Multi-core WCET Analysis Using Non-Intrusive Continuous Observation

Daniel Kästner[1], Gernot Gebhard[1], Markus Pister[1], Simon Wegener[1], Christian Ferdinand[1],
Albert Schulz[2], Martin Sachenbacher[2], Martin Leucker[3], and Alexander Weiss[2]*

Abstract

For safety-relevant real-time applications, worst-case execution time (WCET) bounds have to be determined in order to demonstrate deadline adherence. For timing predictable microprocessors, worst-case execution time guarantees can be computed by static WCET analysis. Hybrid WCET analysis is a solution for covering effects from accesses to interference channels of multi-core processors. In this article we present a seamless approach for hybrid WCET analysis that tightly couples the tools TimeWeaver and CEDARtools. We will describe the underlying concepts, illustrate the tool workflow, and discuss the application of our approach to meet the timing requirements of the EASA AMC 20-193 guidance.

Keywords: DO-178C, multi-core, AMC 20-193, static analysis, real-time tracing, timing predictability, functional safety, certification

1 Introduction

In real-time systems the overall correctness depends on the correct timing behavior: each real-time task has to finish before its deadline. All current safety standards require reliable bounds of the worst-case execution time (WCET) of real-time tasks to be determined.

Until recently, current industry practice—in particular in the automotive domain [23]—oftentimes still relied on end-to-end measurements. However, the timing information obtained with them is only determined for one concrete input, but due to caches and pipelines, the timing behavior of an instruction depends on the execution history. Hence, one needs to take each possible hardware state into account. Therefore, usually no full test coverage can be achieved and there is no safe test end criterion. Techniques based on code instrumentation modify the code, which can significantly change the cache and pipeline behavior (probe effect): the times measured for the instrumented software are not necessarily identical to the timing behavior of the original software. Moreover, the results of end-to-end measurements are hard to interpret, as they are not related to particular parts of the code but only to the whole program.

One safe method for timing analysis is static program analysis by Abstract Interpretation which provides guaranteed upper bounds for the WCET of a task. Static WCET analyzers are available for complex processors with caches and out-of-order pipelines, and, in general, support single-core and multi-core processors. A prerequisite is that reliable models of the processor/System-on-Chip (SoC) architecture can be determined. However, there are modern high performance SoCs which contain unpredictable and/or undocumented components that influence the timing behavior. Analytical results for such processors are unrealistically pessimistic.

A hybrid WCET analysis integrates static value, loop, and path analysis with measurements to capture the timing behavior of tasks. Compared to end-to-end measurements, the advantage of hybrid approaches is that measurements of short code snippets can be taken. Increasing the number of measurements for each snippet increases the chance to catch the possible worst-case state for each of them without the need to trigger the worst-case initial hardware state for the whole task. When the snippets cover the complete program under analysis, a worst-case path can be computed. The probe effect can be avoided by leveraging the embedded trace unit (ETU) of modern processors, which allows a fine-grained observation of a core's program flow. These traces are usually analyzed offline, but new FPGA-based approaches allow to analyze them online, enabling continuous non-intrusive runtime monitoring of embedded software.

For multi-core systems, the main challenge for WCET analysis is the interference generated by other cores running in parallel. AMC 20-193 [9] covers means to bound and mitigate these effects. When static WCET analysis is performed, the maximum costs of possible interference must be included in the result, for example with the help of a WCRA (worst-case resource accesses) analysis that gives safe upper bounds for shared resource accesses. In a second analysis step, the bounds are multiplied with the maximal interference delays of these shared resources, giving the maximal interference costs for a particular shared resource. For hybrid WCET analysis, the picture is simpler: all observable interference is already contained in the measurements, and no extra analysis step is needed.

In this article, we describe TimeWeaver, a hybrid WCET analysis tool, its coupling with CEDARtools to exploit state-of-the-art runtime monitoring, and its use in the context of AMC 20-193.

*The TRISTAN project, nr. 101095947 is supported by Chips Joint Undertaking (CHIPS-JU) and its members Austria, Belgium, Bulgaria, Croatia, Cyprus, Czechia, Germany, Denmark, Estonia, Greece, Spain, Finland, France, Hungary, Ireland, Israel, Iceland, Italy, Lithuania, Luxembourg, Latvia, Malta, Netherlands, Norway, Poland, Portugal, Romania, Sweden, Slovenia, Slovakia, Turkey and including top-up funding by the German Federal Ministry of Education and Research (funding IDs 16MEE0273 and 16MEE0276).

[1] AbsInt Angewandte Informatik GmbH, Germany
{info@absint.com}
[2] Accemic Technologies GmbH, Germany
{aschulz,msachenbacher,aweiss@accemic.com}
[3] University of Luebeck, Germany
{mleucker@isp.uniluebeck.de}

2 EASA AMC 20-193 Objectives

EASA Amendment AMC 20-193 [9], published in 2022, discusses means and defines objectives for the demonstration of compliance with the applicable airworthiness specifications for airborne systems and equipment that contains multi-core processors (MCPs). In the following we will briefly summarize its main contents and motivate the methodology outlined in this article with respect to AMC 20-193 verification obligations.

AMC 20-193 applies to systems with two or more activated cores not executed in lockstep mode for which the item development assurance level (IDAL) of at least one relevant software application is A, B, or C. One of the basic motivations of the amendment is to determine and mitigate inter-core interference, since *"interference between the software applications or tasks executing on an MCP could cause safety-critical software applications to behave in a non-deterministic or unsafe manner, or could prevent them from having sufficient time to complete the execution of their safety-critical functionality"*. Indeed, interference delays can have a huge impact on the memory access latencies. Nowotsch et al. [21] measured maximal write latencies of 39 cycles when only one core of the P4080 [11] was active, and maximal write latencies of 1007 cycles when all eight cores were running.

