Test und Fehlersuche in komplexen autonomen System

Albert Schulz – albert.schulz@accemic.com

Tagung „Echtzeit 2019“ 21.11.2019

Outline

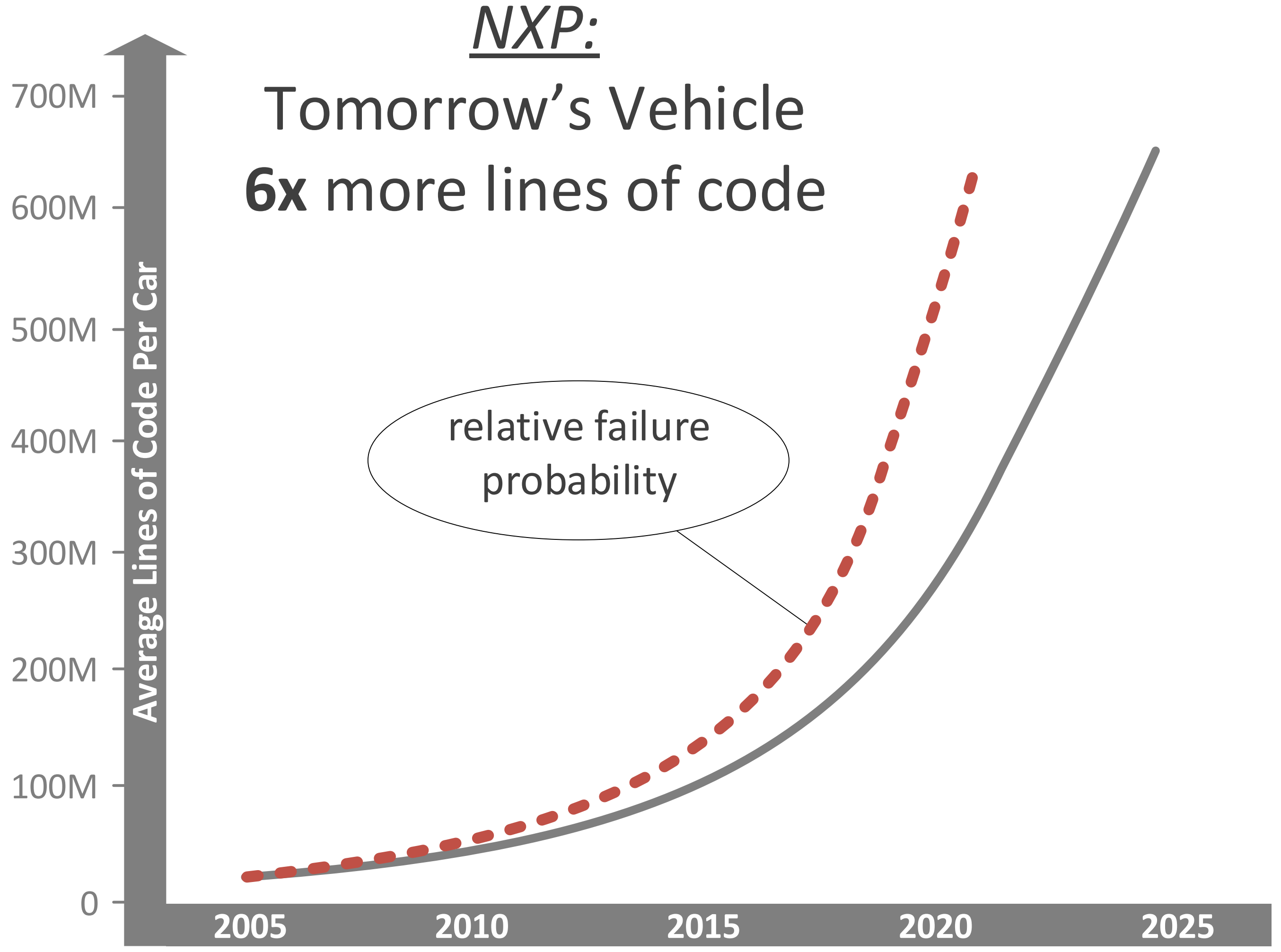
- Introduction
- Hardware Trace
- Dynamic analysis
  - Example: safety-critical control system
- Realtime Code Coverage
- Conclusions

ACCEMIC TECHNOLOGIES

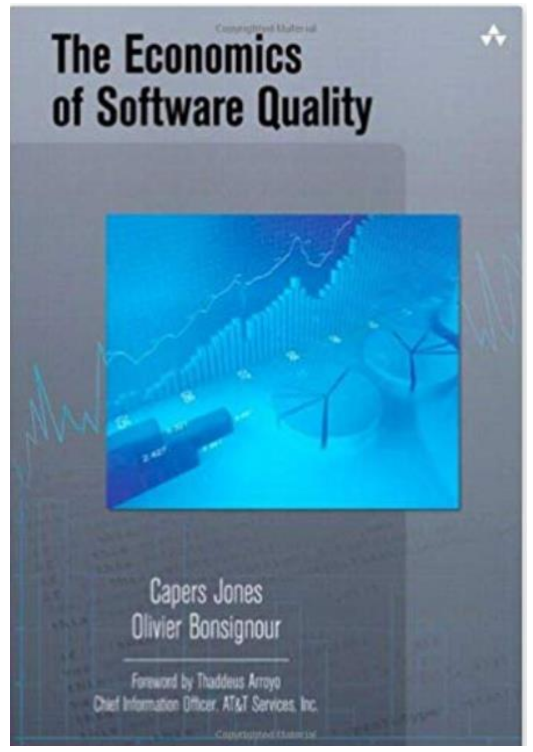
Introduction

- Trend in current systems:
  - autonomous, long runtimes without human interaction
  - Increased complexity → need for multi-core
- Increased chance of bugs, even in post-release code

