Online Analysis of Debug Trace Data for Embedded Systems

Normann Decker[‡], Boris Dreyer[*], Philip Gottschling[*], Christian Hochberger[*], Alexander Lange[†], Martin Leucker[‡], Torben Scheffel[‡], Simon Wegener[§] and Alexander Weiss[†]
[*]Computer Systems Group, Technische Universität Darmstadt, Darmstadt, Germany
[†]Accemic Technologies GmbH, Kiefersfelden, Germany
[‡]ISP, Universität zu Lübeck, Lübeck, Germany
[§]AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany

***Abstract*—Modern multi-core Systems-on-Chip (SoC) provide very high computational power. On the downside, they are hard to debug and it is often very difficult to understand what is going on in these chips because of the limited observability inside the SoC. Chip manufacturers try to compensate this difficulty by providing highly compressed trace data from the individual cores. In the past, the common way to deal with this data was storing it for later offline analysis, which severely limits the time span that can be observed. In this contribution, we present an FPGA-based solution that is able to process the trace data in real-time, enabling continuous observation of the state of a core. Moreover, we discuss applications enabled by this technology.**

I. Introduction

Software development on modern embedded multi-core SoCs is much more difficult than on regular systems. Mainly, this difficulty is caused by the limited observability of internal behaviour of SoCs. Currently, two solutions exist to observe multi-core SoCs: software instrumentation, and embedded trace interfaces.

Software instrumentation is easy to use and well supported by tools. Nevertheless, in realistic scenarios it is impractical. It causes severe changes in the timing behaviour of the system (even reversing order of thread execution) and it requires substantial additional resources. For safety-critical applications, the instrumentation should persist in the final code, as otherwise test results are not reliable [1] and untested effects might remain.

A more sophisticated approach and key element in multi-core observation is the "embedded trace unit" (ETU). A special hardware unit observes SoC internal states, compresses them and outputs the resulting information via a dedicated trace port. Depending on the implementation, an ETU outputs the following information in whole or in part:

- Addresses of instructions executed by the CPU cores
- Task switches and exceptions with their trigger causes
- Occurrences of operations that access registers, memory, performance counters and peripherals
- HW-supported instrumentation (like `debugprintf()`)

Typically, a specialized external device records the trace data stream for later offline analysis on a PC. In contrast to software instrumentation, this approach is more suitable for multi-core SoCs since messages from multiple trace data sources (i.e. multiple cores) can be collected concurrently. Nevertheless, these ETUs come with serious limitations:

- Trace data bandwidth is limited (even at very high rates). Thus, trace information has to be filtered and potentially vital information might be dropped internally. Concurrent observation of multiple hot spots might not be possible. Especially, tracing data accesses requires high bandwidth and thus, is often very limited.
- Trace trigger conditions are limited to the sparse functionality implemented in the ETU.
- Storing trace data severely limits the observation time, since the trace bandwidth can exceed 10 Gbit/s. Typical devices can only collect trace data for several seconds. This problem will be further aggravated in the future.
- There is a mismatch between trace data output and offline processing bandwidth, which is usually several orders of magnitude lower. This results in long trace data processing times, which require a lot of patience by the test/debug engineer and renders the debugging process ineffective and inefficient.

In this paper, we give an overview on the work that started in the CONIRAS project and continued in ARAMiS II and COEMS. We describe a novel observation solution which overcomes the limitations of the state-of-the-art. It provides a new quality in observation, and therefore, in test and debugging efficiency. The presented solution does not store the received trace data for later offline processing but processes it directly online in an FPGA. This approach enables an arbitrary observation time and creates the best observability possible from ETU implementations.

The remainder of this paper is structured as follows: Section II describes our platform for continuous trace analysis. Section III introduces the specification language TeSSLa and describes an efficient way to synthesize suitable monitors to check specifications at runtime. In Section IV, we present several applications that are made possible by our platform. Section V discusses related work. Finally, we give a conclusion and an outlook.

