Best practice for timing optimization

Optimization on RTOS-level and code-level

Embedded Software Engineering Congress 2018
• 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”
- 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

- **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

---

**Level**

**Development phase**
Two dimensions: level and development phase
Overview of timing analysis techniques
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.
• 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

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

- 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
Overview of timing analysis techniques

Pure model based techniques

Simulation based techniques

Observation of the real world

Typically early development phase

Integration/late development phase
Timing parameters

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

CET = CET1 + CET2

NST = NST1 + NST2
Timing Poster – get your copy!
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.
/*----------------------------- The ‘standard’ memcpy routine -----------------------------*/

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;
}

memcpy

*/
Step 0: non optimized version (starting point)

<table>
<thead>
<tr>
<th>Default Memory Locations</th>
<th>CET to Copy 1024 Bytes</th>
<th>CET to Copy 1 Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Code</td>
<td>pDest</td>
<td>pSrc</td>
</tr>
<tr>
<td>Cached Flash0</td>
<td>LMU RAM</td>
<td>Cached Flash0</td>
</tr>
</tbody>
</table>

Assembly code

```assembly
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

CET per Byte
Memory read access times: AURIX™ manual

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.

<table>
<thead>
<tr>
<th>CPU Access Mode</th>
<th>CPU clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data read access to own DSPR</td>
<td>0</td>
</tr>
<tr>
<td>Data write access to own DSPR</td>
<td>0</td>
</tr>
<tr>
<td>Data read access to own or other PSPR</td>
<td>5</td>
</tr>
<tr>
<td>Data write access to own or other PSPR</td>
<td>0</td>
</tr>
<tr>
<td>Data read access to other DSPR</td>
<td>5</td>
</tr>
<tr>
<td>Data write access to other DSPR</td>
<td>0</td>
</tr>
<tr>
<td>Instruction fetch from own PSPR</td>
<td>0</td>
</tr>
<tr>
<td>Instruction fetch from other PSPR (critical word)</td>
<td>5</td>
</tr>
<tr>
<td>Instruction fetch from other PSPR (any remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>Instruction fetch from other DSPR (critical word)</td>
<td>5</td>
</tr>
<tr>
<td>Instruction fetch from other DSPR (any remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>Initial PFlash Access (critical word)</td>
<td>5 + configured PFlash Wait States(^1)</td>
</tr>
<tr>
<td>Initial PFlash Access (remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>PMU PFlash Buffer Hit (critical word)</td>
<td>4</td>
</tr>
<tr>
<td>PMU PFlash Buffer Hit (remaining words)</td>
<td>0</td>
</tr>
<tr>
<td>Initial DFlash Access</td>
<td>5 + configured DFlash Wait States(^2)</td>
</tr>
<tr>
<td>TC1.6E/P Data read from System Peripheral Bus (SPB)</td>
<td>(\frac{f_{CPU}}{f_{SPB}})</td>
</tr>
<tr>
<td>TC1.6E/P Data write to System Peripheral Bus (SPB)</td>
<td>0</td>
</tr>
</tbody>
</table>

\(^1\) FCON.WSFFLASH + FCON.WSECDF (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).
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!

- DSPR = data scratch pad RAM
- PSPR = program scratch pad RAM
- DMI = data memory interface
- PMI = program memory interface

Data read access
Program read access

- Crossbar
- System peripheral bus

Core

DMI

0

0

4..7

5

5+WS

5

5+WS

PMI
Step 1: Use different memory locations

<table>
<thead>
<tr>
<th>Code/Data Memory Locations</th>
<th>CET per byte for 1024 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function Code</strong></td>
<td><strong>pDest</strong></td>
</tr>
<tr>
<td>Cached Flash0</td>
<td>LMU RAM</td>
</tr>
<tr>
<td></td>
<td>LMU RAM</td>
</tr>
<tr>
<td></td>
<td>Local DSPR0</td>
</tr>
<tr>
<td>Local PSPR0</td>
<td>LMU RAM</td>
</tr>
<tr>
<td></td>
<td>LMU RAM</td>
</tr>
<tr>
<td></td>
<td>Local DSPR0</td>
</tr>
<tr>
<td>Un-Cached Flash0</td>
<td>Local DSPR0</td>
</tr>
<tr>
<td>PSPR1</td>
<td>Local DSPR0</td>
</tr>
</tbody>
</table>
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:

```
memcpy_:
  mov.aa      a15,a4
  jlez        d4,0x8020012a
  mov.a       a2,d4
  add.a       a2,#-0x1
  ld.bu       d15,[a5+]0x1
  st.b        [a15+]0x1,d15
  loop        a2,0x80200124
  mov.aa      a2,a4
  ret
```

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Description</th>
<th>MAX</th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasking</td>
<td>Enabling post-increment load and store operations and Loop instruction</td>
<td>65.4ns</td>
<td>59.6ns</td>
</tr>
</tbody>
</table>
**Step 3: manual optimizations**

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

```c
/* 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 /* Else do Manual Loop Unrolling with Switch Case Above */
{
    ....
```
Step 3: manual optimizations (results)

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

Good result!

Best result!
Spinlocks
and how not to use them
Spinlocks – Overview

- **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.

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

```c
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!
Conclusion

A CONCLUSION IS THE PLACE...

WHERE YOU GOT TIRED OF THINKING.
• 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