d15,[%a3]0
80006e84: 2c 40          st.b [%a4]0,%d15
80006e86: b0 1f          add.a %a15,1
80006e88: 3c f5          j 80006e72 <memcpy_+0x4>

```



- No post-increment addressing
- No Loop instruction



## TC27x C-Step

### On-Chip System Buses and Bus Bridges

**Table 3-16 CPU access latency in CPU clock cycles for TC27x**

CPU Access Mode	CPU clock cycles
Data read access to own DSPR	0
Data write access to own DSPR	0
Data read access to own or other PSPR	5
Data write access to own or other PSPR	0
Data read access to other DSPR	5
Data write access to other DSPR	0
Instruction fetch from own PSPR	0
Instruction fetch from other PSPR (critical word)	5
Instruction fetch from other PSPR (any remaining words)	0
Instruction fetch from other DSPR (critical word)	5
Instruction fetch from other DSPR (any remaining words)	0
Initial Pflash Access (critical word)	5 + configured PFlash Wait States <sup>1)</sup>
Initial Pflash Access (remaining words)	0
PMU PFlash Buffer Hit (critical word)	4
PMU PFlash Buffer Hit (remaining words)	0
Initial Dflash Access	5 + configured DFlash Wait States <sup>2)</sup>
TC1.6E/P Data read from System Peripheral Bus (SPB)	4 ( $f_{CPU}=f_{SPB}$ ) 7 ( $f_{CPU}=2*f_{SPB}$ )
TC1.6E/P Data write to System Peripheral Bus (SPB)	0

1) FCON.WSPFLASH + FCON.WSECPF (see PMU chapter for the detailed description of these parameters).

2) FCON.WSDFLASH + FCON.WSECDF (see PMU chapter for the detailed description of these parameters).

### On Chip Bus Access Times

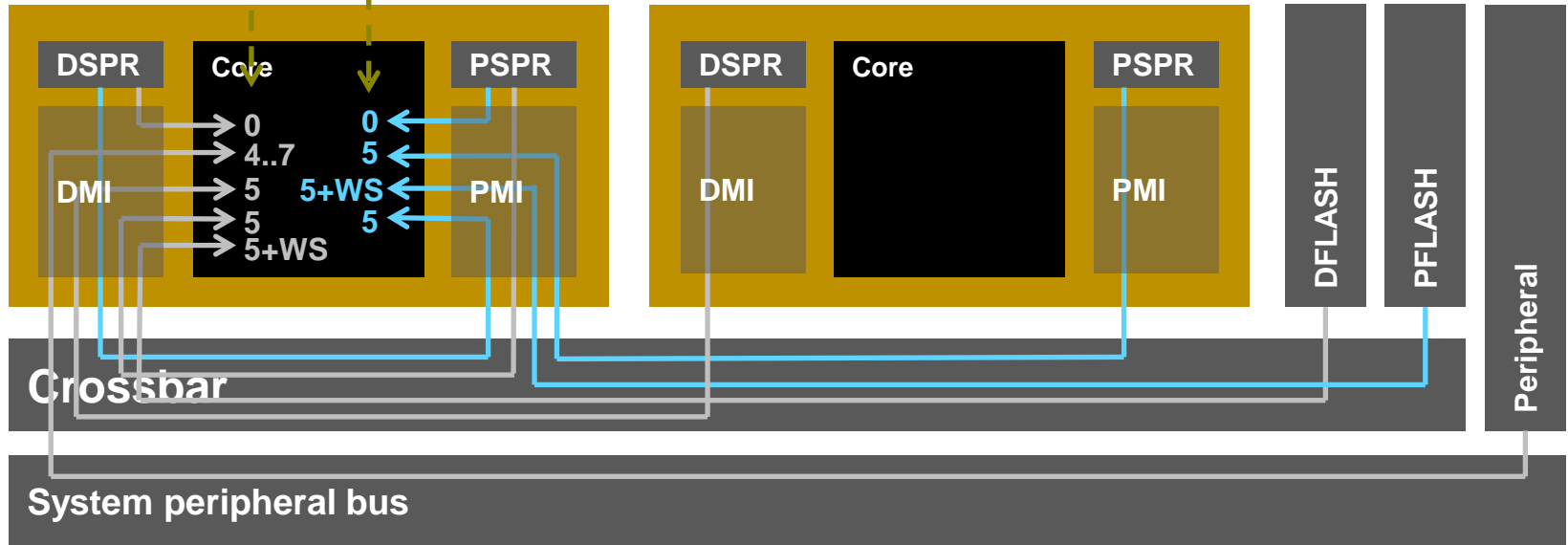
The table describes the CPU access times in CPU clock cycles for the TC27x. The access times are described as maximum CPU stall cycles where e.g. a data access to the local DSPR results in zero stall cycles. Pls. note that the CPU does not always immediately stall after the start of a data read from another SPR due to instruction pipelining effects. This means that the average number will be below the here shown numbers.