II. Continuous Online Analysis Platform

Our method works on the object code level and is split into three phases: an offline pre-processing phase, the continuous

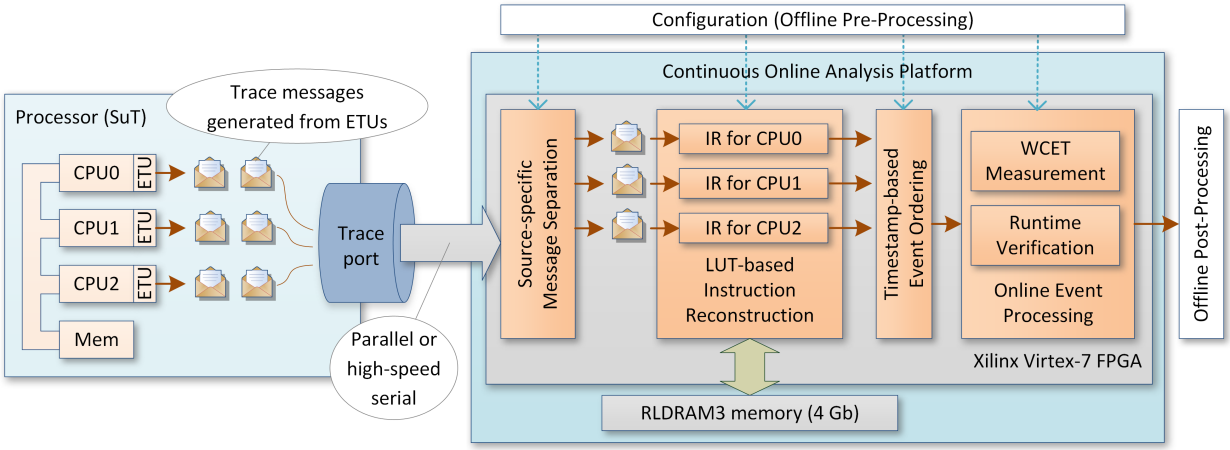

Fig. 1. Overview of our Continuous Online Analysis Platform (blue box). The platform is attached to the system under test via the trace port. Inside the FPGA-based platform, the raw trace is first converted into a stream of watchpoint events (Instruction Reconstruction) and later processed in the individual analysis applications (Online Event Processing).

online analysis phase and an offline post-processing phase. Figure 1 shows the different elements of our approach in relation to each other and the components of the multi-core system under test (SuT).

The ETUs of the SuT (in our case the Cortex-A9 program flow trace units as a part of the ARM CoreSight architecture [2], [3]) produce various trace messages when a program is executed on the SuT. For example, program flow messages are emitted for each non-linear control flow, like interrupts and hardware exceptions, but also for normal branches and calls. Comparable ETUs are available for all major CPU architectures, for instance Intel Program Trace [4] for the latest generations of Intel processors or the Nexus [5] compliant embedded trace units for NXP's QorIQ processors [6]. Besides the program flow trace, there are also units available providing traces for data accesses and the activity of peripheral units.

The ETUs allow to filter the CPU-individual trace messages, for example, to cover only a specific address range. This observation focus can be changed at runtime, if necessary. The resulting stream of trace messages is then emitted via a trace port. Although the trace port bandwidth can reach several Gbit/s, the SuT can generate more trace data than the trace port can output at a given time. Therefore, the ETUs include FIFOs to buffer trace messages. If a large volume of trace messages is generated (e.g. narrow loops with high branch frequency), these processor-internal trace buffers can overflow. A special trace message indicating the overflow is emitted when this happens.

Our FPGA-based continuous online analysis platform is attached to the SuT via the trace port. Inside, instruction reconstruction (IR) units convert the raw stream of trace data into a stream of watchpoint events, which can be further processed on the FPGA by various analyses (see Section IV). A watchpoint event indicates that a specific instruction has been executed or that a control-flow edge has been traversed.

