TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis

Daniel Kästner
AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany

Markus Pister
AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany

Simon Wegener
AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany

Christian Ferdinand
AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany

Abstract

Many embedded control applications have real-time requirements. If the application is safety-relevant, worst-case execution time bounds have to be determined in order to demonstrate deadline adherence. For high-performance multi-core architectures with degraded timing predictability, WCET bounds can be computed by hybrid WCET analysis which combines static analysis with timing measurements. This article focuses on a novel tool for hybrid WCET analysis based on non-intrusive instruction-level real-time tracing.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases Worst-Case Execution Time (WCET) Analysis, Real-time Tracing, Functional Safety

Digital Object Identifier 10.4230/OASIcs.WCET.2019.1

Funding This work was funded by the German Federal Ministry for Education and Research (BMBF) within the project ARAMiS II with the funding ID 01IS16025B, and within the project EMPHASE with the funding ID 16EMO0183. The responsibility for the content remains with the authors.

1 Introduction

In real-time systems the overall correctness depends on the correct timing behaviour: 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.

With end-to-end timing measurements timing information is only determined for one concrete input. Due to caches and pipelines the timing behaviour of an instruction depends on the program path executed before. 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 behaviour (probe effect): the times measured for the instrumented software do not necessarily correspond to the timing behaviour of the original software.

One safe method for timing analysis is static analysis by Abstract Interpretation which provides guaranteed upper bounds for WCET of tasks. Static WCET analyzers are available for complex processors with caches and complex pipelines, and, in general, support single-core processors and multi-core processors. A prerequisite is that good 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 behaviour. Analytical results for such processors are unrealistically

pessimistic. A hybrid WCET analysis combines static value and path analysis with measurements to capture the timing behaviour of tasks. Compared to end-to-end measurements the advantage of hybrid approaches is that measurements of short code snippets can be taken which, in combination, cover the complete program under analysis. Based on these measurements a worst-case path can be computed.

In this article we will focus on the hybrid WCET analyzer TimeWeaver which avoids the probe effect by leveraging the embedded trace unit of modern processors. It reads the executable binary, reconstructs the control-flow graph and computes ranges for the values of registers and memory cells by static analysis. This information is used to derive loop bounds and prune infeasible paths. Then the trace files are processed and the path of longest execution time is computed. The computed time estimate provides valuable feedback for assessing system safety and for optimizing worst-case performance. TimeWeaver also provides feedback for optimizing the trace coverage: paths for which infeasibility has been proven need no measurements; loops for which the analyzed worst-case iteration count has not been measured are reported. This is especially useful for software functions that employ different modes. If the static analysis can prove that the function is only executed in one of the modes, then it is enough to measure those parts of the software that belong to this mode.

2 Related Work

The problem of computing tight bounds of the execution time of a program is an active field of research, with many methods and tools using both static and dynamic analysis approaches [15]. Static analysis methods compute safe upper bounds of the execution time from a mathematical model of the target architecture. Dynamic analysis methods, on the other hand, derive the execution time from measurements performed on real hardware. Hybrid methods, like our approach, combine execution time information extracted from measurements with statically computable information like control flow graphs to improve safety, precision and/or coverage of the result. Probabilistic methods, finally, try to compute statistical models from measurements to compute upper bounds of the execution time.

The most basic version of measurement-based execution time analysis, namely end-to-end measurements, is still in frequent industrial use [12], but its problems are manifold. Not only it is unable to produce safe estimates, as in general not all possible scenarios can be measured, but the results are hard to interpret, too, as they are not related to particular parts of the code but only to the whole program. To overcome this, more structured approaches have been proposed, e.g. by Betts et al. in [4], which combine the measured execution times of small code snippets to form an execution time estimate for the whole program under analysis. Their use of software instrumentation leads to the probe effect, i.e., the timing behaviour of the program under observation changes due to the used observation technique. Moreover, their method does not account for typical cache behaviour, because it does not discriminate between different loop iterations. Hence, their method may be overly conservative. In a more recent publication [7], they use the non-intrusive tracing mechanisms of state-of-the-art debugging hardware. The main obstacle of their method is the limited size of trace buffers and/or the huge amount of trace data. According to their estimates, around half a terabyte of data would be generated in an hour of testing.

Bernat et al. [5, 6] obtain execution time profiles from measurements and combine them to compute a probabilistic timing estimate. Like [4], their tool pWCET[1] uses instrumentation.