# AURIX™ memory *read* access times: interpretation

Maximum CPU stall cycles for **data** reads

Maximum CPU stall cycles for **program** reads

“Maximum” refers to a situation where there are no memory access conflicts. If these occur, the penalty can be **much** higher!



→ data read access  
→ program read access

DSPR = data scratch pad RAM  
PSPR = program scratch pad RAM

DMI = data memory interface  
PMI = program memory interface



# Step 1: Use different memory locations

Code/Data Memory Locations			CET per byte for 1024 bytes
Function Code	pDest	pSrc	
Cached Flash0	LMU RAM	Cached Flash	111.7ns ← Baseline
	LMU RAM	LMU RAM	125.0ns
	Local DSPR0	Local DSPR0	100.6ns ← Fastest
Local PSPR0	LMU RAM	Cached Flash	106.4ns
	LMU RAM	LMU RAM	135.8ns
	Local DSPR0	Local DSPR0	100.6ns
Un-Cached Flash0	Local DSPR0	Local DSPR0	205.1ns ← Slowest
PSPR1	Local DSPR0	Local DSPR0	149.4ns

## Step 2: compiler optimizations

---

- Tasking
  - Function Specific Option Pragmas
    - `#pragma optimize 'o'`, where o stands for option
    - `#pragma endoptimize`. To confine the optimization option
  - Desirable:
    1. Use post-incrementing load and store operations
    2. Use Loop instruction
    3. Use loop unrolling
- These compiler optimizations are only a subset of what was actually analyzed

## Step 2: compiler optimizations (results)

- Use post-incrementing load and store operations
- Use Loop instruction
- Tasking can achieve both at the same time using a compiler environment option `-t0`, which means to optimize for speed
- Assembly:

```

8020011c 40 4f          memcpy_:  mov.aa    a15,a4
8020011e 8e 46              jlez      d4,0x8020012a
80200120 60 42              mov.a     a2,d4
80200122 b0 f2              add.a     a2,#-0x1
80200124 04 5f              ld.bu    d15,[a5+]0x1
80200126 24 ff              st.b     [a15+]0x1,d15
80200128 fc 2e              loop     a2,0x80200124
8020012a 40 42              mov.aa    a2,a4
8020012c 00 90              ret

```

		CET per Byte for 1024	
Compiler	Description	MAX	MIN
Tasking	Enabling post-increment load and store operations and Loop instruction	65.4ns	59.6ns

## Step 3: manual optimizations

- Checking Data Alignment
  - If aligned, we can copy across words each time using word size instructions.

```
/* Divide nBytes by 4. This is to get rid of EXTR.U operation and to get word decrements.
   E.g. 16 bytes is 4 words.. */
GTF_uint32_t wordCount = nBytes >> 2u;

/* Check for word alignment. Casting is needed for bitwise manipulation */
if( 0u == ( ( (GTF_uint32_t)pDest | (GTF_uint32_t)pSrc | nBytes ) & 3u ) )
{
    /* Assign Word Pointers */
    GTF_uint32_t *pD = (GTF_uint32_t *)pDest;
    GTF_uint32_t const *pS = (GTF_uint32_t const *)pSrc;

    while( 0u != wordCount-- )
    {
        *pD++ = *pS++; /* Copy words (4 bytes at a time..not 1 byte) across */
    }
}
/* Else do Manual Loop Unrolling with Switch Case Above */
else
{
    ....
