Best practice for timing optimization

Optimization on RTOS-level and code-level

Embedded Software Engineering Congress 2018

Peter Gliwa, Dr. Nicholas Merriam, Alexander Stassis – Version 1, CC6

Contents

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

RTOS
top down
Code

(*) 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*

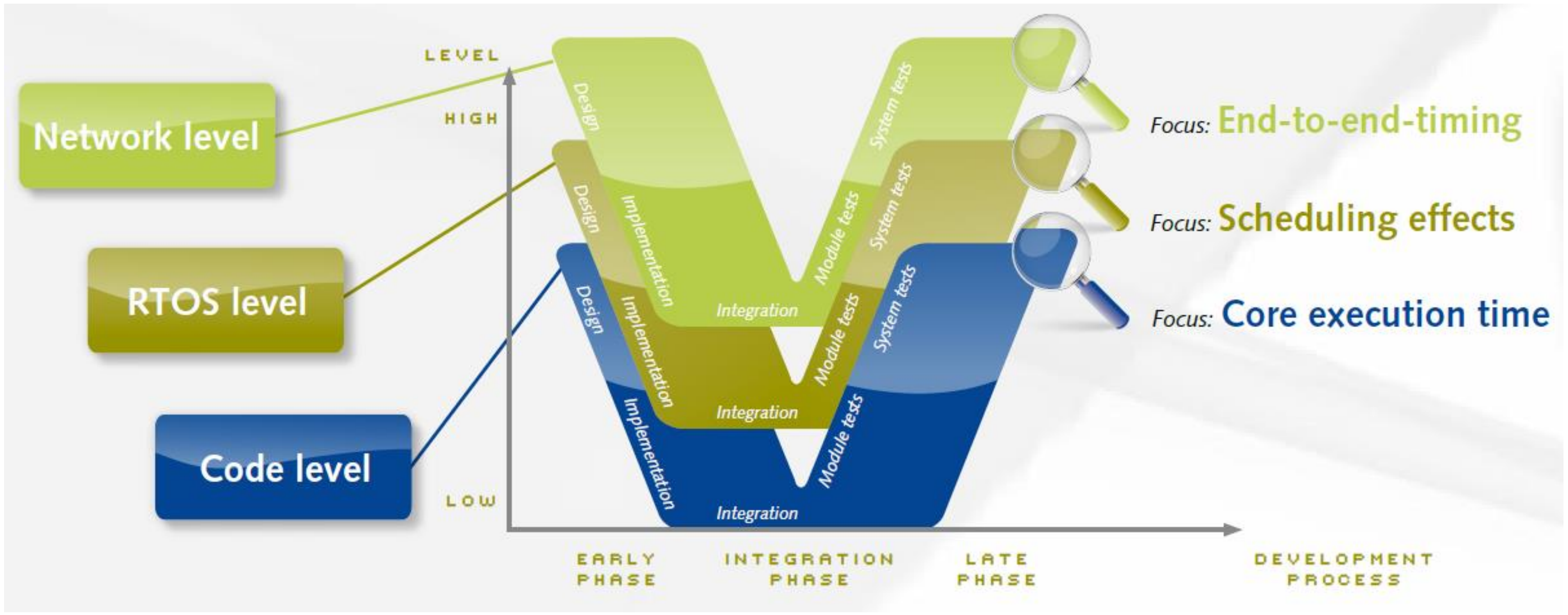
- **Network level**
  - inter ECU communication
  - end-to-end-timing
  - typically OEM business
- **RTOS level (also: scheduling level)**
  - one scheduling entity
  - scheduling effects
  - typically tier-1 business
- **code level**
  - fragment of code (e.g. function)
  - Scheduling not regarded.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

Level

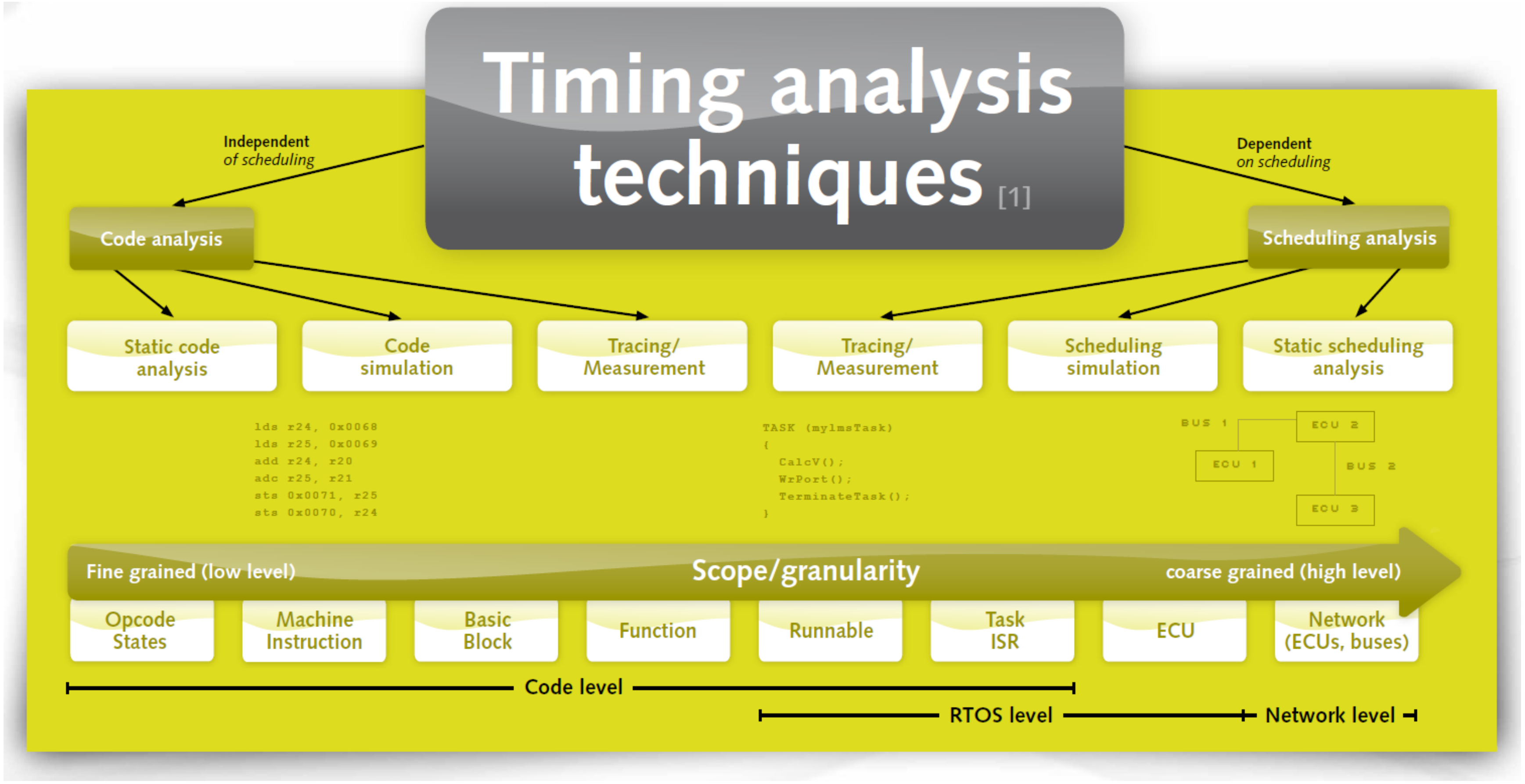
- **Early phase**
  - timing requirements
  - Timing design
  - Hardware selection
  - OS-config, mapping to cores
- **Integration phase**
  - Debug
  - Optimize
- **Late phase**
  - Verify timing against requirements (→ tests)
  - Document actual timing
  - Permanently supervise timing on ECU

