19th International Workshop on Worst-Case Execution Time Analysis

WCET 2019, July 9, 2019, Stuttgart, Germany

Edited by

Sebastian Altmeyer

Editors

Sebastian Altmeyer
University of Amsterdam, The Netherlands
altmeyer@uva.nl

ACM Classification 2012
Theory of computation → Program analysis; Computer systems organization → Real-time systems; Software and its engineering → Software performance; Software and its engineering → Software verification and validation

ISBN 978-3-95977-118-4

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-118-4.

Publication date
July, 2019

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.WCET.2019.0

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Regular Paper

19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019).
Editor: Sebastian Altmeyer
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

Welcome to the **19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)**. WCET 2019 is organized in conjunction with the Euromicro Conference on Real-Time Systems (ECRTS 2019) in Stuttgart, Germany, and is held one day prior to the conference. The WCET workshop is the main venue for research on worst-case execution time analysis in the broad sense and serves as a yearly meeting for the WCET community.

This year, the workshop features an invited keynote talk by Dr. Franz-Josef Grosch from Robert Bosch GmbH entitled *Blech - a synchronous language for embedded real-time programming* and 6 presentations of regular papers. Each of these 6 papers received 3 reviews from members of the program committee. The final selection was then based on an online discussion.

The WCET workshop is the result of the combined effort of many people. First, I like to thank the authors of the WCET 2019 papers, and the keynote speaker Franz-Josef Grosch, for contributing the scientific content of the workshop. I also thank the members of the program committee and the external reviewer for their high-quality reviews and fruitful online discussion. I thank the steering committee for their guidance and advice on the organization of this workshop. Special thanks go to Michael Wagner for the help in publishing the proceedings of WCET 2019.

The WCET exists to exchange ideas and on all WCET-related topics in a friendly atmosphere. I invite you to enjoy the presentations and to actively participate in the discussions!

Amsterdam, The Netherlands
May 26th, 2019
Sebastian Altmeyer

19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019).
Editor: Sebastian Altmeyer

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Committee

Program Chair

- Sebastian Altmeyer – University of Amsterdam, The Netherlands

Program Committee

- Clement Ballabriga – Lille 1 University, France
- Florian Brandner – Télécom ParisTech, Université Paris-Saclay, France
- Adam Betts – Rapita Systems, UK
- Heiko Falk – TU Hamburg, Germany
- Björn Lisper – Mälardalen University, Sweden
- Claire Maiza – Grenoble INP/Verimag, France
- Enrico Mezzetti – Barcelona Supercomputing Center, Spain
- Kartik Nagar – Purdue University, United States
- Isabelle Puaut – University of Rennes I/IRISA, France
- Jan Reineke – Saarland University, Germany
- Christine Rochange – IRIT, France
- Abhik Roychoudhury – National University of Singapore, Singapure
- Martin Schoeberl – Technical University of Denmark, Denmark
- Peter Wägemann – Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
- Simon Wegener – AbsInt Angewandte Informatik GmbH, Germany
- Jakob Zwirchmayr – TTTech Auto AG, Austria

External Reviewers

- Jean Malm – Mälardalen University, Sweden

Steering Committee

- Björn Lisper – Mälardalen University, Sweden
- Isabelle Puaut – University of Rennes I/IRISA, France
- Jan Reineke – Saarland University, Germany

19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019).
Editor: Sebastian Altmeyer

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

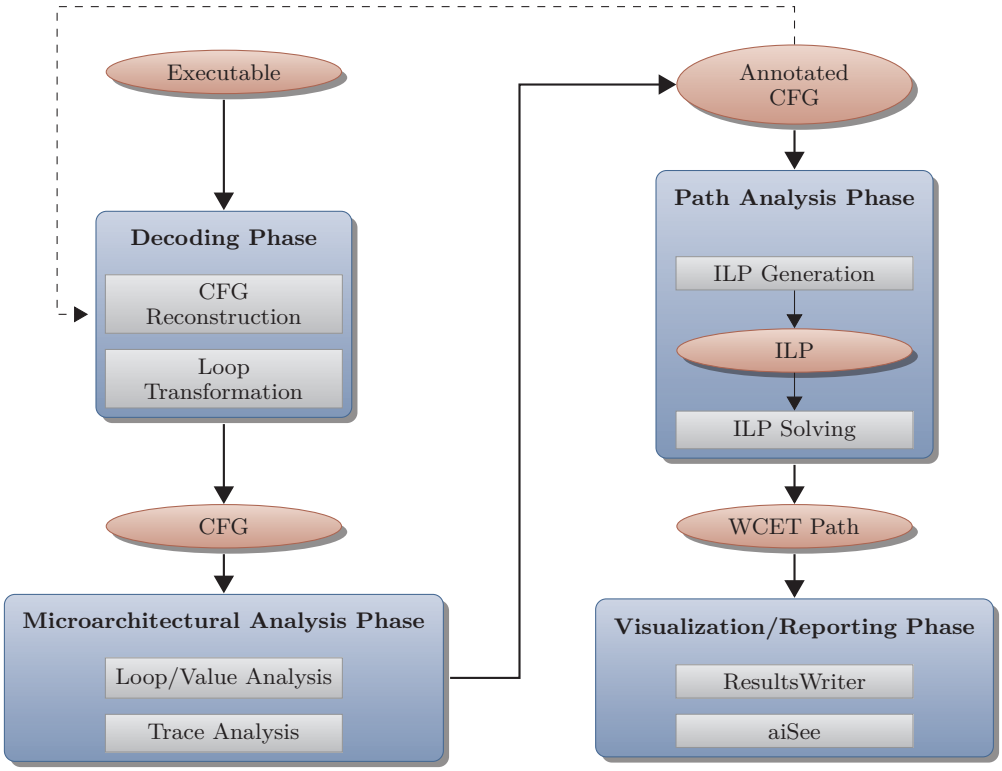


Figure 1 The structure of the TimeWeaver tool chain.

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.

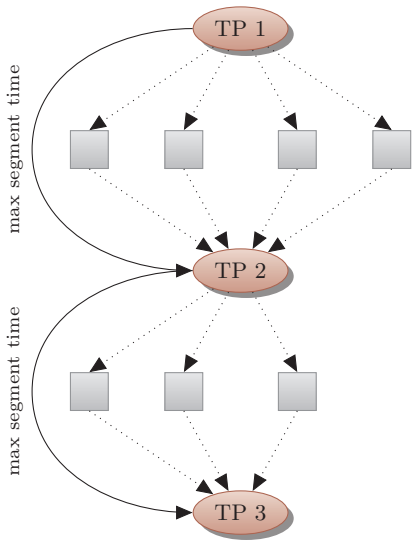


Figure 2 Execution paths between trace points.

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.

Non-Intrusive Online Timing Analysis of Large Embedded Applications

Boris Dreyer
Computer Systems Group, TU Darmstadt, Germany
dreyer@rs.tu-darmstadt.de

Christian Hochberger
Computer Systems Group, TU Darmstadt, Germany
hochberger@rs.tu-darmstadt.de

Abstract

A thorough understanding of the timing behavior of embedded systems software has become very important. With the advent of ever more complex embedded software e.g. in autonomous driving, the size of this software is growing at a fast pace. Execution time profiles (ETP) have proven to be a useful way to understand the timing behavior of embedded software. Collecting these ETPs was either limited to small applications or required multiple runs of the same software for calibration processes. In this contribution, we present a novel method for collecting ETPs in a single shot of the software at very high quality even for large applications.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded systems

Keywords and phrases WCET, Execution Time Profiling, ARM CoreSight, Event Stream Processing

Digital Object Identifier 10.4230/OASIcs.WCET.2019.2

1 Introduction and Background

1.1 Motivation

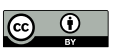
A thorough understanding of the execution time of embedded systems software is crucial for the development of reliable systems. Yet, an analytical approach for this understanding has become almost impossible with modern System-on-Chip (SoC) architectures. The overall execution time is dominated by unpredictable effects like caches with random replacement policy.

Also, this effect demands more understanding of the timing as just min and max times for functions or loops. Rather, it requires a full execution time profile (ETP), which shows the probability for all possible execution times.

Different ways exist to collect such an ETP. Although it is possible to instrument the code to create a statistics for all parts of the code, this way highly influences the execution time and thus comes with severe drawbacks. The best alternative is to use the trace data that is provided by modern SoCs [1]. Unfortunately, commercial tools currently only offer to record this trace data. Given the data rate and the complexity of typical systems, this is not feasible, as the amount of data can easily reach TB. The only truly viable alternative is online processing of trace data.

1.2 Online Processing of Trace Data

To the best of our knowledge, the CONIRAS platform [5] was the first that was able to entirely process the trace data produced by modern ARM processor cores. This is non trivial, as the trace information is highly compressed and packetized. Only in the beginning of a packet an absolute reference is given. The remaining packet uses only relative addressing. Online processing consists of: (a) Reconstruction of the program addresses from the trace



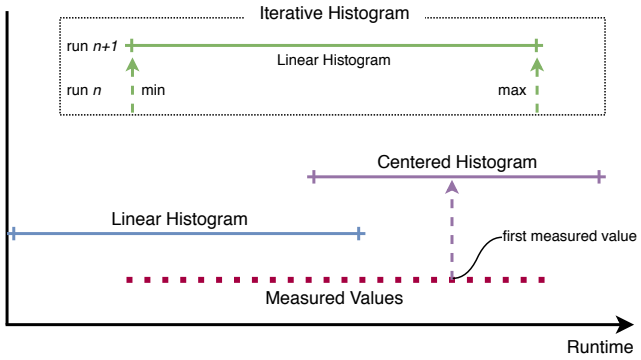


Figure 1 Schematic comparison of our previous histogram algorithms. The range of measured values are shown as red dotted line. The covered value range by a linear histogram is shown as blue line, by a centered histogram as purple line and by an iterative histogram as green line.

stream (b) Mapping these addresses to individual basic blocks of the application under test (AuT) and (c) aggregating the information in a suitable form. Timingwise, the CONIRAS project targeted only worst case execution time (WCET). Support for histograms was added later. We started with linear and centered histograms. Both were using a bin size that the user had to choose. Later, we devised an iterative approach, where the optimal bin size is evaluated by multiple runs of the embedded software.

Essentially, the CONIRAS platform was limited by the amount of memory that was available on the underlying FPGA. All address lookups and all statistics were using BRAMs of the FPGA (even if it was not necessary).

1.3 Related Work

Apart from ETPs, hardware-implemented histogram are often used in image and video processing. Gautam presents a parallel histogram calculation architechture for image processing [7]. A hardware efficient, simplified Histogram of Orientation Gradient (HoG) module in Scale Invariant Feature Transform (SIFT) for describing key-point detected using Gaussian Scale Space (GSS) is proposed in [11]. Maggiani et al. propose in [10] an optimized design of a histogram extractor algorithm targeted to low-complexity and low-cost FPGA-based Smart Cameras. [14] uses them for Adaptive Histogram Equalization. In [8] and [9], they are part of the Histogram of Oriented Gradients algorithm for object detection. [15] presents a sliced Integral Histogram algorithm for efficient histogram computation. [13] uses Local Binary Patterns Histogram for face recognition.

In contrast to these related works, the distribution of function runtimes is highly dynamic. Therefore, we could not use the known methods for constructing histograms in hardware and had to develop new histogram algorithms [2, 3].

The remainder of this paper is structured as follows: Section 2 explains the limitations of the CONIRAS platform and analyzes the causes. Our novel approach for collecting ETPs is presented in Section 3. The overall platform and the tool flow are presented in Section 4. Finally, in Section 5 a conclusion and an outlook are given.

2 Limitations of Previous Hardware-optimized Histogram Algorithms

In this section we want to recapitulate existing hardware-optimized histogram algorithms and explain why these algorithms are not adequate for analyzing large embedded applications. In [2] and [3] we presented a linear, a centered and an iterative histogram algorithm. The schematic bin distributions of these histograms are shown in Figure 1.

Linear Histogram Algorithm. Linear histograms span a range from 0 to an upper bound with equally distributed bins over the full range. The size of the last bin is unlimited. In this way, the histogram uses its last bin to count the amount of values that were outside of its range. This information is used to judge the quality of the histogram. The linear algorithm was developed for the runtime measurement of Waypoint Edge Events (WPEs), as they usually take only a few cycles.

The blue lines of Figure 2 show the coverage of the debie1 task runtimes with linear histograms. It can be seen that a linear distribution with starting point 0 is only suitable for the measurement of functions with short runtimes.

Centered Histogram Algorithm. The centered bin distribution has been developed by assuming that the expected runtimes are clustered around a reference value. To obtain a more detailed view of the measured values, the bins are distributed around this reference value [2]. In contrast to the linear histogram, the first and last bin are used to store the number of values that are outside the distribution. This algorithm requires more memory than the linear algorithm because the reference value of each histogram must be stored during the analysis of the AuT.

The purple lines of Figure 2 show the coverage of the debie1 task runtimes with centered histograms. The histograms have been configured to use the first measured value as reference value and that the reference value is in the upper quarter of the total distribution. For functions with longer runtimes, this distribution is better suited than the linear distribution. Nevertheless, the coverage is not satisfactory to create good ETPs.

Iterative Histogram Algorithm. The iterative algorithm was developed for the runtime measurement of functions. Unlike the execution time of WPEs, the runtime of functions can have high dynamics. For example, if sensors supply input data and the runtime of a function depends on these inputs. The algorithm consists of multiple runs. In each run, a linear histogram is constructed and the minimum and maximum measured values are stored. The lower and upper bound of these histograms are derived from the measured minimum and maximum values of the previous runs. The bounds for the first run are defined by the user. The algorithm terminates when the amount of entries in the smallest and largest bin are below a user defined threshold. It is important to know that adjusting the minimum and maximum boundary of a linear histogram requires to re-synthesize the histogram hardware and this takes several minutes.

Therefore, this iterative approach is unsuitable for large embedded applications. It may take a long time before the function to be analyzed is executed. If the function has high dynamics in its runtime, several iterations are necessary to get good histograms which increases the time drastically.

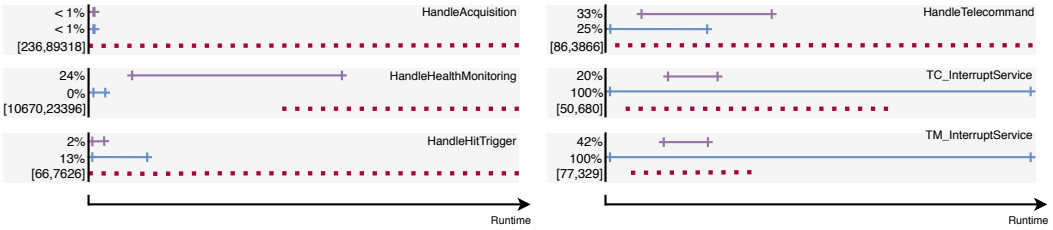


Figure 2 Measured debie1 task execution cycles (red dotted lines) covered by linear (blue lines) and centered (purple lines) histograms with 128 bins of size 8. Numbers for linear and centered distribution are taken from [2]. The iterative algorithm covers the complete range of measured values after the second run and is therefore not drawn as evaluated in [3].

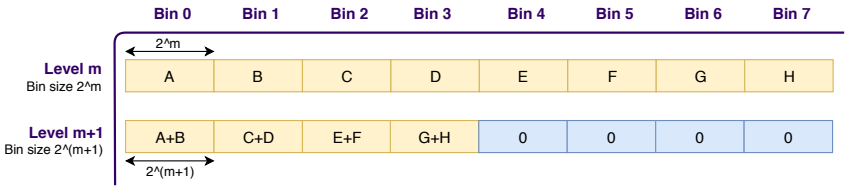


Figure 3 Visualization of one compression step of the scalable histogram algorithm. Two adjacent bins are summed up and the remaining bins are set to zero.

3 Scalable Histogram Algorithm

For the calculation of meaningful ETPs, the underlying histograms must be as detailed as possible. This means that the histograms should have a small bin size and the histogram should be able to assign each measured value to a bin. The runtimes of functions, tasks and loops can differ from execution to execution. This may be caused by data dependencies or cache effects. It is not uncommon for the runtime of a function to depend on the function's parameters. To compute histograms with high event frequency in hardware, we fixed the number of bins of a histogram during the hardware synthesis, see Section 4.

Therefore, the idea of the scalable histogram algorithm is to increase the size of all histogram bins dynamically during the analysis of an AuT. This allows the histogram to process a wider range of measured values if necessary. Each histogram has an initial bin size of 1. Whenever the histogram is not capable to store a measured value because this value is too large to be stored in the highest bin the sizes of all bins are doubled. During this duplication, two adjacent bins are merged, i.e. their values are added and the remaining bins are set to zero, see Figure 3. This phase is called a compression and is repeated until the measured value can be stored in a bin. The number of times a histogram has been compressed is called the histogram's compression level and is stored together with each histogram. This level is later used by the timing analysis software to recalculate the border of each bin.

Example. In this section we want to demonstrate the algorithm with an example. This example uses a scalable histogram with 8 bins to process the event sequence <5,4,11,7,54,10>. Figure 4 shows how this sequence is processed in steps (a) to (j). These steps are now explained.

At the beginning of this example the histogram is in its initial state, shown in step (a). This means that all bins are cleared and the histogram's compression level is 0. Therefore, each bin has a size of 1 and the histogram can process values between 0 and 7 without increasing its compression level, i.e. the need of one or more compressions.

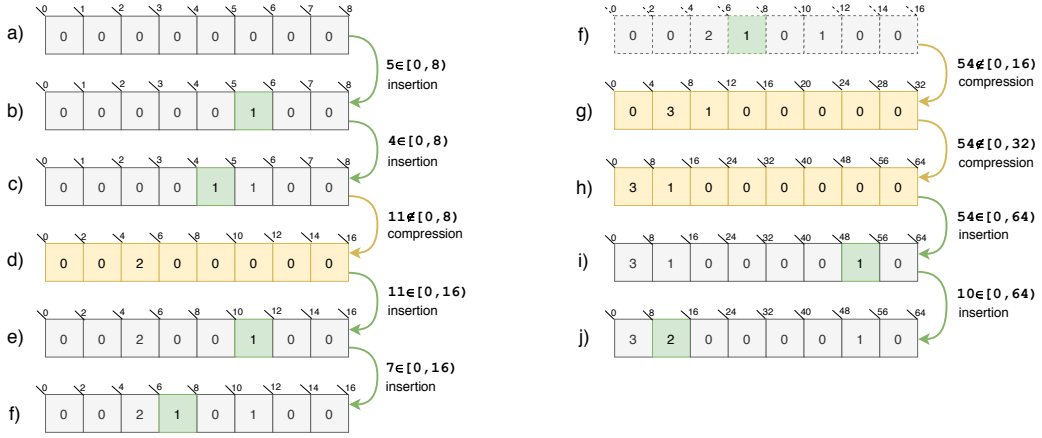


Figure 4 Scalable histogram algorithm example showing an 8 bin histogram processing the event sequence <5,4,11,7,54,10> step by step.

The first sequence value is 5 and because it is within the current range of the histogram the corresponding bin is updated (b). The next sequence value is 4 and is also within the current range of the histogram so the corresponding bin can be updated too (c). The following value is 11 and is not within the range of the histogram. Before this value can be processed, the histogram must be compressed to double its range (d). The compressed histogram can process values between 0 to 15 and is able to process the value 11. The updated bin is shown in step (e). The next sequence value is 7 and can be processed without further compressions (f). Then value 54 is processed, which requires the two consecutive compressions depicted as steps (g) and (h) before the corresponding bin can be updated in step (i). The first compression extends the range of the histogram to process values between 0 and 31 and the second compression to process values between 0 and 63. The final value 10 can be processed without any compressing.

Evaluation. In this evaluation we compare ETPs constructed from scalable histograms with 64 and 128 bins with perfect ETPs. Perfect ETPs are the most detailed ETPs, i.e. they are constructed from histograms with bin sizes of 1 which could not be realized in hardware. Since the number of histogram bins now depends on the measured values, we constructed these perfect ETPs through simulation.

Figure 5 overlays these ETPs for all debie1 tasks and Interrupt Service Routines (ISRs). The debie1 benchmark was chosen to ensure comparability with the evaluation results of [2] and [3]. It can be seen that we can construct ETPs with a good precision/memory trade-off for each debie1 task and ISR with our new algorithm. Unlike our previous algorithms, this construction was done in one analyzing pass and without any previous runtime information about the tasks to be analyzed.

4 Scalable Histogram Platform

In this section we want to introduce a timing analysis platform that calculates scalable histograms and simple statistics in hardware. These calculations are done in realtime during the analysis of an AuT. This platform is an extension of the CONIRAS platform which had the original purpose to compute the WCET of an AuT.

