Software Timing Analysis for Safe and Secure Embedded Systems

Peter Gliwa
Safety & Security / Embedded Testing
June 2019, Munich
Software Timing Analysis for Safe and Secure Embedded Systems

Peter Gliwa
Safety & Security / Embedded Testing
June 2019, Munich
Embedded architecture vs. dancing

- **Single-core (e.g. OSEK)**
  - Cf. one single guy doing break-dance
  - High performance core

- **Multi-core (e.g. AUTOSAR CP)**
  - Cf. dancing chorus
  - Communication between (often very similar) cores

- **"Any"-core (e.g. POSIX, AUTOSAR AP)**
  - Cf. dancing crowd
  - Rather non-deterministic behavior
  - Difficult to control
Contents

• Introduction
• Timing Basics
• Timing analysis techniques
• Tracing
• Timing and Security
• Conclusion
Introduction
Why care about timing?

- Proper timing of ECU software is essential for
  - Reliability / Availability
  - Safety (and often also Security)

- Timing problems are very often difficult
  - to identify as such, to debug, to solve

- ISO 26262 requires “freedom of interference”
  - Can only be guaranteed in the absence of timing problems

- Multicore
  - If your single-core timing is unstable already, multi-core will be a nightmare

Such indicator does not exist (unfortunately)
Software Development Process

Apply your development processes and methodologies also to the timing of your software!
Timing Basics
OSEK / AUTOSAR CP task states

ECC = Extended Conformance Class
TASK = container for code, e.g. runnables
Timing parameters

<table>
<thead>
<tr>
<th>Abr.</th>
<th>Explanation (EN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPT</td>
<td>initial pending time</td>
</tr>
<tr>
<td>CET</td>
<td>core execution time</td>
</tr>
<tr>
<td>GET</td>
<td>gross execution time</td>
</tr>
<tr>
<td>RT</td>
<td>response time</td>
</tr>
<tr>
<td>DT</td>
<td>delta time</td>
</tr>
<tr>
<td>PER</td>
<td>period</td>
</tr>
<tr>
<td>ST</td>
<td>slack time</td>
</tr>
<tr>
<td>PRE</td>
<td>preemption</td>
</tr>
<tr>
<td>JIT</td>
<td>jitter</td>
</tr>
<tr>
<td>CPU</td>
<td>cpu load</td>
</tr>
<tr>
<td>DL</td>
<td>Deadline</td>
</tr>
<tr>
<td>NST</td>
<td>Net slack time</td>
</tr>
</tbody>
</table>

 Task A
 Task B
 Task C

\[
\text{CET} = \text{CET}_1 + \text{CET}_2 + \text{CET}_3
\]

\[
\text{NST} = \text{NST}_1 + \text{NST}_2
\]
Timing analysis techniques
Overview of timing analysis techniques

Timing analysis techniques

- Code analysis
  - Static code analysis
  - Code simulation
- Scheduling analysis
  - Tracing/Measurement
  - Scheduling simulation
  - Static scheduling analysis

Scope/granularity:
- Fine grained (low level): Opcode, Machine Instruction, Basic Block, Function, Runnable, Task ISR, ECU
- Coarse grained (high level): Network (ECUs, buses)

1. Task (myTask)
   - CalcY();
   - Wait();
   - TerminateTask();

Code level

RTOS level

Network level
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

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

- 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, estimaations, 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
Tracing
Timing-visualization
Timing-optimization
Timing-verification
Timing-supervision
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.
GLIWA

Timing-suite T1

- Visualise timing
- Measure timing
  Observed WCET, CPU load, core execution times, response times, jitter, etc.
- Verify timing
- Proof models
  models of static analysis, simulation
- Debug & optimize
- Suitable for mass-production projects and in-car-use
  - Relaxed bandwidth requirements
  - No HW modifications necessary

ISO 26262 ASIL-D certified!

Products for timing- and stack analysis

Services beyond timing analysis
T1 overview

T1-HOST-SW
PC based SW tool for visualization, analysis and configuration

T1-TARGET-SW
Embedded software component which traces, analyses and supervises at run-time
T1 on multi-core (TC399 with 6 cores)
Typical timing test phases

• Collect timing information
  – **Use-case:** Gain e.g. max CETs, max RTs, max. CPU-load
  – **Approach:** collect T1.cont results based on dedicated test-cycles or long term tests

• Trigger on error
  – **Use-case:** Understand the system behaviour in fault situations
  – **Approach:**
    • Instrument user code detecting errors with triggers
    • Trigger on T1 timing constraint violation detection

• Document Timing of PoIs (points of interest)
  – **Use-case:** document the correct behaviour in dedicated situations
  – **Approach:** Trigger on POIs
Timing and Security
Side-channel attacks

- Analyzing the timing of crypto routines
  → example: the more characters of a password match, so greater gets the CET of the password check routine

- Countermeasure: make sure the CET is independent of the input/output
Timing-supervision as a “security watchdog”

- Collect timing parameters (CETs, RTs, etc.) during tests
  → determine min./max. value for each parameter.
- These define a valid range (“timing constraints”).
- Measure timing parameters at run-time in the productive code
- Identify potential security problems (e.g. control-flow-hijacking, code injection, etc.) by constraints supervision
Conclusion

A CONCLUSION IS THE PLACE...
WHERE YOU GOT TIRED OF THINKING.
Download PDF from gliwa.com or request a free hard-copy
Contents

- Basics (Compilers, RTOSs, processors)
- Timing theory
- Timing analysis techniques
- Examples from automotive projects
- Timing optimization
- Multi-core, many-core
- AUTOSAR
- Safety, ISO 26262
Summary

• Timing *does* matter – better not find out the hard way!

• Simply apply existing development processes, concepts, etc. also to timing – it is easy!

• Tracing helps you to understand how your system *really* behaves. Optionally in the production SW.

• Tracing simplifies introducing new technologies dramatically.
Thank you