The guidance formulated by AMC 20-193 is structured in six stages, (i) *planning*, (ii) *setting of MCP resources*, (iii) *interference channels and resource usage*, (iv) *software verification*, (v) *error detection and handling, and safety nets*, and (vi) *data to complement the accomplishment summaries*.

The *planning* stage provides guidance to enhance the system development and verification processes in a way that will enable the hardware and the software hosted on the MCP to satisfy the functional, performance, and timing requirements of the system. Amongst others, Objective *MCP_Planning_1* requires that the applicant specifies the MCP they intend to use, the number of the active cores, and the software architecture hosted on the MCP. In particular, the applicant has to *"identify whether or not the MCP platform will provide robust resource partitioning and/or robust time partitioning"*. Objective *MCP_Planning_2* demands to describe the planned use of the shared resources, taking into account the time interference possibly caused by the usage, as well as the planned means to verify the usage, e.g., the tools and techniques planned for WCET analysis / timing verification.

Stage (ii) is concerned with the configuration of the MCP. According to objective *MCP_Resource_Usage_1*, the applicant has to determine and document any settings that may affect the system's ability hosted on the MCP to satisfy the functional and non-functional requirements. In stage (iii), objective *MCP_Resource_Usage_3* requires identifying and mitigating interference channels, while objective *MCP_Resource_usage_4* demands to identify and allocate resources and to verify that the demands for resources of the MCP and the interconnect do not exceed the available resources.

In the *software verification* stage, objective *MCP_Software_1* aims at providing assurance that the time bounds defined for the system are not violated. If the platform provides robust resource partitioning and robust time partitioning, the WCET of software applications may be determined separately. In this context, tools for computing static WCET guarantees, e.g., aiT WCET Analyzer [10] are applicable. If no robust resource and time partitioning can be guaranteed, the WCET has to be determined with all software components on all cores executing in the intended final configuration. This is the topic on which this article concentrates: we will present an efficient methodology for hybrid WCET analysis that allows interference-aware WCET bounds to be computed non-intrusively and that provides feedback on the trace coverage obtained.

Verification goal *MCP_Software_2* demands to verify that the data and control coupling between all SW components has been exerted, and that it is correct. Here, two aspects are needed: the feasible data and control coupling has to be determined, and the data and control coupling coverage achieved by the requirements-based testing has to be determined. An approach for sound data and control coupling analysis based on the static analyzer Astrée has been presented in [16]; the coverage information needed may be complemented by the trace data of the hybrid WCET analysis presented in the following sections.

Stage (v) is concerned with error detection and mitigation. Effects of any failure that may happen inside the MCP needs to detected and handled according to the safety goals of the system hosted on the MCP. "Safety nets" may provide a fail-safe containment for these failures. Finally, stage (vi) requires the applicant to provide a description of how the objectives of AMC 20-193 are satisfied.

From a timing analysis point of view, the most crucial factors addressed by AMC 20-193 are the selection of the MCP; the identification, documentation, and assessment of interference channels; and the availability of robust resource and robust time partitioning. All these affect the suitability of the applied methods for timing verification. For example, a MCP providing timing predictability [5, 7, 30] and timing compositionality [12] allows for a fully static analysis. The effects of resource conflicts can then be bounded with a worst-case resource accesses (WCRA) analysis.

The availability of robust resource partitioning directly influences the choice of the analysis strategy, as formulated in objective *MCP_Software_1*. Robust partitioning can either be achieved via dedicated hardware features, or via a suitable software architecture. For example, the privatization of shared resources can prevent resource access conflicts and hence, provide robust partitioning. Ways to implement privatization of shared resources include TDMA-based resource scheduling [24] and runtime resource capacity enforcement [21].

However, robust partitioning may be hard to achieve for unpredictable or undocumented features of a MCP, as the same features that prevent the design and implementation of precise static timing analyses often also make the analysis and mitigation of interference channels difficult. Consider for example a shared cache with pseudo-random replacement: The mitigation of such an interference channel may need some kind of software-based cache partitioning [26] which complicates the software architecture and may conflict with other development goals, for example the use of specific operating systems or software libraries.

Micro-benchmarks that intentionally drive contention on shared resources [28] (sometimes called “stressors” or “daemons”) are useful for determining and assessing interference channels, i.e., for platform characterization. They may uncover undocumented interference channels and measure the impact of resource conflicts on the timing behavior, i.e., they may help to determine the interference cost of a resource conflict for that specific interference channel. However, their applicability for measurement-based timing verification is limited. In case robust partitioning is available for the system hosted on the MCP, it will by definition prevent any adverse effects of the stressor on the software tasks for which the timing needs to be verified. If no robust partitioning is available, it will lead to overly pessimistic timing behavior being observed that not necessarily reflect the real timing behavior of the system. Moreover, the use of a stressor during measurement-based timing verification contradicts objective *MCP_Software_1* of AMC 20-193: the WCET has to be determined with all software components on all cores executing in the intended final configuration.

Instead, in accordance with AMC 20-193, we propose to apply hybrid timing analysis for commercial-of-the-shelf (COTS) multicore processors for which robust partitioning cannot be guaranteed. The measurements should be performed in the intended final configuration without any artificial generation of contention. Quite the contrary, the software architecture should prevent any unneeded interference, if possible.

3 Hybrid WCET Analysis

The goal of non-intrusive trace-based WCET analysis is to observe execution times of tasks and interrupt service routines (ISRs) including the timing interference due to concurrent execution and multi-core resource conflicts, while avoiding the probe effect.