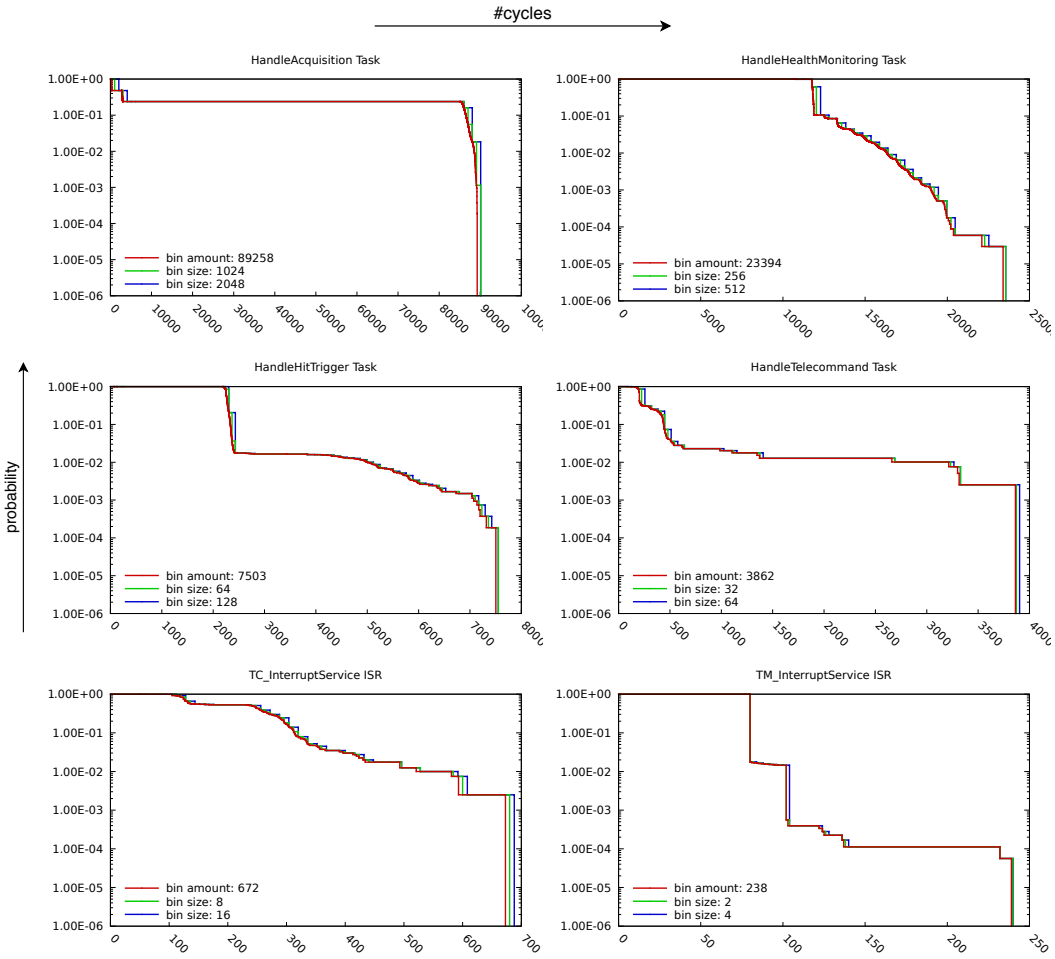


Figure 5 Execution cycles ETPs of all debie1 tasks and ISRs generated from 70,000,000 WPE. Red lines are simulated perfect ETPs i.e. the underlying histograms have a bin size of 1; green lines are ETPs from scalable histograms with 128 bins, blue lines are ETPs from scalable histograms with 64 bins.

4.1 Workflow

Figure 6 shows the workflow to compute timing statistics of embedded applications online in a non-intrusive way. This workflow only requires the application as binary file.

Preprocessing

The first step to analyse an AuT is to reconstruct its Waypoint Graph (WPG) [4] with tools like aiT [6] and Angr [12].

After that, the *Timing Analysis Module Generator* generates the *Timing Analysis Module* that measures the function runtimes and loop iterations of the AuT and also calculates and stores function runtimes and loop iterations statistics in hardware. In order to let the user specify which functions and loops of an AuT should be analyzed and which statistics should be generated we developed the *Statistics and Logging Language* (SALL). SALL is not in focus of the paper and will therefore not be discussed further.

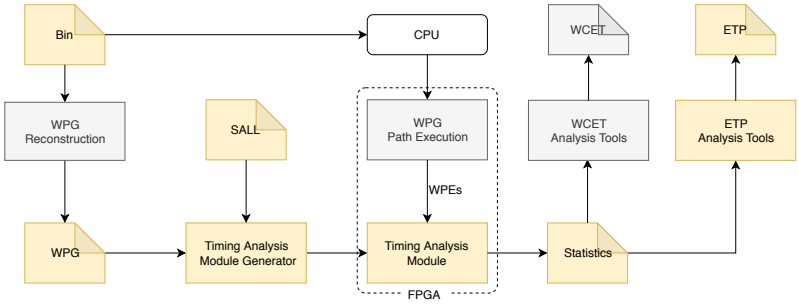


Figure 6 Workflow of our timing analysis platform for analyzing large embedded programs. The *Timing Analysis Module* has been developed for this purpose. Gray parts could be reused from our CONIRAS platform and its extensions.

Online Analysis

The *Timing Analysis Module* uses Waypoint Edge Events (WPE) emitted by the trace-based *WPG Path Execution Module* to calculate and store scalable histograms for function runtimes and loop iterations. A WPE consists of the executed edge of the WPG, i.e. an edge ID and the amount of cycles since the previous WPE was emitted.

Instead of histograms the module can also calculate and store simple statistics. Simple statistics means to compute and store the minimum, maximum and total runtime of a function and the minimum, maximum and total iterations of a loop. It also stores how often a function and loop was executed. Furthermore, it can also calculate and store simple statistics of low-level context-sensitive WPE. Context-sensitive WPE statistics means to compute and store the minimum, maximum and total runtime of an event that were executed during the first iteration of its innermost surrounding loop separated from the measured executions of that event during further iterations of its innermost surrounding loop. It also context-sensitively stores how often that WPE was executed.

Postprocessing

After the analysis is done the *Timing Analysis Module* transfers the generated statistics to the host computer. This computer can than be used to perform high level analysis like ETP or WCET calculation.

4.2 Timing Analysis Module

This module consists of a loop and function automata cluster, forwarding trees, storage controllers and statistics storages. Each automaton generates context-sensitive WPE runtimes, loop and function runtimes, and loop iterations amount data. The forwarding and poll trees are used to serialize these data. The poll tree uses a round robin mechanism to poll information out of the automata's output buffer, depicted in Figure 7. This buffer mechanism is necessary because multiple automata can emit loop and function events at the same time. Whereas a simple forwarding is sufficient to forward the context-sensitive WPE runtimes, because this information is only emitted by the innermost function or loop. The *Storage Controller* uses the statistics layout information of the automata cluster together with their emitted values to generate control signals for the *Statistics Storage Module* to calculate simple statistics and histograms.

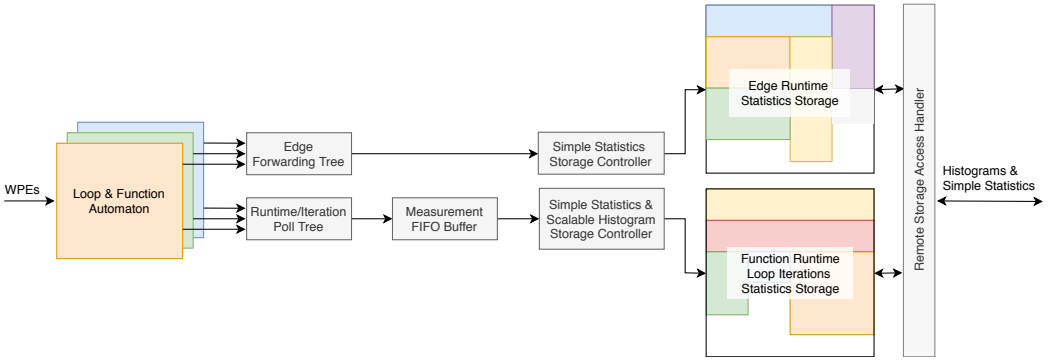


Figure 7 Structure of the *Timing Analysis Module* consisting of a loop and function automata cluster, forwarding trees, storage controllers and statistics storages.

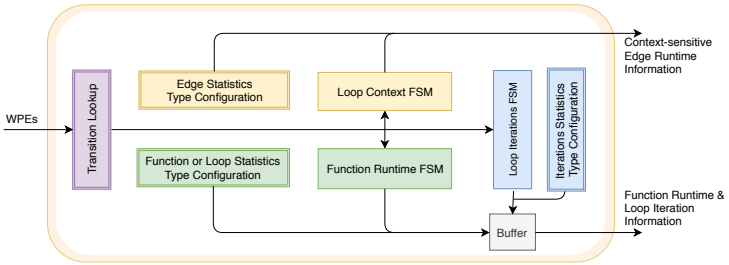


Figure 8 Structure of one *Loop & Function Automaton Module*.

Loop and Function Automaton Module

This module is able to analyze one loop or one function of an AuT and its structure is shown in Figure 8. It processes WPEs which are generated by the *WPG Path Execution Module* by interpreting low level trace packets like atom packets, timestamp packets, branch address packets and waypoint update packets for an ARM CoreSight [1] trace.

The green part of Figure 8 calculates the runtime of a function or loop. The blue part counts the iterations of a loop. The yellow part determines if this automaton represents the currently innermost executed loop or function. It also determines the current context of the executed WPEs. All these parts use Finite-State Machines (FSMs) to manage their computations. The transition events of these FSMs are generated by the purple part. It uses a rewritable lookup to map WPEs to transition events. These lookups are realized with distributed RAM for small functions and BRAM for large functions. By altering the lookup content through meta-configuration at runtime the automaton can be used to analyze another loop or function.

Each automaton also stores information about which statistics should be generated from its measurements and where they should be stored in *Statistics Storage Module*. This information was defined during the preprocessing phase by the *Timing Analysis Module Generator* and is used by the *Storage Controller Module* to calculate the memory addresses for updating the correct statistics.

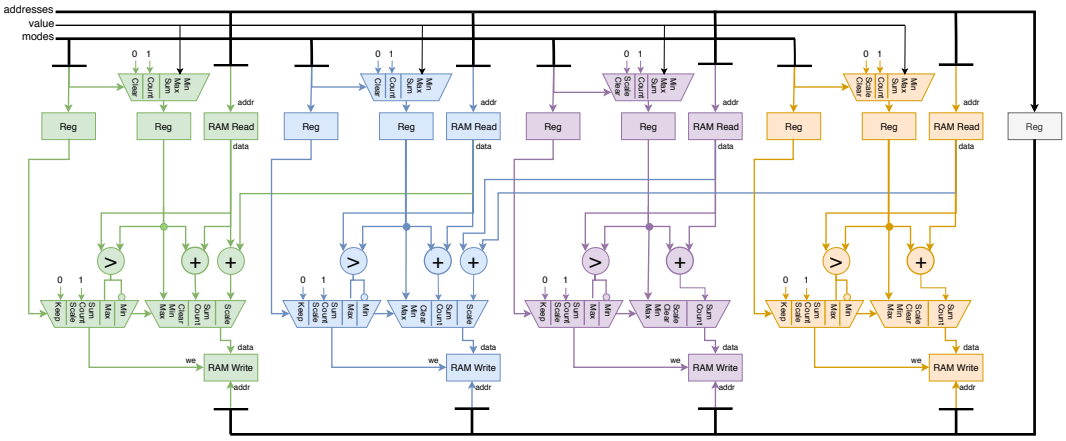


Figure 9 Timing critical part of one *Statistics Module*. It computes and stores histograms with four bins or simple aggregation function, i.e. min, max, sum and count. Forwarding paths to update the same memory location several times in a row are not shown for clarity.

Statistics Storage

Figure 9 gives a detailed insight of the timing critical datapath core of a *Statistics Storage Module* with four statistics cells. Each cell is colored differently for a better overview. The four cells can be used to calculate histograms with up to four bins or simple statistics. Each cell contains a dual ported RAM to store its statistics values and is connected to a *Storage Controller* that drives the individual address, mode and value signals. The mode signal determines the statistics operation which the cell applies to the value signal and the corresponding stored value. The address signal determines which stored value is updated.

Each cell is able to calculate simple min, max, sum and count statistics. The cells can also work together to calculate scalable histograms with four or two bins. If the first two cells work together to calculate histograms with two bins, the remaining cells can be used for simple statistics. This generic cell concept can be scaled to compute histograms with way more bin.

Statistics Controller

The *Simple Statistics and Scalable Histogram Statistics Controller Module* generates the control signals for the *Statistics Storage Module* to update histograms and simple statistics. To this end, the module implements the scalable histogram algorithm and stores the compression level of each histogram.

From the automata cluster it receives the measured function runtimes and loop iterations and the statistics type to use. The statistics type defines whether simple statistics or a histogram should be created and where in the memory of the *Statistics Storage Module* these statistics should be stored.

4.3 Measurement FIFO Buffer Evaluation

As long as no histogram compression is necessary, the *Statistics Controller Module* can process one event every clock cycle. One histogram compression requires one additional cycle. Because the cluster can generate an event every cycle, a first-in-first-out (FIFO) buffer was placed between the cluster and the Statistics Controller.

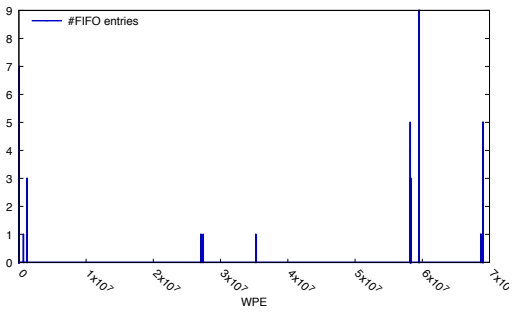


Figure 10 Measurement FIFO buffer utilization during the complete debie1 benchmark analysis.

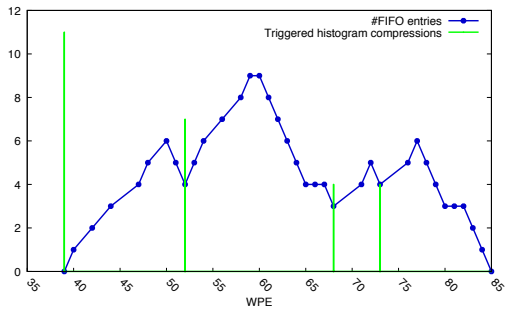


Figure 11 Measurement FIFO buffer utilization between processing WPE 59,529,035 and 59,529,085.

It is important that this buffer does not overflow during the analysis of an AuT, as this falsifies the analysis results. It is also important that this buffer can be implemented in hardware, i.e. it does not require unrealistic amounts of RAM. Therefore, we simulated the utilization of this buffer during the analysis of the debie1 benchmarks with prerecorded WPEs from [3]. This means that our target SoC to record the WPEs was a Xilinx Zynq XC7Z020 featuring a dual-core ARM Cortex-A9 running at 667 MHz. We compiled the benchmark with the C++ compiler `GNU C/C++ 4.9.2 20140904 (prerelease)` to ensure comparability with the evaluation results of [2] and [3]. The benchmark was executed on one core, while the other core executed a program that was used to generate interferences on the shared L2 cache and the shared interconnects.

Figure 10 shows the FIFO utilization during the analysis of the complete benchmark. To this end, almost 70,000,000 prerecorded WPEs were processed and 240 scalable histograms with 64 bins were constructed to be able to analyze all 68 loops and 172 functions of the benchmark. It can be seen that the FIFO is almost always empty and only buffers a maximum of 9 events at the same time.

This is shown in detail in Figure 11. The blue line shows the amount of buffered events for each processed event. The green lines show whenever a processed event triggers one or more compression steps of a histogram. The height of these green lines show the amount of compression steps that are necessary to process this event.

This evaluation has shown that the buffer can be implemented in hardware and requires only a few resources for the debie1 benchmark.

5 Conclusion and Future Work

In this contribution, we have presented a novel approach to collect histograms of execution times for large embedded applications. No instrumentation of the software is needed and thus, the timing behavior of the embedded system is not influenced. The histograms are gathered in a single shot of the AuT, yet they are so precise that we can compute very good ETPs from them. We have presented a measurement platform that can be tailored to the needs of the user. Also, we have analyzed the critical elements of the platform with respect to their HW implementation.

In the future, we want to use our domain specific language to automatize the customization of the measurement platform. This will enable us to automatically realize lookup functions in the best suitable type of memory. Since we will need less BRAMs for the lookup, more histograms can be gathered at the same time then.

References

1 ARM Ltd. CoreSight™ Architecture Specification v2.0, 2013. ARM IHI 0029B.

2 T. Ballenthin, B. Dreyer, C. Hochberger, and S. Wegener. Hardware Support for Histogram-Based Performance Analysis of Embedded Systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, 2017.

3 B. Dreyer, C. Hochberger, T. Ballenthin, and S. Wegener. Iterative Histogram-based Performance Analysis of Embedded Systems. *IEEE Embedded Systems Letters*, 2018.

4 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.

5 Boris Dreyer, Christian Hochberger, Simon Wegener, and Alexander Weiss. Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015.

6 Christian Ferdinand and Reinhold Heckmann. aiT: Worst-case execution time prediction by static programm analysis. In René Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions, 22–27 August 2004, Toulouse, France*, pages 377–384. Kluwer, 2004.

7 K. S. Gautam. Parallel Histogram Calculation for FPGA: Histogram Calculation. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, pages 774–777, February 2016. doi:10.1109/IACC.2016.148.

8 M. E. Ilas. New Histogram Computation Adapted for FPGA Implementation of HOG Algorithm: For Car Detection Applications. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, 2017.

9 C. Kelly, F. M. Siddiqui, B. Bardak, and R. Woods. Histogram of Oriented Gradients Front End Processing: An FPGA based Processor Approach. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.

10 L. Maggiani, C. Salvadori, M. Petracca, P. Pagano, and R. Saletti. Reconfigurable architecture for computing histograms in real-time tailored to FPGA-based smart camera. In *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pages 1042–1046, June 2014. doi:10.1109/ISIE.2014.6864756.

11 E. P. R. Raj, B. S. Paul, and G. L. Narayanan. Simplified SIFT Histogram of Oriented Gradients Bin Locator on FPGA. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, July 2018. doi:10.1109/ICCCNT.2018.8493928.

12 Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

13 N. Stekas and D. v. d. Heuvel. Face Recognition Using Local Binary Patterns Histograms (LBPH) on an FPGA-Based System on Chip (SoC). In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

14 U. K. Urimi, M. R. Kongara, and C. R. Patil. Real-time Implementation of Modified Adaptive Histogram Equalization for High Dynamic Range Infrared Images in FPGA. In *2015 Fifth National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG)*, 2015.

15 Yang Yang, Yun-Xia Liu, and Qi-Fan Dong. Sliced Integral Histogram: An Efficient Histogram Computing Algorithm and its FPGA Implementation. *Multimedia Tools and Applications*, 2017.

ePAPI: Performance Application Programming Interface for Embedded Platforms

Jeremy Giesen
Universitat Politècnica de Catalunya, Spain
Barcelona Supercomputing Center, Spain

Enrico Mezzetti
Barcelona Supercomputing Center, Spain

Jaume Abella
Barcelona Supercomputing Center, Spain

Enrique Fernández
Universidad de Las Palmas de Gran Canaria, Spain

Francisco J. Cazorla
Barcelona Supercomputing Center, Spain

Abstract

Performance Monitoring Counters (PMCs) have been traditionally used in the mainstream computing domain to perform debugging and optimization of software performance. PMCs are increasingly considered in embedded time-critical domains to collect in-depth information, e.g. cache misses and memory accesses, of software execution time on complex multicore platforms. In main-stream platforms, standardized specifications and applications like the Performance Application Programming Interface (*PAPI*) and `perf` have been proposed to deal with variable PMC support across platforms, by providing a shared interface for configuring and collecting traceable events. However, no equivalent solution exists for embedded critical processors for which the user is required to deal with low-level, platform-specific, and error-prone manipulation of PMC registers. In this paper, we address the need for a standardized PMC interface in the embedded domain, especially in view to support timing characterization of embedded platforms. We assess the compatibility of the PAPI interface with the PMC support available on the AURIX TC297, a reference automotive platform, and we implement and validate ePAPI, the first functionally-equivalent and low-overhead implementation of PAPI for the considered embedded platform.

2012 ACM Subject Classification Computer systems organization → Embedded software; Computer systems organization → Real-time systems

Keywords and phrases Monitoring counters, embedded systems

Digital Object Identifier 10.4230/OASIcs.WCET.2019.3

Funding This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), the European Union's Regional Development Fund (ERDF) within the framework of the ERDF (FEDER) program in Catalonia 2014-2020 under the grant SDESI (2016 PROD00115), and the HiPEAC Network of Excellence. Jaume Abella and Enrico Mezzetti have been partially supported by MINECO under Ramon y Cajal and Juan de la Cierva-Incorporación postdoctoral fellowships number RYC-2013-14717 and IJCI-2016-27396 respectively.

1 Introduction

Embedded critical systems must undergo a strict verification and validation (V&V) process to guarantee that the deployed system behaves correctly, both from the functional and non-functional perspectives. For time-critical systems, correctness also depends on the timely delivery of the results. Performance and economic considerations are pushing towards the adoption of complex, heterogeneous high-performance Commercial Off-The-Shelf (COTS) systems even in the most conservative embedded system domains. As a matter of fact, the computational requirements of increasingly advanced software functionalities in automotive [4] and avionics [3] systems, just to mention a few, can only be met with the use of multicore COTS platforms featuring multiple levels of caches and specialized hardware accelerators. However, the architectural complexity of such systems is hindering the effectiveness of consolidated WCET analysis approaches [2].

The problem emanates from the fact that, while several designs [8, 9, 17, 19, 22] have been proposed to better factor in multicore contention in task's WCET estimates, for cost reasons those solutions have not been fully adopted by chip providers yet. In particular, industry is reluctant to re-design and re-verify already-verified functional unit blocks (FUBs).

Software solutions have been proposed to handle multicore contention in COTS processors that have limited hardware support for time predictability [10, 20, 15, 6]. As a first step, these solutions use event monitors to track tasks generated activities (events). Events are mapped to Performance Monitoring Counters (PMC), a set of specialized, software-visible configurable registers. As a second step, these approaches build on limiting per task (core) maximum shared resources utilization. To that end, usually the operating system or the hypervisor monitors task's activities using the available PMCs and suspends or restrains tasks' execution when the assigned budget is exhausted. Tracked and bounded events to control contention among tasks include per-task (and possibly per-type) access counts to different shared resources like caches and memories. Overall, PMCs and Performance Monitoring Units (PMUs) in general, are instrumental to timing analysis on complex COTS high-performance hardware platforms. In particular, PMUs can be exploited to provide finer-grained metrics to support timing analyses by, for example, capturing several aspects of the execution, such as suffered contention and maximum latencies, that can be used to refine static analysis assumptions and to support measurement-based timing analysis [14].

