Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs*

Boris Dreyer[1], Christian Hochberger[1], Alexander Lange[3], Simon Wegener[2], and Alexander Weiss[3]

1 **Fachgebiet Rechnersysteme, Technische Universität Darmstadt**
{dreyer,hochberger}@rs.tu-darmstadt.de
2 **AbsInt Angewandte Informatik GmbH**
wegener@absint.com
3 **Accemic GmbH & Co. KG**
{alange,aweiss}@accemic.com

Abstract

Traditionally, the Worst-Case Execution Time (WCET) of Embedded Software has been estimated using analytical approaches. This is effective, if good models of the processor/System-on-Chip (SoC) architecture exist. Unfortunately, modern high performance SoCs often contain unpredictable and/or undocumented components that influence the timing behaviour. Thus, analytical results for such processors are unrealistically pessimistic. One possible alternative approach seems to be hybrid WCET analysis, where measurement data together with an analytical approach is used to estimate worst-case behaviour. Previously, we demonstrated how continuous evaluation of basic block trace data can be used to produce detailed statistics of basic blocks in embedded software. In the meantime it has become clear that the trace data provided by modern SoCs delivers a different type of information. In this contribution, we show that even under realistic conditions, a meaningful analysis can be conducted with the trace data.

1998 ACM Subject Classification C.4 Performance of Systems, D.2.4 Software/Program Verification

Keywords and phrases Hybrid Worst-Case Execution Time (WCET) Estimation for Multicore Processors, Real-time Systems

Digital Object Identifier 10.4230/OASIcs.WCET.2016.<first-page-number>

1 Introduction

In previous work [8], we showed a novel approach for hybrid execution time estimation. Its main features are the precision that we achieve by taking typical cache behaviour into account, the continuous nature of the FPGA-based online aggregation and the non-intrusiveness, as we exploit the hardware tracing mechanisms of modern state-of-the art SoCs.

One of the underlying techniques used to implement this approach is the notion of the control flow graph. A control flow graph consists of basic blocks – sequences of instructions, where each instruction except the first and the last has exactly one predecessor and one successor – and edges that describe the flow of control in a program, i.e. conditionals, routine calls, loops etc. However, the embedded trace unit (ETU) of modern ARM processors (like the Xilinx Zynq featuring an ARM Cortex-A9) is not fully compatible with this model.

* This work was funded within the project CONIRAS by the German Federal Ministry for Education and Research with the funding ID 01IS13029. The responsibility for the content remains with the authors.

The mental model of the ETU is as follows: For each non-linear control flow, for example interrupts and hardware exceptions, but also normal branches and calls, a so-called waypoint event is emited. These events carry the address where the control flow change happened and the target of the change. Some instructions (the waypoint instructions) always generate a waypoint event [2]. Amongst others, all instructions that possibly modify the program counter are waypoint instructions. This is enough to fully reconstruct the control flow, but less fine grained than the control flow graph.

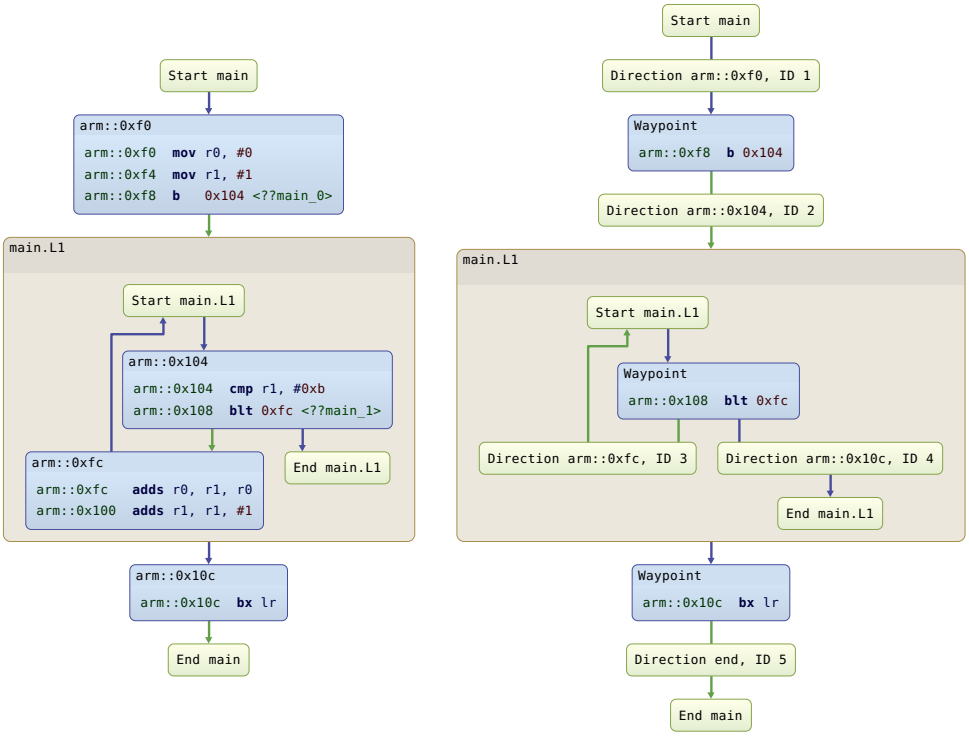

Figure 1 Control flow graph of a small program containing a simple loop (left) and its associated waypoint graph (right).