The solution which is implemented in the hybrid WCET analysis tool TimeWeaver [1] combines static context-sensitive path analysis with non-intrusive real-time instruction-level tracing to provide worst-case execution time estimates. By its nature, an analysis using measurements to derive timing information is aware of timing interference due to concurrent execution and multi-core resource conflicts, because the effects of asynchronous events (e.g. activity of other running cores or DRAM refreshes) are directly visible in the measurements. The probe effect is completely avoided since no code instrumentation is needed. The trace information can be provided out-of-the-box by embedded trace units of modern processors, like Nexus IEEE-ISTO 5001™ [13], Infineon MCDS™ [14], or ARM CoreSight™ [3], as used for example in the NXP Layerscape LX2xxx. These trace protocols allow the fine-grained observation of a program execution and assign timestamps to specific program points during execution. Thus, the traces contain an execution time measurement for each trace segment stretching out between two consecutive trace points. The computed estimates are safe upper bounds with respect to the given input traces, i.e., TimeWeaver derives an overall upper timing bound from the execution time observed in the given traces by employing path extrapolation [18]. Thus, the coverage of the input traces on the analyzed code is an important metric that influences the quality of the computed WCET estimates. ARM, PPC, RH850, and TriCore/AURIX are already supported; a TimeWeaver version targeting RISC-V is currently in development, exploiting the open trace interface developed as part of the European Chips JU TRISTAN [27].

3.1 Structure of TimeWeaver

The main inputs for TimeWeaver are the fully linked executable(s), timed traces, and the location of the analyzed code in the memory (entry point, which usually is the name of a task or function). The analysis proceeds in several stages: decoding, loop/value analysis, trace analysis, and path analysis (see Figure 1). Most steps in this tool chain are shared with aiT WCET Analyzer which provides a fully static analysis targeting timing-predictable processors [10].

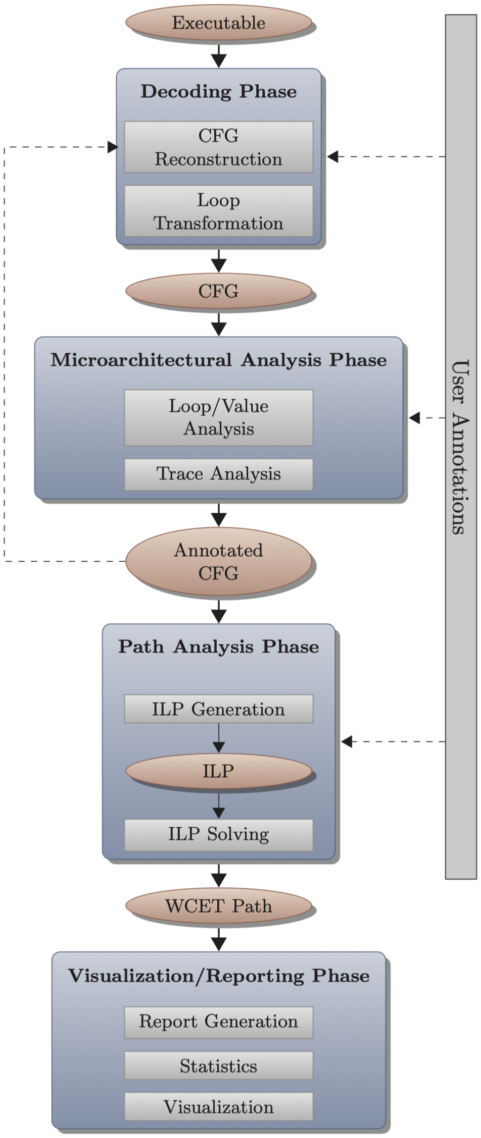


Figure 1: Structure of TimeWeaver. The analysis proceeds in four key stages: decoding, loop/value analysis, trace analysis, and path analysis.

In the decoding phase, the instruction decoder reads and disassembles the input executable(s) into its individual instructions. Architecture specific patterns decide whether an instruction is a control-flow related instruction (e.g., call, branch, return) or just an ordinary instruction. This knowledge is used to reconstruct the basic blocks of the control-flow graph (CFG). Then, the control flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, either the user can write some annotations to guide the control-flow reconstruction, or TimeWeaver can be instructed to extract the targets of unresolved branches or calls from the input traces. To this end there is a feedback loop between the CFG reconstruction and the trace analysis step.

In the next phase, several microarchitectural analyses are performed on the reconstructed CFG starting with the combined loop and value analysis. It determines possible values of registers and memory cells, addresses of memory accesses, as well as loop and recursion bounds. Based on this, statically infeasible paths are computed, i.e., parts of the program that cannot be reached by any execution under the given configuration. This is important because each detected infeasible path increases the trace coverage. Such paths are pruned from further analysis. If the value analysis cannot compute a loop bound or if the computed bound is not precise enough, users can specify custom bounds by means of annotations which are used by the analysis. Loop bounds can also be extracted from the traces.

After value analysis, the analyzer has annotated each instruction in the control-flow graph with context-sensitive analysis results. This context-sensitivity is important because the precision of an analysis can be improved significantly if the execution environment is considered [25]. For example, if a routine is called with different register values from two different program points, the execution time in both situations might be different. Depending on the context settings, this is taken into account leading to higher precision in the analysis result.

In the trace analysis step the given traces are analyzed such that each trace event is mapped to a program point in the control-flow graph. This mapping defines the trace points and trace segments between them and is not only necessary for the whole analysis but also ensures that the input trace matches the analyzed binary. In case a preemptive system has been traced, interrupts are detected and reported. The extracted timing information, i.e., the clock cycles which have been elapsed between two consecutive trace points are annotated to the CFG in a context-sensitive manner.

Afterwards, a CFG which combines the results of value analysis and traced execution timings (both context-sensitive) is available. This graph is the input for the next step, the path analysis phase. Here, the trace segment times alongside the control-flow graph are used to generate an integer linear program (ILP) formulation to compute the worst-case execution path with respect to the traced timings. At this point, the recorded times for each pair of trace segment and analysis context get maximized. The implicit path enumeration technique (IPET) used by TimeWeaver allows to construct WCET estimates for paths that have not been observed themselves during measurements but are only created during path extrapolation. Thus, not every path needs to be explicitly observed in the traces, greatly reducing the number of measurements that need to be taken.