PMC support varies greatly across hardware platforms and even across models of the same platform. This diversity complicates the definition of a structured approach for the use of PMCs to support platform analysis. A structural and reusable approach for configuring and reading PMCs is necessary to abstract away from the low-level hardware details and to guarantee a correct manipulation of the registers, especially with respect to PMU configuration. In this respect, a standardization effort has led to the definition of kernel-level tools, like the Linux set of utilities `perf` [1], or shared common libraries for configuring and using monitoring counters, like the Performance Application Programming Interface (PAPI) [13]. Whereas these tools have reached an exceptional diffusion in mainstream and high-performance platforms, no equivalent solution is currently available for embedded reference platforms and RTOSs.

In this paper, we take a first step towards filling this gap by addressing the implementation of a common abstract PMC library for use in the embedded domain to enable the collection of relevant hardware events in a platform-independent way. To that end, we consider the PAPI library and evaluate the extent at which it could be used for fine-grained platform analysis, especially with respect to timing characterization. After assessing the compatibility

of PAPI with the PMC support available on the Infineon AURIX™ TC297 [11] platform, a reference platform in the automotive domain, we define, implement and validate ePAPI, a functionally-equivalent, low-overhead port of PAPI to the referred platform.

The remainder of this paper is organized as follows: Section 2 provides some background on PAPI and the use of PMCs in embedded critical domains; Section 3 discusses the design choices behind the implementation of PAPI on the AURIX™ TC297; Section 4 describes our validation approach and provides empirical evidence on the correctness and efficiency of the port. Finally, Section 5 provides some conclusions.

2 Background

PMCs in the embedded domain. The need for performance counters originally developed within the scope of mainstream processors, as a means for hardware and software developers to perform low-level debugging. PMCs have been later used also for coarse-grained performance optimization. It is only recently, however, that PMCs have been increasingly considered in embedded systems to support timing [10, 20, 15, 6], thermal [12], and power [21] analyses. With respect to timing analysis, the main scope of this paper, PMCs have been especially considered to analyze the contention effects in multicore COTS platforms, either to build analytical models [10, 5, 6] or to monitor and enforce access or usage quotas [20, 15] on shared hardware resources. All these approaches, however, rely on ad-hoc, platform and RTOS/hypervisor specific low-level functions to configure and collect information from PMCs. In this paper, we support these same methods by showing that a generic, reusable, and validated library can be defined that allows collecting PMC information without the need to enter into the platform-specific details.

Performance Application Programming Interface. PAPI [13], is a cross-platform library with supporting utilities that represents the de-facto standard for the collection of *hardware events* on mainstream hardware devices, including heterogeneous and virtual platforms. A hardware event is capable of detecting the occurrence of a specific activity generated by the running software. Events can be tracked using specific registers called *performance monitoring counters* (PMCs). To that end event monitors – that are not visible from the system/user software – are mapped to software-visible PMCs via some PMC-related control registers. Interestingly, while the number of available PMCs is normally limited to few dozens at most, the number of traceable events depends on the specific platform support and can be in the order of hundreds or even thousands [16]. This requires several runs to capture the desired event monitors, with few event monitors read per run.

PAPI aims to provide a uniform environment across platforms and provides a unified interface across processors. This is achieved by resorting to a two-layer architecture:

- a *portable layer* implements the API and machine-independent functions,
- whereas a *machine-specific layer* defines machine-dependent (and operating system dependent) functions and data structures, using kernel extensions, operating system calls, or assembly language to access and configure monitoring counters.

In practice, however, different events may be supported, indirectly supported, or even unsupported depending on the processor in use. This apparent inconsistency is solved in PAPI by supporting two types of hardware events:

- *preset events*, also known as predefined events, are a common set of events deemed relevant and useful for performance characterization. These events are typically found in CPUs with debug support units and give access to several hardware events related to memory hierarchy, cache coherence protocol, cycle and instruction counts, functional units, pipeline status, etc.

- Some events, however, are only available on specific platforms. The *native events* set identifies the full set of events that can be tracked on a given CPU. Native events can be configured and traced directly, even if there is no corresponding preset event available.

The interface is implemented as an instrumentation library in C and needs to be compiled/linked together with the target application we want to characterize. The interface actually supports two different usage modes, which can be used side by side since they share the internal structures on which PAPI is built. A high-level API provides the basic ability to start, stop, and read the counters for a specified list of events. A low-level API, instead, manages hardware events in user-defined groups called *event sets* (preset or native) providing fine-grained measurement and control of the PAPI interface. The high-level interface offers a more lightweight semantics but at the cost of reduced flexibility.

3 Porting PAPI to an embedded platform

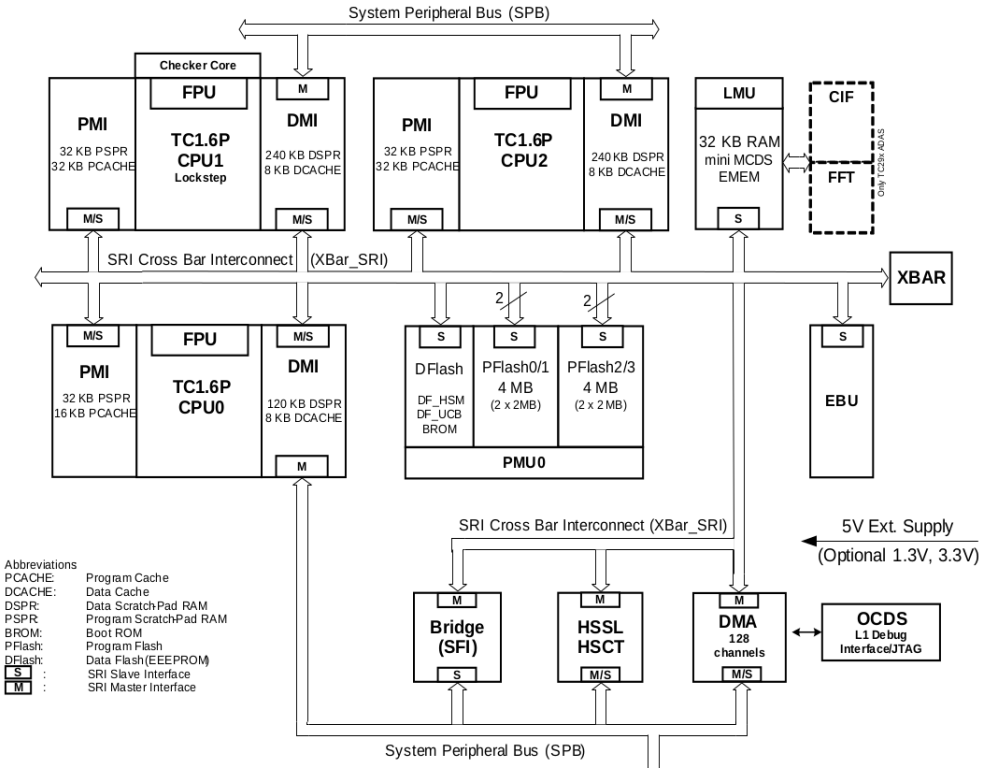


Figure 1 TC297 block diagram (from [11]).

In this work we assess the porting of the general-purpose PAPI library to an embedded target, to evaluate whether the same support for performance monitoring could be efficiently and effectively adopted in embedded systems. In our porting, we target the Infineon AURIX™ TriCore TC297 platform [11], a representative COTS platform in the automotive domain.

3.1 Reference platform

The TC297 block diagram is reported in Figure 1. The platform features three performance-efficient TriCore TC1.6P cores, all being equipped with separate core-local memories (scratchpads and caches) for instructions and data. Processors are then connected to each other and

to a shared "memory system" through the Shared Resource Interconnect (SRI) crossbar, which supports multiple requests to different slaves (memory regions or peripherals) to be served in parallel without incurring contention effects. The shared memory system comprises an SRAM device, accessed via the Local Memory Unit (LMU), and a FLASH device, accessed via the Program Memory Unit (PMU). The PMU offers 4 independent interfaces to the program flashes (2MB each) and 1 interface for the data flash. The LMU provides access to volatile memory resources whose primary purpose is to provide 32 kbytes of local memory for general purpose usage. Each of the TC1.6P cores implements three pipelines that allow the platform to support dual instruction issuing in parallel into an integer pipeline and load/store pipeline. The third pipeline is providing specialized support for zero-overhead loop instructions. Despite belonging to the same processor model, the three cores have slightly different memory specifications. In Core 1 and Core 2, first-level instruction and data caches and scratchpads are 32KB, 8KB, 32KB, and 240KB respectively, with slightly smaller capacities in the case of Core 0's, with 16KB, 8KB, 32KB, and 120KB. Finally, Core 1 is associated with a checker core, in lock-step mode.

Debug support and monitoring counters. A porting of PAPI to any target platform depends on the debug support and observability available. At the core level, the AURIX™ TC297 debug support features 5 dedicated PMC registers per core. Two registers are dedicated to count executed cycles (`CCNT`) and instructions (`ICNT`). The remaining three multiplexing registers (`M1CNT`, `M2CNT`, and `M3CNT`) can be configured to count the different hardware events supported by the platform by setting the proper bits in the Counter Control Register `CCTRL`.

The AURIX™ TC297 multiplexing registers can be enabled and configured to count 12 hardware events related to pipeline stalls (on the three different pipelines), cache behavior, and branch and SRI crossbar statistics:

- `IP_DISPATCH_STALL` : incremented on every cycle in which the Integer dispatch unit is stalled for whatever reason.
- `LS_DISPATCH_STALL` : incremented on every cycle in which the Load-Store dispatch unit is stalled for whatever reason.
- `LP_DISPATCH_STALL` : incremented on every cycle in which the Loop dispatch unit is stalled for whatever reason.
- `MULTI_ISSUE` : incremented in any cycle where more than one instruction is issued.
- `PCACHE_HIT` : incremented whenever the target of a cached fetch request from the fetch unit is found in the program cache.
- `PCACHE_MISS` : incremented whenever the target of a cached fetch request from the fetch unit is not found in the program cache and hence a bus fetch is initiated.
- `DCACHE_HIT` : incremented whenever the target of a cached request from the Load-Store unit is found in the data cache.
- `DCACHE_MISS_CLEAN` : incremented whenever the target of a cached request from the Load-Store unit is not found in the data cache and hence a bus fetch is initiated with no dirty cache line eviction.
- `DCACHE_MISS_DIRTY` : incremented whenever the target of a cached request from the Load-Store unit is not found in the data cache and hence a bus fetch is initiated with the write-back of a dirty cache line.
- `TOTAL_BRANCH` : incremented in any cycle in which a branch instruction is in a branch resolution stage of the pipeline.

- PMEM_STALL : incremented whenever the fetch unit is requesting an instruction and the instruction memory is stalled for whatever reason.
- DMEM_STALL : incremented whenever the Load-Store unit is requesting a data operation and the data memory is stalled for whatever reason.

Table 1 summarizes the events that can be traced through the (multiplexed) performance-monitoring counters with the respective configuration of the CCTRL register.

Table 1 Multi-Count Configuration (TC1.6P).

	Monitoring counters configuration registers		
CCRTL bits	M1CNT	M2CNT	M3CNT
000	IP_DISPATCH_STALL	LS_DISPATCH_STALL	LP_DISPATCH_STALL
001	PCACHE_HIT	PCACHE_MISS	MULTI_ISSUE
010	DCACHE_HIT	DCACHE_MISS_CLEAN	DCACHE_MISS_DIRTY
011	TOTAL_BRANCH	PMEM_STALL	DMEM_STALL

3.2 Selection and mapping of PAPI events

The ePAPI implementation only supports a selection of PAPI events. The selection was made according to two principles. First, while PAPI supports more than 100 preset events, actual implementations normally support only a subset of events. The TC297 is not an exception and several preset events are not supported. For instance, more than 30 events related to the L2 and L3 cache behavior, are inherently not supported in the target system as it exhibits a flat cache hierarchy. Second, the porting is tailored to the use of those performance monitors we consider more relevant for the embedded domain characterization, that is timing and/or energy analyses, multicore contention analysis, as well as average performance analysis.

Table 2 summarizes the results of our selection over PAPI event. Each selected event is associated with the available specification [11] and to the supporting counter in the TC297, with the associated PMC multiplexed register.

Mapping preset events is generally straightforward, with the exception of some PAPI events that could only be covered by combining the information from more than one PMC. In a few cases, the current support in the TC297 could only provide an over-approximation of the PAPI event. As an example, the PAPI_RES_STL event (cycles stalled on any resource) could be loosely upper-bounded by TC297 XMEM_STALL and XX_DISPATCH_STL events (as they partially overlap). Moreover, due to the inflexible configuration of the TC297 CCTRL (see Table 1), some events can only be covered by combining two values read from the same multiplexed PMC register. In this case, two executions are required to collect all necessary evidence from the PMCs. This last class of events is not directly implemented in ePAPI, thus the programmer is responsible for measuring and combining the different event counts as required. In addition, the last six entries in Table 2 report the native events supported by the TC297 that do not fall in PAPI preset event set.

4 ePAPI implementation and validation

The full PAPI specification includes more than 70 different functions. The ultimate objective of this study was not to provide a full implementation of the interface but, more precisely, to instantiate it to a representative embedded platform. The pruning over PAPI functions was mainly a consequence of the particular hardware-software configuration considered

Table 2 AURIX events to PAPI interface analysis.

PAPI event	Description	Counter source	Counter(s)
PAPI_BRU_IDL	Cycles branch units are idle	CCNT − TOTAL_BRANCH	CCNT − M1CNT
PAPI_L1_DCH	Level 1 data cache hit	DCACHE_HIT	M1CNT
PAPI_L1_ICH	Level 1 instruction cache hit	PCACHE_HIT	M1CNT
PAPI_L1_ICM	Level 1 instructions cache miss	PCACHE_MISS	M2CNT
PAPI_L1_DCA	Level 1 data cache accesses	DCACHE_HIT + DCACHE_M_C + DCACHE_M_D	M1CNT + M2CNT + M3CNT
PAPI_L1_DCM	Level 1 data cache misses	DCACHE_M_C + DCACHE_M_D +	M2CNT + M3CNT
PAPI_L1_ICA	Level 1 instruction cache access	PCACHE_HIT + PCACHE_MISS	M1CNT + M2CNT
PAPI_L1_ICR	Level 1 instruction cache reads	PCACHE_HIT + PCACHE_MISS	M1CNT + M2CNT
PAPI_MEM_SCY	Cycles stalled waiting for memory accesses	DMEM_STALL + PMEM_STALL	M2CNT + M3CNT
PAPI_L1_TCA	Level 1 total cache accesses	PCACHE_HIT+ PCACHE_MISS+ DCACHE_HIT+ DCACHE_M_C+ DCACHE_M_D	M1CNT(x2)+ M2CNT(x2)+ M3CNT
PAPI_L1_TCH	Level 1 total cache hits	PCACHE_HIT+ DCACHE_HIT	M1CNT(x2)
PAPI_L1_TCM	Level 1 total cache misses	PCACHE_MISS+ DCACHE_M_C+ DCACHE_M_D	M1CNT(x2)+ M3CNT
PAPI_TOT_CYC	Total cycles	CCNT	CCNT
PAPI_TOT_INS	Instructions completed	ICCNT	ICNT
PMEM_STALL	Cycles where the program memory is stalled.	PMEM_STALL	M2CNT
DMEM_STALL	Cycles where the data memory is stalled.	DMEM_STALL	M3CNT
MULTI_ISSUE	Cycles where more than one instruction is issued.	MULTI_ISSUE	M3CNT
IPDISP_STL	Cycles in which Integer Dispatch Unit is stalled.	IP_DISPATCH_STL	M1CNT
LSDISP_STL	Cycles in which Load-Store Dispatch Unit is stalled.	LS_DISPATCH_STL	M2CNT
LPDISP_STL	Cycles in which Loop Dispatch Unit is stalled.	LP_DISPATCH_STL	M3CNT

since ePAPI has been designed assuming a bare-metal environment and on top of the PMC support offered by the AURIX TC297. In particular, functions have not been selected for implementation in ePAPI for the following criteria:

1. Focus on CPU performance: while the latest versions of PAPI support diverse hardware components (e.g., network), we focused on CPU performance only as the central scope for timing characterization. We, therefore, discarded functions related to component selection.
2. Lack of platform support for Floating-Point Unit (FPU) events: despite the AURIX TC297 is equipped with a fully pipelined single-precision FPU [11], no support is provided to monitor hardware events related to the FPU module.
3. Lack of operating system support: this led us to discard all functions related to the collection of performance metrics on a thread or task basis, as well as those related to process virtualization.

4. Lack of standard output: we assume ePAPI to be deployed in scenarios where no visual output means is available, so that all the library outputs, including raw data and results, are not directly shown to the user but are stored into a predefined, configurable memory region. In reason of this less interactive use of ePAPI, we considered it unnecessary to implement those functions responsible for querying the system for hardware configurations or for printing any information on a screen.

Table 3 Subset of PAPI functions implemented in ePAPI.

High-level supported functions	
int PAPI_start_counters	Starts counting HW events
int PAPI_read_counters	Copies current PMCs to internal array and reset counters
int PAPI_accum_counters	Adds PMCs values to internal array and reset counters
int PAPI_stop_counters	Stops counting events and return current PMCs
int PAPI_num_counters	Gives the number of HW counters available in the system
int PAPI_epc	Gives events per cycle, real and processor time
int PAPI_ipc	Gives instructions per cycle, real and processor time
Low-level supported functions	
int PAPI_start	Starts counting HW events in an event set
int PAPI_read	Read HW events from an event set with no reset
int PAPI_accum	Accumulates and resets HW events from an event set
int PAPI_stop	Stops counting HW events in event set and return PMCs
int PAPI_reset	Resets the HW event counts in an event set
int PAPI_create_eventset	Creates a new empty event set
int PAPI_add_event	Adds a single HW event to an event set
int PAPI_add_events	Adds array of HW events to an event set
int PAPI_add_named_event	Adds an HW event by name to a PAPI event set
int PAPI_remove_event	Removes a HW event from a PAPI event set
int PAPI_remove_events	Removes an array of PAPI events from an event set
int PAPI_remove_named_event	Removes a named event from a PAPI event set
int PAPI_cleanup_eventset	Removes all PAPI events from an event set
int PAPI_event_name_to_code	Translates an PAPI event name into an integer event code
int PAPI_list_events	Returns a list of the HW events in an event set
int PAPI_num_events	Returns the number of HW events in an event set
int PAPI_state	Returns the counting state of an event set

As a result of a preliminary analysis of PAPI high-level and low-level interfaces, we restricted the number of functions to be included in ePAPI. This first ePAPI implementation supports almost all (7 out of 10) functions defined by the PAPI high-level interface but only a small subset (17 out of 66) of the low-level interface. Table 3 shows the subset of PAPI functions for which an implementation is provided in ePAPI on top of the AURIX TriCore. Full function signatures were omitted for the sake of clarity. The only high-level functions not implemented are *int PAPI_num_components*, related to component selection, and *int PAPI_flips, int PAPI_flops*, delivering high-level FPU statistics. Several low-level functions were not ported to ePAPI for the reasons above.

In the current ePAPI implementation, the results of the invocation of ePAPI functions, which includes PMCs and other relevant information, are mapped to a small area in the local Data Scratchpad (DSPR) of the monitored core, where the collected PMC values can be easily gathered. Alternative platform-specific output mechanisms can be defined (e.g., the EMEM module in the AURIX).

```
int Events[NUM_EVENTS] = {PAPI_TOT_CYC};
long long values[NUM_EVENTS];

/* Start counting events */
PAPI_start_counters(Events, NUM_EVENTS);

//Place code under analysis here

/* Read counters */
PAPI_read_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Add the counters */
PAPI_accum_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Stop counting events */
PAPI_stop_counters(values, NUM_EVENTS);
```

(a) PAPI use example.

```
int Events[NUM_EVENTS] = {PAPI_TOT_CYC};
long long values[NUM_EVENTS];

/* Start counting events */
ePAPI_start_counters(Events, NUM_EVENTS);

//Place code under analysis here

/* Read counters */
ePAPI_read_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Add the counters */
ePAPI_accum_counters(values, NUM_EVENTS);

//Place more code under analysis here

/* Stop counting events */
ePAPI_stop_counters(values, NUM_EVENTS);
```

(b) ePAPI use example.

Figure 2 Example uses of PAPI and ePAPI.

For the low-level aspects, ePAPI implementation builds on the adaptation to the TC297 of the `PMClib` library, a low-level, basic PMC interface developed as part of a previous study [6]. This library, entirely developed in assembly, allows direct control of the `CCTRL` register and manipulation of PMCs values.

ePAPI example. To evaluate the signature-level equivalence of our porting, Figure 2 compares two examples of the use of the high-level API of PAPI and ePAPI. In fact, the interface of ePAPI matches perfectly the one offered by PAPI. The only difference between the two implementations is found in the "e" prefix, which was used to make ePAPI implementation distinguishable but can be dropped for full compatibility. Sharing the same signature makes the use of ePAPI easier to users with PAPI background.