Development phase

Two dimensions: *level* and *development phase*



Overview of timing analysis techniques

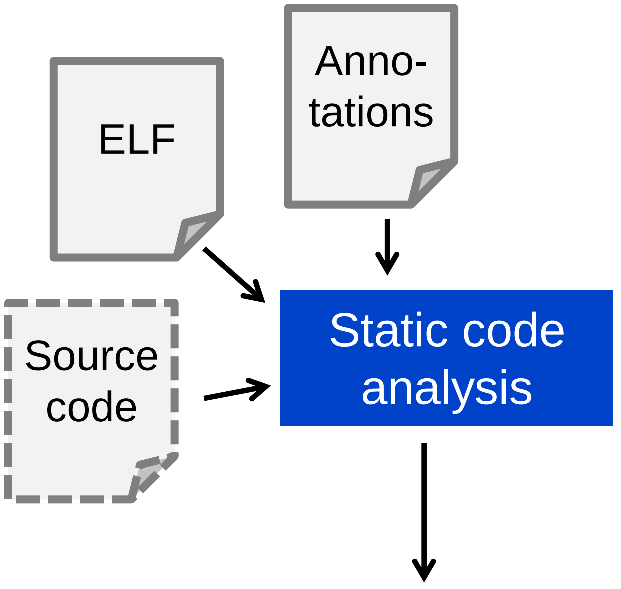


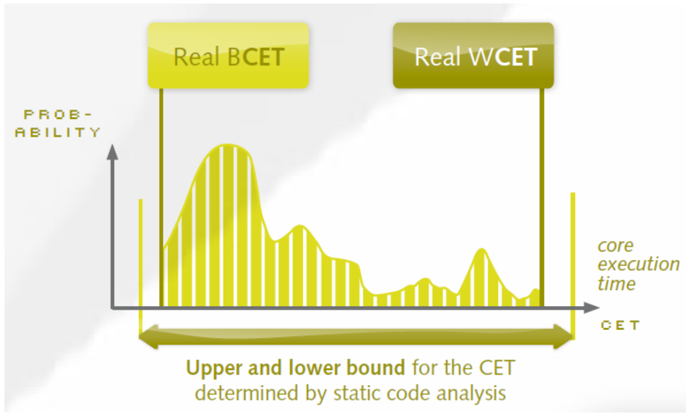
```
lds r24, 0x0068
lds r25, 0x0069
add r24, r20
adc r25, r21
sts 0x0071, r25
sts 0x0070, r24
```

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

Static code analysis

- 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



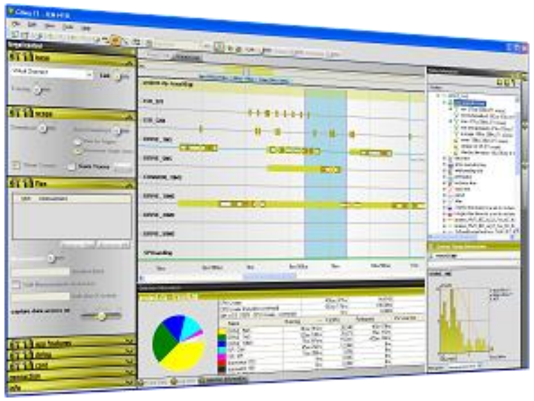


Code simulation

- 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.

Measurement / Tracing

- 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

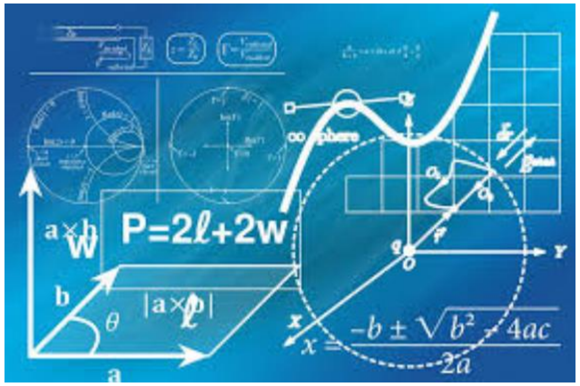
“ ”

- Scheduling Tracing
  - produces traces which can be viewed and from which timing parameters can be derived

“ ”

Static scheduling analysis

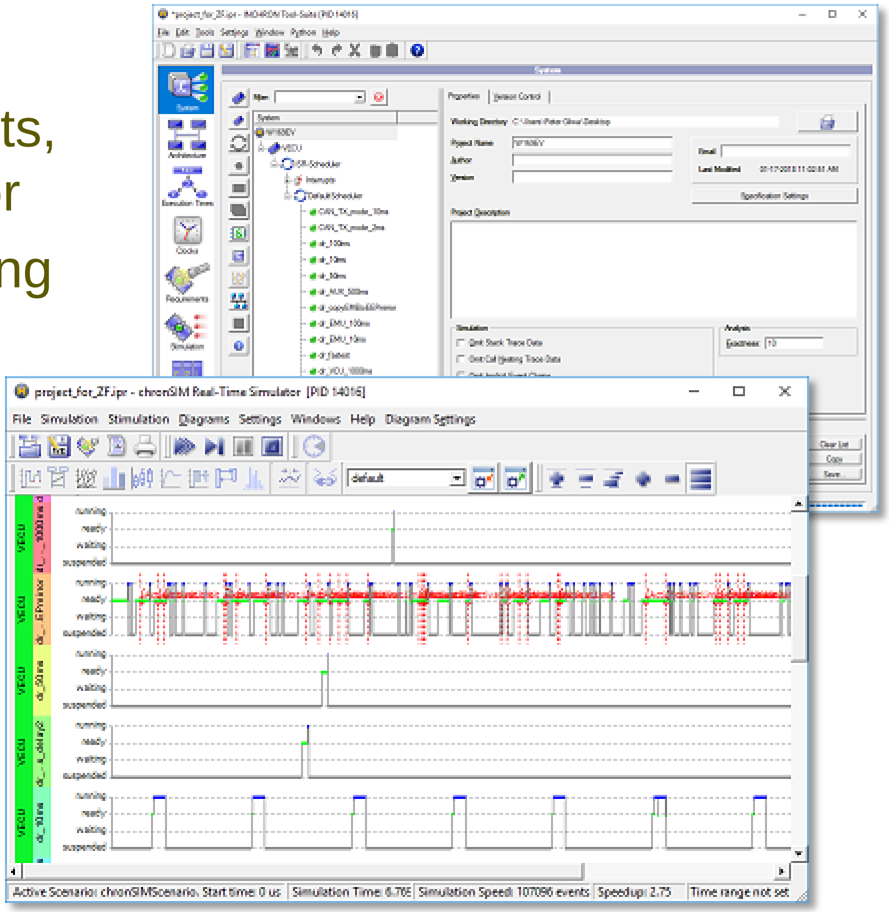
$$RT_i = CET_i + JIT_i + \sum_{j \in \mathrm{hp}(i)} CET_j \left\lceil \frac{RT_i}{T_j} \right\rceil \leq DL_i$$