TimeWeaver also computes the timing contributions of each function and uses debug information to map this information back to the source code. Thus, TimeWeaver allows to have an in-depth look where time is spent and helps to uncover hotspots. These are often unexpected. For example, the accidental use of 64-bit integer division on a 32-bit architecture leads to the inclusion of software routines for this arithmetic operation. Changing the underlying integer type in the source code reduces this overhead.

3.2 Quality of Measurements

Besides the WCET estimate itself, TimeWeaver also gives guidance concerning the quality of the measurements by computing several coverage metrics. In principle, path coverage is the best coverage criterion. Achieving 100 % path coverage means that every possible path through a program has been tested. However, both computing path coverage as well as trying to reach full path coverage is computationally extremely expensive, as there are exponentially many paths for the number of branches in a program. Thus, TimeWeaver employs path extrapolation to reduce the burden of having each path measured at least once. The traces are cut into segments, i.e., the path between two consecutive trace points. These segments may span several basic blocks in the CFG. The ILP formulation of the path analysis allows to construct the longest path in the CFG based on the trace segments even if this path has not been observed directly in the traces.

TimeWeaver computes the following coverage metrics: block/instruction coverage, edge coverage, and flow coverage. The metrics are computed at the machine code level; a mapping to the source code level is available. For each basic block in the CFG, TimeWeaver reports whether it has been covered by measurements and if yes, how often. This information is also used to compute the instruction coverage. Paths for which infeasibility has been proven need no measurements, so associated blocks are excluded while computing coverage. This makes it easy to detect missing tests that are needed to trigger specific execution scenarios.

Checking the number of measurements for each basic block allows to assess the confidence in the measured timings. To support this, TimeWeaver also reports for each trace segment the minimum, maximum, and average observed execution times, plus the standard deviation. The same information is also computed for all traces, making outliers easily detectable. Moreover, loops for which the analyzed worst-case iteration count has not been measured are also reported.

Some basic blocks are reachable from multiple predecessors, so full block coverage does not ensure that each way a block can be reached has been observed. However, this is an important metric for timing analysis as many performance-enhancing features of modern processors take the execution history into account. Hence, TimeWeaver additionally computes the edge coverage and the flow coverage for each block. Full edge coverage means that each possible combination of a block and its predecessors has been observed. Flow coverage improves on this metric by taking the successors into account, i.e., each possible combination of predecessor → block → successor needs to be observed to reach full flow coverage. Flow coverage helps to uncover hidden dependencies in the measurements.

Most implementations of the various trace protocols do not emit a trace message for each branch but only if the branch target has been computed, or if the branch history buffer is full. The program flow is captured by recording single bits for taken/not-taken branches. Thus, there might be some control-flow joins in the CFG for which no trace point exists, preventing the path extrapolation at these program points. To aid the path extrapolation, special code patterns can be used to force the ETU to emit a trace message. For example, the ETU can be configured to emit a trace message for each branch-and-link instruction on the PowerPC architecture. Together with the trace point at return instructions, this feature ensures that trace segments do not cross routine boundaries. Another possibility is the usage of lightweight hardware-supported instrumentation to enforce trace points at specific locations.

Many performance-enhancing features like branch prediction, caches, pipelining, etc. take the execution history into account. Thus, the hardware state influences the timing behavior of a code snippet. The longer the observation period, the greater the likelihood of capturing the WCET situations for each segment in the trace. Although it cannot be guaranteed that the WCET situation for each trace segment occurred during tracing, it is much more likely than trying to trigger the WCET situation for the whole task or ISR. Moreover, since the path extrapolation combines maximum trace segment times that might be mutually exclusive in reality, the resulting WCET estimate is usually larger than the maximum observed execution time of that path, adding some kind of safety margin. However, long observation periods result in large trace files. Hence, a balanced approach is favorable. The key here is not to analyze just any trace sequences, but those with relevant anomalies such as particularly long execution times or specific execution prefixes (see Figure 2). How exactly these sequences are identified is explained in Section 4. A virtually unlimited observation period (typically spanning a few hours to a few days)—as provided by Accemic's CEDARtools—can significantly enhance the statistical relevance and, as a result, increase confidence in the results of the WCET analysis. Thus, we coupled TimeWeaver with CEDARtools, see Section 5.

3.3 Qualification Support

AbsInt provides Qualification Support Kits (QSKs) to assist the automatic qualification of its tools up to the highest criticality levels. The QSKs aim at demonstrating the correct functioning of the tool in the operational context of the tool user with respect to the relevant tool-influencing parameters like options, code constructs, provided external information for the analyzers, etc. The QSK consists of the following parts: specification of the tool functional requirements, test cases and test case procedures, requirements trace data (traceability matrix), test suite and execution framework, and tool lifecycle data. The tool lifecycle data demonstrate development in accordance to safety standards. The QSKs enable the qualification of TimeWeaver in accordance to domain-relevant safety standards like DO-178C / DO-330, ISO-26262, IEC-61508, EN-50128 / EN-50657, and more. More details on the tool qualification strategy of AbsInt can be found in [17].

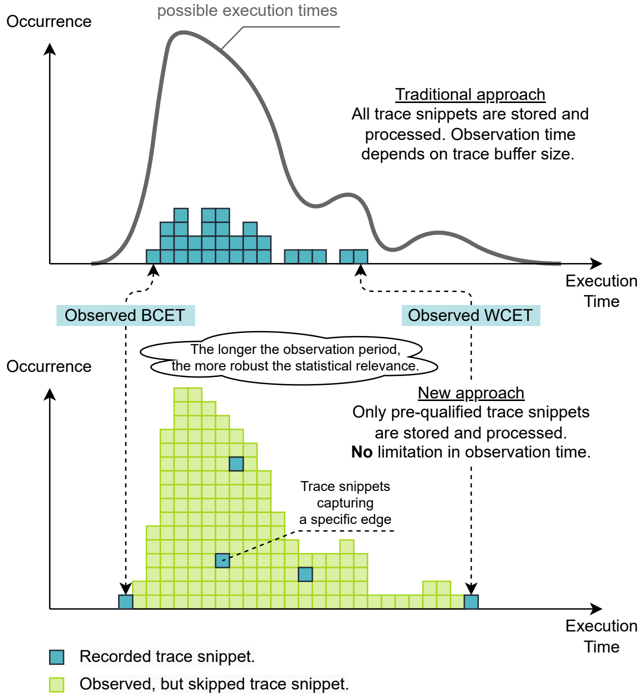


Figure 2: Comparison of traditional trace analysis approaches with the new online analysis implemented by CEDARtools that allows for a virtually unlimited observation period.