A. *Configuration (Offline Pre-Processing)*

In the pre-processing phase, the fully linked binary of the program being executed on the SuT is disassembled and the control-flow graph (CFG) is reconstructed using the executable reader [7] of aiT [8]. The user can specify the targets of computed calls or branches if they cannot be resolved by means of static analysis. From this CFG, the waypoint graph (WPG) is computed. To do so, a pattern matcher checks for each instruction whether it is a waypoint instruction. These instructions are specified in the ARM CoreSight Program Flow Trace manual [3]. Amongst others, all instructions that possibly modify the program counter are waypoint instructions. Afterwards, the edges of the WPG are computed. For each waypoint instruction found, the algorithm follows the edges in the CFG to find reachable waypoints. This gives the direction of an edge in the WPG and its target. A unique ID is assigned to each such edge and stored in the lookup tables (LUTs) of the IR units. These LUTs are used to map trace messages to the WPG.

B. *Instruction Reconstruction*

For the Cortex-A9 program flow trace, the most relevant messages are the cycle-accurate atom packets (Atom), the branch address packets (Branch) and the instruction synchronization packets (I-Sync). Atom indicates whether a branch instruction passed or failed its condition code check and outputs an explicit cycle count indicating the number of cycles since the last cycle count output. Branch indicates a change in the control flow when an exception or a processor security state change occurs, or when the CPU executes an indirect branch instruction that passes its condition code check. The current instruction address and a cycle count are periodically emitted via I-Sync.

Within our continuous online analysis platform, the trace messages received from the system under test are distributed into CPU-specific message streams. Each message stream is

processed in real-time by one IR unit [9]. After receiving the first I-Sync message (which gives us the current instruction address), the IR unit reconstructs the control flow of the CPU by processing the consecutive Atom and Branch messages. Branch messages transfer the branch offset directly and are relatively easy to process. Much more difficult is the processing of the Atom messages, which do not include the offset of a direct branch, but only a flag if a branch is executed or not. To evaluate the branch offset, a preprocessed LUT containing the branch offset for each instruction address is used.

Additionally, this LUT also includes predefined event IDs, indicating whether the branch is of interest for further processing. In case of a non-zero ID, a watchpoint event is emitted by the IR unit, identified by the ID that has been assigned to it during preprocessing. Moreover, the watchpoint event also includes the amount of CPU cycles that passed since the previous watchpoint. These watchpoint events correspond to the traversal of edges in the WPG. A special watchpoint event is emitted when the IR units detect an overflow of the ETU-internal trace buffers.

C. Online Event Processing

The watchpoint event stream emitted by the IR units is further processed in two distinct modules implemented on the FPGA. The first module is a reconfigurable runtime verification engine, see Section III. Its resource consumption depends on the number of necessary monitors. The second module calculates execution time statistics for each edge in the WPG. Its resource consumption increases linearly with the amount of edges.

D. Offline Post-Processing

After the program executed on the SuT has finished (or the test engineer has collected enough data), the post-processing phase is started by downloading the collected information (e.g. timing statistics, code coverage statistics, reports from the runtime verification engine). This data is then presented to the user and can be processed further.

E. Hardware Implementation

Supposing that every 5th instruction is a branch and the CPU runs at 2 GHz, we have to handle 400 M branches per second (multiplied by the count of observed CPUs). The required processing performance can only be achieved with high-end FPGAs and high-performance external memory, optimized for high random-access bandwidth. The system architecture combines a high degree of parallelization and speculative operations.

Our current hardware implementation consists of a VPX format [10] FPGA board, equipped with a Virtex-7 FPGA. Connected to the FPGA are eight RLDRAM3 chips (each 16M x 36). The trace signal input is provided by an FMC card [11]. For arbitrary scalability, additional FPGA cards can be linked together via a VPX backplane.

III. Synthesizing Reconfigurable Monitors

We developed the Temporal Stream-based Specification Language (TeSSLa)[1] to specify properties on the stream of watchpoint events. Besides being able to specify state machines or timing properties, TeSSLa also allows for specifying properties about data and data manipulations, like aggregations. TeSSLa does not allow recursive specifications, because recursion is hard to synthesize on hardware. Instead there are a lot of build-in functions for more complex operations like sum or maximum.

After specifying a property, we map the specification to reconfigurable hardware and check the property during runtime, for example in debugging sessions or while the software is used in the field. Examples for possible TeSSLa specifications are shown later in Section IV. More details can also be found in [12].