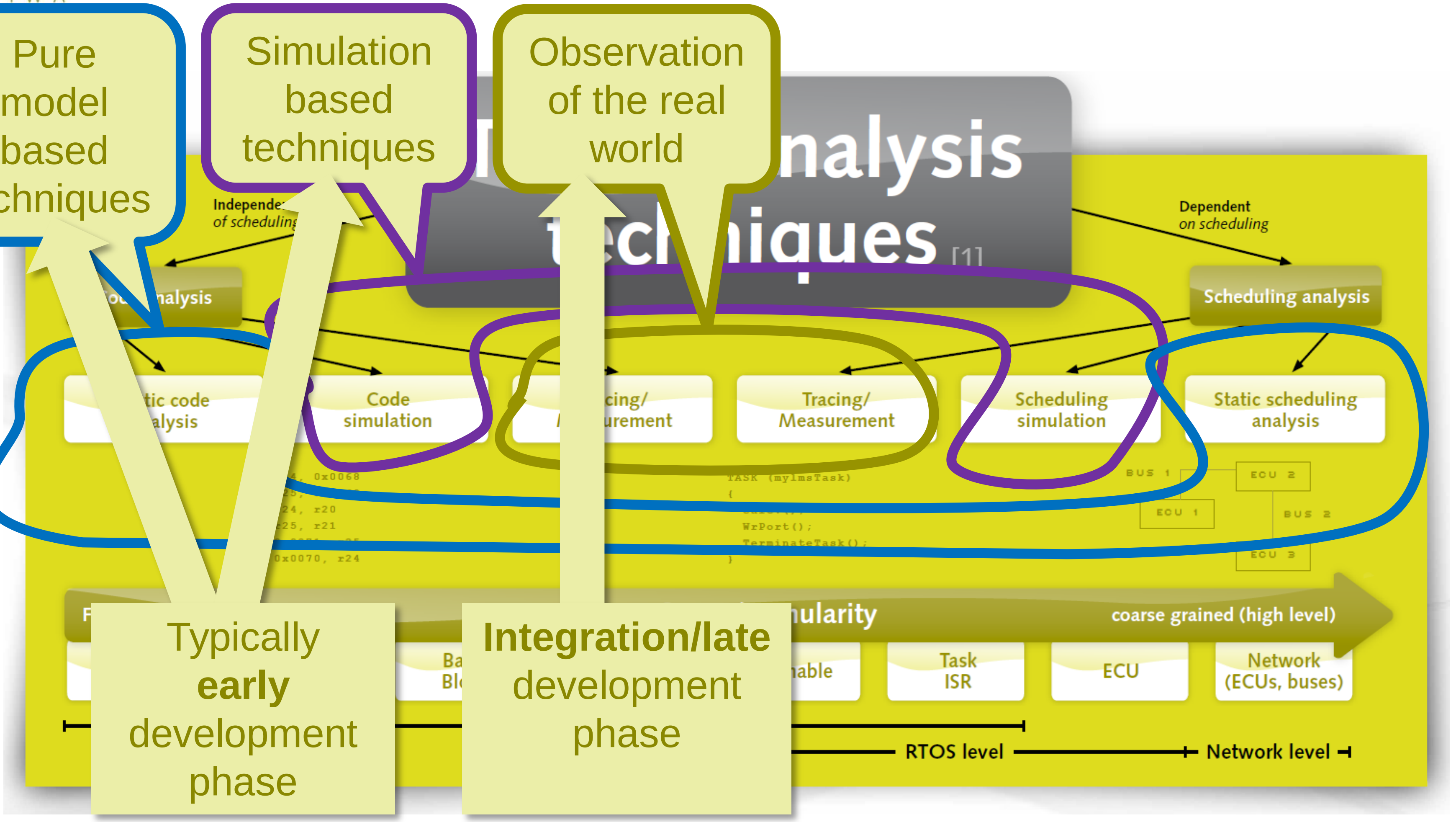
- Input: scheduling model and min/max execution times
- Calculates (no simulation!) the worst case scheduling situation for a given timing parameter, e.g. the WCRT of task A.
- No code or hardware required.
- The execution times fed into the analysis can be either budgets, estimations, or outputs from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

Static scheduling simulation

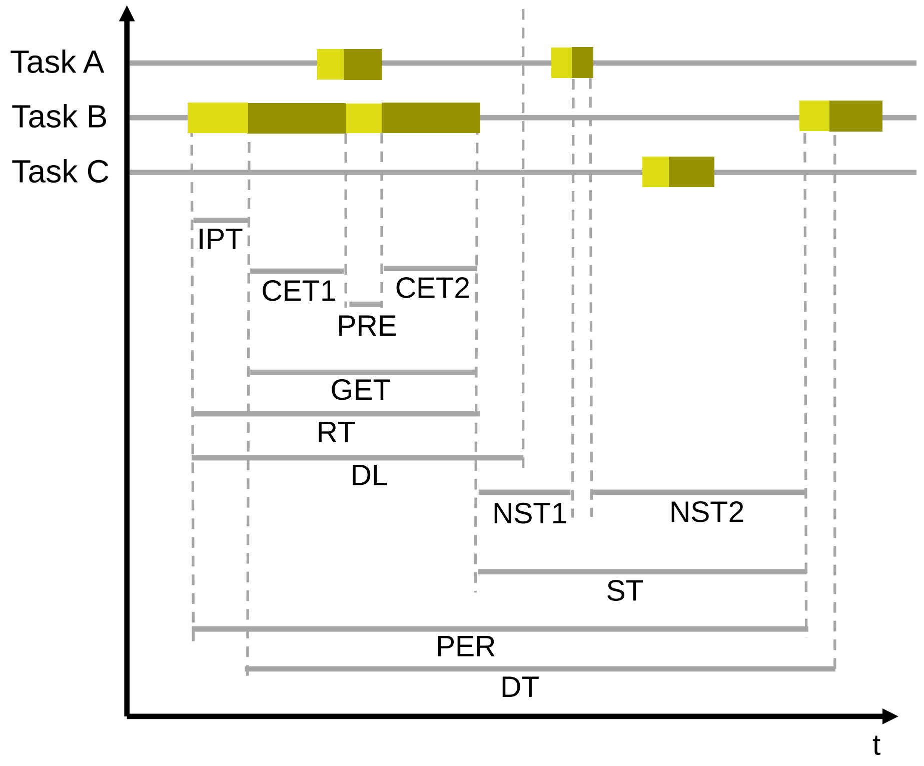
- Similar functionality as the scheduling analysis
- Instead of *calculating* the results, they *simulate* run time behavior
- Main output: the observed timing information and generated traces