[1] RapiTime is the commercial successor of pWCET.

Stattelmann et al. [13] propose the use of context information in order to account for cache effects. As an experimental platform they used the Infineon TriCore TC1797ED which allows automata-based trigger events to start/stop trace event recording. Stattelmann et al. implemented complex trigger specifications to reduce the required amount of trace data. In principle, their work shows that the inclusion of context information leads to more precise execution time results. However, the trigger event implementation in the measurement hardware is not precise enough to ensure valid fine-grained trace recording. This resulted in underestimated basic block timings.

Dreyer et al. [8] use the real-time tracing capability of modern SoCs like Xilinx Zynq to perform non-intrusive measurements. The core parts are implemented on a FPGA to directly process the raw trace data while the program is still running. The FPGA module computes execution time statistics like the minimal and maximal execution time for each edge in the waypoint graph. The presented work relies on the availability of a trace processing module for the underlying hardware architecture to interpret the raw trace data. Additionally, the timing results are only partially context-sensitive for loops, as only the first iteration is treated separately. Our approach, in contrast, processes the trace data offline, exploiting full context-sensitivity and hence, leading to more precise results. Like them, we use non-intrusive hardware tracing mechanisms of state-of-the-art multi-core processors (like NXP T1042, Zynq Ultrascale+ or TriCore AURIX) which try to minimize the amount of recorded data without loosing path information. This makes the offline processing feasible.

3 Hybrid WCET Analysis

Techniques to compute worst-case execution time information from measurements are either based on end-to-end measurements of tasks, or they construct a worst-case path from timing information obtained for a set of smaller code snippets in which the executable code of the task has been partitioned. With end-to-end timing measurements, timing information is only determined for one concrete input. As described above, due to caches and pipelines the timing behaviour of an instruction depends on the program path executed before. Therefore, usually no full test coverage can be achieved and there is no safe test end criterion. Approaches that instrument the code to obtain timing information about the code snippets of a task modify the code which can significantly change the cache and pipeline behaviour (probe effect): the times measured for the instrumented software do not correspond to the timing behaviour of the original software.

The solution which is implemented in the hybrid WCET analysis tool TimeWeaver [2] combines static context-sensitive path analysis with non-intrusive instruction-level real-time 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 multicore 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 computed estimates are safe upper bounds with respect to the given input traces, i.e., an overall upper timing bound is derived from the execution time observed in the given traces (for more details, see Section 3.3). Thus, the coverage of the input traces on the analyzed code is an important metric that influences the quality of the computed WCET estimates.

The required trace information is provided out-of-the-box by embedded trace units of modern processors, like NEXUS IEEE-ISTO 5001™ [9], Infineon TriCore™ MCDS, or ARM CoreSight™ [3]. They allow the fine-grained observation of a program execution on single-core and multicore systems. Examples for processors supporting the NEXUS trace interface are the NXP QorIQ P- and T-series processors (using either an e500mc or an e5500/e6500 core).

Listing 1 A sample NEXUS trace in ASCII format.

```
+056 TCODE=1D PT-IBHSM F-ADDR=F1F4 HIST=2 TS=8847
+064 TCODE=21 PT-PTCM EVCODE=A TS=88F1
+072 TCODE=1C PT-IBHM U-ADDR=03DC HIST=1 TS=8D62
+080 TCODE=21 PT-PTCM EVCODE=A TS=8E2F
+088 TCODE=21 PT-PTCM EVCODE=A TS=8FBA
+096 TCODE=21 PT-PTCM EVCODE=A TS=9105
+104 TCODE=1C PT-IBHM U-ADDR=02CC HIST=1 TS=9275
+112 TCODE=1C PT-IBHM U-ADDR=01F0 HIST=1 TS=93BF
+120 TCODE=21 PT-PTCM EVCODE=A TS=997B
+128 TCODE=1C PT-IBHM U-ADDR=0044 HIST=1 TS=9B02
+136 TCODE=21 PT-PTCM EVCODE=A TS=9F21
```

3.1 Example: NEXUS Traces

On the PowerPC architecture TimeWeaver relies on NEXUS program flow trace messages. Such traces consist of *trace segments* separated by *trace events*. The events are mapped to points in the control-flow graph (trace points) and the segments to program paths between these points. This is done for those parts of the trace that reach from the call of the routine used as analysis entry till the end of that routine or any other feasible end of execution. Such parts are called *trace snippets*. A single trace may contain several trace snippets. TimeWeaver can operate on one or more traces given as trace files, each containing one or more trace snippets.