McKinsey & Company:
*"**Snowballing complexity** is causing **significant software-related quality issues …**"*



Capers Jones:
~5% Post-release defects



Introduction

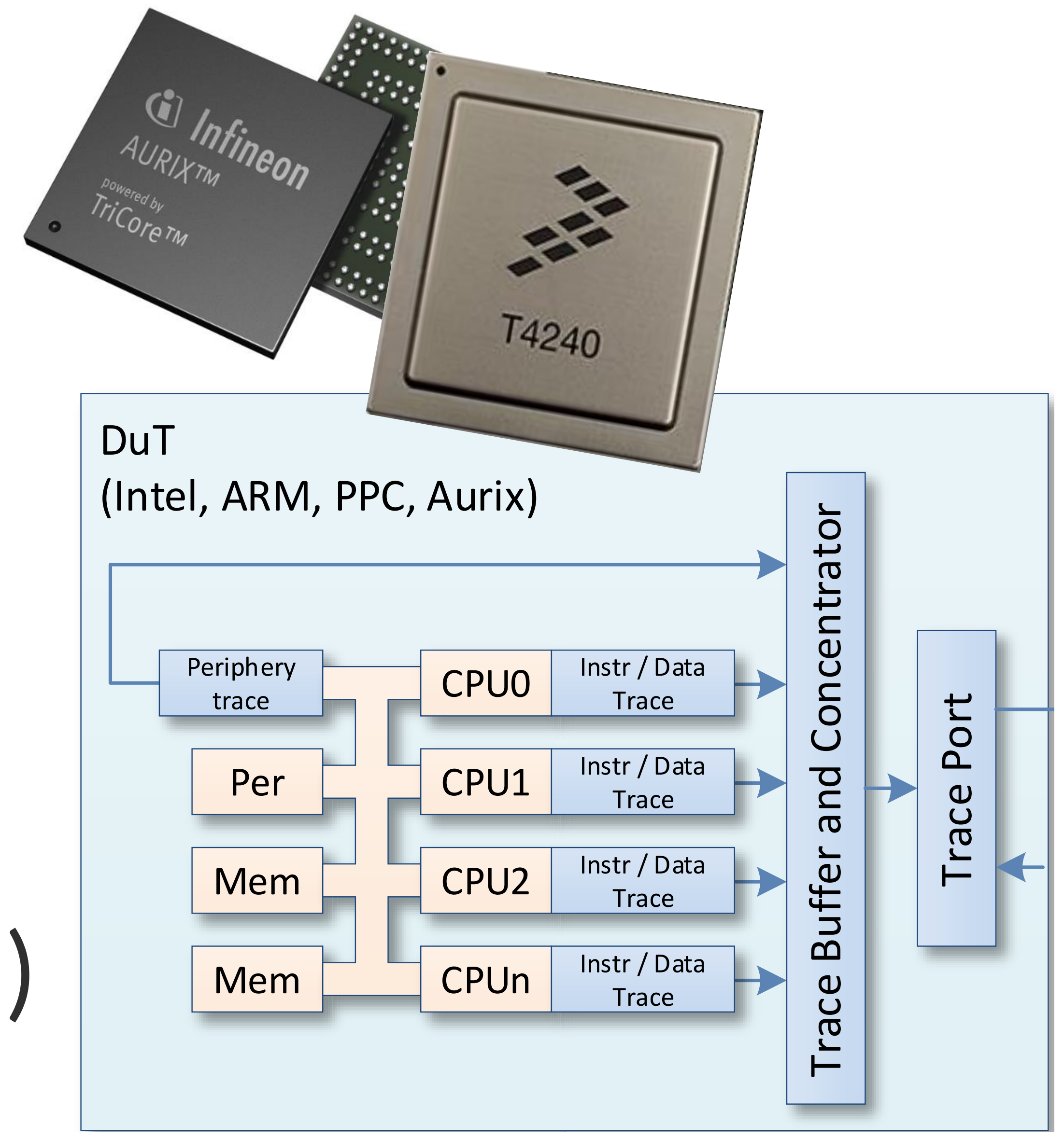
- Certification is challenging for safety-critical systems
- Software instrumentation helps, but interferes with functional Code
  - e.g. code-coverage adds additional code for measurements (e.g. gcov)
  - Software tracing techniques with high overhead in time and space (printf)
- Multi-core makes static analysis challenging
- Certified code contains often additional test code
  - Requires memory space and computation time
- Alternatives?

