Translated article *"Tracing von eingebetteten Multicore-Systemen " HANSER automotive 3 / 2020*

DEVELOPMENT TOOLS

Tracing Embedded Multi-Core Systems

While multi-core processors offer more processing power than single-core architectures, they are more likely to produce hard-to-detect and concurrency-related errors. This article presents a new technology that enables the measurement of the timing behavior of programs on multi-core architectures, the measurement of the coverage achieved by test running on the target system, and the analysis of complex errors.

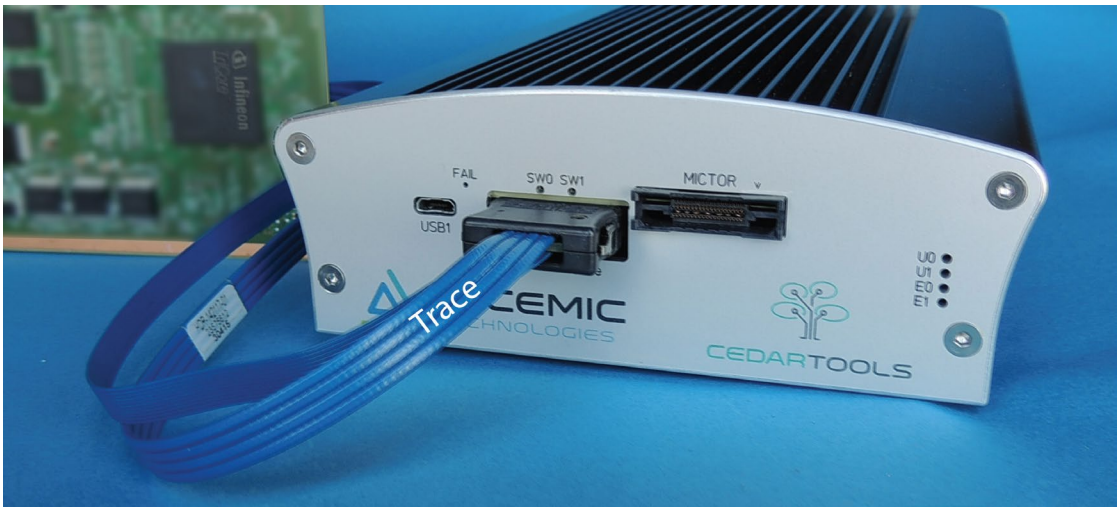

Figure 1: CEDARtools®

For many years, control units in automobiles have been taking over more and more compute-intensive tasks - and no end to this trend is in sight. With the increasing complexity, the probability of errors goes up while the efficiency of their elimination goes down [1]. The constantly increasing computing load makes the use of multi-core processors inevitable for many applications. However, the transition from sequential to parallel processing can introduce errors that are very difficult to reproduce, particularly if the code was never intended for parallel execution.

Without doubt, liability reasons demand special diligence in software development and testing from the automotive industry. However, testing and debugging environments only make sense if they help to assign root causes to observed error symptoms. Observability is, therefore, key. In contrast to traditional single-core systems, modern multi-core architectures present us with special challenges. The more integrated such a system is, the more difficult it is to understand internal processes. This is exactly where the technology presented in this article provides support.

Proven and Novel Tools

Software instrumentation is widely used and established for monitoring internal processes. Automatically added code logs the execution of the program, for exam-

ple, to measure the execution times of code blocks or to determine the coverage of tests. However, instrumentation requires memory space and impacts the temporal behavior of program execution.

Fortunately, there is an alternative: Almost all modern processors feature an embedded trace unit (e.g. Intel Processor Trace [2]), which communicates via a designated (e.g. AGBT) or a standard interface (e.g. USB DCI, PCIe). This unit provides information about the executed program flow without affecting it. Depending on the architecture and trace configuration, the temporal execution behavior and memory data accesses can also be reconstructed. In addition, many trace units enable lightweight hardware-supported instrumentation, which is thus acceptable in the release code, (e.g. PTWrite, MIPI STP [4]) as well as the tracing of peripheral units (memory controllers, communication units).