When we want to implement the given TeSSLa specifications as runtime verification monitors on an FPGA, there are two options. First, direct synthesis, where the monitor descriptions are transformed into a hardware description language (HDL), synthesized and programmed on the FPGA. The drawback here is the high synthesis time of at least minutes and up to hours, depending on the size and the number of monitors. This is not feasible as turnaround time for software verification which is usually an iterative and interactive process. In [13], Reinbacher, Rozier and Schumann show a more elegant way. They synthesize a combination of static hardware and reconfigurable interconnects.

We implemented a similar approach in [14] where our reconfigurable processing system has a turnaround time of less than a second. Multiple control and data flow graphs (CDFGs) that represent different input problems are used to construct the reconfigurable platform. In contrast to [13] our implementation supports the reconfiguration of data flow.

We translate the TeSSLa specifications into CDFGs which mostly follow a similar structure of two stages. The first stage contains arithmetical, logical and relational expressions. Some of the relational expressions directly indicate an error while others generate so-called propositions. These propositions are boolean values that can be evaluated by a finite state machine (FSM), the second stage. For example, the SALT formula given in Listing 1 can be transformed into an FSM.

In the following, we describe the necessary steps to make use of the rapid reconfiguration. At first, the user specifies a set of (exemplary) TeSSLa formulas which are translated into CDFGs. Those are combined into a single super CDFG that forms a superset of all input CDFGs. The major goal in this step is to identify operations that are shared between graphs. As the resulting CDFG should be preferably small, we try to maximize the number of shared resources.

The super CDFG is then translated into a Verilog description of a reconfigurable monitor. Its functionality can be adjusted by employing three different techniques:

[1] For more information on TeSSLa see http://www.isp.uni-luebeck.de/tessla.

- First, while combining the CDFGs multiplexers are introduced that can route different operands to a shared resource. Setting the control bits of those multiplexers allows to construct a desired path through the first stage of the monitor.
- Second, we use registers for constant values instead of hardcoding them, because constants like addresses, watchpoint IDs or boundaries may change.
- The last feature is the FSM which is implemented as a microprogrammed state machine. Its behavior only depends on the microprogram stored in memory and is only limited by the inputs (propositions) and a maximum number of states.

The desired number of monitors are then assembled into a monitor system which is then synthesized. Depending on the size of the target FPGA and the monitor complexity the number of monitors in the system can vary from ten to up to a few hundred. For example, in [14] we implemented 256 (merged) monitors on a Virtex 7 FPGA. These monitors must be configured to realize specific TeSSLa formulas. The configurations are calculated from the CDFGs of TeSSLa formulas and the super CDFG of the reconfigurable monitor. After uploading the configuration, which takes only a few milliseconds, the user can watch the current state of the monitor on the host PC.

IV. Applications

We implemented six applications operating on the stream of watchpoint events. These analyses are presented in the following paragraphs.

A. Measuring Code Coverage

The quality of test suites is often assessed in terms of coverage criteria, for example statement coverage, condition coverage, or branch coverage. Non-intrusive online observation allows for observing a test run without modifying the runtime environment or the program code. Based on the watchpoints, the program flow can be identified and therefore all mentioned coverage criteria can be computed on the FPGA.

B. Hybrid WCET Estimation

Precise estimation of the Worst-Case Execution Time (WCET) of embedded software is a necessary precondition in most safety-critical systems. Hybrid approaches for WCET estimation combine static path analysis on the binary level with measurements on the real hardware which capture the timing behaviour of short code snippets. We presented one such approach in [15]. Its main benefits are:

- The timing of short instruction sequences is measured. This fine-grained approach allows to see where time is spent.
- The use of ETUs makes the measurements non-intrusive. The probe effect is avoided.
- Trace data is aggregated continuously, allowing arbitrarily long periods of observation. This is particularly important for multi-core systems to catch both typical behaviour and rare circumstances.