ACCEMIC TECHNOLOGIES

Hardware Trace

Trace-Information:

- Control-Flow (Branches, Function calls)
- OS-relevant events (context switches),
- Data access (address, data)*,
- Application-specific events (lightweight instrumentation)



Processors with Hardware-Trace Infrastructure:

- Infineon Aurix: Emulation Device
- ARM Cortex-A/-M/-R: CoreSight
- Intel x86: IntelPT
- NXP QorIQ P-series, T-series: Debug Assist Module

*depends on Processor capabilities

ARM® CoreSight™
Architecture Specification

Intel® 64 and IA-32 Architectures
Software Developer's Manual

T4240R2 Advanced QorIQ Debug and Performance Monitoring Reference Manual

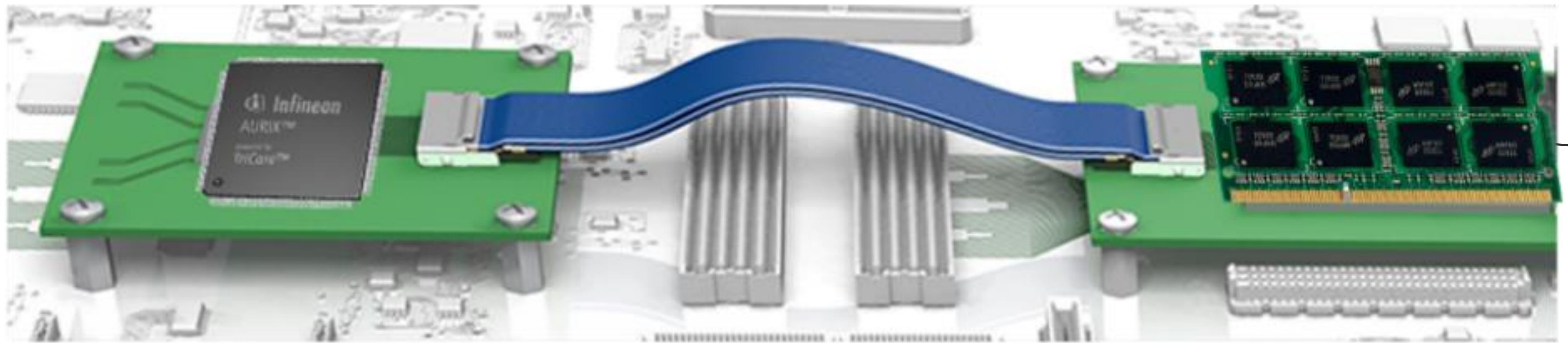
TC29/7/6/3xED
32-Bit Single-Chip Micocontroller

ACCEMIC TECHNOLOGIES

Hardware Trace

State-of-the-art: Offline Analysis (e.g. Lauterbach TRACE32)

Trace-Buffer limits observation time



some GBit/s

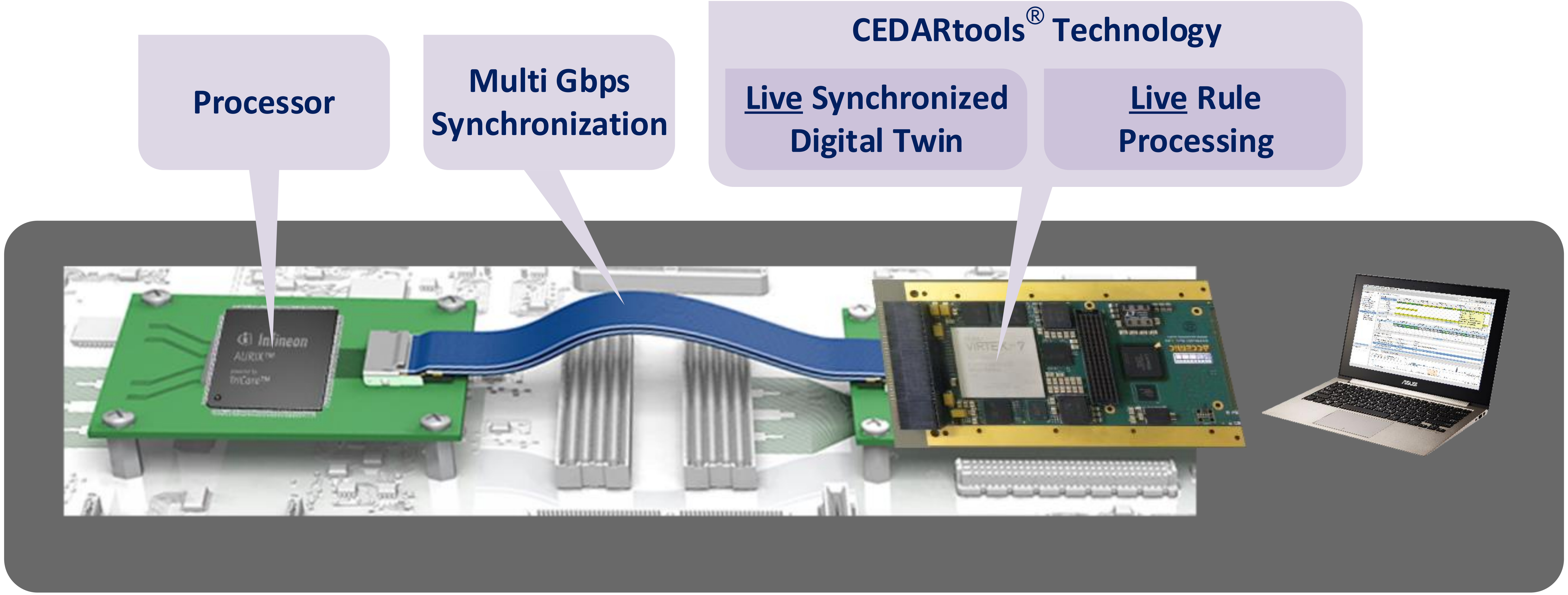
Trace data generation
by processor internal hardware structures