A NEXUS trace event encodes its type, a time stamp containing the elapsed CPU cycles since the last trace event and the contents of the branch history buffer, which can be used to reconstruct execution path decisions and allows to map trace segments to the control-flow graph of the corresponding executable.

Microprocessor debugging solutions like the Lauterbach PowerDebug Pro [10] allow to record NEXUS trace events as they are emitted during program execution and to export them in various formats. For example, the following command in the Lauterbach Trace32 tool produces NEXUS traces in ASCII format:

```
Trace.export.ascii <file> nexus /showRecord
```

A sample exercpt is shown in Listing 1, with some information removed to improve readability. Those exports can be processed for timing analysis as described below.

Each line corresponds to a trace event. The number at the beginning of the line is the trace record number. The second and third column represent the particular trace event type followed by type-specific information like branch history and program address information associated with the event. The TS number at the end is a time stamp.

Debugging solutions differ in the format in which they export trace data. Some debuggers allow the user to configure the output. TimeWeaver can currently import traces which have been exported by Lauterbach, PLS or iSYSTEM debuggers. Whenever the format is configurable, we have identified a minimal set of information needed to perform the analysis. Additionally, the tool chain can be easily extended to support other trace formats.

3.2 TimeWeaver Toolchain

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). Optionally, users can specify further semantical information to the

analysis, like targets of computed calls, loop bounds, values of registers and memory cells. This information is used to fine-tune the analysis. The analysis proceeds in several stages: decoding, loop/value analysis, trace analysis, and path analysis. Most steps in this tool chain are shared with aiT [1], leveraging its powerful static analysis framework.

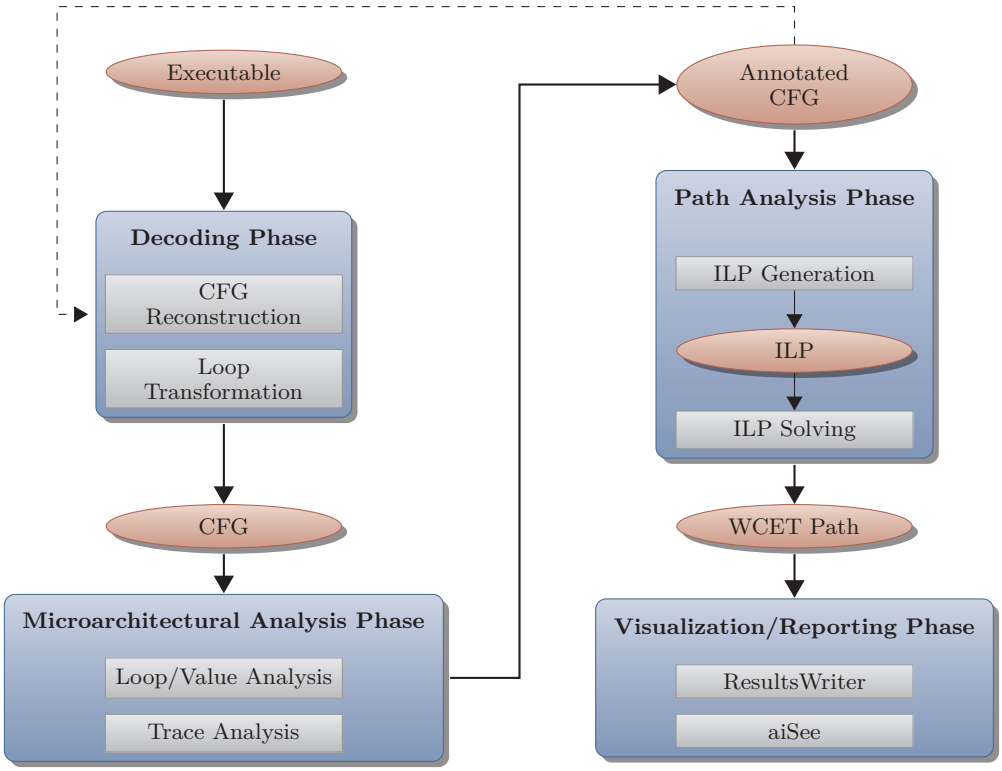

Figure 1 The structure of the TimeWeaver tool chain.