4.1 Validation of ePAPI

The AURIX™ TC297 implementation of ePAPI has been validated to check whether the functional behavior meets the software specification from PAPI documentation, by conducting a proper functional testing campaign. As a major requirement in ePAPI, and as a relevant difference with respect to mainstream PAPI, library calls are also required to incur low overhead, to avoid the performance monitoring process to interfere with the application being monitored, i.e., *probe effect* [18]. As a preliminary step, we validated correctness and accuracy of PMCs exploiting the low-level `PMClib` library (adapted from [6]) to measure small ad-hoc benchmarks that cause a known amount of events to be triggered for each traceable event. The functional correctness of ePAPI has been validated building on the same library and ad-hoc benchmarks for both the high-level and low-level APIs.

We assessed ePAPI consistency and accuracy against the `PMClib` library, which we know manipulates PMCs at a very low level with no unnecessary overhead. Clearly, the generic infrastructure of ePAPI (especially in its high-level) interface cannot have zero impact on the execution, as it necessarily requires more instructions to be executed. However, it is important that this impact is low and bounded. The validation of this specific requirement consisted of executing the same test programs using either the `PMClib` library or ePAPI interfaces (both high- and low-level) and compare the obtained PMC values.

Table 4 Results of ePAPI overheads validation.

PAPI Event	Bare-metal	ePAPI Read	ePAPI accum.	Bare-metal	ePAPI Read	ePAPI accum.	Bare-metal	ePAPI Read	ePAPI accum.
ePAPI_BRU_IDL	Cyc. Idle Branch Unit			Tot. Cycles			Tot. Instructions		
	70.017	+27	+28	100.019	+30	+31	80.011	+11	+11
ePAPI_L1_ICA	ICache accesses			Tot. Cycles			Tot. Instructions		
	499.002	+0	+0	1.029.597	+3	+3	1.017.010	+12	+12
ePAPI_L1_ICR	ICache reads			Tot. Cycles			Tot. Instructions		
	499.002	+0	+0	1.029.597	+23	+23	1.017.010	+12	+12
ePAPI_L1_ICH	ICache hits			Tot. Cycles			Tot. Instructions		
	498.890	+0	+0	1.029.595	+25	+26	1.017.010	+12	+12
ePAPI_L1_ICM	ICache misses			Tot. Cycles			Tot. Instructions		
	124.987	+0	+0	2.879.511	+23	+23	1.513.009	+11	+11
ePAPI_L1_DCA	DCache accesses			Tot. Cycles			Tot. Instructions		
	1.000.000	+0	+0	1.029.599	+20	+31	1.017.009	+11	+11
ePAPI_L1_DCH	DCache hits			Tot. Cycles			Tot. Instructions		
	999.900	+0	+0	1.029.597	+22	+22	1.017.009	+11	+11
ePAPI_L1_DCM	DCache misses			Tot. Cycles			Tot. Instructions		
	6.001	+0	+0	299.616	+16	+17	32.010	+12	+12
ePAPI_MEM_SCY	Stalls on Mem. accesses			Tot. Cycles			Tot. Instructions		
	290.336	+0	+0	430.579	+25	+25	65.011	+11	+11
PMEM_STALL	Stall on Program memory			Tot. Cycles			Tot. Instructions		
	127	+3	+3	160.061	+32	+33	143.011	+11	+11
DMEM_STALL	Stall cycles on Data memory			Tot. Cycles			Tot. Instructions		
	301.714	+0	+0	401.820	+28	+29	40.011	+11	+11
MULTI_ISSUE	Multiple issues			Tot. Cycles			Tot. Instructions		
	100.014	+3	+1	120.040	+33	+34	230.050	+12	+12
IP_DISPATCH_STALL	Instr. dispatch stalls			Tot. Cycles			Tot. Instructions		
	1.000	+0	+0	17.014	+33	+34	17.010	+12	+12
LS_DISPATCH_STALL	Load/store dispatch stalls			Tot. Cycles			Tot. Instructions		
	2.002	+10	+10	17.014	+33	+34	17.010	+12	+12
LP_DISPATCH_STALL	Load/store dispatch stalls			Tot. Cycles			Tot. Instructions		
	1.001	+0	+0	1.008.032	+25	+26	2.006.014	+11	+11

Table 4 compares the PMC values collected running specific benchmarks for all events supported in ePAPI. For a given event, the same benchmark is measured using `PMClib` and ePAPI high-level interfaces, considering "plain" and accumulated PMC reads. We focused on the high-level interface as it is the one potentially incurring more overhead. For each experiment, we report the counts over the relevant event, executed cycles, and instructions.

Results show that the overhead incurred by ePAPI is generally negligible. The only observed differences were on the number of instructions and cycles, while no overhead was observed on the specific event counts. The additional counts for `ePAPI_BRU_IDL` are to be interpreted as ePAPI only slightly affecting the branch unit. In fact, ePAPI is implemented in a way that counters are enabled and disabled to guarantee that the measured events belong to the program under analysis. ePAPI always executes around 11-12 instructions more than `PMClib`, due to the additional instructions and data used for generic PMC management. This is reflected in a difference in executed cycles varying from 25 to 28 cycles.

ePAPI limitations. The porting of ePAPI to the TC297 was motivated by the need for a common standardized interface for performance characterization of embedded systems, mainly to support timing and multicore contention analysis. As a result of our analysis on

the PAPI interface and on the support available in the specific target platform, we identified some limitations as well as some desirable characteristics that could not be matched owing to the limitations of PAPI, peculiarities of embedded targets or of the selected platform itself.

From the PAPI perspective, we observe that the subset of the interface defined by preset events is necessarily generic, and mostly designed to collect performance metrics for optimization purposes. Despite the use of native events can cover more relevant events from the embedded domain perspective (e.g., stalls events to characterize contention), not having those events in the cross-platform interface partially defeats the benefits of having a standardized interface across embedded targets.

From the AURIX™ TC297 perspective, we suffered from the limited PMC support available, which is pretty limited compared to the support available in conventional processors. This is not generally the case in more advanced embedded targets, and it was indeed the main cause for discarding PAPI functionalities from the porting. Considering the potential uses of the interface, the PMC support in the TC297 does not allow to characterize contention effects with satisfactory precision as stall events cannot be associated to the different target in the SRI. For instance, the `PMEM_STALL` event is counting stalls cycles suffered when fetching code from any PFlash interface, the LMU or even non-local scratchpads.

5 Conclusions and future directions

With the goal of supporting timing analysis of complex COTS multicore processors, in this paper we have reported our investigation on the adoption of a standardized performance monitoring interface for embedded targets. To that end, we considered the general-purpose PAPI specification and assessed it against the available PMC support in the AURIX™ TC297, a representative platform in the automotive domain. We identified a compatible subset of PAPI preset events and some platform-specific native events. We have developed and validated ePAPI, a functionally-equivalent and low-overhead implementation of PAPI for the AURIX™ TC297. As future directions, we are interested in the extension of ePAPI to support RTOS-based PAPI functions (e.g., supporting task- and thread-level PMCs) by considering the TC297-compatible Erika RTOS [7]. Further, since the first implementation of ePAPI is tailored to the TC297, we are also considering to extend ePAPI to different embedded platforms, preferably with wider PMC support.

References

1 perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`.

2 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Pérez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, pages 39–48. IEEE, 2015. `doi:10.1109/SIES.2015.7185039`.

3 Airbus. Global Networks, Global Citizens. 2018-2037. Global Market Forecast. `https://www.airbus.com/aircraft/market/global-market-forecast.html`, 2018.

4 ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. `https://www.arm.com/company/news/2015/04/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade`, 2015.

5 Dakshina Dasari, Vincent Nélis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3):272–322, 2016. `doi:10.1007/s11241-015-9229-9`.

6 Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the AURIX^{TM} TC27x. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 97:1–97:6. ACM, 2018. doi:10.1145/3195970.3196077.

7 Evidence. Erika Enterprise RTOS v3. http://www.erika-enterprise.com/, 2019.

8 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Design Autom. Electr. Syst.*, 14(1):2:1–2:24, 2009. doi:10.1145/1455229.1455231.

9 Carles Hernández, Jaume Abella, Francisco J. Cazorla, Alen Bardizbanyan, Jan Andersson, Fabrice Cros, and Franck Wartel. Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, volume 76 of *LIPIcs*, pages 16:1–16:23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ECRTS.2017.16.

10 Rafia Inam, Mikael Sjödin, and Marcus Jägemar. Bandwidth measurement using performance counters for predictable multicore software. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012, Krakow, Poland, September 17-21, 2012*, pages 1–4. IEEE, 2012. doi:10.1109/ETFA.2012.6489714.

11 Infineon. *AURIX™ TC29x B-Step 32-Bit Single-Chip Microcontroller - User's Manual V1.3 2014-12*. 2019.

12 Kyeong-Jae Lee and Kevin Skadron. Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA*. IEEE Computer Society, 2005. doi:10.1109/IPDPS.2005.448.

13 Kevin London, Shirley Moore, Phil Mucci, Keith Seymour, and Richard Luczak. The PAPI Cross-Platform Interface to Hardware Performance Counters. In *Department of Defense Users' Group Conference Proceedings*, pages 18–21, Biloxi, Mississippi, June 2001.

14 Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. High-Integrity Performance Monitoring Units in Automotive Chips for Reliable Timing V&V. *IEEE Micro*, 38(1):56–65, 2018. doi:10.1109/MM.2018.112130235.

15 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael 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. doi:10.1109/ECRTS.2014.20.

16 NXP/Freescle. *e6500 Core Reference Manual - E6500RM Rev 0 06/2014*. 2014.

17 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

18 Young Wn Song and Yann-Hang Lee. On the existence of probe effect in multi-threaded embedded programs. In Tulika Mitra and Jan Reineke, editors, *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 18:1–18:9. ACM, 2014. doi:10.1145/2656045.2656062.

19 Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

20 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 55–64. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531079.

21 Reza Zamani and Ahmad Afsahi. A study of hardware performance monitoring counter selection in power modeling of computing systems. In *2012 International Green Computing Conference, IGCC 2012, San Jose, CA, USA, June 4-8, 2012*, pages 1–10. IEEE Computer Society, 2012. doi:10.1109/IGCC.2012.6322289.

22 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 101–110. IEEE Computer Society, 2014. doi:10.1109/RTAS.2014.6925994.

Worst-Case Energy-Consumption Analysis by Microarchitecture-Aware Timing Analysis for Device-Driven Cyber-Physical Systems

Phillip Raffeck
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Christian Eichler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Peter Wägemann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Many energy-constrained cyber-physical systems require both timeliness and the execution of tasks within given energy budgets. That is, besides knowledge on worst-case execution time (WCET), the worst-case energy consumption (WCEC) of operations is essential. Unfortunately, WCET analysis approaches are not directly applicable for deriving WCEC bounds in device-driven cyber-physical systems: For example, a single memory operation can lead to a significant power-consumption increase when thereby switching on a device (e.g., transceiver, actuator) in the embedded system.

However, as we demonstrate in this paper, existing approaches from microarchitecture-aware timing analysis (i.e., considering cache and pipeline effects) are beneficial for determining WCEC bounds: We extended our framework on whole-system analysis with microarchitecture-aware timing modeling to precisely account for the execution time that devices are kept (in)active. Our evaluations based on a benchmark generator, which is able to output benchmarks with known baselines (i.e., actual WCET and actual WCEC), and an ARM Cortex-M4 platform validate that the approach significantly reduces analysis pessimism in whole-system WCEC analyses.

2012 ACM Subject Classification Hardware → Static timing analysis; Hardware → Power and energy; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases WCEC, WCRE, WCET, michroarchitecture analysis, whole-system analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2019.4

Supplement Material Source code: `https://gitlab.cs.fau.de/syswcec-uarch`

Funding This work is supported by the German Research Foundation (DFG), in part by Research Grant no. SCHR 603/9-2, no. SCHR 603/13-1, no. SCHR 603/14-2, and the Bavarian Ministry of State for Economics, Traffic and Technology under the (EU EFRE funds) grant no. 0704/883 25.

Acknowledgements We thank Simon Schuster for his insightful comments and the support with Platin's source base. We also thank Dominik Huber, Aaron Strahlberger, and Julius Wiedmann for their help with reducing the pessimism of the microarchitecture analysis.

1 Introduction

Most embedded systems have restrictions on their execution performance and energy availability. Tasks in such systems must, therefore, execute within execution-time and energy-consumption budgets arising from the system's schedule and the state of charge. To guarantee the execution within these budgets, reliable upper bounds on the worst-case execution time (WCET) and the worst-case energy consumption (WCEC) are essential. In contrast to WCEC analysis, numerous WCET analysis approaches are available [1, 3, 17, 18, 24, 25, 30].

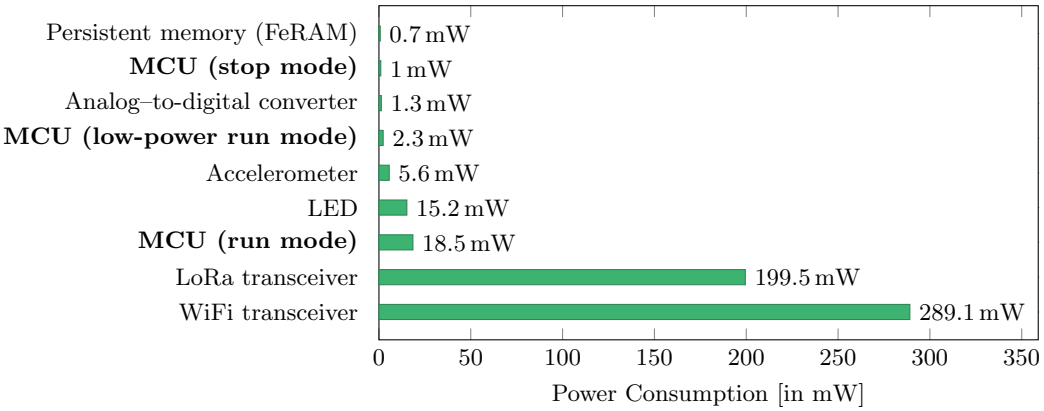


Figure 1 Consumers and their power consumption in systems with devices [7, 13, 15, 37].

In analogy to real-time systems where timely execution must be guaranteed, energy-constrained systems can have hard energy constraints [10], and WCEC bounds on the energy consumption of each task, which is the integral of a task's power consumption over its execution time, are essential to ensure the execution within energy budgets [39, 42]. More specifically, the energy demand from the task's start until its completion with all interferences (i.e., asynchronous interrupts, preemptions) is necessary, which is referred to as the *worst-case response energy (WCRE)*. For example, the WCRE is essential to guarantee the completion of a task storing critical system-state data to persistent memory without risking the battery's depletion during its execution.

These energy-constrained cyber-physical systems are integrated into physical processes, and thus have to interact with their environment, which is achieved by sensors (e.g., analog-to-digital converters), actuators (e.g., motors), or transceivers for communication (e.g., WiFi, LoRa). Figure 1 illustrates the power consumed by devices often integrated into cyber-physical systems. Especially transceivers, being software-controlled by the processor, consume an order of magnitude more than the processor's power demand. Also, CPU-internal (timer) devices significantly contribute to the overall consumption. That is, to precisely determine WCRE bounds, static worst-case analyses have to model these components. Existing approaches to determine WCEC bounds focus on the processor and microarchitecture-aware power modeling [21, 26, 29, 36, 38]. However, they ignore the influence of power-hungry devices.

Considering the temporal behavior of the microarchitecture, including hardware features such as caching and pipelining, is crucial and well explored in WCET analysis to determine sound and precise bounds. For some processors, this temporal behavior is provided by the documentation [20]. Regarding energy consumption, the energetic microarchitectural behavior is not available for commercial platforms. However, the maximum power consumption in specific states is available for some processors [14, 34] and peripherals [7, 37], which is information we use together with the worst-case execution time to bound energy consumptions.

In this paper, we present an approach to precisely account for all context-sensitive power consumers in an embedded system by a microarchitecture-aware WCET analysis. The whole-system WCRE analysis decomposes the input system into blocks of constant maximum power consumption. Based on this abstraction, a system-wide path analysis is conducted that includes all possible preemptions and the fixed-priority scheduling scheme. A microarchitecture-aware timing analysis accounts for the durations the devices are kept (in)active. Finally, with the gathered information, an optimization problem is formulated, whose solution eventually yields WCRE bounds for tasks in the system.

This work is based on prior work on WCRE analysis [40] and extends it by the following aspects: We now consider microarchitectural effects in the processor's pipeline, the influence of caches, and especially cache-related preemption delays in the fixed-priority scheduling scheme. Additionally, we further discuss the necessity to model energy consumption by microarchitecture-aware timing analysis in device-driven embedded systems. Finally, we evaluate our prototype with GenEE [12], which is able to generate device-aware benchmarks with known WCEC paths, which again serve as baselines for the presented evaluations.

The paper is structured as follows: Section 2 outlines the major challenges when determining precise upper bounds on the system's energy consumption. In Section 3, we present our approach to solve these challenges and give an overview of the implementation of our prototype, which is evaluated in Section 4. Section 5 outlines related and future work in the context of system-wide WCEC analyses, and Section 6 concludes.

2 Problem Statement

We aim to tighten the results of whole-system resource analysis by considering the microarchitecture state in the WCRE analysis. In this section, we present an overview of the assumed system and hardware model (see Section 2.1) and identify three challenges that arise due to the incorporation of the microarchitecture state (see Section 2.2–2.4).

2.1 System Model

For the analysis, we assume an OSEK-compliant [28] real-time system comprising tasks, interrupts, and temporarily active devices. According to the OSEK standard, our system model includes a fixed-priority scheduling scheme. Tasks may be interrupted by asynchronous interrupts and, according to their priority, preempted by higher-priority tasks. Furthermore, the task set is known at compile time and configured using OSEK's implementation language (i.e., OIL [27]). Our system model assumes that interrupt service routines (ISRs) have run-to-completion semantics. If ISRs activate higher-priority tasks, these are executed after the handler. All interrupts that potentially preempt a task during execution are known at compile time, and their occurrence is bounded by their minimum inter-arrival time.

All peripheral devices have two different power consumption modes (i.e., on/off), which are software-controlled via explicit system calls. Currently, we do not consider devices that have hardware-induced mode changes or have multiple modes, which are aspects of future work. Switching on a device subsequently leads to a constant increase in power consumption.

We consider small, embedded single-core processors with instruction caches and pipelines, whose complexity is similar to CPUs commonly used in safety-critical, hard real-time systems. An example for our system model is the Infineon XMC4500 [20] – an ARM Cortex-M4 processor we used in our implementation – that features an in-order 3-stage pipeline and a 4 KiB instruction cache (2-way set associative). The instruction cache employs an LRU cache-replacement policy, which is known to be free of timing anomalies [6, 16].

2.2 Challenge # 1: Microarchitecture-Aware Execution-Time Modeling

Modern embedded processors have complex hardware components to speed up the average execution time of instructions. Thus, the state of such microarchitecture components directly influences the execution time of machine-code instructions and therefore the execution time of every task in the system, including the operating system. This influence has to be considered when determining upper bounds for both energy consumption and execution time.

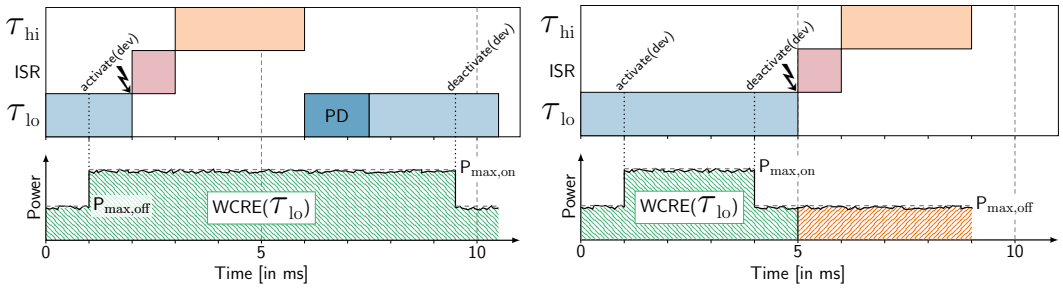


Figure 2 Preemption delays (PD) prolong execution time and also lead to higher response energy consumption. Furthermore, depending on the system's context-sensitive set of active devices, preemptions can lead to a higher response energy consumption.

Our approach: We employ well-known techniques from the literature to model the state of the pipeline and the instruction cache of our target platform at all relevant program points. Specifically, we perform cache analyses to obtain information about persistent data [8] and represent the microarchitecture state using a *microarchitecture execution graph* (MEG) [35].

2.3 Challenge # 2: Inter-Task Effects on the Microarchitecture State

Considering not only tasks in isolation but whole systems introduces another influence on the microarchitecture state: *inter-task effects*. As tasks interact with other tasks or the operating system, these interactions leave traces in the microarchitecture state. Consequently, the preempting entities disturb the microarchitecture state of the analyzed task, thereby influencing its execution time. These effects are known as *pipeline- and cache-related preemption delays* (PRPDs, CRPDs) [5, 31]. Figure 2 demonstrates how such delays contribute both to the worst-case response time (WCRT) and also the WCRE of a task in the presence of devices. Here, the low-priority task (τ_{lo}) experiences a preemption delay (PD) after being preempted by an ISR, which again leads to a resumption of the high-priority task τ_{hi}.

Our approach: To bound the number of possible preemptions, we use an existing analysis method that relies on an enumeration of possible (operating) system states [9]. We handle the costs of these preemptions with established techniques for PRPD and CRPD [5, 31] under consideration of recent findings on the effectiveness of these approaches [2, 33].