Trace data buffer
by a few GByte RAM buffer

Trace data processing
usually magnitudes slower then generation

User interface
for observation result output

Hardware Trace Processing in Realtime



Dynamic Analysis

Non-intrusive Continuous Timing Verification

- Use case: Safety-critical application to control breaks
- Requirement:
  - Ensure Timing Constraint from **pressing the breaks**, until their **activation**
  - Constraint: Should react within 5ms!

Implementation

```c
void run_task()
{
    float break_angle = read_break_sensor();

    int strength;
    strength = calculate_break_strengh_for_angle(break_angle);

    int motor_control;
    motor_control = calculate_motor_control_value(strength);

    if (motor_control == 1) {
        activate_breaks();
    }
    else if (motor_control == -1) release_breaks();
}
```

```c
float read_break_sensor()
{
    float sensor_value = rand()%91;
    return sensor_value;
}

int calculate_break_strengh_for_angle(float angle)
{
    usleep(1e3+rand()%5*1e3); /// Sleep randomly between 1ms and 5ms
    float strength = angle/10.0;
    return round(strength);
}

int calculate_motor_control_value(int strength)
{
    usleep(1e3+rand()%5*1e3); /// Sleep randomly between 1ms and 5ms
    if (strength > 3) return 1;
    else if (strength == 0) return 0;
    else return -1;
}
```

- **run_task()** executed periodically every second
- **Calculations** have variable execution durations
  - simulates dynamic events due to multicore environment
- **Breaks** are only **activated** sometimes, depending on the break angle

ACCEMIC TECHNOLOGIES

Dynamic Analysis

Non-intrusive Continuous Timing Verification

- Use case: Safety-critical application to control breaks
- Requirement:
  - Ensure Timing Constraint from **pressing the breaks**, until their **activation**
  - Constraint: Should react within 5ms!

Implementation

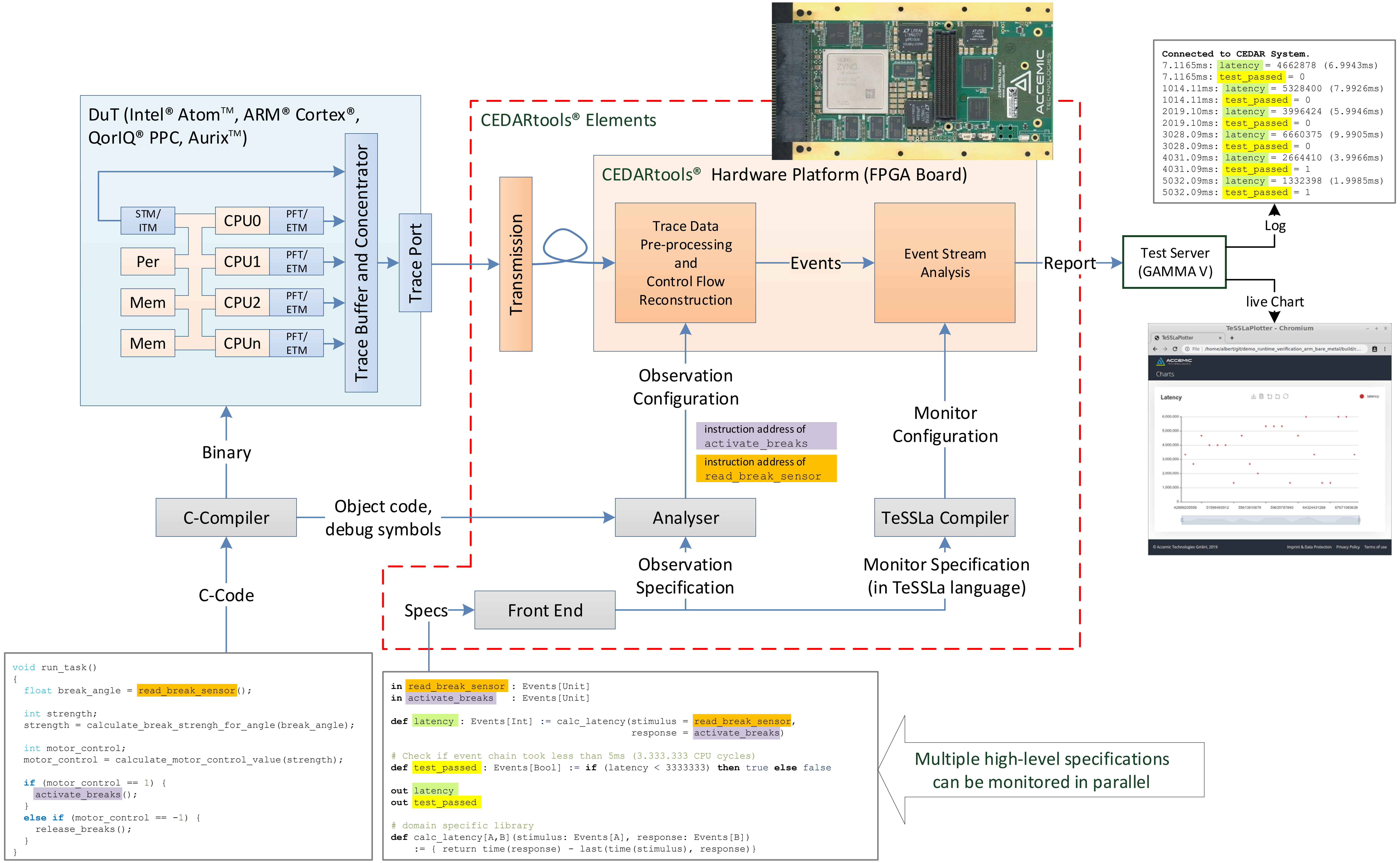
```c
void run_task()
{
    float break_angle = read_break_sensor();

    int strength;
    strength = calculate_break_strengh_for_angle(break_angle);

    int motor_control;
    motor_control = calculate_motor_control_value(strength);

    if (motor_control == 1) {
        activate_breaks();
    }
    else if (motor_control == -1) release_breaks();
}
```