*project_for_ZF.ipr - INCHRON Tool-Suite [PID 14016]
project_for_ZF.ipr - chronSIM Real-Time Simulator [PID 14016]
File Simulation Stimulation Diagrams Settings Windows Help Diagram Settings
default
running
ready
waiting
suspended
VECU
Active Scenario: chronSIMScenario. Start time: 0 us
Simulation Time: 6.78
Simulation Speed: 107096 events
Speedup: 2.75
Time range not set

Overview of timing analysis techniques


Pure model based techniques
Simulation based techniques
Observation of the real world
Timing analysis techniques [1]
Independent of scheduling
Dependent on scheduling
Code analysis
Scheduling analysis
Static code analysis
Code simulation
Tracing/ Measurement
Tracing/ Measurement
Scheduling simulation
Static scheduling analysis
Granularity
coarse grained (high level)
Basic Block
Runnable
Task ISR
ECU
Network (ECUs, buses)
RTOS level
Network level
Typically early development phase
Integration/late development phase

Timing parameters

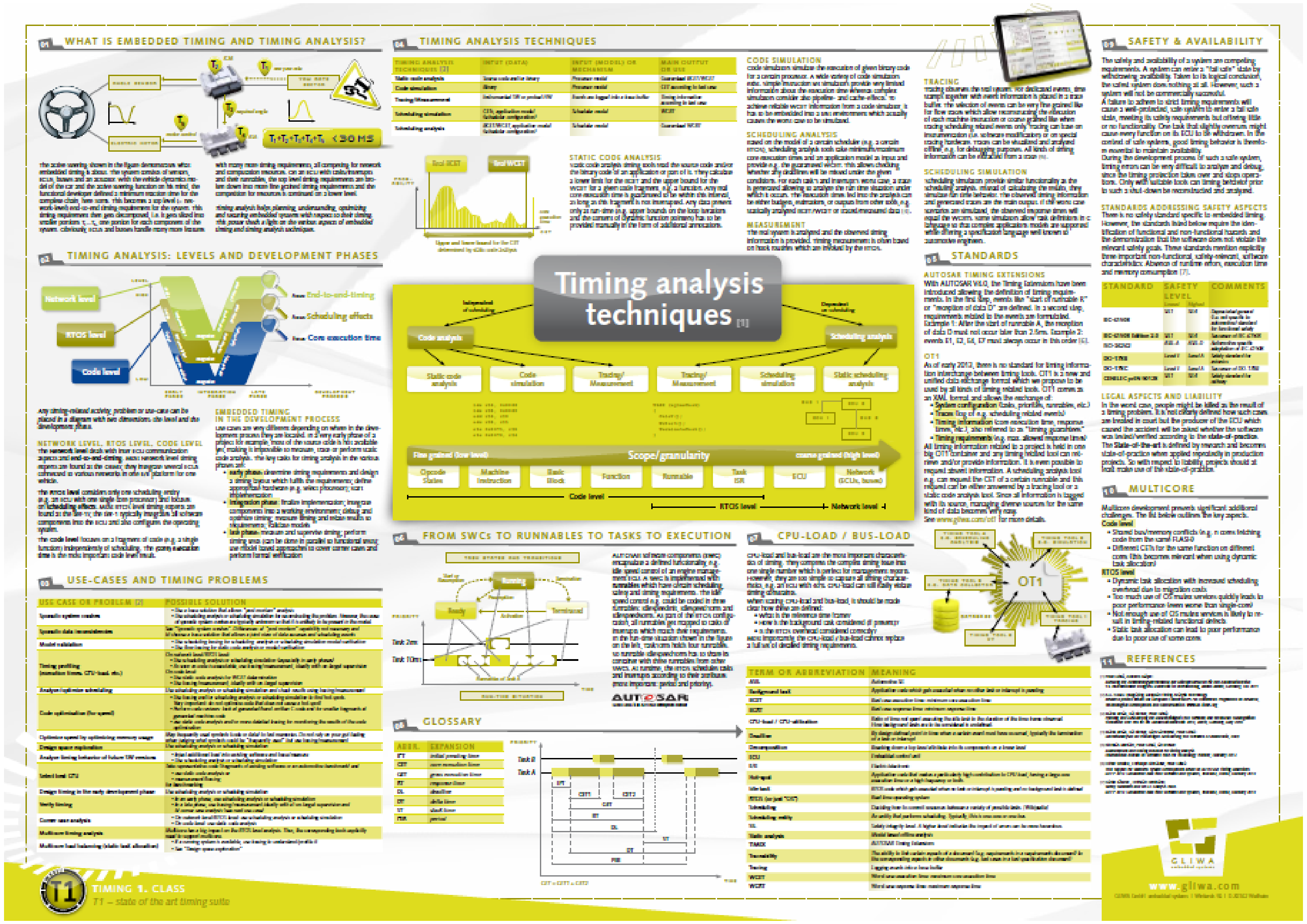


CET = CET1 + CET2

NST = NST 1 + NST2

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

Timing Poster – get your copy!



01 WHAT IS EMBEDDED TIMING AND TIMING ANALYSIS?

02 TIMING ANALYSIS: LEVELS AND DEVELOPMENT PHASES

03 USE-CASES AND TIMING PROBLEMS

04 TIMING ANALYSIS TECHNIQUES

Timing analysis techniques [1]