The decoding phase of TimeWeaver is mostly identical to the decoding phase of aiT. One important difference is that when encountering call targets which cannot be statically resolved, 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 (cf. Figure 1). As an alternative, the same user annotations can be used as in the aiT tool chain.

In the next phase, several microarchitectural analyses are performed on the reconstructed CFG starting with the combined loop and value analysis, again equal to the aiT tool chain. 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. The loop transformation allows loops in the CFG to be handled as self-recursive routines to improves analysis precision [13].

After value analysis, the analyzer has annotated each instruction in the control-flow graph with context-sensitive analysis results. Context-sensitivity in our analysis framework means to differentiate between different call stacks and between different loop iterations. The

length of the call string and the number of distinguished loop iterations can be configured by the user. This context-sensitivity is important because the precision of an analysis can be improved significantly if the execution environment is considered [13]. 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 segments mentioned above 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.

After the trace conversion, 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 ILP formulation is structurally the same as in the path analysis of aiT [14] with the exception that the involved execution times are not computed by a micro-architectural pipeline analysis but are extracted from the input traces. The generated ILP is fed to a solver whose solution is the worst-case execution path alongside its costs, i.e., the WCET estimate of the analyzed task. This solution is annotated to the CFG for the final step, namely reporting and visualization. Here, not only the global WCET estimate and the execution path triggering it are reported, but also detailed results per routine including the effective as well as the analyzed and trace loop bounds. Moreover, TimeWeaver reports the trace coverage as well as statistical information regarding the trace segments (minimal and maximal observed execution times, variance, distribution graphs, ...). This enables the user to reason about the quality of the measurements.

3.3 WCET Estimate Extrapolation

As mentioned above, a global WCET estimate is computed based on the observed execution times of trace segments. The times are maximized per trace segment and the maximized times are composed to identify the worst-case path with respect to those figures.

Where in general, one would need to measure all possible execution paths (which is impractical on real-world applications) of the analyzed program for coverage reasons, TimeWeaver allows to compute an upper bound on the global execution time of the analyzed program based on the trace segment times extracted from the input traces. The underlying assumption is based on the observation that it is quite hard to stimulate both the worst-case path for a whole program as well as the worst-case hardware state in which the execution starts [15]. However, it is much easier to observe the possible maximal execution times for short program snippets, in particular, if those snippets are measured many times. Additionally, the static path analysis allows to construct worst-case paths from the trace segments. In contrast to static timing analysis, where timing anomalies are possible due to the stateful analysis, the path analysis in our hybrid approach maximizes the execution time independent of the hardware state (which is not directly visible in the traces). Hence, timing anomalies are avoided, as local maximization cannot lead to a globally smaller estimate.

This way, it is only necessary to trace all possible execution paths between two consecutive trace points. By inserting custom trace points, the user can further decrease the required number of measurements. Figure 2 illustrates this by showing three consecutive trace points (TP1, TP2, and TP3) and the possible execution paths between each of them. The WCET estimate for the time between TP1 and TP3 is computed as the sum over the maximized trace segment time between TP1→TP2 and the maximized trace segment time between TP2→TP3. Thus, the measurements need to cover the four execution paths between TP1→TP2 as well as between three execution paths between TP2→TP3. Without that time composition, all 12 execution paths between TP1→TP3 need to be measured.

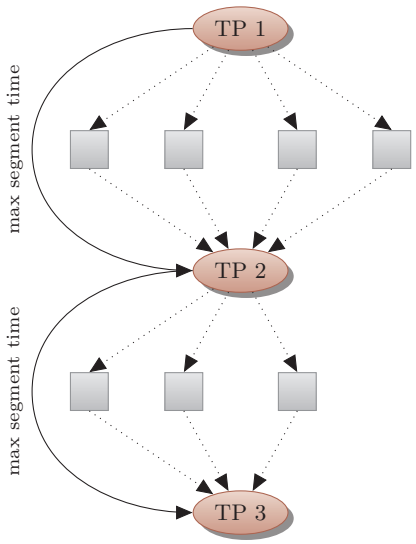

Figure 2 Execution paths between trace points.

