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
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 ..."

*NXP:*
Tomorrow’s Vehicle 6x more lines of code

*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?
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

© 2019 Accemic Technologies GmbH. All Rights Reserved.
Hardware Trace
State-of-the-art: Offline Analysis (e.g. Lauterbach TRACE32)

Trace-Buffer limits observation time

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

© 2019 Accemic Technologies GmbH. All Rights Reserved.
Hardware Trace Processing in Realtime

Processor

Multi Gbps Synchronization

CEDARtools® Technology

Live Synchronized Digital Twin

Live Rule Processing
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_strength_for_angle(break_angle);
    int motor_control;
    motor_control = calculate_motor_control_value(strength);
    if (motor_control == 1) activate_breaks();
    else if (motor_control == 0) release_breaks();
}

float read_break_sensor()
{
    float sensor_value = rand();
    return sensor_value;
}

int calculate_break_strength_for_angle(float angle)
{
    // Sleep randomly between 1ms and 1ms
    return rand() / 0.09;
}

int calculate_motor_control_value(int strength)
{
    // Sleep randomly between 1ms and 1ms
    return rand() / 0.09;
}
```

- `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
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

Constraints Specification (TeSSLa)
Dynamic Analysis
Non-intrusive Continuous Timing Verification

DuT (Intel® Atom™, ARM® Cortex®, QoriQ® PPC, Aurix™)

CEDARtools® Elements

CEDARtools® Hardware Platform (FPGA Board)

Observation Configuration

TeSSLa Compiler

Monitor Specification (in TeSSLa language)

Monitor Configuration

Event Stream Analysis

Events

Trace Data Pre-processing and Control Flow Reconstruction

Transmission

Trace Buffer and Concentrator

CPU0

CPU1

CPU2

CPU

Mem

Per

CPU

Mem

CPU0

CPU1

CPU2

CPU

Mem

CPU0

CPU1

CPU2

CPU

Mem

CPU0

CPU1

CPU2

CPU

Mem

Binary

Object code, debug symbols

C-Compiler

C-Code

TeSSLa Compiler

Monitor Configuration

Observation Specification

Analysers

Specs

Front End

Object code, debug symbols

C-Compiler

C-Code

TeSSLa Compiler

Monitor Configuration

Observation Specification

Analysers

Specs

Front End

Multiple high-level specifications can be monitored in parallel

© 2019 Accemic Technologies GmbH. All Rights Reserved.
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
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

```c
unsigned fib(unsigned const n)
{
    return (n < 2) ? n : fib(n-2) + fib(n-1);
}

unsigned collatz_depth(unsigned n)
{
    while(n > 1) {
        n = (n&1)? 3*n+1 : n/2;
    }
}
```
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

Measurable statement of the quality of High Level Requirements

Measurable statement of the quality of High Level Tests

Increased effectivity of High Level Tests

Save 30..40% of today’s effort to design structural tests

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
1    #include "stdio.h"
2    #include "string.h"
3    #include "assert.h"
4
5    char* pass_fail(char grade) {
6        static char msg[2][8] = {"pass", "fail");
7        int pass = 0;
8        if (grade == 'd' || grade == 'f') {
9            pass = 1;
10       } else if (grade == 'a' || grade == 'b' || grade == 'c') {
11            pass = 1;
12       } else { pass = -1; }
13       return pass ? msg[pass] : msg[0];
14    }
15

Listing 2: x86 code compiled with -O0
1    /* if (grade == 'd' || grade == 'f' */
2    8048439: cmpb $0x64,-0x14(%ebp)
3    804843d: je 804844b // jump if grade==d'
4    8048443: jmp 80484ae // jump if grade==f'
5    8048445: movl $0x0,-0x6(%ebp) // pass=0
6    804844c: jmp 8048470 // jump to return
7    /* else if (grade==a'||||grade==c') */
8    804844e: cmpb $0x61,-0x14(%ebp)
9    8048452: je 8048460 //jump if grade==a'
10   8048456: cmpb $0x62,-0x14(%ebp)
11   804845a: je 8048469 //jump if grade==b'
12   804845e: jmp 804846e //jump if grade==c'
13   8048460: movl $0x1,-0x6(%ebp) // pass=1
14   8048467: jmp 8048470
15   (pass=-1 */
16   8048469: movl $0xffffffff,-0x4(%ebp)
17   ...

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

```c
char* pass_fail(char grade) {
  static char msg[2][8] = {"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 −O0

```assembly
8048455: push %ebp
8048456: mov $0x804a01c,%eax // %eax:=msg[0]
804845b: mov %esp,%ebp
804845d: mov $08(%ebp),%edx // %edx:=grade
5 // if (grade == ’d’ || grade == ’f’) */
6 8048460: mov %edx,%cl // %cl:=grade
7 // ASCII(’d’)==0x64, ASCII(’f’)==0x66,
8 // ’f’==0xfffd==’d’, ’d’==0xfffd==’d’
9 8048462: and $0xffffffff,%eax
10 8048465: cmp $0x64,%cl // ’d’, grade
11 8048468: je 804847e; // %cl==’d’>return
12 // else if (grade==’a’ || grade==’c’) */
13 // */ else */
14 804846a: sub $0x61,%edx // %edx:=grade==’a’
15 804846d: cmp $0x3,%edx // CF==edx<371:0
16 8048470: shr %eax,%eax // %eax:=CF?1:0
17 8048472: and $0x2,%eax // %eax:=CF?2:0
18 8048474: dec %eax // %eax:=CF?1:-1
19 // return pass % msg[pass] := msg[0]; /*
20 // */
21 804847e: imul $0x25,%eax,%eax
22 // %eax:=msg[1]eax
23 804847f: add $0x804a01c,%eax
24 8048480: ...
```