2.4 Challenge # 3: Device-Aware Energy-Consumption Modeling

To our knowledge, documentation on the microstructural energy-consumption behavior is available for neither the processor nor peripherals for any existing platform. To tackle this problem, several approaches to microstructural energy models exist to account for the number of transistor switches [21, 26, 29, 36, 38]. However, these methods fail when analyzing complex architectures and are not directly applicable to commercial off-the-shelf devices. The problem of energy-consumption modeling becomes even worse with the context-sensitive state of devices since the entire state of the system needs to be considered for the WCRE analysis: Reconsidering Figure 2, the energy consumption of τ_{lo} depends on whether the ISR occurs prior to or after activating a device after 1 ms. Consequently, a sound WCRE analysis has to be aware of possible device activations and system-wide power states.

Our approach: We overcome the challenge of missing microstructural energy models by leveraging the fact that the variances at the instruction level are minor compared to the power consumption of devices. Therefore, our approach bounds the WCEC via an *indirection* over

the execution time by performing microarchitecture-aware WCET analysis of regions that have a fixed set of active devices and thus a constant maximum power consumption P_{max}. These regions are context-sensitively determined by a system-wide path analysis and again used in a sound WCRE problem formulation [40].

3 Approach

In the following, we present our whole-system approach to tighten the WCRE estimates of tasks. We tackle the challenges introduced in Section 2 by microarchitecture-aware timing analysis of system regions with constant maximum power consumption.

3.1 Design Overview

Our approach tightens WCRE bounds by incorporating knowledge of the microarchitecture into the analysis. To achieve this, we increase the accuracy of the timing analysis by modeling an abstract microarchitecture state using microarchitecture execution graphs (MEGs) [35]. A MEG is a directed graph with nodes representing the abstract processor state (i.e., the state of the cache and the pipeline). Edges connect each node to its successor states and are weighted by the number of processor cycles it takes to complete the transition. We construct the MEG for a given instruction stream by computing the influence of each processor cycle on the microarchitecture state, thus creating nodes for each possible state.

For the pipeline, the necessary knowledge for the abstract state comprises the instructions currently processed in the pipeline and the number of CPU cycles needed to finish the processing. For the instruction cache, the interesting properties are the known cache contents and the order in which these contents are replaced in the case of a cache conflict. This analysis implies knowledge of both which datum possibly resides in which cache set as well as the replacement policy for cache ways. Having derived the necessary knowledge about pipeline and cache behavior by obtaining a hardware model and utilizing common cache analysis methods, we can construct a MEG for each basic block. The MEG enables to obtain tight bounds for the timing behavior and thus address Challenge # 1.

Constructing the MEG allows us to analyze tasks in isolation, but we are still missing the influence of inter-task effects on the microarchitecture state, as described in Challenge # 2. To consider these effects correctly, we utilize common estimation methods established in the literature [2, 31] to obtain bounds for the preemption delays in both pipeline and caches.

To address Challenge # 3, we extend the SysWCEC approach [40]. SysWCEC provides means to perform device-aware WCRE analysis of a whole system by formulating an integer linear program (ILP) over the possible (device-aware) system states. The optimization objective of the ILP is the WCRE. However, by design, the ILP also yields the execution time of the path corresponding to the WCRE of the analyzed task, which is not necessarily the same as the task's WCRT path. For example, short computations can have small or high energy consumption depending on the set of active devices. For the ILP formulation, basic blocks of the control-flow graph are aggregated to *power atomic basic blocks (PABBs)*, which represent single-entry single-exit regions that are atomic both from a scheduling perspective and regarding maximum power consumption. During the execution of a PABB no system calls are performed. Interrupts, however, can occur and potentially dispatch higher-priority tasks. Subsequently, all possible transitions between PABBs are context-sensitively enumerated and thereby inserted into the *power-state–transition graph (PSTG)*. The explicit enumeration also contains transitions for possible interrupts. Explicit path enumerations in the context of WCET analysis are often problematic due to the state-explosion problem [22]. However, for

system states, this explicit enumeration for a real-world benchmark (i.e., UAV platform), containing around 10 000 states, is conducted within a few seconds [9]. As the PSTG keeps track of the state of all devices and their power consumption, it represents a global, context-sensitive, power-aware control-flow graph, which enables to perform device-aware WCRE analysis. For further details on the PSTG and SysWCEC, we refer to previous work [40].

With all possible transitions, including interrupts, encoded in the PSTG, we bound the influence of an interrupt on a specific PSTG node. Each PSTG node again references the MEGs for each basic block it comprises. After determining the PRPD and CRPD induced by each possible interrupt for each PSTG node, we incorporate these preemption delays as additional constraints in the ILP of the SysWCEC approach. With this extension, the preemption delays of all interrupts are considered. By modeling the microarchitecture state, we are able to perform device-aware WCRE analyses with microarchitecture awareness through an indirection over a microarchitecture-aware WCET analysis. To summarize, the cost of each PSTG node i is expressed as $WCEC_i = WCET_{\mu,i} \cdot P_{max,i}$, where $WCET_{\mu,i}$ denotes the microarchitecture- and PD-aware WCET and $P_{max,i}$ the node's context-sensitive maximum power consumption. Note that this equation only holds for single PSTG nodes, which have a constant maximum power consumption. For the final WCRE determination, we formulate an ILP that contains these context-sensitive WCEC costs of PSTG nodes.

3.2 Implementation

The target platform for the analysis prototype is the Infineon XMC4500 development board [19], which features an ARM Cortex-M4 processor, as described in Section 2.1. We assume interrupt release jitter to be zero, to rule out the possibility that the analyzed scenario is interrupted by an ISR that occurred before the start of the scenario. Currently, we do not model inter–basic-block effects, which is part of our future work. Therefore, the analysis of every basic block starts with an empty pipeline. The cache states before and after the execution of a basic block are determined using `may-based persistence` analysis [8].

To bound the influence of interrupts on a PSTG node, we identify the point with the maximum PRPD among all instructions of the basic blocks associated with that PSTG node. For the CRPD, we compute the set union of *evicting cache blocks* (ECBs) for each interrupt and all tasks these interrupts potentially dispatch. We calculate the CRPD using an ECB-only approach, as recent evaluations have shown that it yields satisfactory results [33]. Both PRPD and CRPD are not integrated into the MEG, but rather considered as additional delays in the ILP and related to their corresponding interrupt occurrence.

Implementation Limitations: To bound the energy consumption associated with preemption delay, we currently assume the highest possible power state the preempted PABB could ever be executed in as the power state for the preemption delay. As it is possible to extract all possible successor states by searching the PSTG and finding the maximum possible power state, at which execution could continue, we consider this aspect as future work. Furthermore, the implementation does not include the semantics of ARM's assembly instruction for switching task contexts. As this code comprises only a few processor cycles, the influence of this code is minor compared to the overall energy consumption. Considering this aspect as well as bounding the influence of barrier instructions is part of our future work.

Our implementation is an extension to the SysWCEC [40] approach using the LLVM-based [23] analysis toolkit Platin [18, 30], which was originally developed for the PATMOS architecture [32]. Provided with a model of the target microarchitecture and the extended PSTG, Platin is now capable of generating an ILP including both microarchitecture-aware timing costs and constraints, which eventually reduce pessimism of the WCRE bounds.

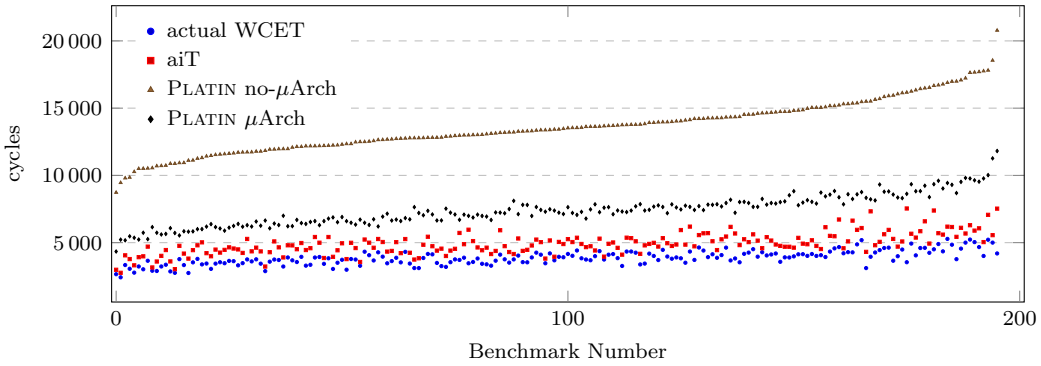


Figure 3 Actual WCET value and the WCET estimates of aiT and Platin with and without microarchitecture awareness for 196 generated benchmarks.

4 Evaluation

In this section, we present and discuss the experimental results obtained using our prototype. In Section 4.1, we first evaluate the influence of the microarchitecture awareness on the results of WCET analysis. We then set in relation the energy-consumption bounds estimated by our approach to both the actual WCRE as well as a bound computed from an approach that misses knowledge about the microarchitecture. Consequently, to obtain sound worst-case bounds, this approach has to make several pessimistic assumptions, for example, for the preemption delay. We describe the general evaluation setup for the WCRE analysis in Section 4.2 and evaluate power-consumption characteristics of our target system (see Section 4.3). We provide the results of WCRE analysis in Section 4.4 and additionally evaluate the applicability of our approach in Section 4.5.

4.1 WCET Analysis of Generated Benchmarks

To assess the accuracy gained by incorporating microarchitectural influences into timing analysis, the computed WCET of two versions of Platin, one with microarchitecture awareness and one without, are compared. For this, we generate benchmarks using the GenE benchmark generator [41], which provides automatically generated benchmarks with known WCET paths. For comparing the measurements and the static analyses (Platin no-μArch, Platin μArch), we use GenE's surrounding evaluation framework Aladdin [11]. Since Aladdin has integration for the commercially available WCET analysis tool aiT [1] on the ARM Cortex-M4, we also use these reported upper bounds for a comprehensive comparison.

Figure 3 shows the actual WCET value of the generated benchmarks obtained by measuring the known worst-case path in comparison to the analysis results. The microarchitecture-aware approach yields results, which are 39 % to 51 % smaller than the results of the approach without microarchitecture awareness. We are thus able to improve the accuracy of the analysis results by considering microarchitectural influences during the analysis. The additional comparison with aiT, an analysis tool with accurate hardware modeling as a defining feature, shows that there is still room for improvement, as in the worst-case Platin yields bounds 106 % larger than the bounds by aiT. Nonetheless, there are also benchmarks for which the Platin results are on par with the results obtained by aiT, being only 11 % larger.

4.2 Evaluation Setup for WCRE Analysis

As described in Section 3.2, the target platform for our prototype and thus the following evaluations is the Infineon XMC4500 development board [19]. We emulate the existence of devices by using four 56 Ω resistors as load. These resistors are driven by 3.3 V via external transistors, which the platform controls using GPIO pins. We measure the voltage drop over a 0.51 Ω shunt resistor and additionally over the whole system using a Tektronix MSO4034 oscilloscope with a sampling rate of 2.5 GS/s.

For the experiments, we use generated, OSEK-compliant task sets, which toggle I/O pins connected to the resistors. This way, we emulate realistic task sets interacting with devices. As a generator, we now employ GenEE [12] that creates benchmarks with a known worst-case path for energy consumption, which leads to the benchmark's WCEC. The benchmark-generation approach allows us to measure the energy consumption of that worst-case path and thus to compare the analysis results against this known baseline (i.e., actual WCEC).

4.3 Power Trace of Target System

We first examine the power-consumption behavior of our target platform. Figure 4 shows a comparison of the power consumption of one generated task set with and without attached electrical load. For the baseline trace without load (depicted below in black), we have an average power consumption of 499 mW with a standard deviation of 5 mW. These fluctuations are small in comparison to the changes in power consumption when switching devices on and off (depicted above in green). This observation underlines our approach to model the energy consumption via an indirection over microarchitecture-aware WCET analysis ($WCEC = P_{max} \cdot WCET$), where P_{max} denotes the maximum power consumption in states with a fixed set of active devices. To evaluate the overestimation of this abstraction from the dynamic power behavior, we observed the CPU's minimum and maximum power. Here, we found power variances from $P_{min} = 481$ mW up to $P_{max} = 524$ mW, which means a maximum dynamic power variation of 43 mW. Thus, always assuming a maximum constant power of the CPU leads to a worst-case overestimation of 8 %. We consider this worst-case overestimation as minor compared to the impact of transceivers in the range of hundreds of mW. Furthermore, for a reliable determination of P_{max} the ambient temperature needs to be also considered, which goes beyond the scope of this paper. Since we measure the entire system including all peripherals, parasitic capacitance, such as from the driving transistors, reduce the power consumption's transition steepness. That is, when switching on a device, it takes several microseconds until the maximum power of the new power state is reached.

4.4 WCRE Analysis With Devices

To assess the validity of our approach, we perform WCRE analysis on generated task sets with a known worst-case path for energy consumption. This way, we can compare the WCRE bound obtained with our approach both against a known baseline as well as against approaches that have to make pessimistic assumptions. This comparison allows us to evaluate the quality of our WCRE estimate compared to the actual WCRE value. Additionally, we assess the improved tightness of the analysis result gained by knowledge of the microarchitecture.

We generated task sets comprising between three and nine tasks with different execution times. Figure 5a shows the actual WCRE values of these task sets as well as the analysis results of our approach and two pessimistic approaches. One of these approaches is SysWCEC without microarchitecture awareness (labeled *pessimistic SysWCEC*), which is forced to make

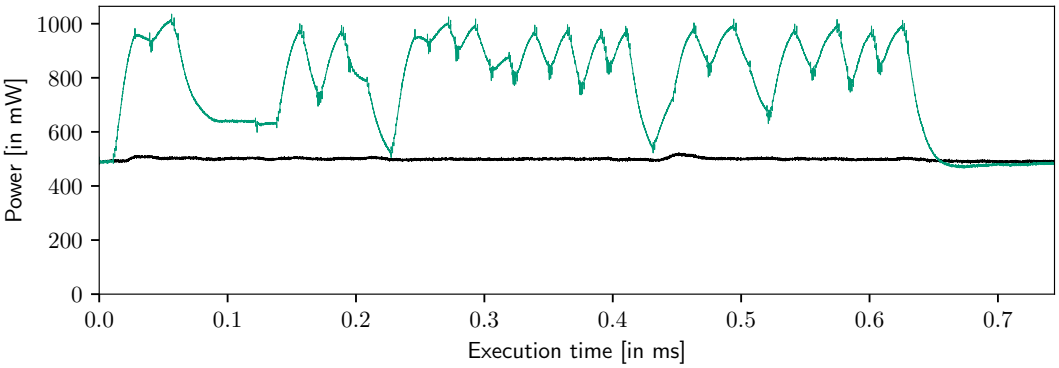


Figure 4 Trace of the power consumption of one evaluation benchmark with (green, above) and without (black, below) attached devices. The toggling of devices entails power-consumption changes which are much larger than fluctuations observed due to microarchitecture effects.

pessimistic assumptions, for example, for instruction fetch latencies (i.e., cache misses). The other (labeled *µArch always-on*) is a microarchitecture-aware approach lacking device-state modeling, which thus has to assume that all devices are always on.

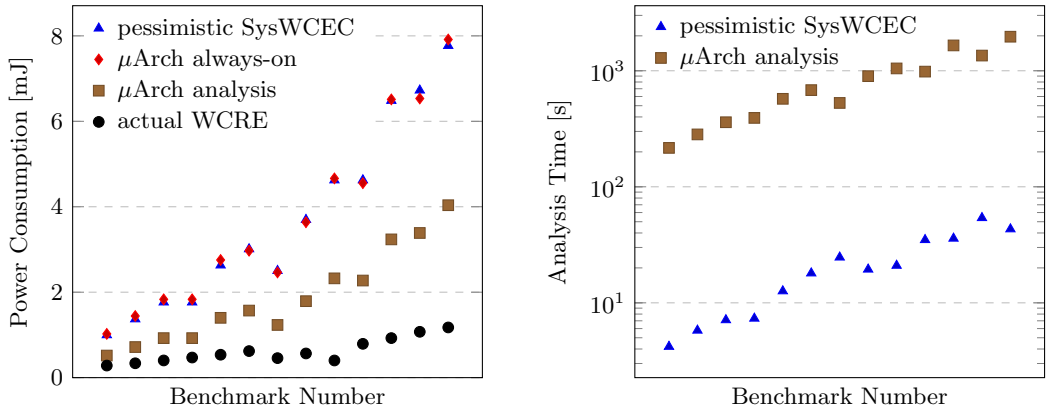
Our approach yields overestimations of 83 % up to 479 %, with a median of 170 % and a geometric mean of 173 % (an overestimation of 0 % represents the actual WCRE). While bringing potential for improvements to light, the results show that by leveraging both microarchitecture awareness and knowledge about the device states, we are able to give tighter bounds in comparison to the pessimistic approaches. We achieve between 46 % and up to 51 % tighter bounds compared to the analysis without the microarchitecture model, and between 49 % and up to 52 % tighter bounds compared to the always-on approach. With overestimations of around 170 %, our approach is capable of providing acceptable bounds on the WCRE, while yielding significantly better results than both pessimistic approaches. This confirms the validity of our approach and demonstrates the necessity to consider both the microarchitecture state and the state of devices to achieve tight WCRE bounds. The results further support our solution to circumvent the microarchitecture modeling of energetic behavior through an indirection over execution time analysis.

4.5 Applicability and Scalability

In the previous section, we showed that our approach is capable of computing tighter results. This improvement comes, however, at the cost of increased analysis times due to the complexity of modeling the microarchitecture state. Figure 5b depicts the execution times of the WCRE analyses on an Intel i5-4590 CPU with 16 GB of RAM. PLATIN without microarchitecture awareness finishes the analysis in a few seconds, whereas the microarchitecture-aware approach takes up to 50 x longer. However, the analysis times are still in the range of minutes for benchmarks with up to 20 000 processor cycles. We argue that this increase is acceptable for the up to 51 % tighter results gained by microarchitecture awareness.

5 Related & Future Work

To the best of our knowledge, this approach is the first for WCRE in cyber-physical systems where microarchitecture-aware timing analysis is exploited to account for on/off timespans of devices. There exists a large body of related work on microarchitecture-aware energy modeling [21, 26, 29, 36, 38]. In contrast to these approaches, we do not directly consider



(a) Actual WCRE value (circles) and the estimates of PLATIN with (squares) and without (triangles) microarchitecture awareness, and an always-on approach (diamonds).

(b) Analysis time needed with (squares) and without (triangles) microarchitecture awareness to compute the WCRE estimates for the generated benchmarks shown on the left.

Figure 5 Results of the WCRE analysis for 13 generated benchmarks.

dynamic power variations due to the number of transistor switches on the microarchitectural level. In this context, Morse et al. presented an overview of the limitations of WCEC modeling on the microarchitecture level [26]. The overview outlines the problem of data-dependent power consumption of instructions. For processors with limited microarchitectural complexity, we found power-consumption variances being only a minor fraction of the overall power consumption of device-driven systems [40], where active transceivers consume hundreds of mW. Nevertheless, approaches to reduce the pessimism of the processor, as proposed by Morse et al. [26], are directly applicable to our approach. Furthermore, we see potential to model data-depending operations again with existing WCET approaches for data-flow analysis [4]. This data-dependent analysis potentially improves modeling transistor switches.

A further aspect of future work is handling devices that, for example, reduce their energy consumption after a defined timespan once activated. Since our approach is aware of (microarchitecture-aware) WCETs along system-wide paths, we are able to apply functions for the energy-consumption cost to account for these complex energy-consumption behaviors.

Additionally, we believe that we can further reduce the pessimism of the WCRE analysis results by including inter–basic-block and inter-PABB effects in the analysis, for example, by modeling pipeline interleaving and performing data-flow analyses on the PSTG.

6 Conclusion

Static worst-case energy-consumption analysis gains importance with the increasing number of safety-critical cyber-physical systems that are battery-operated. To guarantee safe operation, determining reliable and precise WCRE bounds that account for the whole system is crucial.

This paper presents an approach that decomposes the system into blocks with a fixed set of active devices, which is used to determine system-wide control flows. A microarchitecture-aware WCET analysis along these paths precisely determines the durations devices are active under consideration of preemption delays. This information is used as costs in a WCRE formulation. Our evaluations show that this composite analysis significantly reduces analysis pessimism. The source code of our analysis approach is publicly available.

Source code: `https://gitlab.cs.fau.de/syswcec-uarch`

References

1 AbsInt. aiT WCET analyzers. `https://www.absint.com/ait/`.

2 S. Altmeyer, R. I. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

3 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Proc. of the 8th Int'l Work. on Software Technolgies for Embedded and Ubiquitous Systems (SEUS '10)*, pages 35–46, 2010.

4 Johann Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Systems*, 22(3):183–227, 2002.

5 J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proc. of the 2nd Real-Time Technology and Applications Symp. (RTAS '96)*, pages 204–212, 1996.

6 F. Cassez, R. Rydhof Hansen, and M. C. Olesen. What is a timing anomaly? In *Proc. of the 12th Int'l Work. on Worst-Case Execution Time Analysis (WCET '12)*, 2012.

7 Shenzhen HOPE Microelectronics Co. RFM95/96/97/98(W) - low power long range transceiver module, 2019.

8 C. Cullmann. Cache Persistence Analysis: Theory and Practice. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 12(1s):40:1–40:25, 2013.

9 C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. SysWCET: Whole-system response-time analysis for fixed-priority real-time systems. In *Proc. of the 23nd Real-Time and Embedded Technology and Applications Symp. (RTAS '17)*, pages 37–48, 2017.

10 N. Duda, T. Nowak, M. Hartmann, M. Schadhauser, B. Cassens, P. Wägemann, M. Nabeel, S. Ripperger, S. Herbst, K. Meyer-Wegener, F. Mayer, F. Dressler, W. Schröder-Preikschat, R. Kapitza, J. Robert, J. Thielecke, R. Weigel, and A. Koelpin. BATS: adaptive ultra low power sensor network for animal tracking. *Sensors*, 18(10):1–34, 2018.