3.4 Loop Scaling

For loops, there might be a gap between the maximum of the observed iteration counts in the input traces (traced bound) and the statically possible maximum iteration count (analyzed bound) which is computed by the value analysis. A typical example is the `memcpy` function. This function could contain a loop that iterates five times in one call context, four times for a different one, and seven times in yet another calling context. The value analysis might compute a loop bound of [0..7] iterations, while the traces only contain occurrences of the loop with 4 or 5 iterations. The bound actually used for the ILP generation – the so-called effective bound – is the analyzed bound if it is finite and applicable (cf. scaling conflicts below) and otherwise the traced bound. Per user request, the (interval) intersection of analyzed and traced bound is used. In the example above, the intersection result would be [4..5] iterations.

If the effective bound is higher than the traced bound, the maximum observed execution time (context-sensitively) for one loop iteration is scaled up to the effective bound. This overcomes the necessity to trace each loop in the analyzed task with its worst-case iteration count, which might be hard to achieve because loop conditions often are data-dependent and thus, can be complex to trigger.

However, loop scaling as described above is not always directly applicable. It requires each trace to pass a trace point inside the loop body. If there is at least one traced execution path through the loop body without a trace point, scaling cannot be done and only the

traced bounds are used for this loop. Such a situation is called an *event loop scaling conflict*. Event loop scaling conflicts usually happen for very short loops, both in the size of the loop body as well as in the number of iterations. Then, it may happen that the trace segments covering the loop start before the loop is entered and end after the loop has been left, because the embedded trace units often do not emit a trace event for every branch but only if the branch history buffer is full. Hence, no trace point lies inside the loop body, and no timing information can be extracted for one iteration of the loop alone. The solution is to either trace the worst-case loop iteration count or to ensure that each traced path through the loop body passes a trace point (by inserting custom trace points).

There is another situation which triggers a loop scaling conflict: if due to the context settings of the analysis a loop is virtually unrolled more times than the corresponding loop body has been executed in the trace, scaling cannot be applied, too. The reason is that the scaling is applied in the last loop context, i.e., in that context which represents the last loop iteration(s). In that case, there is no traced loop body time in the trace mapped to this context which prevents scaling. Such a conflict is called an *unroll loop scaling conflict*. Consider again the `memcpy` example from the beginning of this section. Assume now that the user decided to let the analysis differentiate between each of the first six loop iterations and all other loop iterations. The traces only cover executions where the loop iterates four or five times. Hence, no timing information can be extracted for the accumulative loop context that covers the seventh and all later iterations of the loop, and loop scaling fails. To solve this conflict, one can either trace the worst-case iteration count of the corresponding loop or the (virtual) loop unroll during analysis of this particular loop can be decreased to the traced bound.

3.5 Observing Interference

TimeWeaver can also be used to observe interference from tasks running on other cores or other asynchronous events. If enough measurements have been taken, all interferences that influence the timing behaviour will be visible. However, this also means that if one wants to exclude interferences from the results, one needs to use only traces as input that are representative for such a scenario. Some of these events can be excluded quite easily. For example, task switches and interrupts can be detected because they leave the precomputed control-flow graph. This is not the case for resource conflicts, e.g., on the shared bus. Here, it is unfortunately quite hard to identify whether the increased waiting time is caused by the intrinsic behaviour of the processor or by external interferences. The identification of different kinds of interferences in trace data is a topic for future research.

For some (rather regular) types of interference, a possibility is to filter out some trace segment times. For example, the input traces might contain asynchronous events like DRAM refreshes which can lead to exceptionally high trace segment times. TimeWeaver allows to address these with a filter for trace segment times based on their cumulative frequency, i.e., their occurrence percentage. The threshold refers to a percentage of occurrences ordered by execution times. A threshold of 0% is passed by all occurrences. A threshold of 5% is passed by all but the 4 most expensive ones (in terms of execution time) if there are 100 occurrences, by all but the 9 most expensive ones if there are 200 occurrences, etc. Trace segment times that do not pass the specified threshold are ignored in the ILP generation. The filter function is applied for each trace segment separately. TimeWeaver allows to simulate the effect of the cumulative frequency filter in a dedicated statistics view. This enables the user to experiment with different filter values.