Listing 1
EXAMPLE FOR A TESSLA SPECIFICATION OF A STRUCTURAL PROPERTY.

```
define line32_exec := codeLine main.c:32
define line47_exec := codeLine main.c:47
define monitor_output := monitor(always
  (line32_exec implies next
  (not(line32_exec) until line47_exec)))
out monitor_output
```

For the description of the workflow of our hybrid WCET estimation approach, we refer to Figure 1. First, in the preprocessing phase, we use the WPG to configure the WCET measurement module. A watchpoint is set for each waypoint edge. During the continuous analysis phase, the WCET module computes for each edge the minimal and the maximal associated execution time, how often an edge has been traversed, and the sum of all observed execution times for a particular edge. The last two data points allow us to compute average execution times. Afterwards, in the post-processing phase, these statistics are downloaded from the FPGA's memory. Subsequently, the WPG together with the timing statistics are used to construct a maximisation problem encoded as an integer linear program (ILP). Solving this ILP gives a path with maximal execution time (and consequently, an estimate of the worst-case execution time).

We also implemented an extension of this approach where we computed timing histograms instead of simple min/max statistics. We refer to [16] for the details.

C. Finding Functional Bugs

Functional bugs can occur in any type of system when functions or code lines are executed in the wrong order. By observing the watchpoint stream emitted by the IR units, we get information about the order in which functions were executed, whether the if-branch or the else-branch of a conditional was taken, or about the order in which code lines were executed.

Assume for example a C application where the code lines 32 and 47 in the file main.c always have to be executed in order. Every time line 32 is executed, line 47 has to be executed afterwards at least once before line 32 is executed again. This specification has been encoded in the TeSSLa specification shown in Listing 1. It uses a SALT [17] formula with LTL_3 semantics [18] to describe a monitor in TeSSLa. It expresses that every time line 32 is executed, line 32 is not allowed to be executed again until line 47 has been executed at least once. The *out* statement marks the *monitor_output* stream as a stream that is added to the information that is downloaded in the offline post-processing phase.

D. Finding Timing Bugs

Besides bugs in the program logic, there might also occur non-functional bugs, for example caused by the violation of timing requirements of the system. Finding such bugs is really

Listing 2
EXAMPLE FOR A TESSLA SPECIFICATION OF A TIMING PROPERTY.

```
define firstCalls := function_call first
define triggerCalls := function_call trigger
define error := triggerCalls implies not inPast(
    firstCalls, 2s)
out error
```

difficult without observing the trace of the execution of the SuT. Since a timestamp is attached to each watchpoint, it is possible to check properties over real-time constraints.

Assume a C application with two functions *first* and *trigger*. It is important that each call to function *trigger* is preceded by a call to function *first* that happened less than two seconds in the past.

To check such a timing constraint in a system, we wrote the TeSSLa specification shown in Listing 2. This specification defines two streams for the function calls and then checks that, if *trigger* is called, *first* has been called at most two seconds ago.

E. Continuous Timing Validation

Often, the execution time depends on the data being processed. If the data (e.g. the characteristic curve of a control software) changes, the timing behaviour changes as well. In case of a deadline miss, this effect is directly visible. This may not be the case if the system has been designed with some slack intended for future enhancements. Consequently, the increased execution time of a task might not lead to any visible effect, but the assumptions about the system's state might be entirely wrong.

Our solution to this problem is to describe the expected timing behaviour in a TeSSLa specification. We use the timing statistics recorded during hybrid WCET estimation to automatically generate such a specification.

Consider for example the CFG shown in Figure 2. It consists of a function f calling function g, which contains some nested loops. Our hybrid WCET estimation computes for function g a minimal execution time of 16 μs and a maximal

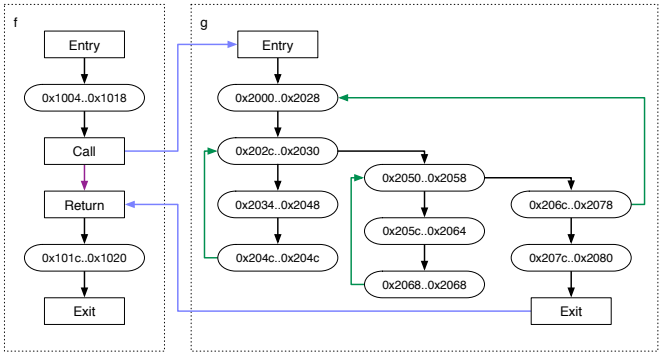

Fig. 2. A simple CFG with two functions f and g. Basic blocks are shown with the address of their first and last instruction.