11 C. Eichler, P. Wägemann, T. Distler, and W. Schröder-Preikschat. Demo Abstract: Tooling Support for Benchmarking Timing Analysis. In *Proc. of the 23rd Real-Time and Embedded Technology and Applications Symp. (RTAS '17)*, pages 159–160, 2017.

12 C. Eichler, P. Wägemann, and W. Schröder-Preikschat. GenEE: A benchmark generator for static analysis tools of energy-constrained cyber-physical systems. In *Proc. of the 2nd Work. on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench '19)*, 2019. URL: `https://www4.cs.fau.de/Publications/2019/eichler_19_cpsiotbench.pdf`.

13 Freescale Semiconductor, Inc. *KL46 Sub-Family Reference Manual*, 2013.

14 Freescale Semiconductor, Inc. *Kinetis KL46 Sub-Family*, 2014.

15 Fujitsu Semiconductor. *FRAM MB85RC256V*, 2013.

16 G. Gebhard. Timing anomalies reloaded. In *Proc. of the 10th Int'l Work. on Worst-Case Execution Time Analysis (WCET '10)*, 2010.

17 D. Hardy, B. Rouxel, and I. Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *Proc. of the 17th Int'l Work. on Worst-Case Execution Time Analysis (WCET '17)*, pages 8:1–8:12, 2017.

18 S. Hepp, B. Huber, D. Prokesch, and P. Puschner. The platin Tool Kit – The T-CREST Approach for Compiler and WCET Integration. In *Proc. of the 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS '15)*, pages 277–292, 2015.

19 Infineon Technologies AG. Evaluation Board XMC4500 Relax Kit & XMC4500 Relax Lite Kit, 2014.

20 Infineon Technologies AG. *XMC4500 Microcontroller Series for Industrial Applications — Data Sheet*, 2017. V1.5 2017-12.

21 R. Jayaseelan, T. Mitra, and X. Li. Estimating the Worst-Case Energy Consumption of Embedded Software. In *Proc. of the 12th Real-Time and Embedded Technology and Applications Symp. (RTAS '06)*, pages 81–90, 2006.

22 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proc. of the 21st Int'l Conf. on Real-Time Networks and Systems (RTNS '13)*, pages 161–170, 2013.

23 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization (CGO '04)*, pages 75–86, 2004.

24 Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007.

25 B. Lisper. SWEET – A tool for WCET flow analysis. In *Proc. of the 6th Int'l Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '14)*, pages 482–485, 2014.

26 J. Morse, S. Kerrison, and K. Eder. On the Limitations of Analyzing Worst-Case Dynamic Energy of Processing. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 17(3):59:1–59:22, 2018.

27 OSEK/VDX Group. OIL: OSEK implementation language. Technical report, OSEK/VDX Group, July 2004.

28 OSEK/VDX Group. Operating System Specification 2.2.3. Technical report, OSEK/VDX Group, February 2005.

29 J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis. In *Proc. of the 20th Int'l Work. on Software and Compilers for Embedded Systems (SCOPES '17)*, pages 51–59, 2017.

30 P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST Approach of Compiler and WCET-Analysis Integration. In *Proc. of the 9th Work. on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS '13)*, pages 33–40, 2013.

31 J. Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proc. of the 21st Real-Time Systems Symp. (RTSS '00)*, pages 195–204, 2000.

32 M. Schoeberl et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61:449–471, 2015.

33 D. Shah, S. Hahn, and J. Reineke. Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis. In *18th Int'l Work. on Worst-Case Execution Time Analysis (WCET '18)*, pages 7:1–7:11, 2018.

34 Silicon Laboratories, Inc. *EFM32GG Reference Manual*, 2016.

35 I. J. Stein. *ILP-based path analysis on abstract pipeline state graphs.* epubli, 2010.

36 S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proc. of the Int'l Work. on Power And Timing Modeling, Optimization and Simulation (PATMOS '01)*, 2001.

37 Espressif Systems. ESP8266EX Datasheet Version 6.0, 2018.

38 V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing Systems*, 13(2-3):223–238, 1996.

39 M. Völp, M. Hähnel, and A. Lackorzynski. Has Energy Surpassed Timeliness? Scheduling Energy-Constrained Mixed-Criticality Systems. In *Proc. of the 20th Real-Time and Embedded Technology and Applications Symp. (RTAS '14)*, pages 275–284, 2014.

40 P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat. Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems. In *Proc. of the 30th Euromicro Conf. on Real-Time Systems (ECRTS '18)*, pages 24:1–24:25, 2018. doi:10.4230/LIPIcs.ECRTS.2018.24.

41 P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat. Benchmark Generation for Timing Analysis. In *Proc. of the 23rd Real-Time and Embedded Technology and Applications Symp. (RTAS '17)*, pages 319–330, 2017.

42 P. Wägemann, T. Distler, H. Janker, P. Raffeck, V. Sieh, and W. Schröder-Preikschat. Operating Energy-Neutral Real-Time Systems. *ACM Trans. on Embedded Computing Systems (ACM TECS)*, 17(1):11:1–11:25, 2017.

WCET of OCaml Bytecode on Microcontrollers: An Automated Method and Its Formalisation

Steven Varoumas
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6
F-75005, Paris, France
Cnam, Centre d'études et de recherche en informatique et communications, Cédric
292 rue Saint Martin, 75003, Paris, France

Tristan Crolard
Cnam, Centre d'études et de recherche en informatique et communications, Cédric
292 rue Saint Martin, 75003, Paris, France

Abstract

Considering the bytecode representation of a program written in a high-level programming language enables portability of its execution as well as a factorisation of various possible analyses of this program. In this article, we present a method for computing the worst-case execution time (WCET) of an embedded bytecode program fit to run on a microcontroller. Due to the simple memory model of such a device, this automated WCET computation relies only on a control-flow analysis of the program, and can be adapted to multiple models of microcontrollers. This method evaluates the bytecode program using concrete as well as partially unknown values, in order to estimate its longest execution time. We present a software tool, based on this method, that computes the WCET of a synchronous embedded OCaml program. One key contribution of this article is a mechanically checked formalisation of the aforementioned method over an idealised bytecode language, as well as its proof of correctness.

2012 ACM Subject Classification Computer systems organization → Embedded software; Computer systems organization → Real-time systems

Keywords and phrases Worst-case execution time, microcontrollers, synchronous programming, bytecode, OCaml

Digital Object Identifier 10.4230/OASIcs.WCET.2019.5

Supplement Material `http://stevenvar.github.io`

1 Introduction

Due to their low cost and efficient power use, microcontrollers are heavily used by the embedded system industry. Nonetheless, the very limited memory and power resources of these devices has lead developers to use traditional low-level languages such as C or assembly. While being precise and powerful, these languages often lack the hardware abstraction that would allow programmers that are not system programming experts to write various applications, from home automation to more critical applications. For this reason, many projects have emerged that make it possible to run, on microcontrollers with less than 10 kilo-bytes (kb) of RAM, programs written in higher-level languages such as Java [3], Scheme [19] [6], or OCaml [24]. The runtime environments developed in these projects usually include an optimised *virtual machine* that interprets bytecode of the language directly on the device. Beyond the usual hardware abstraction layer, these languages offer a higher-level expressiveness for the development of programs, and more guarantees over them (thanks to strong static type systems and automatic memory management among other features).

Timing constraints are frequent in embedded systems: they must often retain quick reaction times, in order to handle rapid changes in their environments. Computing the worst-case execution time (WCET) of the program's routine responsible for reacting to input stimuli is thus necessary, in order to ensure that no input is ignored during execution, and thus prevent a potential incorrect or dangerous behaviour from the system. A usual method of handling these timing constraints is to rely on schedulability analyses together with the use of *real-time operating systems* such as FreeRTOS [7]. However, those systems are often quite heavy and not particularly adapted for microcontrollers with low memory resources. For this reason, the approach we present relies instead on the synchronous programming model. This model offers a lightweight reactive and concurrent programming model, particularly adapted to the applications of microcontrollers and their material limitations [21], as well as advantageous properties for WCET computing: by construction, the generated code does not contain loops and does not need annotations.

In this article, we use a toolchain stemming from our previous works that consists in an OCaml virtual machine implementation fit for running on various models of microcontrollers, as well as a compatible *synchronous programming* extension to the OCaml language. This solution enjoys the high-level features of the OCaml programming language in order to develop *safer* programs for embedded system, and benefits from a model of concurrent programming adapted to the scarce memory resources of microcontrollers. Our approach, which uses a bytecode representation of embedded OCaml programs, provides *portability*, and enables a *modularisation* of various analyses done over these programs.

The main contribution presented in this article is an automated method that computes the WCET of a bytecode program on partially unknown inputs, together with its correctness proof. The method is implemented as a prototype tool called *Bytecrawler* which supports any OCaml bytecode instruction found in a compiled synchronous program. The formalisation is currently based on a small idealised bytecode language, all the proofs have however been mechanically checked.

Section 2 presents some preliminaries: we introduce the OMicroB virtual machine [23] and OCaLustre, a synchronous extension to the OCaml language [21]. Section 3 describes our method for computing the WCET of OCaml bytecode, and its implementation. Section 4 contains the formalisation and the correctness proof of the method. Section 5 concludes with a short discussion about related and future works.

2 Preliminaries

2.1 OMicroB: an OCaml virtual machine for microcontrollers

OMicroB [23] is an implementation of the OCaml virtual machine (named *ZAM - Zinc Abstract Machine* [11]) dedicated to run OCaml bytecode on microcontrollers with very scarce resources (typically, less than 10kb of RAM and less than 100kb of flash memory). Lightweight and configurable, it provides every feature of the language and its runtime, such as a strong static type system with type inference, the use of various programming paradigms (functional, imperative, modular, and object-oriented), as well as automatic memory management with garbage collection (GC). OMicroB is designed to be portable, due to its implementation in standard C, that we consider as a *portable assembly language*, since most of the target models of microcontrollers come with a dedicated C compiler. We were successful in porting OMicroB to AVR microcontrollers, and ports to ARM Cortex-M0 (used by BBC micro:bit cards) and PIC32 are promising works in progress.

The main component of OMicroB is an optimised stack-based bytecode interpreter, capable of running all the 149 bytecode instructions of the language. This interpreter handles various registers (such as a stack pointer, a code pointer, an accumulator, etc.) whose values are accessed and modified by the instructions of the running OCaml program. Moreover, the interpreter may invoke various low-level primitives (mostly dealing with input/output operations) that are directly implemented in C.

As shown in Figure 1, in order to execute an OCaml program on a microcontroller, the source code is first compiled into bytecode (by the standard *ocamlc* compiler), which is then "cleaned" by the *ocamlclean* tool which performs dead-code elimination. This bytecode program is then embedded into a C source code as an array of bytes by a tool called *bc2c*. It is then compiled together with the OCaml runtime library and interpreter by some platform-dependent C compiler and linker, and thus turned into an executable. For simulation purposes, the program can also be compiled for (and executed on) a generic PC.

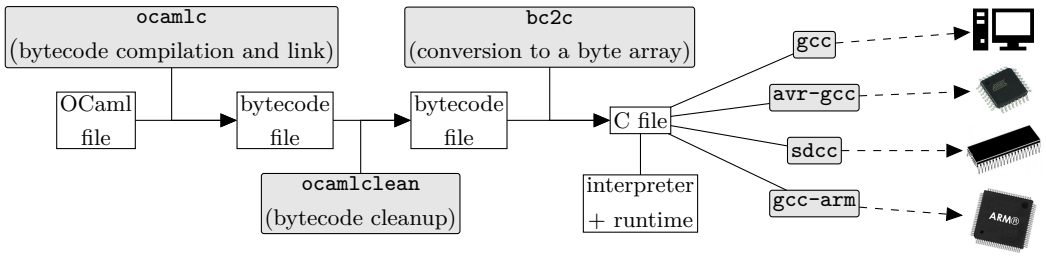


Figure 1 Compilation of an OCaml program with OMicroB (taken from [23]).

OMicroB offers a safer and more expressive model of programming than the classical C and assembly languages that are traditionally used to program microcontrollers. The robustness and the greater hardware abstractions gained by such a higher-level programming language makes it easier for developers that are not particularly well-versed in hardware specifics to develop programs for embedded systems.

2.2 OCaLustre: a synchronous extension of OCaml

Due to their constant interactions with their environment, embedded systems must react quickly to various stimuli. A model of concurrent programming, suitable for the size of the limited resources of microcontrollers might thus be a welcomed addition to the programming of microcontrollers [21]. Therefore, we use OCaLustre [22], a synchronous programming extension of OCaml, inspired by the Lustre [4] language. Synchronous programming rely on the main principle that computations made during a *logical instant* are instantaneous: that is, the time required at each instant by the program to compute its output from its input should be ignored when reasoning about programs.

In OCaLustre, as in Lustre, the elementary synchronous component is called a *node*: a function that takes flows of values as input and computes output flows. The body of a node is a system of equations, defining the set of output or local values. For example, the `count` node of figure 2 takes an `r` flow as an input, and returns the `cpt` flow as its output. The `-»` operator, equivalent to the `fby` ("followed-by") Lustre operator, has the following semantics: `x = a -» b` means that `x` is equal to `a` for the first instant, and then to the *previous* value of `b` (i.e. the value computed for `b` at the preceding synchronous instant) for the subsequent instants. Therefore, `count` computes the series of natural numbers, reset to zero when the value of the `r` flow is true.

```
let%node count r ~return:cpt =
  aux = 0 ->> (cpt + 1);
  cpt = if r then 0 else aux
```

Figure 2 A synchronous node.

During the compilation of an OCaLustre program, each node is separately turned into standard OCaml code, compatible with the `ocamlc` bytecode compiler, and thus with OMicroB. Our compilation method, derived from the *single loop code generation* used by most Lustre compilers, produces lightweight code, compatible with the resources of microcontrollers. It also provides guarantees over the program, such as the absence of *causality loops* (i.e. flows should not mutually depend on each other during the same instant).

3 Worst case execution time analysis of OCaml bytecode

In order to compute the WCET of a synchronous OCaLustre program, we created *Bytecrawler*, a tool that computes an upper bound of the execution time of each synchronous instant. This tool has the main advantage of working over the *bytecode* instructions of the virtual machine, rather than the underlying native-code instructions. It relies on the fact that bytecode analyses and platform-dependent analyses are separated ([9]) to free the application developer from needing to put additional annotations in their program: loop bounds for the program are not required due to the chosen programming model, and lower-level annotations, that can appear in the bytecode interpreter or the runtime library, are provided by the platform developer.

3.1 Bytecrawler WCET computation function

Values of variables that come from the environment of the program (such as values returned by electronic sensors) are unknown at compile-time, and thus introduce an *external non-determinism*. Bytecrawler can thus be seen as an *abstract* bytecode interpreter, extended to deal with unknown values. It is based on the standard method of [14] that considers all possible paths of the program to compute the costlier one, but Bytecrawler refines this evaluation by performing a hybrid execution that uses *concrete* values whenever possible. The program is normally executed by Bytecrawler with all values that are *known* at compile-time, but the evaluation function is generalised to handle *unknown* values: when reaching a conditional branching bytecode instruction (namely, `BRANCHIF` and `BRANCHIFNOT`) whose condition's value is unknown, Bytecrawler will explores the different possible paths.

Figure 3 is an excerpt of Bytecrawler's WCET computation function. It computes an upper bound of the timing cost (in cycles) of a given OCaml program using a *cost* function that maps each bytecode instruction to its cycle count, and following the bytecode semantics. In particular, the `CCALL` instruction corresponds to the call to an I/O primitive (written in C): the return value of such a call is always unknown at compile time.

Note that, while this method might work for *any* OCaml program, some limitations quickly appear: any looping program whose number of iterations depends of an unknown value could not be bounded, and the triggering of the garbage collection (GC) algorithm could break the WCET estimation. For this reason, Bytecrawler is better suited to work only on the synchronous extension of OCaml, which enjoys the nice properties that instants do not contain any loop, and they handle only basic data types which never trigger the GC.

```
let rec wcet state =
  let state' = {state with pc = state.pc + 1} in
  let instr = state.instrs.(pc) in
  cost instr + match instr with
  | CONST i -> wcet {state' with accu = Int i}
  | BRANCH ptr -> wcet {state with pc = ptr}
  | BRANCHIF ptr ->
    (match state.accu with
    | Int 0 -> wcet state'
    | Int _ -> wcet {state with pc = ptr}
    | Unknown -> max (wcet {state with pc = ptr}) (wcet state'))
  | ADDINT ->
    (match state.accu, state.stack with
    | Int x, (Int y)::s -> wcet {state' with accu = Int (x+y); stack = s}
    | _, _ -> wcet {state' with accu = Unknown})
  | CCALL _ -> wcet {state' with accu = Unknown}
  | STOP -> 0
```

Figure 3 Bytecrawler WCET computation function (excerpt).

▶ Remark. Symbolic execution [1] might allow us to refine even more the WCET estimate by pruning unreachable paths. Such an analysis for a stack-based virtual machine seems more involved than for imperative languages. For instance, the symbolic execution of Java bytecode is clearly not straightforward [13]). Moreover, functional programs usually feature higher-order functions and the OCaml virtual machine is tailored for an efficient evaluation of such programs. A recent experiment with a symbolic execution for Haskell [8] shows that defunctionalisation into some intermediate first-order functional language might be required. While our synchronous extension does not currently include higher-order functions, translating back from this higher-order bytecode language to a first-order intermediate representation would already prove difficult. We thus keep the study of this method for a future work.

3.2 Cost function for bytecode instructions

The cost function which associates each bytecode instruction to a cycle count is a finite map, represented as a table. This table can be computed by a classic WCET tool (such as the Bound-T execution time analyser [10]), configured for the correct model of microcontroller. A key benefit of our approach resides in the fact that this table must be computed *only once* for each microcontroller. The targeted devices are supposed to not induce *timing anomalies* [15] due to their simple hardware model (cache-less, with in-order pipelines which are not timing anomalous [5]): this ensures that the computed cost of each bytecode instruction remains the same for every given OCaml program, and provides *compositionality* of the timing analysis.

For most of the bytecode instructions, the execution time is a constant, and our method thus does not overestimate their execution time. For some other instructions, the execution time is dependent on some argument (for example, the `APPTERM` instruction which performs tail calls depends on the number of parameters of the function) which is explicit in the bytecode: their cost can be pre-computed for every realistic instruction/parameter pair. For some of the remaining bytecode instructions (for instance, those dealing with higher-order closures), the execution time might be more difficult to bound statically. However, these instructions do not appear, by hypothesis, in the considered bytecode. Lastly, calls to C primitives (via the `CCALL` bytecode instruction) may require an execution time that depends

on the values of the arguments of the function. The WCET of these calls might thus overestimate the actual execution time of the `CCALL` evaluation. Handling these primitive calls relies on another cost table that maps each C primitive name to its maximum cycle count. The costs of these primitives are hardware-specific and need to be provided by the platform developer. Each primitive cost should be computed using classical methods and tools for WCET analysis.

▶ **Example.** The excerpt of OCaml bytecode on the left side of figure 4 corresponds to the body of the step function created by compilation of the `count` node of figure 2. On the right side of this figure is displayed a table associating a cost to every bytecode instruction, which has been computed by *Bound-T* (configured for an AVR ATmega32U4 microcontroller). From this table, Bytecrawler computes a maximum cost of 2121 cycles for this function. For sake of simplicity, calls to I/O primitives (via the `CCALL` bytecode instruction) are not mentioned in this example. For instance, in a complete synchronous program, calls to input (resp. output) primitives happen in the beginning (resp. end) of each synchronous instant. These calls should thus also be taken into account.

```
(...)
69  ACC0
70  GETFIELD0
71  PUSHACC2
72  BRANCHIFNOT 75
73  CONST0
74  BRANCH 76
75  ACC0
76  PUSHACC0
77  OFFSETINT 1
78  PUSHACC0
79  PUSHACC4
80  SETFIELD0
81  ACC1
82  PUSHACC4
83  SETFIELD1
84  CONST0
```

Instruction	Cost (cycles)
ACC0	74
ACC1	74
CONST0	66
GETFIELD0	96
SETFIELD0	145
SETFIELD1	150
PUSHACC0	95
PUSHACC2	115
PUSHACC4	115
BRANCH	299
BRANCHIFNOT	315
OFFSETINT	301
(...)	(...)

Figure 4 OCaml bytecode (left) and instructions cost table (right).

4 Formalisation and correctness proof

In this section, we describe a formalisation of the inner working of the Bytecrawler tool. For this purpose, we first introduce an idealised bytecode language that serves as an illustration on a small subset of OCaml bytecode instructions. We then formalise how to compute an upper bound of the WCET for a program written with these instructions, and we prove the correctness of this computation. Our conjecture is that, because of the imperative nature of the OCaLustre programs, and because of the absence of dynamic memory allocation during the execution of a synchronous instant, the results over this idealised bytecode language can be transposed to the actual subset of the OCaml instructions coming from the compilation of an OCaLustre source code.

The specifications presented in this section have been developed with the *Ott* tool [18], and the various lemmas and theorems were proven using the *Coq* proof assistant [20]. The corresponding proof scripts are available at `http://stevenvar.github.io`.