05 STANDARDS

06 FROM SWCs TO RUNNABLES TO TASKS TO EXECUTION

07 CPU-LOAD / BUS-LOAD

08 GLOSSARY

09 SAFETY & AVAILABILITY

10 MULTICORE

11 REFERENCES

TIMING 1. CLASS

T1 – state of the art timing suite

www.gliwa.com

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

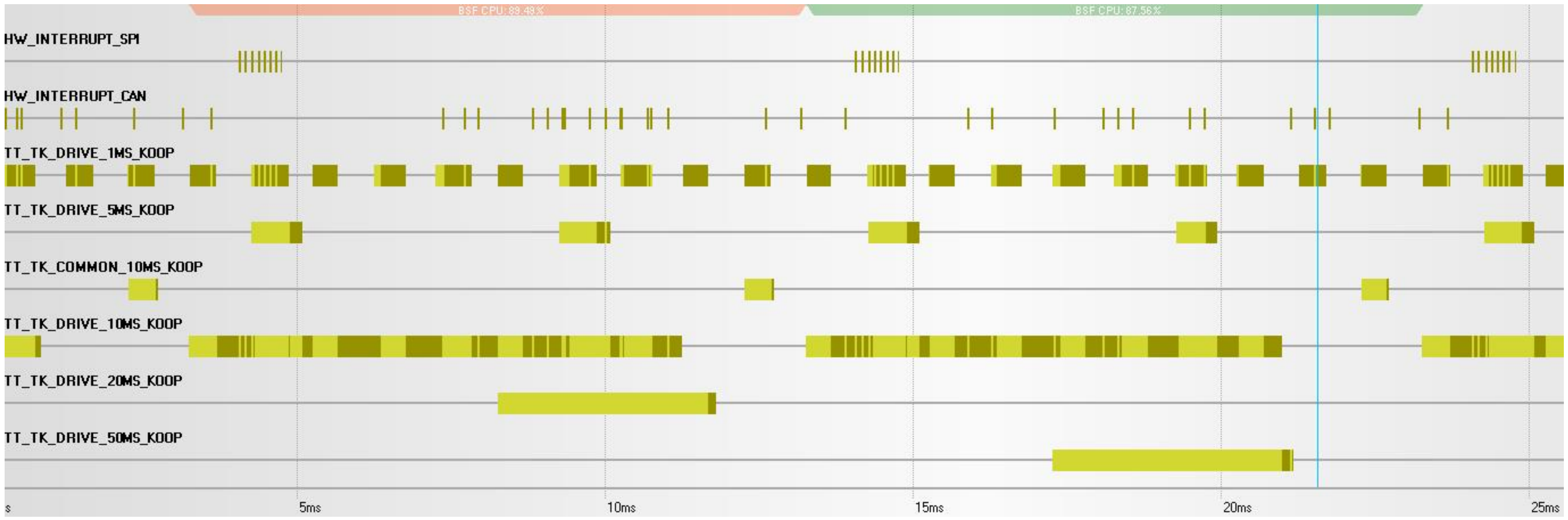
RTOS
top
down
Code

- 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

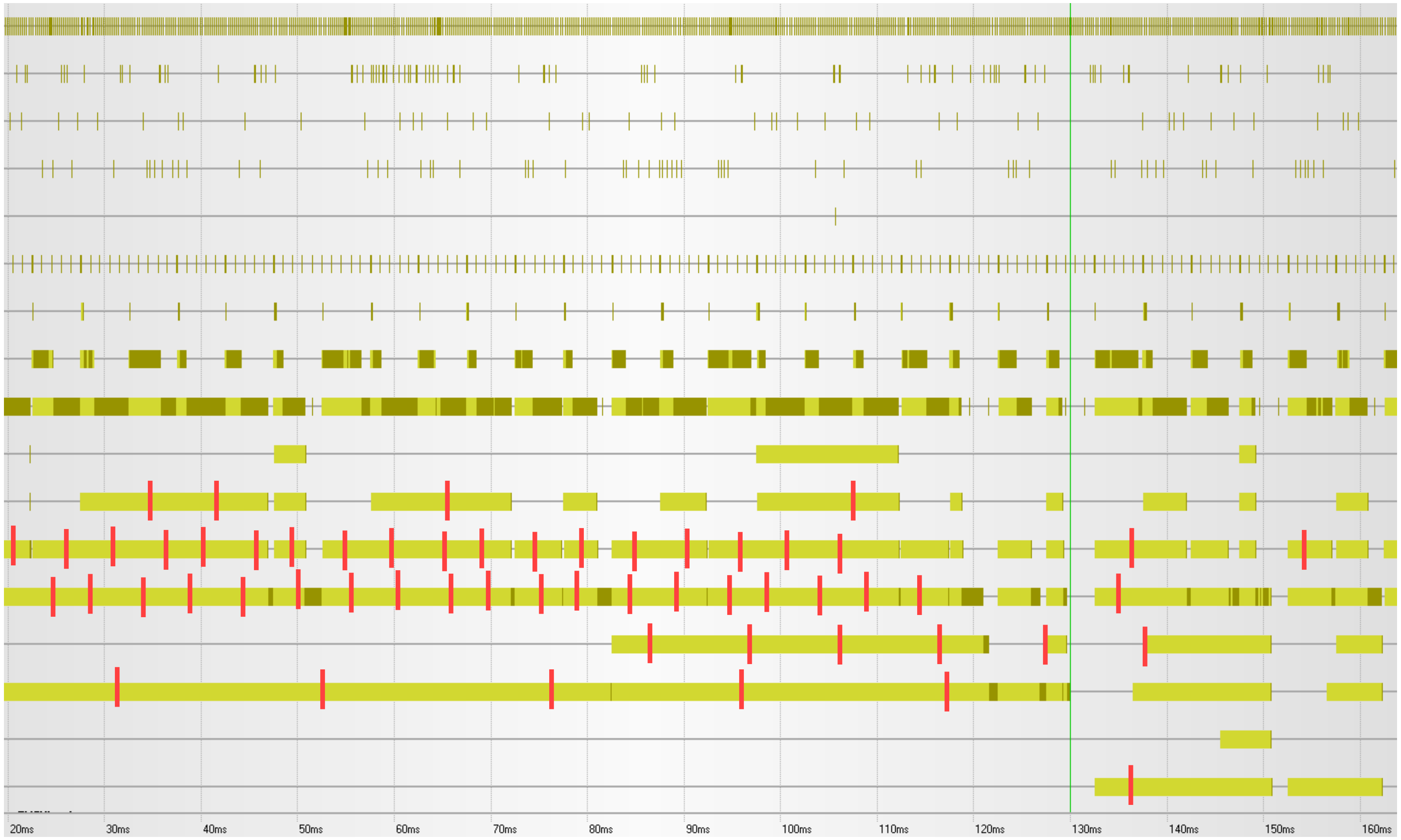


- Highly loaded (up to 93%)
- As a result of optimizations, a less powerful (and cheaper) processor than in the previous generation could be used
- Cooperative scheduling avoiding costly protection mechanisms

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()`** / **`ReleaseSpinnLock()`** 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