4 Embedded Trace

Embedded trace [22] is a method for non-intrusively monitoring processors, providing valuable insights into their program execution at the machine code level. By leveraging *embedded trace*, engineers can gain a deep understanding of processor operations without disrupting the normal execution of software or hardware. This technology is implemented through specific hardware structures that are tightly coupled to the CPU(s) and are used to capture data about the execution of code. The major advantage of using embedded trace is that the embedded trace unit (ETU) does not influence the CPU when monitoring the program flow, i.e., from the application's point of view it is not possible to tell whether the ETU is active or not.

Figure 3 provides an overview of the individual elements and the trace data flow of a processor equipped with embedded trace. We refer to these elements by their encircled identifiers in the following detailed explanation.

There are several options for transferring the trace data generated by the ETU to an external tool. The trace data can be temporarily stored inside the processor in special embedded trace buffers (ETB) *(A)* or in the system memory *(B)*. Trace data can also be output via dedicated embedded trace interfaces (ETI) *(C1)* or via system interfaces *(C2)* to avoid the limited observa-

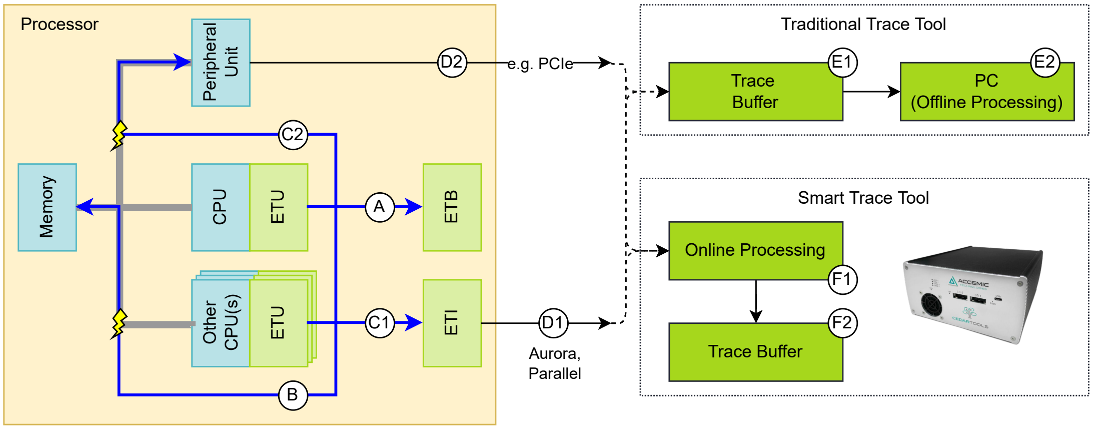


Figure 3: Elements of embedded trace. The target system producing trace data is shown on the left, while the off-chip processing of trace data is shown on the right.

tion time caused by the memory limitations. The direct output of the trace data via ETI is technically the most elegant solution, since it does not interfere with the processor. However, implementing an interface with the required high bandwidth is expensive, so some processor designs compromise by storing the trace data in the main memory *(B)* or outputting it via a system interface *(C2)*. However, this is also done at the expense of the desired non-intrusiveness—the required bus operations interfere with the application (indicated by the yellow flash symbols in Figure 3). To allow the use of trace information outside the SoC, trace data is output either directly using dedicated trace interfaces (parallel [20] or high-speed serial like Aurora [31]) *(D1)*, or via fast system interfaces (usually PCIe, sometimes USB) *(D2)*.

This trace data is then received by an external trace tool. There exist simple trace buffers and advanced smart trace tools. Trace buffers *(E1)* typically contain several GiB of memory where the received trace data is temporarily stored before the decoding and further processing in a PC *(E2)*. The disadvantage of this approach is the limited size of the trace buffer, which limits the observation time and thus contradicts the requirement of being able to observe and analyze a system for as long as possible. This problem is addressed by a new generation of smart trace tools, which continuously process incoming trace data *(F1)* and perform real-time reconstruction of the control flow, including timing information. Scalable parallelization with FPGA-based hardware acceleration enables control-flow reconstruction for processors with more than 2 GHz operating clock (for example the Layerscape® LX2160A). Control-flow reconstruction is also supported for applications running on multitasking operating systems such as Linux or VxWorks®, and for applications using dynamically loaded libraries. Based on the live reconstructed control flow and a set of filters, the system can now qualify and store relevant trace snippets in a segmented buffer with a capacity of several GiB *(F2)*. This smart architecture has been implemented in CEDARtools, which are presented in Section 5.

A crucial requirement for trace-based WCET analysis is precise timing information about the observed program execution. However, a high-frequency output of timestamps could significantly inflate the required trace bandwidth. Furthermore, modern trace protocols like Intel PT, Arm Coresight ETM v4, and Nexus Branch History Trace focus on bandwidth optimization and hence, emit trace data not for each executed instruction, but only for the execution of conditional and computed branches. In addition, it cannot be relied upon to generate a timestamp for every jump in every case, but rather to output a timestamp only after a group of jumps. Some trace protocols support setting a minimum cycle count (Arm® Cortex®-A53: 4 CPU clock cycles [4]) at which a new timestamp is sent for a branch command. For other protocols, the sending of a new timestamp is linked to the execution of an indirect jump [13]). Depending on the monitoring requirements (accuracy, acceptable intrusiveness) and the capabilities of the embedded trace unit implemented in the processor, a suitable trace strategy can be found for each application. Balancing the capture of sufficient timing information while optimizing trace bandwidth is a critical challenge when utilizing *embedded trace* for WCET analysis.