Listing 3
EXAMPLE FOR A TIMING VALIDATION TESSLA SPECIFICATION.

```
define block_start := onExecuting 0x2000
define block_end := onExecuting 0x101c
define timing_min_violation := on block_end if
    inPast(block_start, 16us)
define timing_max_violation := on block_end if not
    inPast(block_start, 23us)
out timing_min_violation
out timing_max_violation
```

Listing 4
EXAMPLE FOR A TESSLA SPECIFICATION CALCULATING A STATISTIC.

```
define algoStart := codeLine algo.c:10
define recCalls := function_call rec
define max_num_calls := maximum(eventCount(recCalls,
     algoStart))
out max_num_calls
```

execution time of 23 μs. We translate this result into the TeSSLa specification shown in Listing 3, which is compiled to a monitor configuration that validates the timing behaviour in the field. Violations of the timing behaviour assumptions can thus be detected in a fine-grained way.

F. User-Defined Metrics and Complex Triggers

Sometimes the statement that a property is fulfilled (or not) is not enough, but instead one wants to get more detailed information about the system's behaviour. Examples contain performance metrics (i.e. profiling), statistical measures over function or code line executions, or accumulation of some key parameters during program execution. This can help to observe rare circumstances, by logging the system's state every time a given condition occurs. Obtaining such information in a non-intrusive manner is essential for certification of safety-critical systems. With the data delivered from the IR unit, we can calculate such user-defined metrics using TeSSLa.

Assume a C application implements a recursive algorithm. It contains a function which calls itself in a loop. We know the highest number of calls to the recursive function for a correct implementation of the algorithm. For each run of the complete algorithm, we want to get the maximum number of calls to that function. To collect this information, we wrote the TeSSLa specification in Listing 4.

In this specification, an *eventCount* function is used to count the number of events and reset this counter to zero when an event happens on *algoStart* and start counting again when the recursive function *rec* is called. Then we take the maximum of these counted calls to always get the highest number of calls to the recursive function that occurred in one call to the algorithm until this point in time. The output stream *max_num_calls* contains the maximal number of calls to function *rec* that happened in one execution of the algorithm.

V. Related Work

Guo, Bhakta and Harris [19] present a hardware-based intrusion detection approach for software applications. In a preprocessing step, they analyse the software under test statically to determine all direct branches. Afterwards, they try to cover the destinations of all computed branches by executing the software in a trusted zone. Finally, the software is executed in the field and abnormal branch execution is detected. Our approach is not limited to the detection of abnormal branch execution.

Scherer and Horváth [20] present an online approach for measuring the code and statement coverage of a program. They use an FPGA to decompress the program flow trace which mainly consists of bit-level operations to take the load of the host PC. The major drawback is that they require a high bandwidth (>100MB/s) connection to the host even for microprocessors with only around 100MHz clock frequency. Our solution can cope with microprocessors with clock frequencies up to 2GHz as we only transfer already analyzed and thus, strongly reduced information to the host.

Rapita Systems markets a trace logging solution called RTBx [21]. They claim a capacity of several days' worth of trace data. The analysis is very flexible in terms of application (runtime verification, code coverage, timing analysis) as it is done offline after data acquisition. Their system is built to fit into a 19" rack which makes it difficult to use in space-limited environments. In contrast to them, we analyze the trace data online.

Lee et al. [22] propose a hardware-based solution to detect return-oriented programming (ROP) attacks. They use the ARM CoreSight debug port and offline generated binary meta-data to calculate a shadow call stack online. The authors decided to store the meta-data and the program binary in the same memory resulting in a 2.39% runtime overhead. We use dedicated external memory for storing the meta-data which makes our approach non-intrusive, i.e., it does not influence the SuT in any way.