Constraints Specification (TeSSLa)

```
in read_sensor : Events[Unit]
in activate_break : Events[Unit]

def latency : Events[Int] := time(activate_break) - last(time(read_sensor), activate_break)

# Check if event chain took less than 5ms (3.333.333 cycles on ARM A9)
def test_passed : Events[Bool] := if (latency < 3333333) then true else false

out latency
out test_passed
```

Dynamic Analysis

Non-intrusive Continuous Timing Verification



Dynamic Analysis

Non-intrusive Continuous Code Coverage

Continuous and non-intrusive

- Statement Coverage
- Branch Coverage (EX/NEX)
- Performance measurement
  (count executed instructions)

➢ Measured on object code level
➢ Measured on release code
➢ No instrumentation
➢ No limitation due to trace buffers

Allows measurements on release-code

```
70    10062C: e51b3010 ldr r3, [fp, #-16]
70    100630: e3530001 cmp r3, #1
70    100634: 9a000010 bls 10067c <_Z13collatz_depthj+0x68>
61    100678: eaffffeb b 10062c <_Z13collatz_depthj+0x18>
61    100638: e51b3010 ldr r3, [fp, #-16]
61    10063C: e2033001 and r3, r3, #1
61    100640: e3530000 cmp r3, #0
61    100644: 0a000005 beq 100660 <_Z13collatz_depthj+0x4c>
16    100648: e51b2010 ldr r2, [fp, #-16]
16    10064C: e1a03002 mov r3, r2
16    100650: e1a03083 lsl r3, r3, #1
16    100654: e0833002 add r3, r3, r2
16    100658: e2833001 add r3, r3, #1
16    10065C: ea000001 b 100668 <_Z13collatz_depthj+0x54>
45    100660: e51b3010 ldr r3, [fp, #-16]
45    100664: e1a030a3 lsr r3, r3, #1
61    100668: e50b3010 str r3, [fp, #-16]
```

Dynamic Analysis

Non-intrusive Continuous Code Coverage

Continuous and non-intrusive

- Statement Coverage
- Branch Coverage (EX/NEX)
- Performance measurement (count executed instructions)

➢ Measured on object code level
➢ Measured on release code
➢ No instrumentation
➢ No limitation due to trace buffers