}
```

## Step 3: manual optimizations (results)

		CET per byte for 1024 bytes
Compiler	Description	
Other (not TASKING)	Manual Loop Unrolling Depth Of 4 Switch Case below	65.4ns
	Manual Loop Unrolling Depth of 4 Switch Case above	63.5ns
	Manual Loop Unrolling Depth of 4 Switch Case above and Removing EXTR.U operation	63.5ns
	Duff's Device	71.3ns
	Copying Words across. Union declared outside the function	<b>18.6ns</b>
TASKING	Manual Loop Unrolling Depth Of 4 Switch Case below	58.6ns
	Manual Loop Unrolling Depth of 4 Switch Case above	59.6ns
	Manual Loop Unrolling Depth of 4 Switch Case above and Removing EXTR.U operation	55.8ns
	Duff's Device	57.6ns
	Copying Words across. Union declared outside the function	<b>14.7ns</b>

← Good result!

← Best result!

# Spinlocks

and how not to use them



## Spinlocks – Overview

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

- **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.
- **ReleaseSpinlock** releases a spinlock. Obtained spinlocks must be released in the correct order, the last obtained spinlock must be released first.

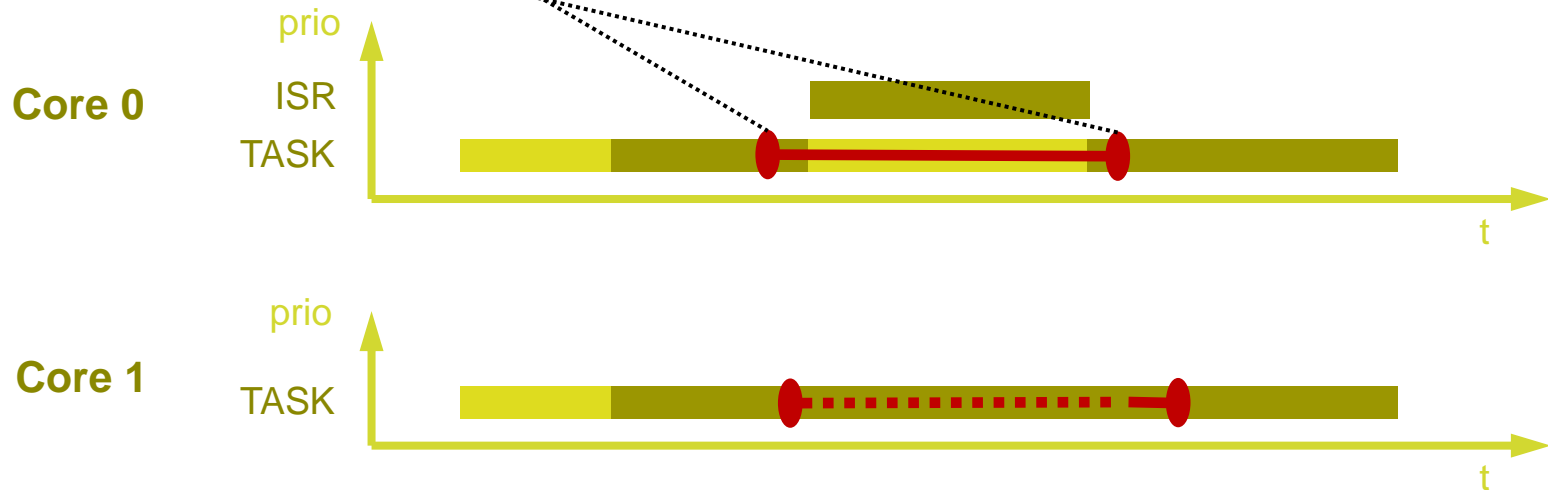
# Spinlocks – problematic straight forward usage

Imagine a situation where a Task gets interrupted by an ISR while holding a spinlock. Although not related at all to the spinlock, **the ISR can now delay TASKS on other cores** waiting (i.e. spinning) for the spinlock.