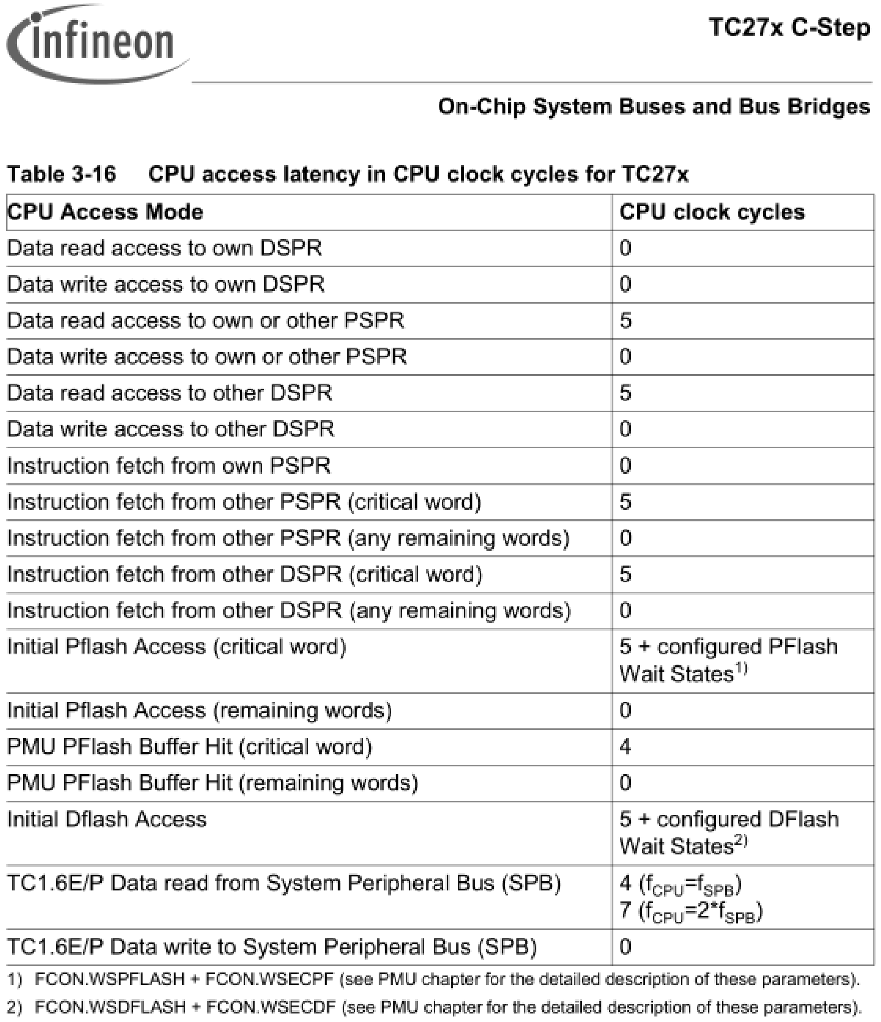
CET per Byte

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

Memory read access times: AURIX™ manual



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 States1)
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 States2)
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).

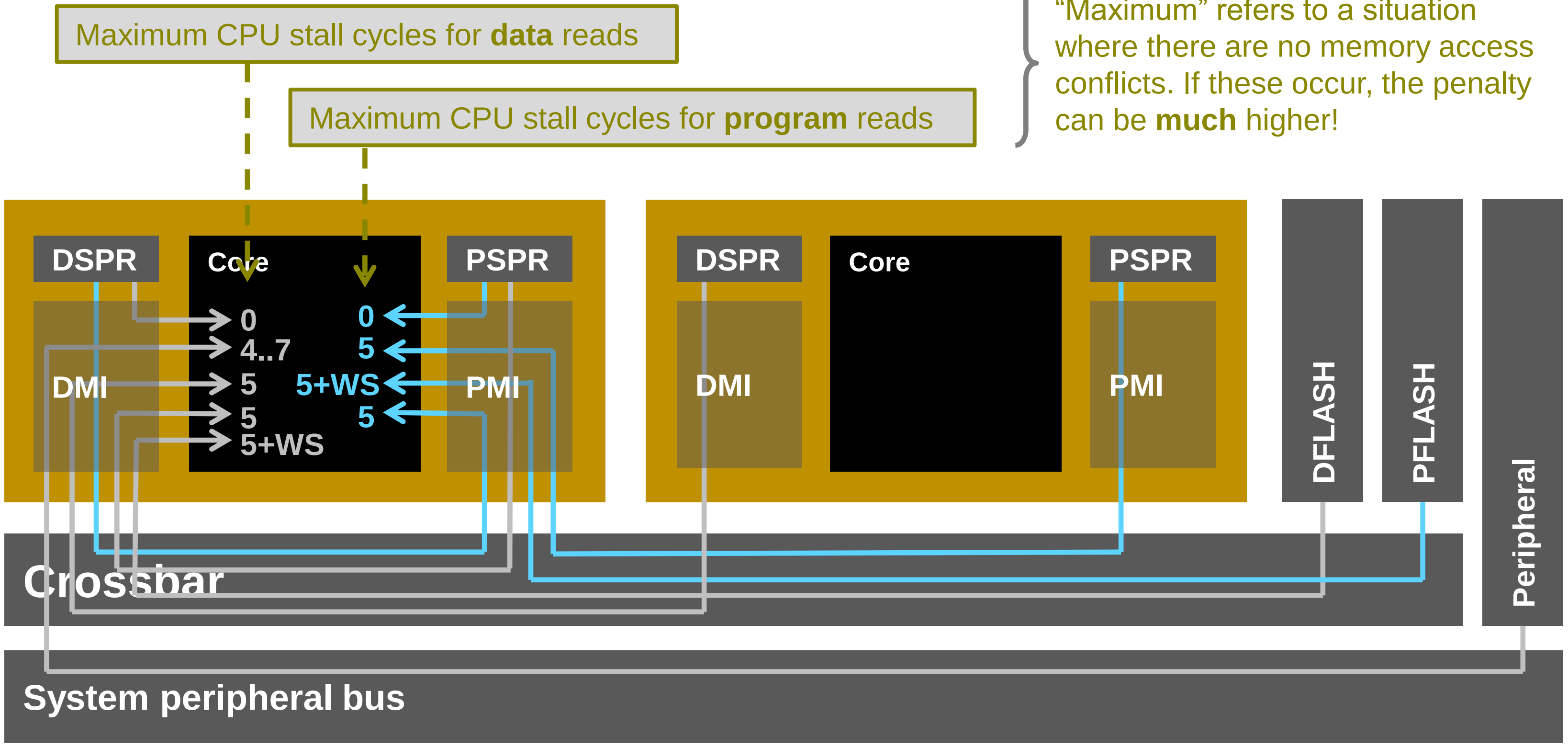
2) FCON.WSDFLASH + FCON.WSECDF (see PMU chapter for the detailed description of these parameters).

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



data read access	DSPR = data scratch pad RAM	DMI = data memory interface
program read access	PSPR = program scratch pad RAM	PMI = programmemory 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	**18.6ns**
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	**14.7ns**

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.