Allows measurements on release-code

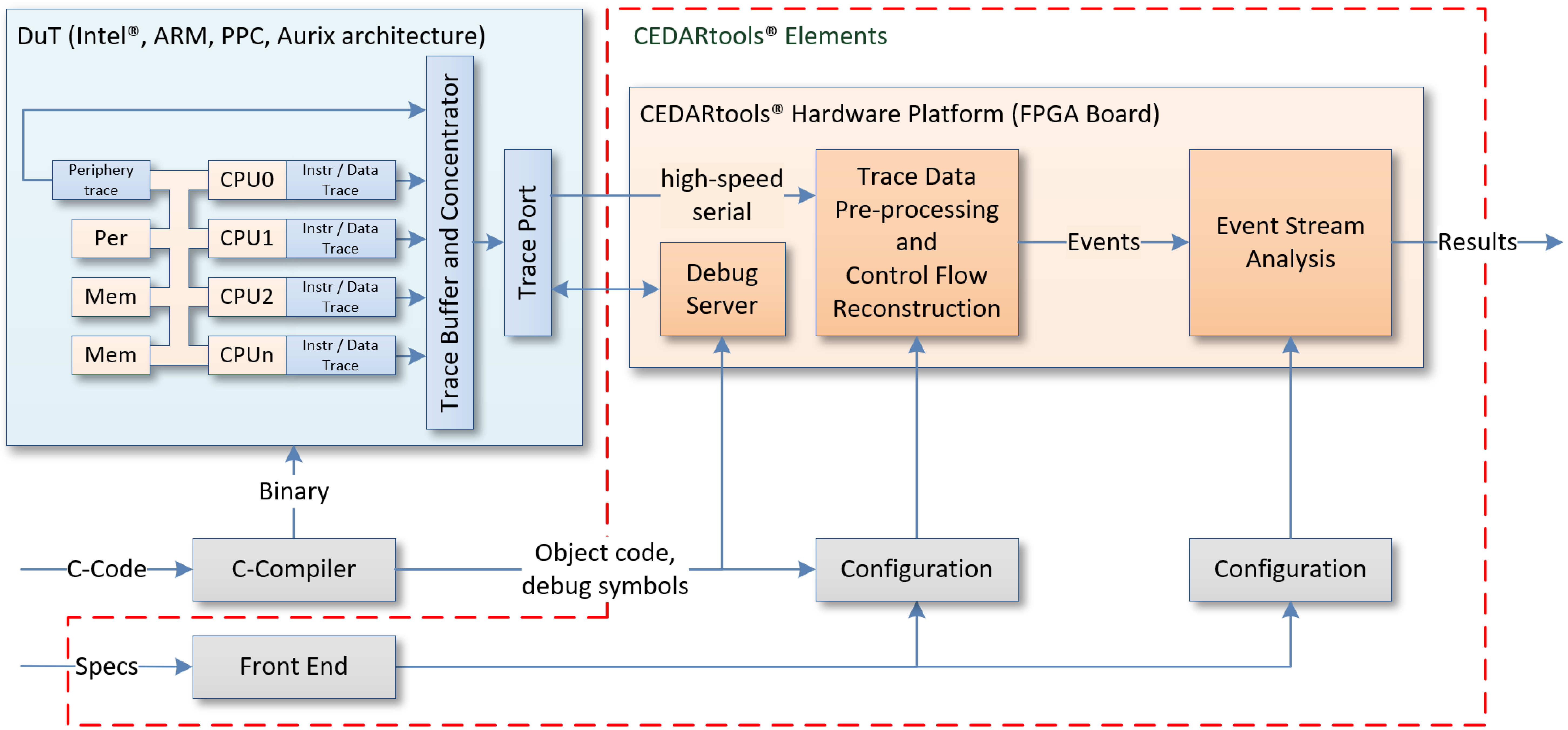
		// Compute n-th Fibonacci number using recursion.
		// - n < 2 does not trigger the else branch.
	275	unsigned fib(unsigned const n) {
[+,+]	275	return (n < 2)? n : fib(n-2) + fib(n-1);
	275	}
		// Unfold Collatz sequence and return its length.
		// - n <= 1 will not execute the while loop at all.
		// - n = 2^k will never trigger the 3*n+1 path.
	9	unsigned collatz_depth(unsigned n) {
	9	unsigned depth = 0;
[+,+]	70	while(n > 1) {
	70	*10062C: e51b3010 ldr r3, [fp, #-16]*
	70	*100630: e3530001 cmp r3, #1*
[+,+]	70	*100634: 9a000010 bls 10067c <_Z13collatz_depthj+0x68>*
	61	*100678: eaffffeb b 10062c <_Z13collatz_depthj+0x18>*
[+,+]	61	n = (n&1)? 3*n+1 : n/2;
	61	*100638: e51b3010 ldr r3, [fp, #-16]*
	61	*10063C: e2033001 and r3, r3, #1*
	61	*100640: e3530000 cmp r3, #0*
[+,+]	61	*100644: 0a000005 beq 100660 <_Z13collatz_depthj+0x4c>*
	16	*100648: e51b2010 ldr r2, [fp, #-16]*
	16	*10064C: e1a03002 mov r3, r2*
	16	*100650: e1a03083 lsl r3, r3, #1*
	16	*100654: e0833002 add r3, r3, r2*
	16	*100658: e2833001 add r3, r3, #1*
	16	*10065C: ea000001 b 100668 <_Z13collatz_depthj+0x54>*
	45	*100660: e51b3010 ldr r3, [fp, #-16]*
	45	*100664: e1a030a3 lsr r3, r3, #1*
	61	*100668: e50b3010 str r3, [fp, #-16]*
	61	depth++;
		}
	9	return depth;
	9	}

ACCEMIC TECHNOLOGIES

Conclusions

- Novel approach for test and debugging based on hardware trace presented
- New potential due to
  - Non-intrusiveness
  - Higher chance to catch sporadic issues using long-running tests
  - Code coverage on integration and system tests
- With goal of increased product quality, reliability and decrease fatal post-release defects