```

GetSpinlock (spinlock) ;
... /* do what you need to do with spinlock obtained */
ReleaseSpinlock (spinlock) ;

```



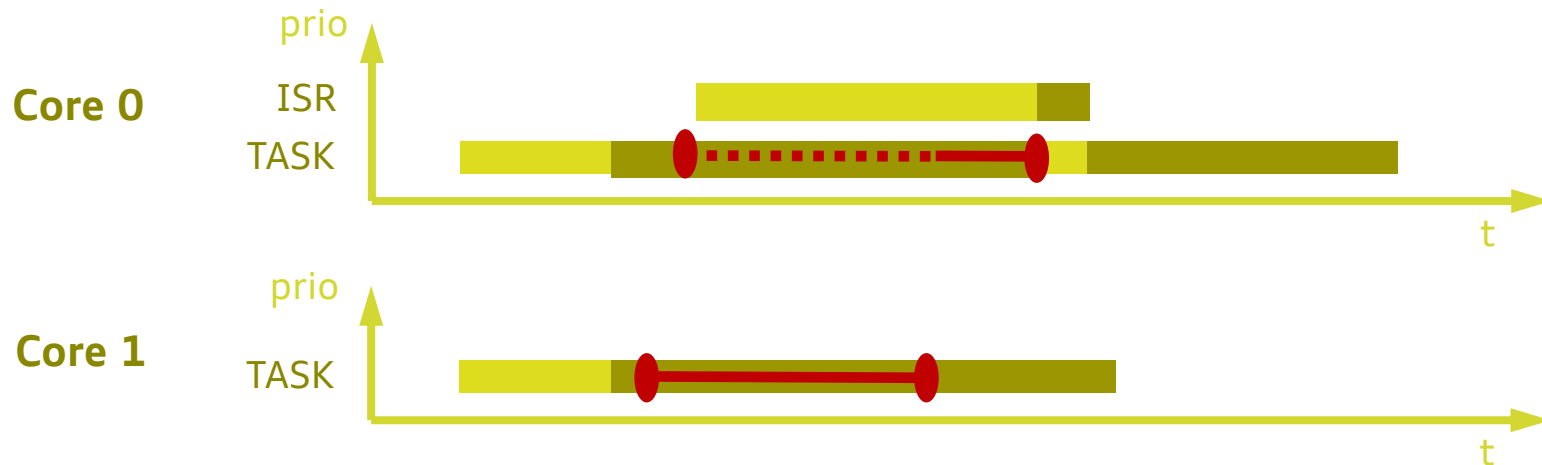


# Spinlocks – pseudo clever usage

To overcome the problem, we could disable/enable interrupts. However, this might lead to a considerable **delay of the ISR caused by TASKs on other cores.**

```

DisableOSInterrupts ( );
GetSpinlock (spinlock);
... /* do what you need to do with spinlock obtained */
ReleaseSpinlock (spinlock);
EnableOSInterrupts ( );
  
```



## Spinlocks – TryToGetSpinlock: best practice

```
TryToGetSpinlockType success;  
DisableOSInterrupts( );  
(void)TryToGetSpinlock( spinlock, &success );  
while( TRYTOGETSPINLOCK_NOSUCCESS == success )  
{  
    EnableOSInterrupts( );  
    /* Allow preemption. */  
    DisableOSInterrupts( );  
    (void)TryToGetSpinlock( spinlock, &success );  
}  
/* Region with spinlock obtained and interrupts disabled. */  
... /* do what you need to do with spinlock obtained */  
ReleaseSpinlock( );  
EnableOSInterrupts( );
```

- Are we there yet? Is this the best implementation?
- Actually no.
- The best spinlock is the one you do not need!

# The Multi-core Poster – Multi-core on one sheet of paper

## Multi-core

AN INTRODUCTION TO AUTOMOTIVE EMBEDDED SOFTWARE TIMING

### 01 INTRODUCTION

**A table of parallel processing:**

- Imagine you want to have a kitchen built in one day (a 10-hour job).
- You ask a contractor to do it but he says: "I will take one full hour."
- So you might have a second one in order to get the job done in 1 day.
- But! While one contractor comes to the site, they said themselves later the first one, the other one can not 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 blocked and you agree to plan next time.

The poster checks right 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 IC 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. Moore's law is called "WAG" (What A Good Number) – says the number of transistors in a device (integrated circuit) doubles approximately every two years.

**Good reasons for more computing power include:**

- More and more advanced vehicle functions (autonomous driving, car-to-X communication, etc.)
- Stricter safety requirements (stricter computing, memory protection, on-target supervision, etc.)
- Increasing use of standards and generated code leads to the scope of optimization.