The idealised bytecode language contains instructions for initialising variables (Init), assigning values to variables (Assign), arithmetic operations over integers (Sub and Add), branching (Branch and Branchif), as well as an instruction to terminate execution (Stop).

$$instr ::= \mathsf{Init}\ x\ v \mid \mathsf{Assign}\ x\ y \mid \mathsf{Add}\ x\ y \mid \mathsf{Sub}\ x\ y \mid \mathsf{Branch}\ v \mid \mathsf{Branchif}\ x\ v \mid \mathsf{Stop}$$
$$values, v, w ::= int\_litteral \mid v + w \mid v - w$$

A state σ of a program is a tuple (P, pc, M) that contains an array P representing the instructions of the program, a code pointer pc, and a memory M that associates any initialised variable name to its value. The small-step operational semantics of the language is defined on figure 5.

$$\frac{P[pc] = \mathsf{Init}\ x\ v}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v])} \qquad \frac{P[pc] = \mathsf{Branch}\ v}{(P, pc, M) \longrightarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Assign}\ x\ y \quad M[y] = v}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v])} \qquad \frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = 0}{(P, pc, M) \longrightarrow (P, pc+1, M)}$$

$$\frac{P[pc] = \mathsf{Add}\ x\ y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v+w])} \qquad \frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] \neq 0}{(P, pc, M) \longrightarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Sub}\ x\ y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v-w])}$$

Figure 5 Operational semantics of the idealised bytecode language.

▶ **Definition 1** (Execution trace). *An execution trace T is the sequence of states taken by the program, following the operational semantics of the language.*

During the actual execution of a program, the values of all variables are *known*, and the corresponding execution trace is unique. Moreover, when the computation terminates normally, the trace ends on instruction Stop. We call such a trace *deterministic*:

▶ **Definition 2** (Deterministic trace). *The* run *function computes the deterministic execution trace beginning with a state σ:*

$$\frac{P[pc] = \mathsf{Stop}}{run((P, pc, M)) = [(P, pc, M)]} \qquad \frac{\sigma \longrightarrow \sigma' \quad run(\sigma') = \mathcal{T}}{run(\sigma) = \sigma\ ::\ \mathcal{T}}$$

▶ Remark. Note that, by definition, a deterministic trace is always finite. Indeed, since it is a certainty (due to absence of loops or recursion) that the actual execution of a synchronous instant will terminate, we are only interested in finite traces.

We associate a cost to every language instruction with the $cost_{instr} : instr \to nat$ function. The cost of a transition is equal to the cost of the corresponding instruction, and the cost of a trace is the sum of the costs of all transitions:

$$cost_{step}(P, pc, M) = cost_{instr}(P[pc]) \qquad cost(\mathcal{T}) = \sum_{\sigma \in \mathcal{T}} cost_{step}(\sigma)$$

4.1 Erasure of variables

In order to compute an upper-bound for the execution time of an OCaLustre program, we reason over an *abstract* representation of the program which considers every possible path that the control flow can take. In this abstract representation, variables can hold *unknown* values, that represent the values that depend on the state of the program environment during execution. The grammar of the idealised bytecode language is extended accordingly, where unknown values are denoted by $\perp$:

$$values, v, w ::= int\_litteral \mid v + w \mid v - w \mid \perp$$

The memory of an abstract execution of a program might thus contain unknown values. We call such a memory an *erasure* of the actual memory of the program:

▶ **Definition 3** (Erasure). *A memory M' is an erasure of a memory M (written $M \searrow M'$) if M' and M share the same set of variables, but M' contains more unknown variables than M. The following induction rules formally define the erasure of a memory:*

$$\frac{}{\varnothing \searrow \varnothing} \qquad \frac{M \searrow M'}{M \cup x = v \searrow M' \cup x = v} \qquad \frac{M \searrow M'}{M \cup x = v \searrow M' \cup x = \perp}$$

By extension, if two states σ and σ' differ only by the fact that the memory of σ' is an erasure of the memory of σ, we will use the same notation: $\sigma \searrow \sigma'$ and say that σ' is an erasure of σ.

Since it depends only on the code pointer and the program instructions, the cost of a transition stays the same after an erasure of the memory of the program:

▶ **Lemma 4.** $\forall \sigma\ \sigma', \sigma \searrow \sigma' \Rightarrow cost_{step}(\sigma) = cost_{step}(\sigma')$

The `wcet` function of figure 3 computes the worst-case execution time of a program by always considering the maximum cost between two possible branches. In our formalism, this function is implemented by the $cost_{max}$ function as follows:

▶ **Definition 5** (Maximum cost). *The $cost_{max}$ function computes the maximum cost of a program:*

$$\frac{P[pc] = \mathsf{Stop}}{cost_{max}((P, pc, M)) = cost_{instr}(\mathsf{Stop})} \qquad \frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\ v \\ M[x] = \perp \\ cost_{step}((P, pc, M)) = c \\ cost_{max}((P, pc + 1, M)) = k \\ cost_{max}((P, v, M)) = k' \end{array}}{cost_{max}((P, pc, M)) = c + max(k, k')}$$

$$\frac{\sigma \longrightarrow \sigma' \quad cost_{step}(\sigma) = c \quad cost_{max}(\sigma') = k}{cost_{max}(\sigma) = c + k}$$

The main result of this paper can now be stated as follows: the maximum cost of a program, computed from an erasure of its initial state, is greater than the actual execution time of the program.

▶ **Theorem 6** (Correctness). $\forall\ \sigma\ \mathcal{T}, run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\mathcal{T})$

In particular, the $cost_{max}$ function can be applied to a memory where all variables are unknown (equivalent to the initial state of a synchronous instant) since this memory is an obvious erasure of all possible initial states. The result obtained is thus an upper bound of the execution times of all possible program runs.

The remainder of this section is devoted to the proof of this theorem.

4.2 Non-deterministic evaluation

The use of unknown values ($\bot$) induces a non-determinism on the evaluation semantics of the language: branching choices might depend on such variables, and thus it becomes statically impossible to guess which path would be the actual one. We thus introduce a *non-deterministic operational semantics* of the language which is a standard abstraction of the deterministic operational semantics where conditional branching is generalised to non-deterministic branching. The rules for this new semantics, denoted by $\rightsquigarrow$, are the same as the rules for the deterministic semantics, extended with the two rules given in figure 6 (and the obvious lifting of arithmetic operations to account for $\bot$).

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \bot}{(P, pc, M) \rightsquigarrow (P, pc + 1, M)} \qquad \frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \bot}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

Figure 6 Rules for non-deterministic branching.

▶ **Definition 7** (Non-deterministic trace). *A non-deterministic trace is a finite sequence of transitions following the non-deterministic semantics of the language.*

Because the non-deterministic semantics of the language is an extension of its deterministic semantics, a transition in the former stays the same in the latter. We say that it is *embedded* into the non-deterministic semantics:

▶ **Lemma 8.** $\forall\ \sigma\ \sigma',\ (\sigma \to \sigma') \Rightarrow (\sigma \rightsquigarrow \sigma')$

Moreover, after erasing a memory, any transition from the deterministic semantics is still possible (possibly among other possible non-deterministic transitions):

▶ **Lemma 9.** $\forall \sigma_1\ \sigma_2\ \sigma_1', (\sigma_1 \rightsquigarrow \sigma_2) \wedge (\sigma_1 \searrow \sigma_1') \Rightarrow (\exists\ \sigma_2',\ (\sigma_1' \rightsquigarrow \sigma_2') \wedge (\sigma_2 \searrow \sigma_2'))$

By combining the two previous lemmas, we state that any erasure preserves the transition when going from the deterministic semantics to the non-deterministic one:

▶ **Theorem 10** (Preservation). $\forall\ \sigma_1\ \sigma_2\ \sigma_1'$,
$(\sigma_1 \to \sigma_2) \wedge (\sigma_1 \searrow \sigma_1') \Rightarrow (\exists\ \sigma_2',\ (\sigma_1' \rightsquigarrow \sigma_2') \wedge (\sigma_2 \searrow \sigma_2'))$

As illustrated by figure 7, the previous embedding and preservation properties hold for traces.

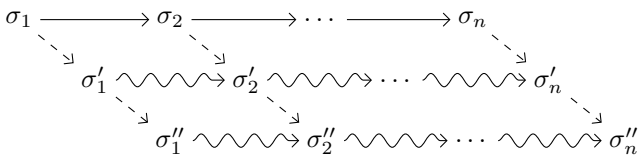


Figure 7 Trace preservation.

4.3 Maximum trace

In order to prove that $cost_{max}$ is the cost of the "most expensive" run of the program, we need to formally define the notion of a maximum trace. We thus defined the run_{max} function and proved that it actually computes a non-deterministic trace with the maximum cost.

▶ **Definition 11** (Maximum trace). *The run_{max} function is inductively defined as follows:*

$$\frac{P[pc] = \mathsf{Stop}}{run_{max}((P, pc, M)) = (P, pc, M) \ :: \ \varnothing} \qquad \frac{\sigma \longrightarrow \sigma' \quad run_{max}(\sigma') = \mathcal{T}}{run_{max}(\sigma) = (\sigma \ :: \ \mathcal{T})}$$

$$\frac{\begin{array}{c} P[pc] = \mathsf{Branchif}\ x\ v \\ M[x] = \bot \\ run_{max}((P, pc+1, M)) = \mathcal{T} \\ run_{max}((P, v, M)) = \mathcal{T}' \\ cost(\mathcal{T}) \leq cost(\mathcal{T}') \end{array}}{run_{max}((P, pc, M)) = ((P, pc, M) \ :: \ \mathcal{T}')} \qquad \frac{\begin{array}{c} P[pc] = \mathsf{Branchif}\ x\ v \\ M[x] = \bot \\ run_{max}((P, pc+1, M)) = \mathcal{T} \\ run_{max}((P, v, M)) = \mathcal{T}' \\ cost(\mathcal{T}) > cost(\mathcal{T}') \end{array}}{run_{max}((P, pc, M)) = ((P, pc, M) \ :: \ \mathcal{T})}$$

As expected, $cost_{max}$ and run_{max} are related by the following equality:

▶ **Lemma 12.** $\forall\ \sigma,\ cost_{max}(\sigma) = cost(run_{max}(\sigma))$

Using the trace preservation lemma, we derive the key property of this proof of correctness: the cost of the maximum trace is greater than the cost of every other finite trace, even when considering an erasure of their memories. In other words, the cost of the maximum trace computed from an erased state is greater than the actual cost of the program:

▶ **Lemma 13.** $\forall\ \sigma\ \mathcal{T},\ run(\sigma) = (\sigma :: \mathcal{T}) \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\sigma :: \mathcal{T})$

The combination of the previous lemmas allows us to conclude with our main theorem stating that the cost, estimated from an erasure of the initial memory, is an upper bound of the real cost of any actual run of the program:

▶ **Theorem 14** (Correctness). $\forall\ \sigma\ \mathcal{T},\ run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\mathcal{T})$

5 Conclusion, related and future works

We presented a virtual machine approach to microcontrollers programming that makes it possible to run programs developed with a high-level, multi-paradigms, programming language on devices with limited resources. This language, OCaml, is extended with a synchronous programming model which is adapted to the concurrent nature of embedded programs. In particular, our approach provides a tool, called Bytecrawler, that makes it possible to compute the WCET of a synchronous instant by reasoning over the bytecode instructions of the program, thus providing an automated, factorised method to compute the reaction time of a program. The aforementioned method has been formalised on a idealised language, and a proof of its correctness has been developed in the Coq proof assistant. To the best of our knowledge, a formalised and automated WCET estimation of OCaml bytecode has not been done before. Similar studies exist on other embedded virtual machines for different programming languages (in particular, multiple works have been done on Java bytecode [2][16][9]), although these results do not seem to have been formally verified. Some recent work on proving the correctness of a WCET estimation tool is available [12], but this tool is integrated within the CompCert C compiler, and thus operates only over C programs.

As a work in progress, we are adapting the formalisation and proof presented in this article to the actual, stack-based, language of OCaml bytecode instructions, in order to extract a certified implementation of Bytecrawler from Coq. As discussed in section 3, symbolic execution could improve the WCET estimate, but adapting it to the OCaml virtual machine might be difficult. Concolic execution [17], a hybrid model of execution that mixes *concrete* and *symbolic* execution, seems close to our method, and might also refine WCET estimation.

Finally, it is worth mentioning that our solution could quite easily be adapted to other kinds of analysis: for example, modifying the cost function to indicate the number of words allocated in memory by each bytecode instruction could provide an upper bound for the total amount of memory used by the program, and thus be an estimate of the maximum memory usage of any given OCaml program.

References

1. Clément Ballabriga, Julien Forget, and Giuseppe Lipari. Symbolic WCET computation. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):39, 2018. doi:10.1145/3147413.
2. Guillem Bernat, Alan Burns, and Andy J. Wellings. Portable worst-case execution time analysis using Java Byte Code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000), Stockholm, Sweden, June 19-21, 2000*, pages 81–88, New York, NY, USA, 2000. IEEE. doi:10.1109/EMRTS.2000.853995.
3. Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a Feature-rich VM for the Resource Poor. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems (SenSys 2009), Berkeley, CA, USA, November 4-6, 2009*, pages 169–182, New York, NY, USA, 2009. ACM. doi:10.1145/1644038.1644056.
4. Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. In *Proceedings of the 14th annual ACM Symposium on Principles of Programming Languages (POPL 1987), Munich, Germany, January 21-23, 1987*, pages 178–188, New York, NY, USA, 1987. ACM. doi:10.1145/41625.41641.
5. Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a Timing Anomaly? In *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012), Pisa, Italy, July 10, 2012*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2012.1.
6. Marc Feeley and Danny Dubé. PICBIT: A Scheme system for the PIC microcontroller. In *Proceedings of the 4th Workshop on Scheme and Functional Programming, Boston, MA, USA*, pages 7–15, 2003. URL: http://www.schemeworkshop.org/2003.
7. Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. Open Source FreeRTOS As a Case Study in Real-time Operating System Evolution. *Journal of Systems and Software*, 118(C):19–35, August 2016. doi:10.1016/j.jss.2016.04.063.
8. William Hallahan, Anton Xue, and Ruzica Piskac. Building a Symbolic Execution Engine for Haskell. In *Proceedings of the 8th Workshop on Tools for Automatic Program Analysis (TAPAS 2017), New York, NY, USA, August 29, 2017*, 2017. URL: https://cs.nyu.edu/acsys/tapas2017.
9. Trevor Harmon and Raymond Klefstad. A Survey of Worst-Case Execution Time Analysis for Real-Time Java. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), 26-30 March 2007, Long Beach, CA, USA*, pages 1–8. IEEE, 2007. doi:10.1109/IPDPS.2007.370422.
10. Niklas Holsti and Sam Saarinen. Status of the Bound-T WCET tool. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, Technical University of Vienna, Austria, June 2002. URL: http://www.cs.york.ac.uk/rts/wcet2002.

11 Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990. URL: https://hal.inria.fr/inria-00070049/file/RT-0117.pdf.

12 André Oliveira Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut. A Formally Verified WCET Estimation Tool. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), Ulm, Germany, July 8, 2014*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–20, Dagstuhl, Germany, 2014. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2014.11.

13 Corina S. Pasareanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), Antwerp, Belgium, September 20-24, 2010*, pages 179–180, New York, NY, USA, 2010. ACM. doi:10.1145/1858996.1859035.

14 Peter P. Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, 1989. doi:10.1007/BF00571421.

15 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET '06), Dresden, Germany, July 4, 2006*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2006.671.

16 Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES '06), Paris, France, October 11-13, 2006*, pages 202–211, New York, NY, USA, 2006. ACM. doi:10.1145/1167999.1168033.

17 Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, USA, November 5-9, 2007*, pages 571–572, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321746.

18 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

19 Vincent St-Amour and Marc Feeley. PICOBIT: a compact Scheme system for microcontrollers. In *Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (IFL 2009), South Orange, NJ, USA, September 23-25, 2009*, pages 1–17. Springer, 2009. doi:10.1007/978-3-642-16478-1_1.

20 The Coq Development Team. The Coq Proof Assistant, version 8.9.0, January 2019. doi:10.5281/zenodo.2554024.

21 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. Concurrent Programming of Microcontrollers, a Virtual Machine Approach. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016. URL: https://hal.archives-ouvertes.fr/ERTS2016.

22 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, 2017. URL: https://hal.archives-ouvertes.fr/JFLA2017.

23 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project. In *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, 2018. URL: https://hal.archives-ouvertes.fr/ERTS2018.

24 Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. Programming Microcontrollers in OCaml: The OCaPIC Project. In *Proceedings of the 17th International Symposium on Practical Aspects of Declarative Languages (PADL 2015), Portland, OR, USA, June 18-19, 2015*, pages 132–148. Springer, 2015. doi:10.1007/978-3-319-19686-2_10.

Validating Static WCET Analysis: A Method and Its Application

Wei-Tsun Sun
IRT Saint Exupéry, Toulouse, France
weitsun.sun@irt-saintexupery.com

Eric Jenn
IRT Saint Exupéry, Toulouse, France
Thales AVS, Toulouse, France
eric.jenn@irt-saintexupery.com

Hugues Cassé
IRIT, Toulouse, France
University of Toulouse, France
casse@irit.fr

Abstract

WCET analysis is a key activity in the development of safety critical real-time systems. Whether upper bounds on WCETs are obtained using static analysis or measurements, the confidence on the compliance of a system with its temporal requirements directly depends on the confidence on these estimations. Static WCET analysis based on abstract interpretation takes benefits from its formal foundations. However, it also strongly depends on the correctness of the underlying models. We hereby show how we have validated the version of the data flow static analyser of OTAWA applied to the AURIX TC275 target processor.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases validation of WCET tools, ISS, nML

Digital Object Identifier 10.4230/OASIcs.WCET.2019.6

1 Introduction

In a real-time system, correctness in the time domain is as important as correctness in the value domain. For instance, a sensor value (resp. a message) may be meaningless if it is acquired (resp. delivered) too early or too late. For a software system, demonstration of temporal correctness relies on the estimation of the application's Worst Case Execution Times (WCET).

WCET may be used to perform schedulability analysis or response time analysis, or to demonstrate compliance with the synchronous execution model hypothesis. WCET can be estimated empirically on the basis of measurements, or analytically on the basis of a model [19]. Among various WCET-estimation strategies, static WCET analysis relies on an appropriate abstraction of the application and execution platform to compute a safe upper bound of the actual WCET, by means of abstract interpretation [4].

Confidence on the results produced by static analysis directly depend on (1) the correctness on the processor model, and (2) the correctness of the analyses. In this paper, we present the method that we have used to validate these two elements for the Aurix TC275 static analyser for data flow. To ensure (1), we propose to compare the behaviour of an Instruction Set Simulator (ISS) generated from the processor model with a reference ISS or the actual platform. To ensure (2), we propose to compare the abstract states built by abstract interpretation with the concrete states generated from a reference ISS or the actual platform. Note that the proposed approach neither verifies all the analyses performed by the static

analyser, nor guarantees the absolute absence of errors for the analyses that are covered – it basically relies on testing –, but it does not require huge effort on modifying the existing framework, stays modular and can be automated.

This paper is organised as the follows: The background is presented in section 2; an overview of our approach is described in section 3; the application of our approach on the AURIX TC275 is detailed in section 4; the experimental results of the application are shown in section 5; the related and the on-going works are described in section 6; finally section 7 concludes the paper.

2 Background

This section introduces the OTAWA framework used to develop WCET static analysers, and the Sim-NML language used to describe ISA (Instruction Set Architectures).

2.1 OTAWA - an open source framework to compute WCET

OTAWA [3] is an open-source framework to build static WCET analysers. A typical WCET analyser is built on the following facilities provided by OTAWA: (1) the binary decoder used to extract the program instructions and other information contained in the executable file, such as the initial memory contents; (2) a set of static analyses based on abstract interpretation [4] to capture the behaviours of the underlying hardware; and (3) other components to aid the WCET computation, e.g. an ILP solver.

The binary decoder is generated by the the GLISS [13] tool from a description of the target processor's ISA written in Sim-NML [18]. An ISS (instruction-set simulator) is also generated from the same description.

Being open-source and its implementation structure, the framework can be easily extended to provide new capabilities, including the generation of any intermediate result of the static analyses. For instance, to support the validation activities presented in this paper, it was modified to produce the concrete processor states from the customised- and generated-ISS, and the abstract states produced by the abstract analyser. The same strategy can be applied to any other open-source WCET computation framework.

2.2 Sim-NML: the language to describe ISA and semantic instructions

OTAWA is designed to be platform-independent. To achieve this independence, the model of the hardware target – including its ISA and the behaviour of its instructions – are described in an external file using the dedicated language Sim-NML.

Listing 1 shows the ***mov*** instruction from the Tricore architecture described in Sim-NML. The first line declares that ***mov*** operates on two data registers (of type reg_d) named c and b. The second line describes the bit-level structure (or "image") of the instruction: the first 4 bits contains the ID of the register c, followed by the 8-bit value 0x1F, and so forth. An "X" indicates an unused bit. The third line describe the syntax of the operation in assembly code. Finally, the action segment describes the behaviour of the instruction. In this example, it assigns the value of register b to register c. This segment is used to generate the ISS using GLISS2, and to perform some analyses such as program slicing (i.e. to propagate the register values, the analyser needs to identify the registers that are written/read).

In order to make static analysis as independent as possible from the target platform, OTAWA translates each instruction into a set of more primitive ***semantic instructions*** [2]. Those semantic instructions must capture the behaviours of all instructions encountered in an execution scenario.

Listing 1 the structure of NMP that describes an instruction - mov.

```
op mov_reg (c:reg_d, b:reg_d)
 image = format("%4b %8b XXXX %4b XXXX %8b",c.image,0x1F,b.image,0x0B)
 syntax = format("mov %s,%s",c.syntax,b.syntax)
 action = { c = b; }
```

Listing 2 Provide the instruction type as an attribution to the mov instruction.

```
extend mov_reg
  sem = { SET(D(c.i), D(b.i)); }
```

Semantic instructions are added to the existing operation description as shown in Listing 2 using attributes. In this example, the semantics of ***mov*** is expressed using the ***SET*** semantic instruction; it states that register c shall get the value of register b. Note that the register naming follows the declaration of the original operation (as shown in the second line).