Thanks for your Attention



Contact:
Accemic Technologies GmbH
Franz-Huber-Str. 39
83088 Kiefersfelden

www.accemic.com

Contact partner:
Dr. Alexander Weiss
aweiss@accemic.com
+49 8033 6039795

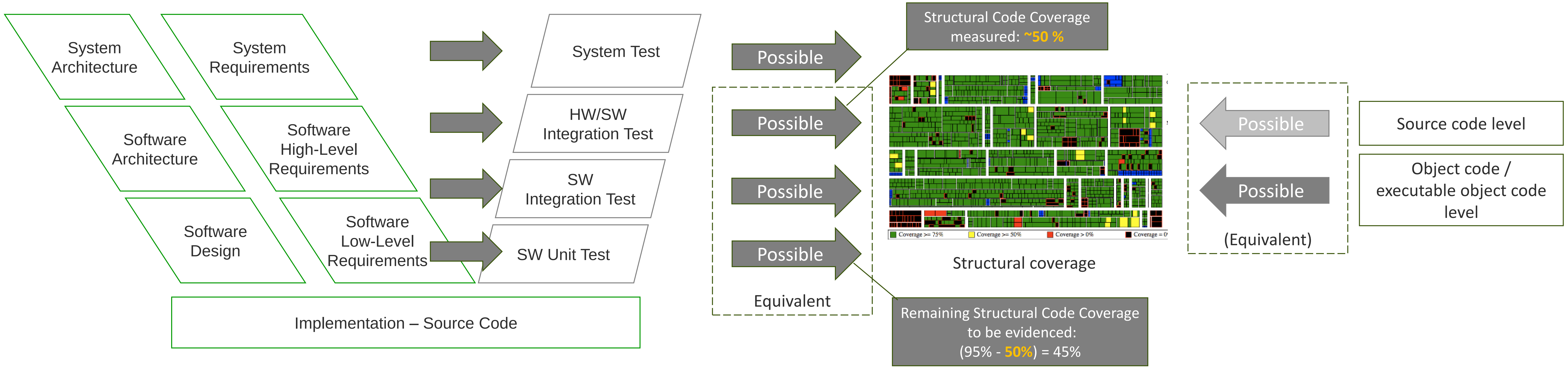
Presenter:
Albert Schulz
aschulz@accemic.com

This work was funded in part by the EU H2020 Project 732016 COEMS and the BMBF Project ARAMiS II (ID 01 IS 16025)

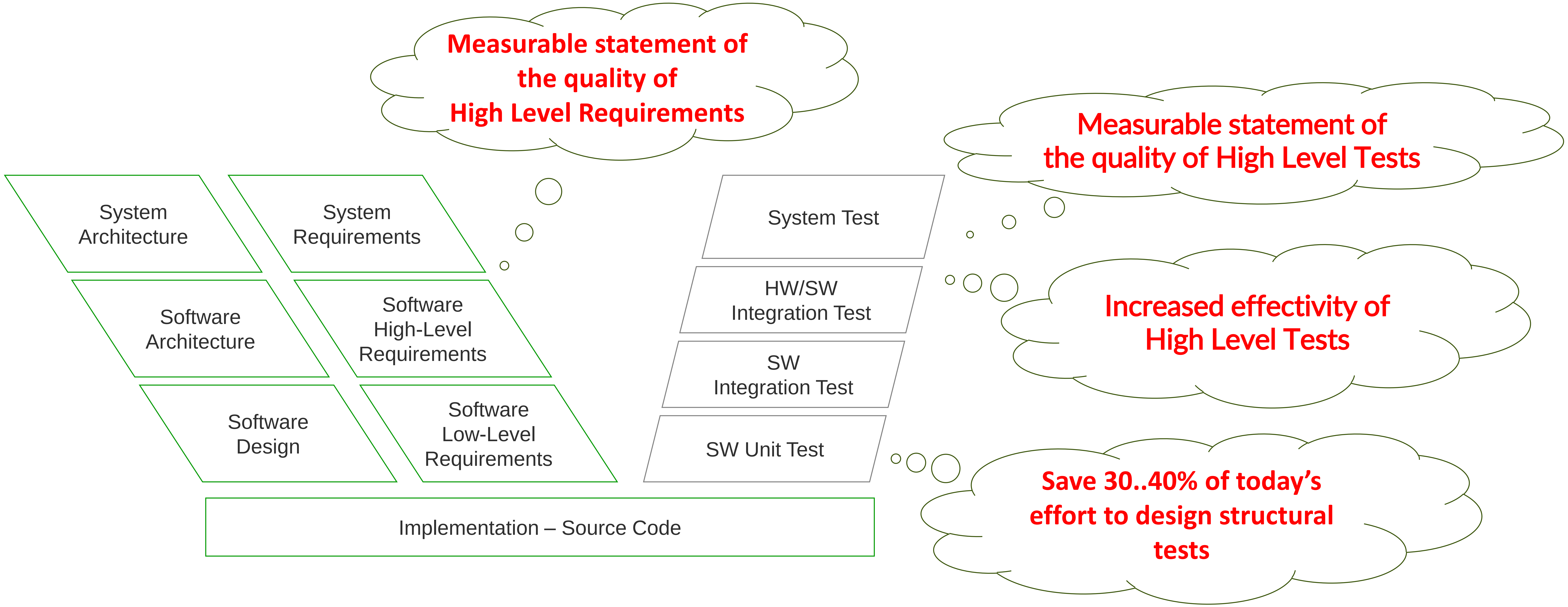
Backup Slides

Dynamic Analysis
@ Object Code Level

- Non-intrusive monitoring and unlimited monitoring period (up to hours, days).
- Structural code coverage can be measured at all test levels.
- Measurable statement of the quality of High-Level Requirements.
- Measurable statement of the quality of High-Level Tests.