4 Experimental Results on TimeWeaver

To evaluate TimeWeaver for PowerPC, we recorded program executions on an NXP T1040 [11] evaluation board using a Lauterbach PowerDebug Pro JTAG debugger. For each application, the maximum observed end-to-end time has been extracted from the traces and compared with the WCET estimate computed by TimeWeaver. In order to enable comparable results, the loop bounds in the analysis have been chosen equal to the traced loop bounds. This way, the increase in the estimate is due to the static path analysis (see Section 3.3). Otherwise, the results of TimeWeaver could easily exceed the measured end-to-end times solely by using loop bounds larger than the observed ones. Table 1 shows the results of this comparison. The difference represents the overestimation resulting from the composition of trace segment times to a global estimate. Some programs consist only of a single path (e.g., edn and nestedDepLoops) while other contain quite complex control flow (e.g., the avionics and automotive tasks). For the former, the difference between TimeWeaver result and maximally observed execution time is mostly due to the ability of the hybrid analysis to combine measurements of different hardware states, while for the latter, the static path analysis is able to construct scenarios for which no single trace has been recorded.

It would have been interesting to compare the results of TimeWeaver to those of aiT, but unfortunately, no abstract timing model of the NXP T1040 exist, which is also one of the main reasons for TimeWeaver to exist: to enable some kind of meaningful timing analysis for processors for which static WCET analysis is impossible due to unpredictable or undocumented hardware features. However, since the value and path analysis parts are shared between TimeWeaver and aiT, the only difference would stem from the fact whether (a) the abstract timing model is tight enough and (b) all possible hardware states have been observed during measurements. Hence, the results show the importance of the path analysis, as the hybrid analysis lifts the burden of stimulating the overall worst-case path from the user.

Table 1 Comparison of TimeWeaver results with maximum observed end-to-end times.

Application	Trace [cycles]	Estimate [cycles]	Diff [%]
crc	809068	829039	2.47
edn	4788025	4791420	0.07
eratosthenes sieve	368345	369803	0.40
dhrystone	168093	177314	5.49
md5	127857	131718	3.02
nestedDepLoops	2747357	2747359	0.00
sha	23426161	23815350	1.66
Avionics Task	420677	498028	18.38
Automotive Task 1	65058	71964	10.62
Automotive Task 2	27215	28967	6.44
Automotive Task 3	17386	18595	6.95
Automotive Task 4	101749	109302	7.42

5 Conclusion

Hybrid worst-case execution time analysis allows to obtain worst-case execution time bounds even for systems where the timing behaviour of the processor is not well-specified, or where asynchronous interferences can be neither be controlled nor bounded. We have given an

overview of the hybrid WCET analyzer TimeWeaver which combines static value and path analysis with timing measurements based on non-intrusive instruction-level real-time traces. The trace information covers interference effects, e.g., by accesses to shared resources from different cores, without being distorted by probe effects since no instrumentation code is needed. The analysis results include the computed WCET bound with the time-critical path, and information about the trace coverage obtained. They provide valuable feedback for optimizing trace coverage, for assessing system safety, and for optimizing worst-case performance. Experimental results show that with good trace coverage safe and precise WCET bounds can be efficiently computed.

References

1 AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. URL: http://absint.com/ait.

2 AbsInt Angewandte Informatik GmbH. TimeWeaver: Hybrid Worst-Case Timing Analysis. URL: http://absint.com/timeweaver.

3 ARM Ltd. CoreSight™ Program Flow Trace™ PFTv1.0 and PFTv1.1 Architecture Specification, 2011. ARM IHI 0035B.

4 Guillem Bernat and Adam Betts. Tree-Based WCET Analysis on Instrumentation Point Graphs. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 558–565, 2006. doi:10.1109/ISORC.2006.75.

5 Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, 2002.

6 Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET: A tool for probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. YCS-2003-353, Department of Computer Science, University of York, February 2003.

7 Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 54–63, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2010.54.

8 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 4:1–4:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2016.4.

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

10 Lauterbach GmbH. Lauterbach Website. URL: http://www.lauterbach.com.

11 NXP Semiconductors. *QorIQ™ T1040 Reference Manual*, 2015.

12 Karsten Schmidt, Denny Marx, Jens Harnisch, Albrecht Mayer, Udo Dannebaum, and Herbert Christlbauer. Non-Intrusive Tracing at First Instruction, 2015. SAE Technical Paper 2015-01-0176. doi:10.4271/2015-01-0176.

13 Stefan Stattelmann and Florian 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 *OpenAccess Series in Informatics (OASIcs)*, pages 64–76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010. doi:10.4230/OASIcs.WCET.2010.64.

14 Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis. Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis*. PhD thesis, Saarland University, 2003.

15 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008. doi:10.1145/1347375.1347389.