Currently, the descriptions for each instruction is still produced manually, and this can be an error-prone process. Errors may affect the selection of the registers (e.g., specifying register d as the target instead of register c) or the translation into the semantic instructions (which would not correctly capture the meaning of the processor instruction)[1].

3 The proposed approach

Out of our experience, errors are essentially introduced (i) during the modelling of the processor's ISA, and (ii) during the implementation of the static analysis based on abstract domains. To identify errors introduced in (i), we propose to cross-check the ISS (instruction-set simulator) provided by the manufacturer (or by any other trusted source) with the ISS generated from the processor ISA model in Sim-MNL. To identify errors introduced in (ii), we propose to cross-check the abstract states computed by the abstract interpreter against the concrete states computed by some trusted ISS or by the ISS generated out of the processor ISA model.

3.1 Verification of the processor ISA model

Figure 1 shows the different steps necessary to verify the processor ISA model. In our case, the model is crafted using the NMP format based on Sim-NML [18]. It is then fetched to an ISS generator, e.g. GLISS2 [13], to generate an ISS automatically.

The verification process consists of executing one or several test programs (with their stimuli) on both the real processor (or some trusted ISS, such as the Infineon's TSIM for the TC275) and the generated ISS, and to compare the execution traces obtained from the two executions. Execution traces contain the successive states of the processor registers and the memory contents. If both traces are identical, the processor model and the actual processor are deemed equivalent (for the test programs considered). Otherwise, the execution traces are investigated to find the causes of the discrepancy(ies), and to correct the processor Sim-NML model. This is an iterative process that stops when equivalence is shown for all test programs.

[1] An automatic translation from the behaviour description of an operation to semantic instructions is relatively complex and not implemented in OTAWA: actual machine instructions are often very complex and provides much more details than required by dataflow analyses.

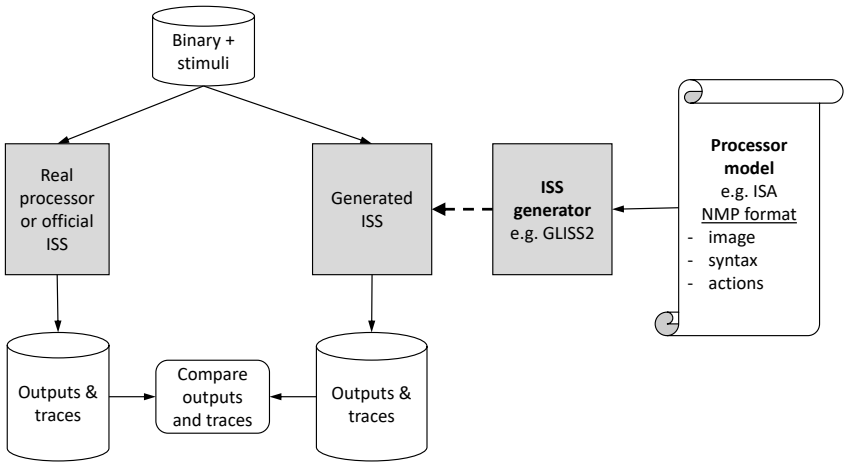


Figure 1 Verification of the processor model.

3.2 Verification of the static analyser

Abstract interpretation proceeds by executing a program in an abstract domain that preserves the properties of interest. This interpretation relies on the semantics of the instructions given in the ISA model which verification has been described in the previous section.

The static analysers interpret all the instructions of a given binary to determine the successive states of the execution platform components, including registers, memories, etc. To verify that the analysis is correct, we propose the approach depicted in Figure 2.

The program object code (binary) is processed by both the static analyser and the actual processor (or an ISS). They produce a series of abstract states and concrete states, respectively. Comparison between the two traces is performed at specific "comparison points" corresponding to specific program locations. Comparison can be done at "fine-grain" (e.g., at each instruction) or at "coarse-grain" (e.g., at the end of each function call). For each comparison point, the concrete state is checked against the corresponding abstract state. Abstraction is deemed correct if, for all comparison points, the abstract state is a correct abstraction of the corresponding concrete state. In Abstract Interpretation, an abstract state is sound at a particular program point if it includes any valid concrete state generated at this point. Note that, as for the previous verification, validity is demonstrated with respect to one or several input programs and on or several input stimuli for each program, so, the quality of the verification depends on the "coverage" of the program set and input stimuli.

4 Applying the proposed approach for support AURIX TC275 on OTAWA

Applying the proposed method on the AURIX TC275 is illustrated in Figure 3, where each step of the method is numbered. The figure is partitioned into two parts: the light gray area covers the validation of the processor model, such as the instruction decoding and branch-target identification. The dark grey area covers the validation process for any static analysis used in the overall WCET estimation process.

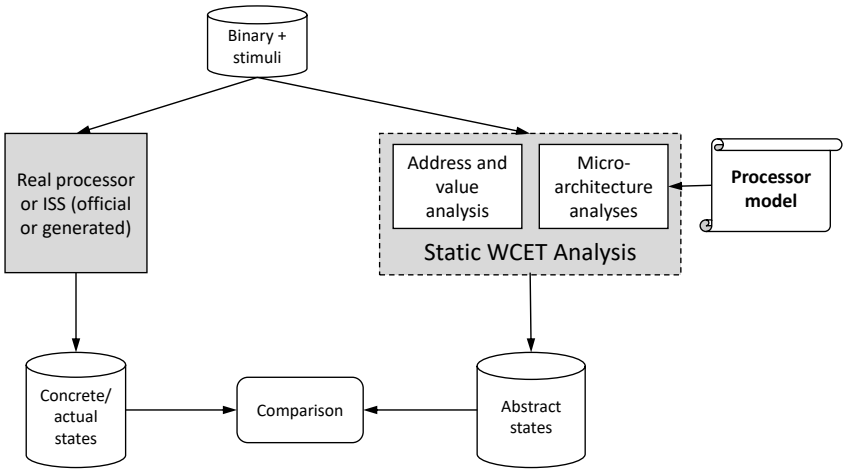


Figure 2 Verification of abstract states versus the concrete states.

4.1 Verification of the TC275 processor ISA model

In the first phase, the NMP files are created (1) to capture the description of the ISA, which consists of the decoding information of the instructions as well as their behaviour (the semantic instructions) to enable architecture-independent static analyses. The content of the NMP files are created out of the documentation provided by the chip manufacturer. For the TC275, for instance, the instruction timings are extracted from the data-sheet [1] and the instruction formats and behaviours are obtained from the instruction-set user manual [17].

In the second phase, the GLISS2 tool uses the NMP files to generate a library for instruction decoding and a standalone ISS (3).

To validate the NMP model, a program binary is loaded and executed by both the generated ISS and a reference execution structure (simulator or actual processor). In our case, the reference is the TSIM ISS provided by Infineon (4). The two execution traces (i.e. the sequence of the instructions and their addresses, the register values, and the accessed memory content) are then compared (5). Any discrepancy found during the comparison shall lead to the correction of the ISA description and to a new iteration of the process.

4.2 Verification of the abstract interpreter for the TC275

A database of operations (IRG file) is generated by processing the ISA description using the GLISS2 tool (6). The semantic instructions are added to operations through the use of the ***extend*** keyword (7) as presented in Section 2.2. The target binary file is then given as input to OTAWA for WCET estimation (8).

As the first step of the WCET estimation, OTAWA processes the binary thanks to the instruction decoder provided with the ISS (see (3)). The structural elements of the program – the CFGs (control-flow graph) and BBs (basic blocks) – are also identified in this step. Then the corresponding semantic instructions are interpreted, the abstract states are computed and associated with each BB.

Finally, an ILP problem is created with an objective function composed from the results of the aforementioned analyses and the constraints from the program structure. The solution of the ILP problem determines the WCET.

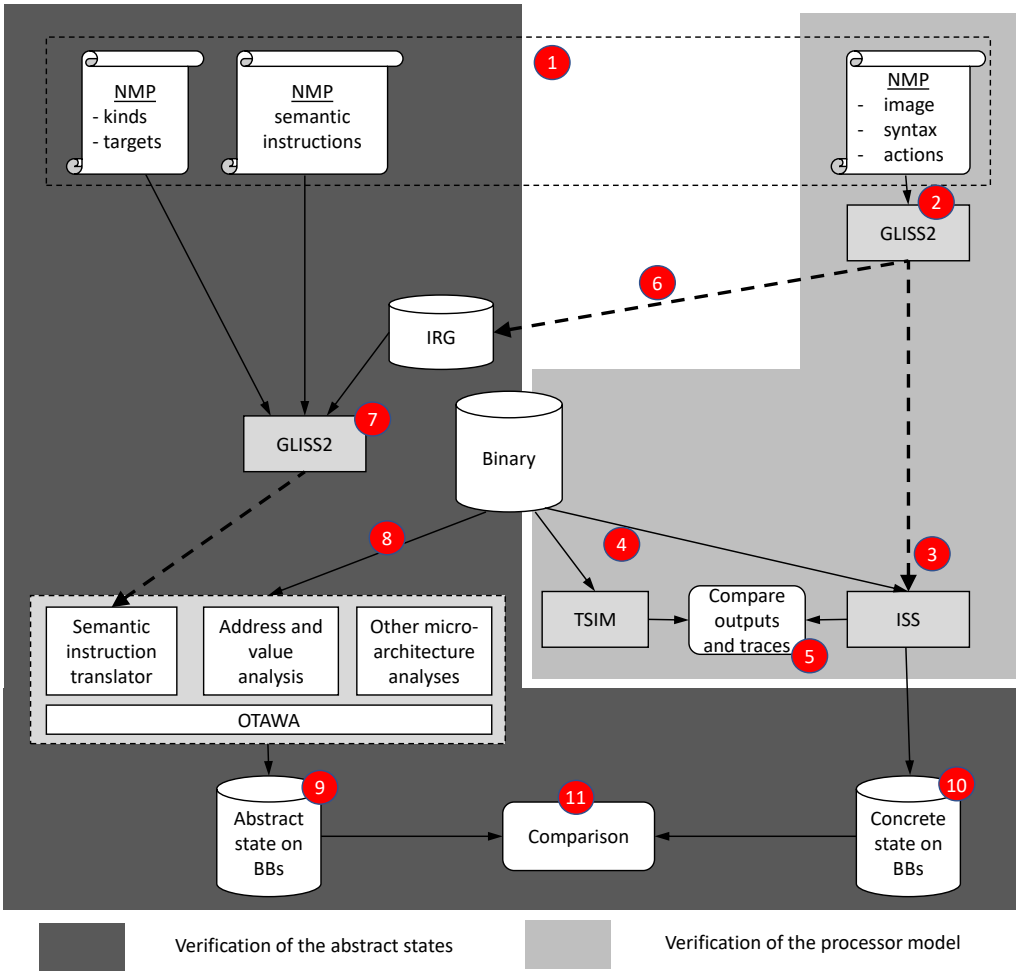


Figure 3 The overall workflow that combines both validation approaches.

To validate the static analysis, the abstract states associated with each BB are extracted (9). At each check point, e.g., the end of a BB, the concrete state (10) is compared with the abstract state (11). If the concrete state is part of the abstract state, then we consider that the verification has passed for this given binary. Verification shall be performed on several binaries to improve confidence. Similar to the verification process of the processor ISA model, the validation of the static analyser is also iterative: for any discrepancy detected, a correction is applied, and the verification process is performed again.

Address/value analysis in OTAWA uses the CLP abstract domain [15, 2]. As the CLP abstract domain is used for several subsequent analyses as data-cache analysis (to know which address the program tries to access), dynamic-branch resolution [16], etc., its soundness and precision is crucial for the WCET estimation. The CLP abstract states consist of the values of each register and each accessed memory location expressed in the CLP domain. The evolution of the abstract states are based on the interpretation of the semantic instructions of the given program.

An example of verification is depicted on Figure 4. The target program consists of a single if-else statement. Abstract states (D_A) are displayed at the end of each BB. Only the values of the data register D1 are shown for simplicity. The abstract state in CLP domain

is represented as a range. At the end of BB1, D1 contains value [1]. Since static analysis considers all possible scenarios, both execution paths to BB2 and BB3 are taken into account, which leads to two different abstract states: [0-2] at the end of BB2, and [1-5] at the end of BB3. These states are ***joined*** at the beginning of BB4. The resulting state is [0-5] which encompasses both ranges. As D1 is not modified in BB4, this is also the value of D1 at the end of the program.

The concrete states of D1 (i.e., the values computed during a concrete execution) are denoted by D_C on the figure. Note that there is no concrete state associated with BB2 because only the right branch (leading to BB3) was executed during the test. Whenever the execution reaches the end of a BB, the current concrete state is compared with the corresponding abstract state, e.g. for BB 3, D1 = 3 which falls in the range [1-5], so the test passes. The validation of the address/data analyses provides two "guarantees": (1) the correct translation of semantic instructions in the NMP file, and (2) the correct abstract interpretation.

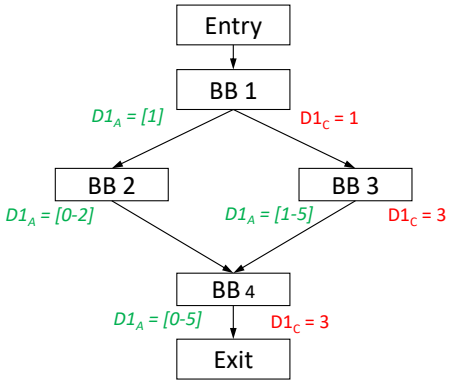


Figure 4 An example of checking if the abstraction is satisfied.

5 Experiments and results

Our approach has been applied using two sets of input programs: a subset of the Mälardalen benchmarks suite [8], and the set of software components of our robotic demonstrator (TwIRTee). The coverage of the verification, expressed by the ratio between the number of different operations executed in the programs, and the number of possible operations s (i.e., all the combinations of the opcodes, operands, and addressing modes, described as different "op" in the NMP file). Due to the space limitation (35 benches from Mälardalen), partial results are shown in Table 1.

There are 814 opcode-operand combinations (instruction types) in total for the TC1.6 (the CPU family used in TC275), As for any test-based verification approach, the test coverage – and so the confidence on the verification results – is highly dependant on the test inputs. In particular, the complexity and code size of an individual benchmark does not have direct relationship with the coverage (the number of different opcode-operand combination used). For instance, program ***adpcm*** is around one-third smaller than program ***nsichneu*** but it gives a higher coverage. Similarly, program ***bs*** executes around 3 times less instructions than ***nsichneu***, but gives a similar coverage. Moreover, all benchmarks share a common sequence

of instructions (the function ***__start*** which initialises the context area for function call and stack) which can represent a significant part of the instruction coverage of a simple program (this explains why the very simple binary-search application ***bs*** covers 80 combinations).

Mälardalen benchmarks cover 191 combinations (23.46%) while the TwIRTee robotics application alone covers 209 combinations (25.68%). This tends to show that using actual application code (involving a greater variety of algorithms) allow achieving a higher coverage level. Together, Mälardalen benchmarks and TwIRTee only cover 28.64% of all combinations. This may indicate that the compiler privileges a small set of opcode-operand combinations. This limit could be overcome by directly synthesising assembly code test programs.

Table 2 gives the number of errors found in the TC275 NMP files using our method. For each test, nearly 1/3 of the covered opcode-operand combinations were erroneous. Even though the ratio of errors are similar among the tests, errors are not correlated. Common errors are: (1) wrong bit-ordering, i.e. a bit range "m to n" is replaced by "n to m", or an erroneous bit-index is used; (2) incorrect sign-ness; (3) register-type error, e.g. using a data register in place of an address register; and (4) mis-interpreting the ISA from the user manual. The mappings of the semantic instructions are prone to the following errors: (1) using the content of a register as a value rather than an index; (2) mis-interpreting the behaviours of instructions.

Concerning the CLP analysis, few errors were detected, including one affecting the implementation of the widening operation between two specific CLP ranges: one "corner case" had not been considered. As a by-product of the validation process, some improvements of the accuracy of the analysis were also identified, when applying the fixes to the errors found, where the re-structuring of the existing codes. This includes the implementation of the bit-wise AND operation between two CLP values. For instance, applying a bit-wise AND with a range [-0xFF, 0xFF] and a constant 0xFF was producing the range [-0xFF, 0xFF]. While safe (i.e., all possible values were covered), this range contains negative values (sign-extended); after modification, the result is now [0, 0xFF].

Table 1 Verification of the ISA through ISS executions.

Name	Size (Byte)	Instructions executed	Covered op types (/814)	Coverage (%)
adpcm	17,624	246,510	95	11.67%
bs	10,776	4,867	80	9.83%
fft1	13,364	83,052	148	18.18%
nsichneu	48,284	15,140	80	9.83%
Mälardalen (all)	419,728	2,795,331	191	23.46%
twIRTee	50,194	193,487,525	209	25.68%
Mälardalen + twIRTee			225	27.64%

Table 2 Results of detection.

Type	Error found (full / 814, tested)
ISA model	67 (8.2%, 29.8%)
CLP analysis	71 (8.7%, 31.5%)

6 Related and on-going works

The soundness verification of analyses involved in WCET calculation is a relatively old topic. In 2001, Ferdinand et al. [7] presents the main principles of their tool suite, named later *aiT*, as being sound-by-construction. This claim concerns mainly the use of static analysis applied to obtain the WCET and is supported (a) by a rich scientific amount of surveys devoted to the formal definition and verification of analyses based on Abstract Interpretation and (b) by the automatic generation of analyses using a tool named PAG. Yet, the authors admits that the architecture model, provided by hand, weakens the approach. More recently, [11] proposes a verified tool that is able to compute WCET using standard IPET method. The tool is embedded in CompCERT [10] but only targets high-level analyses – value, loop bounds and ILP generation, and the issue of hardware verification is not treated.

In [14], Schlickling *et al.* present an approach to derive a timing model[2] usable in a WCET tool from a pipeline description in VHDL. The approach involves several passes of dead-code or useless code elimination combined with abstractions driven by human interaction. The approach is promising but (a) needs reduced but necessary human action and (b) requires the VHDL model of the pipeline. A similar approach might be applied to automatically extract semantics of instructions but we are not aware of any work in this direction.

There are alternatives to OTAWA's semantic language. ALF [9] was designed as an independent language to represent semantics of binary programs. Although several translators from binaries to ALF exist, we are not aware of any work towards verification. CRL2 is used in aiT [6] to represent the program whatever the underlying machine code but very few information about it are available and specially about the support of instruction semantics.

Currently the ISA model (used in instruction decoding) and the semantic instructions (used in the CLP analysis) are constructed separately. This prevents the bugs in the ISA model being entailed in the semantic instructions. The human-error is the main cause when creating both models, in particularly, i.e. two times of efforts are made to create models which are relevant. An on-going work is to create a single model (e.g. ISA) and then the mapping of semantic instructions will be performed automatically. To increase the coverage of our validation process, using benchmarks such as Papabench [12] and systematic constructed programs [5] is also under the investigation.

7 Conclusions

In this paper we have proposed and applied a method to verify some important components of a static WCET analyser: the ISA model and the associated semantic model used by the abstract interpreter, and some parts of the abstract interpreter itself. The verification process is achieved by cross-checking the result produced by a reference implementation of the target processor (a simulator or the actual hardware) and some intermediate results produced by the WCET analysis tool. This approach has been applied on the WCET analyser for the TC275 processor implemented using OTAWA and leads to the detection of numerous errors in the TC275 ISA model and associated semantic model, and a few errors in the abstract interpreter.

[2] The obtained timing model is mainly an abstract simulator of the microprocessor states.

References

1 AURIX TC27x D-Step 32-Bit Single-Chip Microcontroller User's Manual V2.2 2014-12.

2 Hugues Cassé, Florian Birée, and Pascal Sainrat. Multi-architecture value analysis for machine code. In *13th International Workshop on Worst-Case Execution Time Analysis*, pages pp–42, 2013.

3 Hugues Cassé and Pascal Sainrat. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software*, volume 1, 2006.

4 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM symposium on Principles of programming languages*, pages 238–252, 1977.

5 Thanh Nga Dang, Abhik Roychoudhury, Tulika Mitra, and Prabhat Mishra. Generating test programs to cover pipeline interactions. In *2009 46th ACM/IEEE Design Automation Conference*, pages 142–147. IEEE, 2009.

6 Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

7 Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *International Workshop on Embedded Software*, page 469–485. Springer, 2001.

8 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

9 Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF-a language for WCET flow analysis. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

10 Xavier Leroy et al. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 53, 2012.

11 André Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut. A formally verified WCET estimation tool. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

12 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

13 Tahiry Ratsiambahotra, Hugues Cassé, and Pascal Sainrat. A versatile generator of instruction set simulators and disassemblers. In *2009 International Symposium on Performance Evaluation of Computer & Telecommunication Systems*, volume 41, pages 65–72. IEEE, 2009.

14 Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. In *ACM Sigplan Notices*, volume 45, 4, pages 67–76. ACM, 2010.

15 Rathijit Sen and YN Srikant. Executable analysis with circular linear progressions. Technical report, Technical Report IISc-CSA-TR-2007-3, Computer Science and Automation Indian . . . , 2007.

16 Wei-Tsun Sun and Hugues Cassé. Dynamic branch resolution based on combined static analyses. In *16th International Workshop on Worst-Case Execution Time Analysis-WCET 2016*. OASICs, Dagstuhl Publishing, 2016.

17 TriCore™ TriCore™ V1.6 Microcontrollers User Manual (Volume 2).

18 Johan Van Praet, Dirk Lanneer, Werner Geurts, and Gert Goossens. nML: A structural processor modeling language for retargetable compilation and ASIP design. In *Processor Description Languages*, pages 65–93. Elsevier, 2008.

19 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.