An enhancement of the *embedded trace* approach, tackling the two issues discussed before, is currently being developed within the TRISTAN project [27]. In a processor-side trace subsystem for the RISC-V architecture, the application is empowered to communicate relevant trigger points to the trace unit through minimally intrusive instrumentation. Ideally, dedicated static analysis tools can automatically identify these trigger points, and the compiler can insert the appropriate instrumentation into the binary code as non-functional instructions. This instrumentation performs deterministic access to special CPU registers, leaving the registers relevant to the application unaltered. From the application's point of view, it is equivalent to a sequence of `nop` instructions, which consumes only a few CPU clock cycles and does not access the system bus. In most cases, it is therefore justifiable to leave such minimally intrusive and deterministic instrumentation in the release code. The identified trigger points could, for example, involve measuring the time between instruction A and instruction B, storing the trace sequence from 1 ms before A to 1 ms after B for every case where the time between A and B reaches a new maximum and discarding the previously stored trace sequence. This foreknowledge of the code segments relevant for WCET measurements, known at design time, can also be used for precise and bandwidth-efficient control of the output of timing information (CPU clock cycles, wall clock time) by the CPU. Instead of sending timestamps indiscriminately as it was done previously, precise timing information can now be embedded into the trace data stream, ensuring that no redundant information is transmitted while also preventing any gaps in the timing data.

More complex functional tests can be executed on the system traces going beyond simple code coverage or timing measurements. Runtime verification [19] is a formal dynamic method that considers actual runs of a system, and checks properties on streams of events—i.e., system traces—using so-called monitors constructed from high-level specifications. Given such a specification formulated in an appropriate specification language, a monitor is synthesized that runs in parallel to the execution of the system. It accepts exactly the traces of the system adhering to the specification. All other executions are identified as failures. Some level of system resilience can be achieved by so-called runtime reflection [19], which aims to devise mitigation actions in the case of failures to restore some of the system's functionality. This is particular useful for safety-critical systems for which no safe state exists, and thus, being fail-safe is not sufficient but being fail-operational is necessary. The same techniques can also be used to implement complex trigger conditions for trace recording.

The smart trace approach has been implemented in the form of software tools and specialized hardware, and demonstrated in a number of pilot applications. Theoretical and practical aspects of it have been described in several previous publications [22, 15, 29, 6, 8].

5 TimeWeaver/CEDARtools Coupling

Nowadays, in many recent computing platforms for embedded systems, trace data of the program execution is provided by the target system via dedicated, often already existing processing interfaces. Existing trace tools are logging such trace data in a file for offline analysis. This allows to analyze the cause of complex error patterns even after a system has been released. However, this allows only post-mortem analysis and due to the sheer data volume of traces (several GiB/s even for medium-sized processors), the time span that can be observed is limited.

As an alternative, we present with CEDARtools [2] an approach for dynamic analysis and inspection of embedded systems that is based on the idea of on-the-fly analysis of trace data at run-time. We argue that the approach offers several advantages compared to existing static and dynamic analysis methods. The trace analysis capability can be fully separated from the target system; in our case, through specialized high-performance hardware (FPGAs) that can keep up with the speed of trace generation. This means that no instrumentation of the system under scrutiny is necessary, and its original, unaltered behavior can be observed (non-intrusiveness). This is especially important for precise timing analysis in real-time applications, and for analyzing systems with non-deterministic compute architectures and parallelism. In addition, in our solution the on-the-fly processing of traces obliterates the need for storing (most or all of) the trace data[1], thus enabling long-term or even continuous monitoring to catch also sporadic rare events that are otherwise hard to track down. With a capacity of several GiB, the trace memory of CEDARtools is large enough to store relevant sequences for evidentiary purposes. Both simple and complex triggers can be defined in a high-level specification language [15] and make it possible to precisely identify relevant parts of the trace data stream. Hence, only those relevant parts of the trace stream need to be stored in a (partitioned) ring buffer. For example, each time a new maximum execution time a task is observed, the corresponding trace data replaces the old measurements, ensuring that a trace of the currently observed worst-case scenario is stored for later offline analysis. Furthermore, the efficiency of the analysis is significantly increased by monitoring not only one point of interest, but up to 32 of them simultaneously.

The coupling between TimeWeaver and CEDARtools is depicted in Figure 4. It consists of three main components: the target system on the left, the FPGA-based CEDARtools trace box in the middle, and the offline hybrid timing analysis on the right side of the figure.

The binary executable of the software under analysis is loaded onto the target system. The test engineer decides which parts of the software system are of particular interest for the timing analysis, for example tasks or ISRs. As part of a test campaign, they are executed inside a test harness that generates the necessary input stimuli to trigger the intended behavior under scrutiny (unit test / integration test). While the target system is running, trace data is emitted via the trace port of the target system. Attached to this port is the CEDARtools trace box.

The trace box processes the incoming trace data at runtime. The highly compressed trace stream is decoded inside a FPGA to reconstruct the control flow of the software system under observation. The processing speed matches the execution speed of the target system to enable the live monitoring of the software system. The trace box allows virtually unlimited observation periods since only those parts of the trace stream are stored for later offline processing that match predefined criteria. The criteria a given by the test engineer. One possibility is the collection of bad-case scenarios for the points of interests (i.e., the tasks and ISRs of the system). Here, the internal ring buffer of the trace box is partitioned in up to 32 slices. Each slice can store the trace of one observation of a task/ISR together with the observed execution time of this instance. If the currently running instance of the task/ISR exceeds the previous maximum observed execution time, the currently running instance replaces the one previously stored in the slice of the ring buffer. Another possibility is the collection of traces for specific trigger conditions, for example only after a specific combination of tasks has been executed. The trace data stored in the ring buffer can be exported for further offline processing or as a witness for specific execution scenarios.