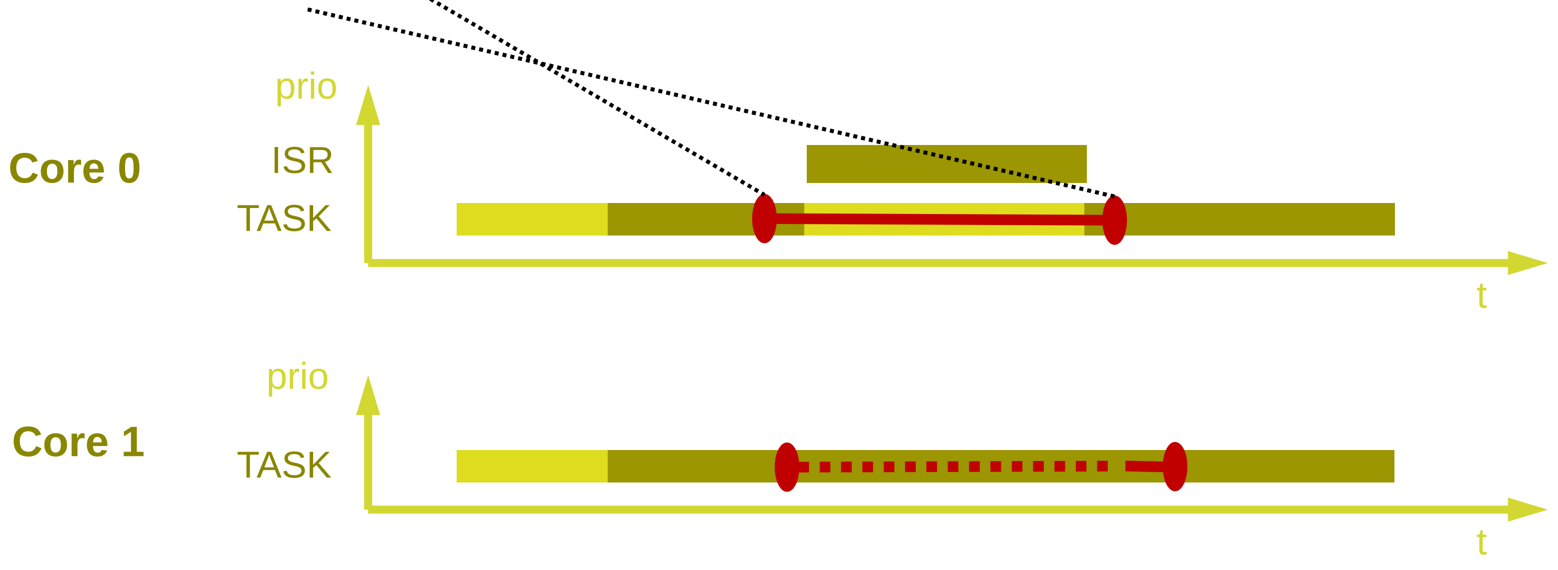
- **`TryToGetSpinlock`** is a non-blocking version of `GetSpinlock`. It always returns immediately with no spinning.

- **`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.

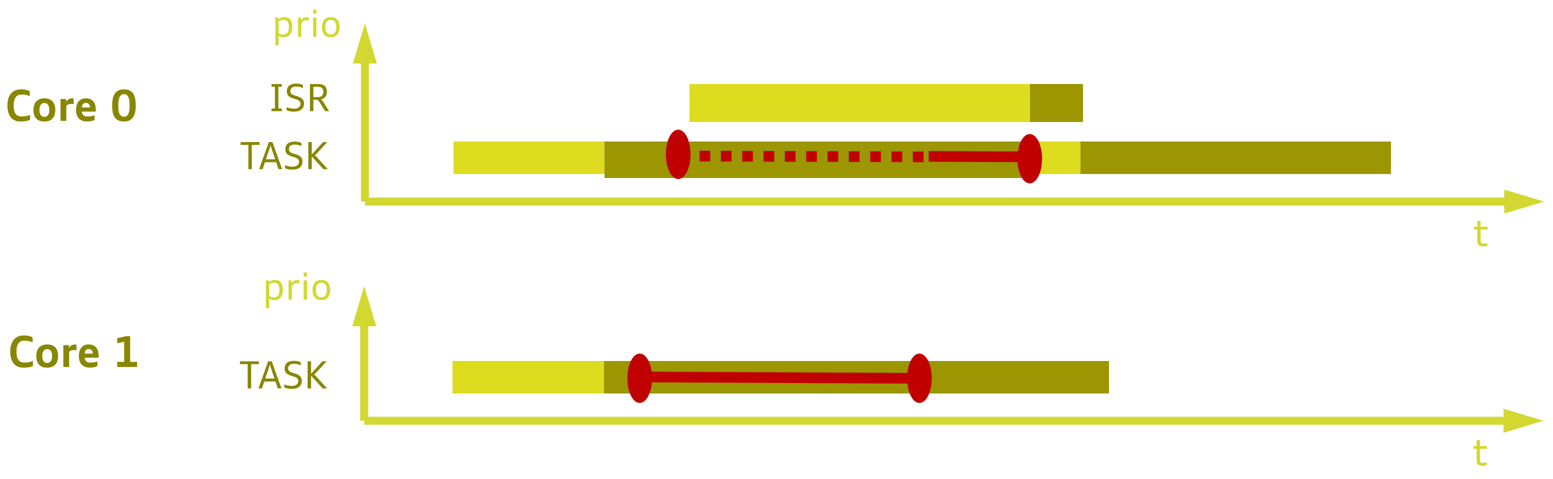
```
GetSpinlock(spinlock);
... /* do what you need to do with spinlock obtained */
ReleaseSpinlock(spinlock);
```



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( );
```

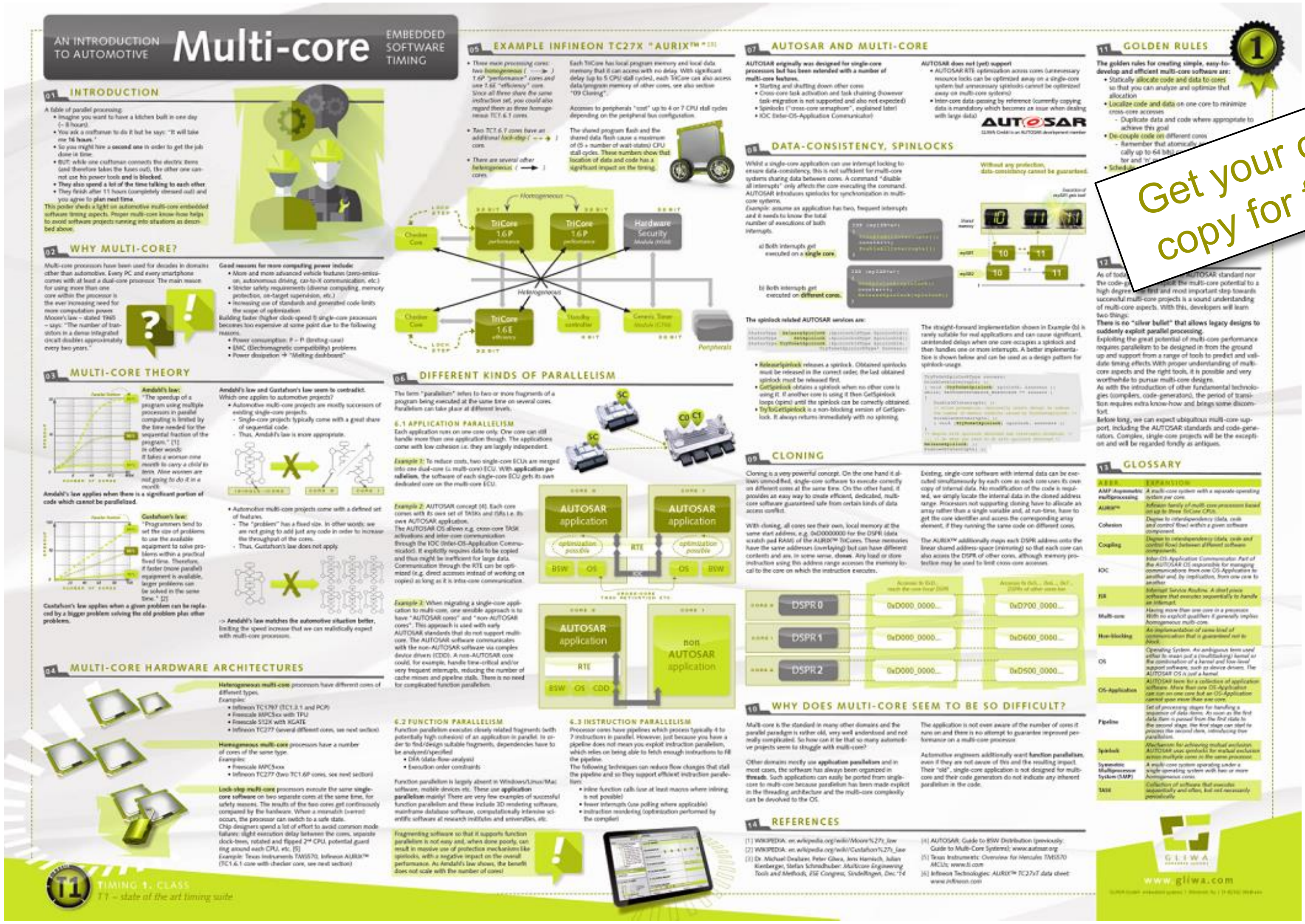


Spinlocks – `TryToGetSpinlock`: best practice