The embedded trace solutions commonly used today store the broadband trace data (several Gbit/s) in temporary buffer memory. At the end of a test run, the program flow is reconstructed on a PC, and the structural coverage is calculated. The limits of this procedure are the observation time bounded by the size of the buffer memory and the additional computing time required for the offline reconstruction of the program flow.

```
17                                  // Unfold Collatz sequence and return its length.
18                                  // - n <= 1 will not execute the while loop at all.
19                                  // - n = 2^k will never trigger the 3*n+1 path.
20                        1970      unsigned collatz_depth(unsigned n) {
21                        1970      unsigned depth = 0;
22          [+,+]         13987     while(n > 1) {
                          ↪ 1970      40089B: eb 24 jmp 4008c1
                          13987       4008C1: 83 7d ec 01 cmpl $0x1,-0x14(%rbp)
                [+,+]     1970 ↪
                          12017       4008C5: 77 d6 ja 40089d
23              [+,+]     12017     n = (n&1)? 3*n+1 : n/2;
                          12017       40089D: 8b 45 ec mov -0x14(%rbp),%eax
                          12017       4008A0: 83 e0 01 and $0x1,%eax
                          12017       4008A3: 85 c0 test %eax,%eax
                [+,+]     3152 ↪ 8865 4008A5: 74 0e je 4008b5
                          3152        4008A7: 8b 55 ec mov -0x14(%rbp),%edx
                          3152        4008AA: 89 d0 mov %edx,%eax
                          3152        4008AC: 01 c0 add %eax,%eax
                          3152        4008AE: 01 d0 add %edx,%eax
                          3152        4008B0: 83 c0 01 add $0x1,%eax
                          ↪ 3152      4008B3: eb 05 jmp 4008ba
                          8865        4008B5: 8b 45 ec mov -0x14(%rbp),%eax
                          8865        4008B8: d1 e8 shr %eax
                          12017       4008BA: 89 45 ec mov %eax,-0x14(%rbp)
24                        12017     depth++;
25                                  }
```

Figure 2: Object code coverage and its mapping to source code level

INFO

Use Case: Dynamic Software Architecture

The relevance of software in vehicles has been increasing for several years and it is likely that this trend will continue. Especially in highly automated driving, many innovations are software-based [9] [10].

In conjunction with equally increasing time-to-market requirements, OEMs face a variety of challenges at validating their systems. In addition to the increasing complexity of software architectures, the integration of legacy components as well as a combined scheduling on multi-core platforms are particularly aggravating.

The LET concept [1] (Logical Execution Time) [11] provides a framework to meet these challenges. However, it relies on one important prerequisite: It must be ensured that each task is safely processed within an assigned time window. Static analyses are currently not providing sufficient precision. Thus, measurement methods are employed additionally to check whether deadlines are met. Measurement approaches that do not require software instrumentation are exceedingly interesting in this context as they measure the actual software status.

[1] To simplify the design process, the LET concept abstracts from the physical execution time on a given platform by only looking at the times of read and write accesses. There is no consideration of the actual execution time as long as deadlines are met.

The live analysis is a further enhancement beyond the established offline analysis of recorded trace data. It requires two technical challenges to be mastered. On the one hand, the highly compressed trace data stream must be pre-processed and the control flow of the CPU(s) must be reconstructed. This computation must work for fast CPUs (>1 GHz) and for different operating systems. On the other hand, the generated event stream must be processed as, for example, by recording jump information to facilitate the calculation of the structural test coverage or by dynamically monitoring a large number of properties using data flow processors that can be configured in a high-level language.

Use Case A: Code Coverage

The standards for the development of safety-critical systems (e.g. [4]) define requirements for the test process, the test techniques to be applied and the verification of the completeness of these tests by measuring the structural coverage. For the latter, it must be shown, depending on the criticality of the application, that all instructions (Statement Coverage), all jumps (Branch Coverage) or all relevant combinations of conditions for branches (MC/DC) have been invoked during the tests. In general, the standards leave it largely open, on which test level (system test, integration test, module test) the structural coverage is validated. Ideally, the requirements specification is included in the test so that the structural coverage can also make a statement about the quality of the requirements.

As discussed in detail in the CAST-17 position paper [5], the measurement of the structural coverage on object code level provides different information than a measurement on source code level. Leveraging the debug information generated by the compiler, the measured object code coverage can be mapped to the corresponding source code coverage, as illustrated in Figure 2.