VI. Conclusion

In this contribution, we have presented a new way to use trace data provided by modern multi-core SoCs. The presented platform is capable of processing trace data in real-time. This enables many applications that have previously been impossible. These applications—including hybrid WCET estimation, code coverage, finding functional or timing bugs, continuous timing validation, and gathering complex statistics—greatly simplify debugging and validation of embedded SW.

In the future, we will develop more applications that are possible in the context of online trace analysis. Also, this approach has led to discussions with SoC manufacturers, how trace interfaces should be built to optimally support the online analysis.

Acknowledgment

This work was funded by the German Federal Ministry for Education and Research within the projects CONIRAS (funding ID 01IS13029) and ARAMiS II (funding ID 01IS16025), and by the European Union's Horizon 2020 research and innovation programme within the project COEMS (grant agreement no. 732016). The responsibility for the content remains with the authors.

References

[1] S. Grünfelder, *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*. dpunkt.verlag GmbH, 2017.

[2] ARM Ltd., "CoreSight™ Architecture Specification v2.0," 2013, ARM IHI 0029B.

[3] ——, "CoreSight™ Program Flow Trace™ PFTv1.0 and PFTv1.1 Architecture Specification," 2011, ARM IHI 0035B.

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

[5] IEEE-ISTO, "IEEE-ISTO 5001™-2012, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface," 2012.

[6] Freescale Semiconductor, Inc., "P4080 Advanced QorIQ Debug and Performance Monitoring Reference Manual," 2012, Rev. F.

[7] H. Theiling, "Control flow graphs for real-time system analysis. reconstruction from binary executables and usage in ilp-based path analysis," Ph.D. dissertation, Saarland University, 2003.

[8] C. Ferdinand and R. Heckmann, "aiT: Worst-case execution time prediction by static programm analysis," in *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions, 22–27 August 2004, Toulouse, France*, R. Jacquart, Ed. Kluwer, 2004, pp. 377–384.

[9] A. Weiss and A. Lange, "Trace-Data Processing and Profiling Device," 2016, US Patent 9286186B2.

[10] ANSI/VITA, "ANSI/VITA 46.0-2013 VPX: Base Specification," 2013.

[11] ——, "ANSI/VITA 57.1-2010 FMC: FPGA Mezzanine Cards Base Standard," 2010.

[12] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss, "Rapidly Adjustable Non-Intrusive Online Monitoring for Multi-core Systems," in *20th Brazilian Symposium on Formal Methods (SBMF 2017)*. Springer, 2017.

[13] T. Reinbacher, K. Y. Rozier, and J. Schumann, "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems," in *20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, ser. LNCS, vol. 8413. Springer, 2014, pp. 357–372.

[14] P. Gottschling and C. Hochberger, "ReEP: A Toolset for Generation and Programming of Reconfigurable Datapaths for Event Processing," in *24th Reconfigurable Architectures Workshop (RAW 2017)*. IEEE, 2017, pp. 141–149.

[15] B. Dreyer, C. Hochberger, A. Lange, S. Wegener, and A. Weiss, "Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016, pp. 4:1–4:11.

[16] T. Ballenthin, B. Dreyer, C. Hochberger, and S. Wegener, "Hardware Support for Histogram-based Performance Analysis of Embedded Systems." in *20th IEEE International Symposium On Real-time Computing (ISORC 2017)*. IEEE, 2017.

[17] A. Bauer and M. Leucker, "The Theory and Practice of SALT," in *NASA Formal Methods (NFM)*. Springer, 2011, pp. 13–40.

[18] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[19] Z. Guo, R. Bhakta, and I. G. Harris, "Control-flow checking for intrusion detection via a real-time debug interface," in *2014 International Conference on Smart Computing Workshops*, 2014, pp. 87–92.

[20] B. Scherer and G. Horváth, "Measurement based software execution tracing in HIL (Hardware In the Loop) tests," in *2014 IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, 2014, pp. 1–5.

[21] Continuous tracing with the RTBx data logger. [Online]. Available: https://www.rapitasystems.com/system/files/downloads/mc-pb-301_rtbx_product_brief_v4.pdf

[22] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP'15)*, 2015, pp. 3:1–3:8.