Finally, the exported trace files are used in a hybrid timing analysis of the software system. Here, the binary executed on

[1] Certification of safety-critical systems may, however, sometimes require evidence that is documented in the form of traces. Hence, CEDARtools allow to export selected traces.

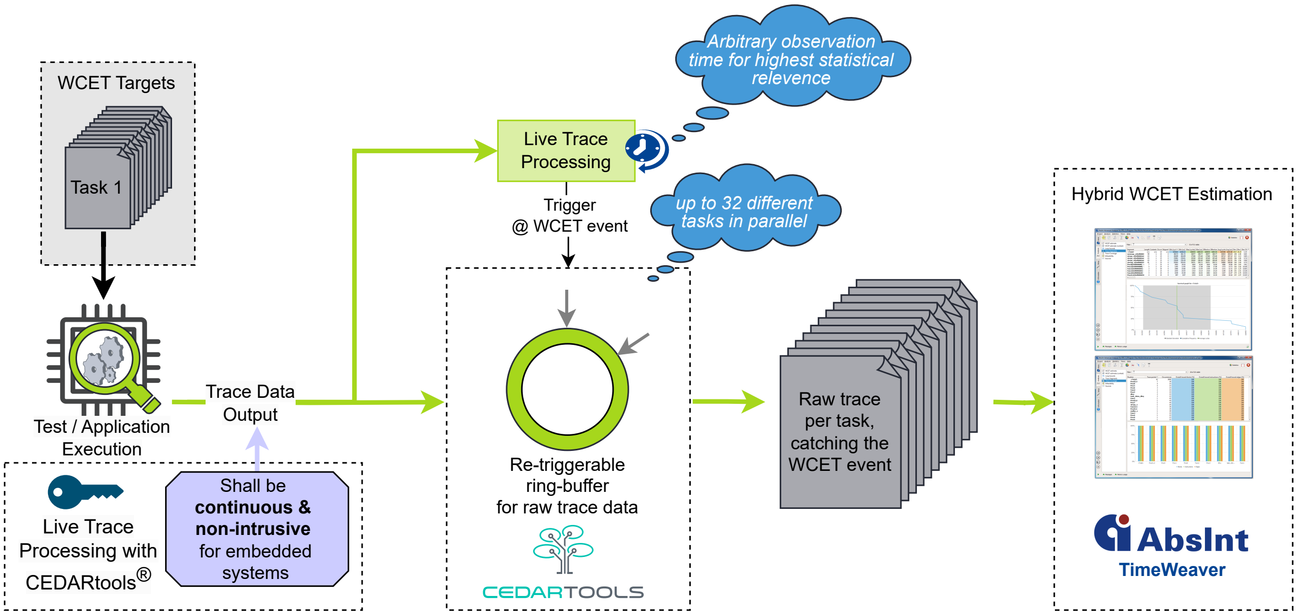

WCET Targets
Task 1
Test / Application Execution
Trace Data Output
Live Trace Processing with CEDARtools®
Shall be continuous & non-intrusive for embedded systems
Live Trace Processing
Trigger @ WCET event
Arbitrary observation time for highest statistical relevence
up to 32 different tasks in parallel
Re-triggerable ring-buffer for raw trace data
CEDARTOOLS
Raw trace per task, catching the WCET event
Hybrid WCET Estimation
AbsInt
TimeWeaver

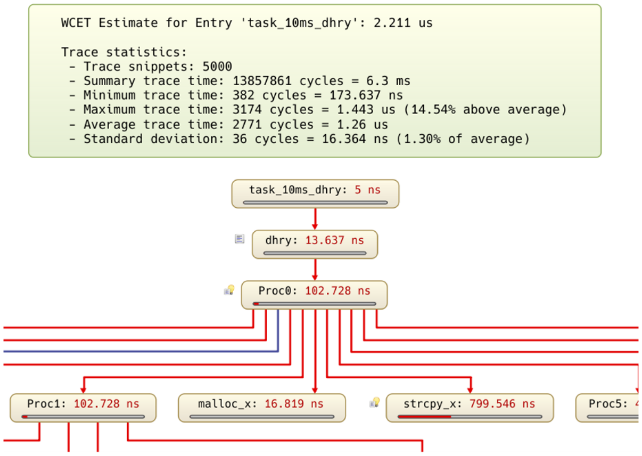
Figure 4: Coupling of CEDARtools and TimeWeaver for hybrid WCET estimation of multicore systems.

the target system, the trace files, and the entry points of the tasks and ISRs that are of particular interest are the input for TimeWeaver. The inputs are processed as explained in Section 3. Path extrapolation during the static path analysis phase of TimeWeaver constructs a critical path from the trace segments obtained via the CEDARtools trace box. Thus, the coupling will find the worst possible path through the CFG even if it has never been observed directly. The result of the hybrid timing analysis is an estimate of the WCET, together with the visualisation of the critical path and statistics for the various trace segments used to compute the estimate. One example for a timing analysis result is depicted in Figure 5. In case the coverage report of TimeWeaver uncovers code regions that have less than the required percentage of instruction, edge, or flow coverage, CEDARtools can be instructed to save exactly those trace sequences that cover the code region in question.

We argue that the targeted selection of trace snippets within an arbitrarily long observation period has a higher statistical relevance than the recording of trace snippets that are randomly located within a limited observation period. The analysis of several avionics applications, which we are not allowed to publish, has confirmed this assumption. To demonstrate how the method works, a sample application was developed with a random Gaussian-distributed execution time. As expected, we obtained a realistic WCET estimate with the presented methodology which is 52% higher than when using a randomly selected full trace sequence within a limited observation period. However, these results cannot be generalized and are highly dependent on the application under investigation.