- Building faster clock speed (1 single-core processors become too expensive at some point due to the following reasons):

  - Power consumption (P = P (dissipating))
  - EMC (Electromagnetic compatibility) problems
  - Power dissipation or "heating" problems

### 03 MULTI-CORE THEORY

**Andrius's law:** The speed of a program using multiple processors is parallel computing is limited by the slowest part of the program. [1]

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

**Goldman's law:** Programmers tend to use the size of the problem to solve the problem to solve the problem. Therefore, a faster (more parallel) processor is available, larger problems can be solved in the same time. [2]

**Goldman's law applies when a given problem can be split up by a larger problem solving the old problem plus other problems.**

### 04 DIFFERENT KINDS OF PARALLELISM

The term "parallelism" refers to how many fragments of a program being executed at the same time or several cores.

#### 6.1 APPLICATION PARALLELISM

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

**Example 1:** In multi-core, two single-core ECUs are merged into one dual-core or 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. Each core comes with its own set of RTOS and RTE. It is its own AUTOSAR application. The AUTOSAR OS allows each core to communicate through the RTE (inter-OS-Application Communication). If application requires data to be copied and then might be performed for large data. Communication through the RTE can be optimized e.g. direct access 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 suitable approach is to use "AUTOSAR user" and "non-AUTOSAR cores". This approach is used with early AUTOSAR standards that do not support multi-core. The AUTOSAR software components with the non-AUTOSAR software via complex domain drivers (CDD), a non-AUTOSAR core could, for example, handle time-critical and/or very frequent interrupt, reducing the number of cache misses and pipeline stalls. There is no need for complex hardware parallelism.

#### 6.2 FUNCTION PARALLELISM

Function parallelism executes clearly related fragments (with pointer high cohesion) of the same program in parallel in order to find a high-level fragment, dependencies have to be independent and simple.

- DA (data flow analysis)
- DA (data flow analysis)

Function parallelism is largely absent in Windows/Linux/Mac software, multi-core domains. There are some application parallelism mainly. There are very few examples of successful function parallelism and these include: 3D rendering software, multimedia databases software, computationally intensive scientific software of research institutes, etc.

Engineering software that is designed for parallelism is not easy and, when done poorly, can make a system in a multi-core environment less performant. A multi-core system with a negative impact on overall performance. An Andrius's law shows, the benefit does not scale with the number of cores.

### 05 EXAMPLE INFINEON TC27X "AURIX™" [3]

Each TC27X has local program memory and local data memory. But it can access via a shared. With significant delay lag to 5 CPU stall cycles, each TC27X can also access data (program memory) of other cores, see also section "OS Cloning".

Access 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. Each core is a maximum of 10 a number of wait states CPU stall cycles. These numbers show that location of data and code has a significant impact on the timing.

**At Both interrupt get executed on a single core.**

**At Both interrupt get executed on different cores.**

The spillover related AUTOSAR services are:

- Relocation/spillover: releases a spillover. Released spillover must be released first.
- OS Cloning: OS Cloning is a special case where one core is using it. Another core is using it. Then each spillover logs (spillover) and the spillover can be correctly obtained.
- Hybrid/Spillover: is a non-blocking error of Cloning log. It always returns immediately with no spillover.

### 06 AUTOSAR AND MULTI-CORE

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

- OS Cloning and OS Cloning: allow other cores.
- Core lock activation and lock sharing (however lock acquisition is not supported and also not expected).
- Spillover (Spillover-Concept, explained later)
- Inter-core data passing by reference (currently copying data is mandatory which becomes a issue when dealing with large data).

AUTOSAR does not (yet) support:

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

Without any problem, data consistency cannot be guaranteed.

**DATA-CONSISTENCY, SPINLOCKS**

What if 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.

**At Both interrupt get executed on a single core.**

**At Both interrupt get executed on different cores.**

The straight-forward implementation shown in Example 04 is rarely suitable for real applications and can cause significant overhead when one core acquires a spinlock and then handles one or more interrupts. A better implementation is shown below and can be used as a single pattern for spinlock usage.

The straight-forward implementation shown in Example 04 is rarely suitable for real applications and can cause significant overhead when one core acquires a spinlock and then handles one or more interrupts. A better implementation is shown below and can be used as a single pattern for spinlock usage.