Dynamic Analysis
@ Object Code Level



Hardware-based monitoring infrastructure is integrated in most processors - *and already paid by you …*

Mind trace interface access opportunities:

- **in your hardware system requirement specifications and**
- **in your buying decisions!**

Dynamic Analysis

Object Code vs. Source Code Level (see also CAST 17)

PROs

- It can demonstrate full code coverage at the object code level.
- It can support more “valid” coverage.
- It is closer to the final airborne software .
- It can be implemented with source code programming language independence.
- It can reduce time-consuming manual analysis.
- No instrumentation is required.
- It can also be used for the objective measurement of the quality of integration and system tests.
- It can reduce the test effort by substituting low-level tests.
- Incomplete requirements and tests are found at the system level

Listing 1: Illustrative example, source code in C

```c
char* pass_fail(char grade) {
  static char msg[2][5] = {"pass", "fail"};
  int pass;
  if (grade=='d'||grade=='f') {
      pass = 0;
  } else if (grade=='a' || grade=='b' ||
      grade=='c') {
      pass = 1;
  } else { pass = -1; }
  return pass ? msg[pass] : msg[0];
}
```

Listing 2: x86 code compiled with -O0

```
/* if (grade == 'd' || grade == 'f') */
8048439: cmpb  $0x64,-0x14(%ebp)
804843d: je    8048445 // jump if grade=='d'
804843f: cmpb  $0x66,-0x14(%ebp)
8048443: jne   804844e // jump if grade!='f'
8048445: movl  $0x0,-0x4(%ebp) // pass:=0
804844c: jmp   8048470 // jump to return
/* else if (grade=='a'||...||grade=='c') */
804844e: cmpb  $0x61,-0x14(%ebp)
8048452: je    8048460 //jump if grade=='a'
8048454: cmpb  $0x62,-0x14(%ebp)
8048458: je    8048460 //jump if grade=='b'
804845a: cmpb  $0x63,-0x14(%ebp)
804845e: jne   8048469 //jump if grade!='c'
8048460: movl  $0x1,-0x4(%ebp) // pass:=1
8048467: jmp   8048470
/* else        (pass:=-1) */
8048469: movl  $0xffffffff,-0x4(%ebp)
...
```

T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. P. Heimdahl, ‘Toward Rigorous Object-Code Coverage Criteria’, in 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), 2018, vol. 00, pp. 328–338.

Dynamic Analysis

Object Code vs. Source Code Level (see also CAST 17)

CONs

- Source code to object code traceability can be difficult (depending on compiler support).
- Optimizing compiler can use difficult-to-monitor flags to process multi-conditions. (we are working on solutions…)
- Typical tools usually use the source code level.

Listing 1: Illustrative example, source code in C

```
char* pass_fail(char grade) {
  static char msg[2][5] = {"pass", "fail"};
  int pass;
  if (grade=='d'||grade=='f') {
      pass = 0;
  } else if (grade=='a' || grade=='b' ||
      grade=='c') {
      pass = 1;
  } else { pass = -1; }
  return pass ? msg[pass] : msg[0];
}
```

Listing 3: *x86* code compiled with `-Os`

```
8048455: push  %ebp
8048456: mov   $0x804a01c,%eax// %eax:=msg[0]
804845b: mov   %esp,%ebp
804845d: mov   0x8(%ebp),%edx // %edx:=grade
/* if (grade == 'd' || grade == 'f') */
8048460: mov   %dl,%cl         // %cl:=grade
// ASCII('d')=0x64, ASCII('f')=0x66,
// 'f'^0xfffd='d', 'd'^0xfffd='d'
8048462: and   $0xfffffffd,%ecx
8048465: cmp   $0x64,%cl // 'd', grade
8048468: je    804847e    // %cl=='d'->return
/* else if (grade=='a'||...||grade=='c') */
/* else */
804846a: sub   $0x61,%edx // %edx=grade-'a'
804846d: cmp   $0x3,%dl    // CF=%edx<3?1:0
8048470: sbb   %eax,%eax   // %eax:=CF?-1:0
8048472: and   $0x2,%eax   // %eax:=CF?2:0
8048475: dec   %eax        // %eax:=CF?1:-1
/* return pass ? msg[pass] : msg[0]; */
// %eax:=5*%eax
8048476: imul  $0x5,%eax,%eax
// %eax:=msg+%eax
8048479: add   $0x804a01c,%eax
804847e: ...
```

T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. P. Heimdahl, 'Toward Rigorous Object-Code Coverage Criteria', in 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), 2018, vol. 00, pp. 328–338.

ACCEMIC TECHNOLOGIES