6 Conclusion

For multi-core systems, the main challenge for WCET analysis is the interference generated by other cores running in parallel. If the platform provides no robust resource partitioning and robust time partitioning, static WCET guarantees are unrealistically pessimistic. In this article we have presented a hybrid WCET analysis that combines static value, loop, and path analysis with non-intrusive measurements to compute interference-aware WCET bounds and provide feedback on the trace coverage obtained. Key to reliable hybrid WCET estimation is the ability to observe a processor in detail over arbitrary time periods, with no or at most exceedingly minimal instrumentation. This is achieved by embedded trace units usually already implemented in all processors in combination with new powerful live control-flow analysis tools. Our approach has been implemented by a coupling and extension of the tools TimeWeaver and CEDARtools. It is compliant to the software verification requirements of the EASA AMC 20-193 guidance.



```
WCET Estimate for Entry 'task_10ms_dhry': 2.211 us

Trace statistics:
 - Trace snippets: 5000
 - Summary trace time: 13857861 cycles = 6.3 ms
 - Minimum trace time: 382 cycles = 173.637 ns
 - Maximum trace time: 3174 cycles = 1.443 us (14.54% above average)
 - Average trace time: 2771 cycles = 1.26 us
 - Standard deviation: 36 cycles = 16.364 ns (1.30% of average)
```

Figure 5: Result visualisation of a WCET analysis using the TimeWeaver/CEDARtools coupling for the NXP LX2160 processor, highlighting the extrapolated critical path that leads to the WCET estimate in red. The statistics include the observed BCET (”minimum trace time”) and the observed WCET (”maximum trace time”).

References

[1] AbsInt Angewandte Informatik GmbH. TimeWeaver. https://www.absint.com/timeweaver.

[2] Accemic Technologies GmbH. CEDARtools. https://accemic.com/cedartools/.

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

[4] ARM Ltd. Arm Cortex-A53 MPCore Processor Technical Reference Manual, 2018. ARM DDI 0500J.

[5] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 13(4):82:1–82:37, 2014.

[6] L. Convent, S. Hungerecker, T. Scheffel, M. Schmitz, D. Thoma, and A. Weiss. Hardware-based runtime verification with embedded tracing units and stream processing. In C. Colombo and M. Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2018.

[7] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza (Burguière), J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingenieurs de l'Automobile*, 807:26–42, 2010.

[8] N. Decker, B. Dreyer, P. Gottschling, C. Hochberger, A. Lange, M. Leucker, T. Scheffel, S. Wegener, and A. Weiss. Online analysis of debug trace data for embedded systems. In J. Madsen and A. K. Coskun, editors, *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 851–856. IEEE, 2018.

[9] EASA. General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20) – Amendment 23. AMC 20-193 Use of multi-core processors, 2022.

[10] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001.

[11] Freescale Semiconductor, Inc. QorIQ™ P4080 Communications Processor Product Brief, Rev. 1, 2008. http://cache.freescale.com/files/32bit/doc/prod_brief/P4080PB.pdf.

[12] S. Hahn, M. Jacobs, and J. Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real Time and Networks Systems*, October 2016.

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

[14] Infineon Technologies AG. Debug Support – AURIX™ TC2xx Microcontroller Training V1.0. https://www.infineon.com/dgdl/Infineon-AURIX_Debug_Support-Training-v01_00-EN.pdf?fileId=5546d46269bda8df0169ca5c0d9a2537[retrieved: January 2021].

[15] H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, and A. Weiss. Tessla - an ecosystem for runtime verification. In T. Dang and V. Stolz, editors, *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *Lecture Notes in Computer Science*, pages 314–324. Springer, 2022.

[16] D. Kästner, L. Mauborgne, S. Wilhelm, C. Mallon, and C. Ferdinand. Static Data and Control Coupling Analysis. In *11th Embedded Real Time Systems European Congress (ERTS2022)*, Toulouse, France, June 2022.

[17] D. Kästner, M. Pister, and C. Ferdinand. Obtaining DO-178C Certification Credits by Static Program Analysis. In *ERTS2022*, Toulouse, France, June 2022.

[18] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In S. Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *Open Access Series in Informatics (OASIcs)*, pages 1:1–1:11, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[19] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.

[20] MIPI Alliance, Inc. MIPI PTI v2.0 Specification for Parallel Trace Interface, May 2011.

[21] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 109–118. IEEE Computer Society, 2014.

[22] T. B. Preußer, S. Gautham, A. D. Rajagopala, C. R. Elks, and A. Weiss. Everything you always wanted to know about embedded trace. *Computer*, 55(2):34–43, 2022.

[23] K. Schmidt, D. Marx, J. Harnisch, A. Mayer, U. Dannebaum, and H. Christlbauer. Non-Intrusive Tracing at First Instruction, 2015. SAE Technical Paper 2015-01-0176.

[24] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing predictability on multi-processor systems with shared resources. In *Workshop on Reconciliating Predictability and Efficiency at EMSOFT 2009*, 2009.

[25] S. Stattelmann and F. Martin. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *Open Access Series in Informatics (OASIcs)*, pages 64–76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[26] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 300–303, New York, NY, USA, 2008. ACM.

[27] TRISTAN. Together for RISc-V Technology and ApplicatioNs. https://tristan-project.eu.

[28] S. H. VanderLeest and S. R. Thompson. Measuring the impact of interference channels on multicore avionics. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–8, 2020.

[29] A. Weiss, S. Gautham, A. V. Jayakumar, C. R. Elks, D. R. Kuhn, R. N. Kacker, and T. B. Preußer. Understanding and fixing complex faults in embedded cyberphysical systems. *Computer*, 54(1):49–60, 2021.

[30] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[31] Xilinx, Inc. Aurora Protocol Specification, Sept. 2007.