### 07 MULTI-CORE HARDWARE ARCHITECTURES

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

Examples:

- Infineon TC27X (TC27A 1 PCP)
- Infineon BRC2xx with TPU
- Infineon TC277 (several different cores, see next section)

Heterogeneous multi-core processors have a number of cores of the same type.

- Infineon BRC2xx
- Infineon TC277 (two TC27A cores, see next section)

Lock-like multi-core processors consist of one single-core software on two separate cores at the same time, for safety reasons. The result of the two cores get continuously compared by the hardware. When a mismatch (error) occurs, the processor can switch to safe state.

Chip designers spend a lot of effort to avoid common mode failures (light emission delay between the cores, separate clock-bus, related and flipped 271 CPU potential guard ring around each CPU, etc. [5])

Example: Texas Instruments TMS320C4x potential AURIX™ (TC2A) core with checker core, see next section)

### 08 CLONING

Cloning is a very powerful concept. On the one hand it allows the same code to be executed on different cores at the same time. On the other hand, it provides an easy way to make efficient, dedicated, multi-core software guaranteed full on both levels of data access control.

With cloning, all cores see their own, local memory at the same exact address. e.g. 0x00000000 for the DSPR (data scratch pad) of the AURIX™ TC27X. These memories have the same address (locking) but can have different contents and, in some cases, allow. Any local or slave instructions using the address access the memory local to the core on which the instruction executes.

Example: AURIX™ TC27X (TC27A) core with checker core, see next section)

Address in CPU	Address in CPU	Address in CPU
DSPR0	0x0000_0000...	0x0700_0000...
DSPR1	0x0000_0000...	0x0600_0000...
DSPR2	0x0000_0000...	0x0500_0000...

**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, well understood and not really new. Software applications for many automotive systems 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- to multi-core because parallelism has been well established in the hardware and the multi-core complexity can be divided to the OS.

### 09 REFERENCES

[1] WIKIPEDIA, en.wikipedia.org/wiki/Moore's\_Law [2] WIKIPEDIA, en.wikipedia.org/wiki/Castorion\_TC27X [3] D. Michael Decker, Peter Glaw, Axel Harms, Julian Koenig, Stefan Krennhuber, MultiCore Engineering Tools and Methods, CRC Catalog, Heidelberg, Dec. 14

[4] AUTOSAR, Guide to SW Development (previously: Guide to Multi-Core Systems), www.autosar.org [5] Texas Instruments, Overview for Hercules TMS320C4x, www.ti.com

[6] Infineon Technologies, AURIX™ TC27X data sheet, www.infineon.com

### 10 GLOSSARY

**AMP (Application Migration Platform)**  
A multi-core system with a specific operating system for multi-core.

**Autosar**  
A multi-core system with a specific operating system for multi-core.

**Cloning**  
Duplicate to inter-processor data, code and control flow within a given processor.

**IOE**  
Inter-OS-Application Communication. Part of the AUTOSAR OS responsible for managing communication between one OS-Application to another and by themselves, from one core to another.

**Multi-core**  
Having more than one core on a processor with each core running a generally separate AUTOSAR multi-core.

**Non-blocking**  
An implementation of a resource that is guaranteed not to block.

**OS**  
Operating System. An autonomous time and other resources and scheduling being used by the combination of a kernel and the free time of the processor, such as a driver driver. The AUTOSAR OS is not a kernel.

**OS-Application**  
AUTOSAR is a collection of application software. Each core of an application software runs on one core but as an OS-Application set of processing steps for handling a task. The task is split from the first state and state is passed from the first state to the second state, including the task.

**Spillover**  
Allocation for allowing multiple assignments. The AUTOSAR RTE is responsible for managing data across multiple cores in the same processor.

**System/OS**  
A multi-core system operating under a single operating system with two or more processors and their processors do not include any relevant parallelism in the code.

**Task**  
Collection of software that controls a specific task. The task is split from the first state and state is passed from the first state to the second state, including the task.