This analysis of the debug information can now be combined with the live analysis: Without any instrumentation

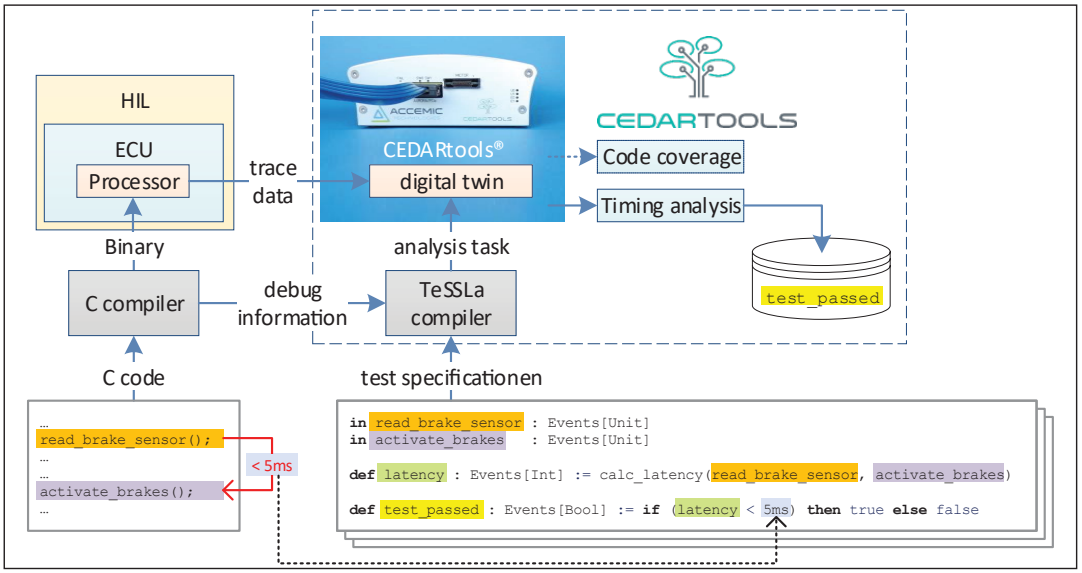

Figure 3: Dynamic event analysis with CEDARtools®

(and thus without changing the dynamic behavior of the target system), the test coverage for the target system is determined. An example of such an analysis system is the CEDARtools® system shown in Figure 1, which performs live analysis using an FPGA.

Measuring structural test coverage during integration and system testing is able to identify gaps in these tests, and replace structural unit tests.

Use Case B: Dynamic Analysis

Static analysis is an essential element in the development process. Due to missing or oversimplified models, this method is increasingly reaching its limits rendering the complementary dynamic analysis of embedded systems more and more important. Also, this approach can be implemented using the hardware outlined above.

For this purpose, selected instruction addresses are marked during the continuous reconstruction of the control flow. At the execution of these instructions, specific events are automatically injected into the emitted event stream, which can then be analyzed online for specific properties. The used event processing units can be configured in a high-level language (www.tessla.io [6]). A large number of temporal properties (e.g. defined by AUTOSAR TIMEX [7] or Amalthea) can be monitored in parallel. The FPGA only requires a parameterization of the event processing units, an individual synthesis of logic structures is not necessary. Thus, a change of the high-level language property description can be applied to a trace data stream within seconds. An example is shown in Figure 3.

Dynamic analysis is also a powerful tool for debugging, especially for complex non-deterministic error patterns. Depending on the probability of occurrence and the area of application, the search for one of these defects can quickly lead to six-digit costs [8].

INFO

With **CEDARtools®**, Accemic Technologies have developed a tool for analyzing complex embedded systems. It solves the problem of the limited size of trace buffers. Major automotive and aerospace OEMs and Tier1s are already evaluating and using the system.

Instead of storing the trace data and later analyzing it offline, the multi-Gbit data stream is analyzed on-the-fly using sophisticated hardware. The following challenges have to be met:

- Live reconstruction of the control and data flow of one or more CPUs using trace data - a rather demanding task, since the trace data is output by the processor in a highly compressed form
- highly configurable live analysis of the generated event stream (e.g. instruction hits, data accesses, task changes, up to ~100M events/s), simultaneous investigation of a multitude of complex properties (minimum and maximum runtimes in effect chains, statistics, sequences, checking of value ranges etc.)
- continuous live measurement of structural test coverage.