```
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



AN INTRODUCTION TO AUTOMOTIVE **Multi-core** EMBEDDED SOFTWARE TIMING

01 INTRODUCTION

02 WHY MULTI-CORE?

03 MULTI-CORE THEORY

04 MULTI-CORE HARDWARE ARCHITECTURES

05 EXAMPLE INFINEON TC27X "AURIX™"

06 DIFFERENT KINDS OF PARALLELISM

07 AUTOSAR AND MULTI-CORE

08 DATA-CONSISTENCY, SPINLOCKS

09 CLONING

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

11 GOLDEN RULES

13 GLOSSARY

14 REFERENCES

T1 TIMING 1. CLASS

T1 – state of the art timing suite

www.gliwa.com

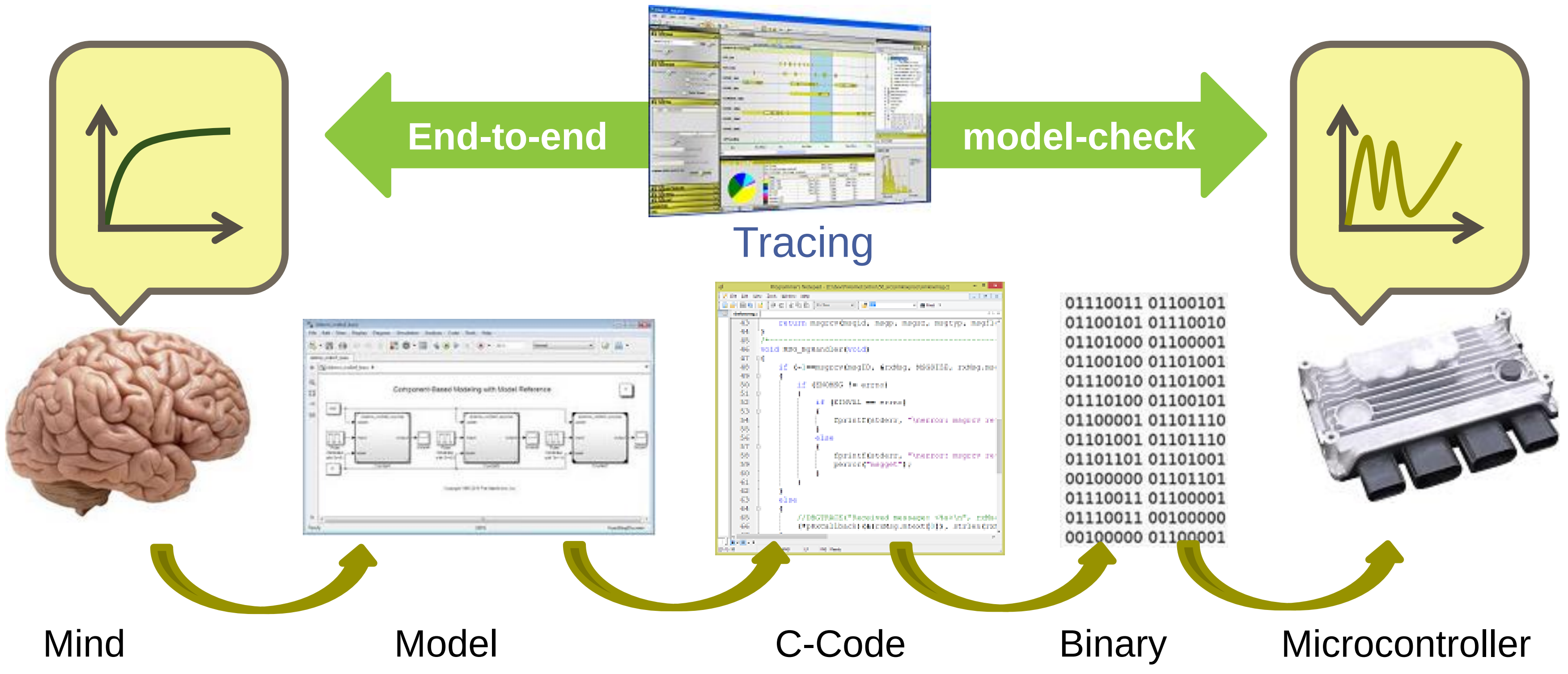
Get your own copy for free!

Conclusion

A CONCLUSION IS THE PLACE...
WHERE YOU GOT TIRED OF THINKING.

Tracing: End-to-end model-check

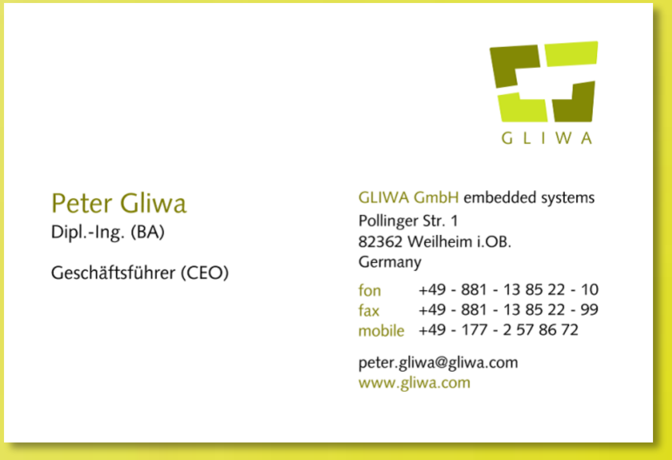
- On its way from the **mind** to the **microcontroller**, an **idea** can suffer from **transition-errors.**
- Tracing allows an **end-to-end model-check**.



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



GLIWA

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
www.gliwa.com