### 11 TIMING CLASS

11 - state of the art timing suite

www.gliwa.com

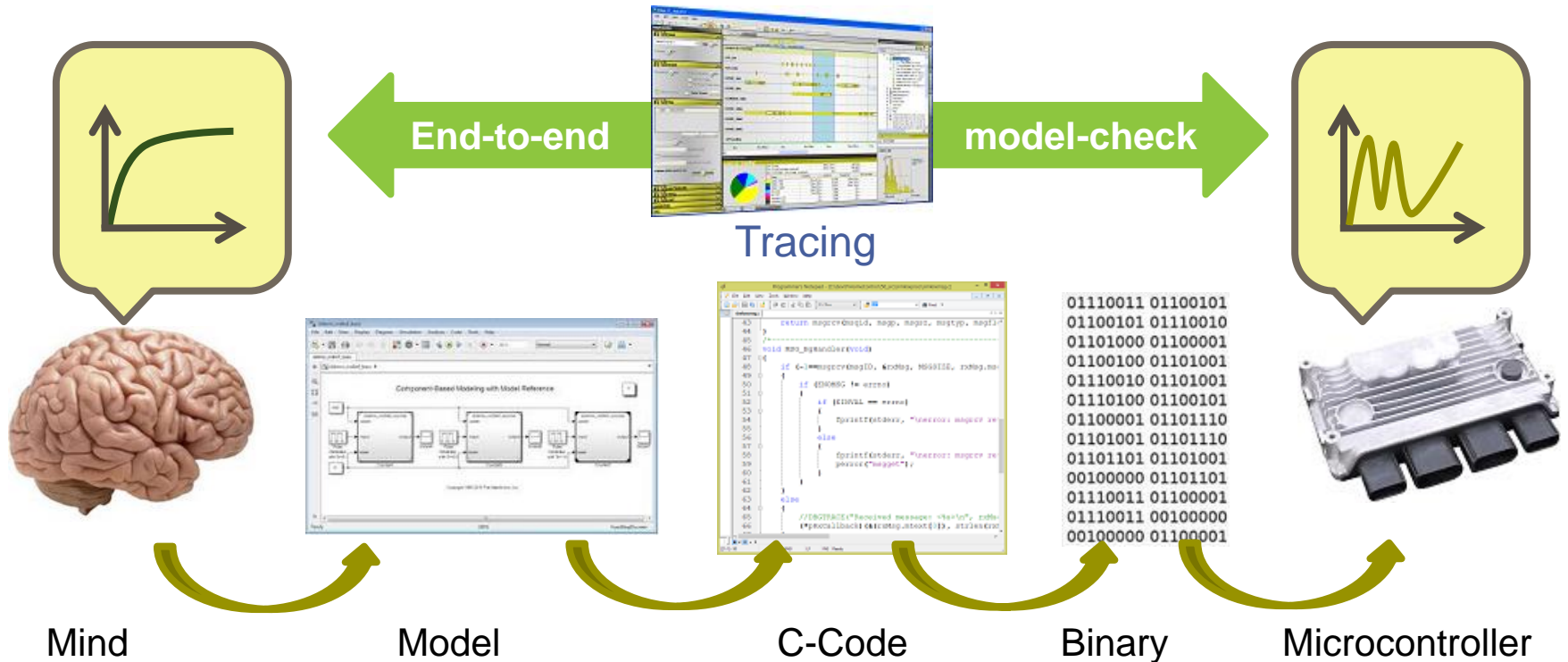
© 2018 GLIWA Embedded Systems. All rights reserved.

# Conclusion



# Tracing: End-to-end model-check

- On its way from the **mind** to the **microcontroller**, an **idea** can suffer from **transition-errors**.
- Tracing allows an **end-to-end model-check**.



## Conclusion

---

- Performance optimization is complex
  - there is no “*press this button to get the perfect software*” solution
- However, tools can significantly reduce the effort
  - In the early phase, in the integration phase, in the late phase
  - On RTOS level, on code level
- Understand your system before starting optimizing
  - Find the critical hot-spots

# Thank you



**Peter Gliwa**  
Dipl.-Ing. (BA)

Geschäftsführer (CEO)

GLIWA GmbH embedded systems  
Pollinger Str. 1  
82362 Weilheim i.OB.  
Germany

fon +49 - 881 - 13 85 22 - 10  
fax +49 - 881 - 13 85 22 - 99  
mobile +49 - 177 - 2 57 86 72

[peter.gliwa@gliwa.com](mailto:peter.gliwa@gliwa.com)  
[www.gliwa.com](http://www.gliwa.com)