The new paradigm is to no longer collect enormous amounts of raw trace data over long periods of time for their later evaluation in a time-consuming process but to analyze the trace data on-the-fly using predefined high-level language properties. This can be done over minutes, hours and days. However, there is no need to dispense the raw data: Using complex triggers, it is possible to define exactly which raw data and events are relevant for the analysis and stored in an existing trace buffer (4 GByte).

Supported processors*:

- Arm® Cortex®-A, -R, -M
- Infineon Aurix™ TC2xx, TC3xx
- Power Architecture® (including NXP QorIQ® P-/T-Series)
- Intel® Atom® (among others E39x0)

Supported Trace Interfaces*:

- High-speed serial interfaces (Aurora): NEXUS, HSSTP, AGBT
- Standard interfaces: PCIe, USB
- Parallel interfaces (Mictor, NEXUS)

* partly still under development

Design Considerations

Already in the specifications, suitable precautions should be defined to achieve comprehensive observability of the Electronical Control Unit (ECU) both during the tests and after the release. The trace interface should be available in specific ECU versions during the test phase as well as in production vehicles. In addition, for tests on pre-production and production vehicles, the interference-free transmission of trace data from the ECU to the analyzer (which is usually located inside the vehicle) must be ensured.

Summary

The continuously increasing number of post-release defects with increasing complexity requires the application of new test methods. These include the measurement of the structural test coverage and the automated execution of runtime analyses in the fully integrated system. Thanks to a new technology, this analysis is now possible without software instrumentation and thus without influencing the runtime behavior.

In addition, it allows the cause of complex error patterns to be efficiently analyzed even after a system has been released. To be able to use this technological capability, wide-bandwidth access to the trace data output by a processor is required. ■ (oe)

www.accemic.com

This work was funded with funds from the EU H2020 project 732016 "COEMS" and from the BMBF project "ARAMiS 2" (FKZ 01IS16025).
An in-depth investigation of the measurement of structural test coverage in integrated systems is carried out in the "CoCoSI" research project (BMBF KMU Innovativ, FKZ 01IS19044) with the project partners Accemic Technologies, Fraunhofer IESE, Heicon and Intel.

List of references

[1] C. Jones and O. Bonsignour, The Economics of Software Quality. Addison-Wesley, 2011.

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, 2016.

[3] 'Specification for System Trace Protocol (STP)'. MIPI Alliance, Inc., 2015.

[4] 'ISO 26262:2018. Road vehicles – Functional safety'. International Organization for Standardization Std., 2018.

[5] 'Position Paper CAST-17: Structural Coverage of Object Code'. FAA Certification Authorities Software Team (CAST), Jun-2003.

[6] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, 'TeSSLa: Temporal Stream-Based Specification Language', in Formal Methods: Foundations and Applications, Cham, 2018, pp. 144–162.

[7] 'AUTOSAR-TIMEX: Specification of Timing Extensions'. [Online]. Available: http://www.autosar.org/.

[8] B. Hanke and F. Schulz, 'Master Thesis: Assessment of multi-core integration infrastructure', University of German Armed Forces Munich, Munich, 2014.

[9] Y. Dajsuren and M. van den Brand, Eds., 'Automotive Software Engineering: Past, Present, and Future', in Automotive Systems and Software Engineering – State of the Art and Future Trends, Springer, 2019, pp. 3–8.

[10] P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha, 'Autonomous Vehicles: State of the Art, Future Trends, and Challenges', in Automotive Systems and Software Engineering – State of the Art and Future Trends, Y. Dajsuren and M. van den Brand, Eds. Springer, 2019, pp. 347–367.

[11] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, 'Giotto: a time-triggered language for embedded programming', Proc. IEEE, vol. 91, no. 1, pp. 84–99, 2003.

Max Jonas Friese
Software Architect
Mercedes-Benz AG

Dr. Stephan Grünfelder
Test Trainer
embedded-test.webs.com

Dr. Michael Paulitsch
Dependability Research
Architect, Principal Engineer
Intel Deutschland GmbH

Dr.-Ing. Alexander Weiss
CEO
Accemic Technologies GmbH