Consider Figure 1. It contains a control flow graph on the left and its associated waypoint graph on the right. On the left, inside the loop `main.L1`, two basic blocks are shown. The second one, starting at address `0xfc`, does not contain any change-of-flow instruction, but performs always a fall-through to the basic block at `0x104`. Consequently, no waypoint instruction exists that represents this second basic block, but only one for the first basic block (the instruction `blt` at address `0x108`). Each outgoing edge of a waypoint instruction is annotated with the target address given by its waypoint event. Hence we can distinguish the two possible paths through the loop.

To cope with the changed setting, we had to rework large parts of our approach presented in [8]. We spend higher effort in the preprocessing phase, but we are rewarded by a simplified implementation of the runtime phase. This paper presents the changes that we made to our former approach in order to achieve precise continuous non-intrusive measurement-based execution time estimation for waypoint graphs.

The paper is structured as follows: First, in Section 2, we discuss related work. We continue with a recapitulation of our method's workflow in Section 3. Then, in Section 4,

we discuss different hardware tracing units, the quality of the trace they produce and their usefulness for our approach. Afterwards, in Section 5, we highlight the changes between our revised approach and our original approach. Moreover, we introduce a new tool to determine in which context an instruction sequence is executed, the so-called loop automata. We continue with an evaluation of our approach on the TACLeBench benchmark suite [9] in Section 6. Finally, we conclude our work and discuss future work in Section 7.

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 [14]. 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. Probabilistical 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 [6], which combine the measured execution times of small code snippets to form an overall execution time estimate. 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 and 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.

Stattelmann et al. [13] propose the use of context information in order to account for cache effects. Their work shows that the inclusion of context information leads to more precise execution time results. However, their approach is limited to processors with sophisticated trace trigger mechanisms and performs again an offline analysis of the collected data.

Most measurement-based methods suffer from one or the other problem mentioned above. Our approach, in contrast, circumvents these drawbacks:

- We measure the timing of short instruction sequences. This fine grained approach allows to see where time is spent.
- We use non-intrusive hardware tracing mechanisms of state-of-the-art processors to produce timestamps. The probe effect is avoided.
- We process the trace events online. There is no need to store huge amounts of trace data.
- We process the trace events continuously. The aggregation can literally run for weeks. The possibilities to catch rare circumstances are increased.
- We incorporate the execution context of instructions and account for typical cache behaviour. The results are thus much more precise.
- The use of an FPGA allows us to adapt our method to different hardware tracing units.

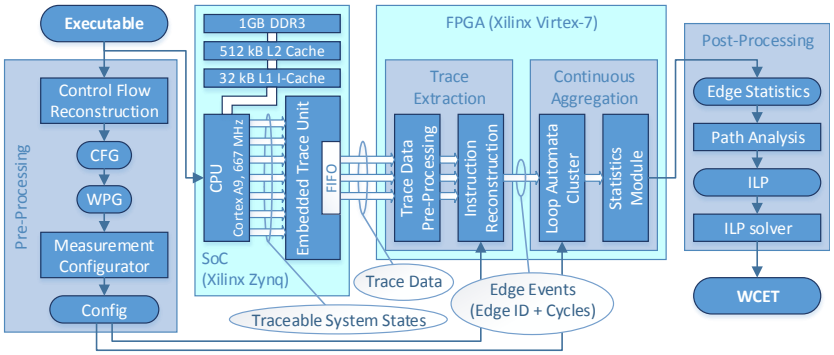

Figure 2 Workflow. Our approach is splitted into three phases: an offline pre-processing phase, the continuous online aggregation phase and an offline post-processing phase.

3 Workflow

Our method works on the object code level and is split into three phases: an offline pre-processing phase, the continuous online aggregation phase and an offline post-processing phase. The workflow of our method is shown in Figure 2. The control flow reconstruction and the ILP-based path analysis are re-used from the `aiT` tool chain [1].

We assume that a set of tasks is distributed over the cores of a multicore processor such that each task runs on exactly one core. Each task uses its own trace extraction and continuous aggregation modules. Hence it suffices to describe the workflow for a single core.

Control Flow Reconstruction and Waypoint Graph Computation First, the binary reader disassembles a fully linked binary executable into its individual instructions. Architecture specific patterns decide whether an instruction is a call, branch, return or just an ordinary instruction. This knowledge is used to form the basic blocks of the control flow graph (CFG).

Then, the control flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, then the user needs to write some annotations to guide the control flow reconstruction.

After finishing the reconstruction of the CFG, the waypoint graph (WPG) is computed. To do so, a pattern matcher checks for each instruction whether it is a waypoint instruction. Afterwards, the edges of the WPG are computed. For each waypoint instruction found, the algorithm follows the edges in the CFG to find reachable waypoints. This gives the direction of a waypoint edge and its target.

Configuration of the Continuous Online Aggregation Then, the WPG is used to create a configuration for the trace extraction module as well as for the continuous online aggregation module on the FPGA. This configuration assigns an unique ID to each edge in the waypoint graph. Moreover, it instantiates the lookup tables in the loop automata cluster (see Section 5 for a detailed description).

Trace Extraction During the program's execution, the ETU continuously emits raw trace data. This stream of data is fed into the trace extraction module. There, the raw data is decoded and compiled into an event stream. An event is generated for each traversal of a waypoint and consists of an ID and a timestamp. The special ID 0 is used if the waypoint

does not belong to the WPG computed during the pre-processing phase. This happens for example in case of an interrupt. Otherwise, the ID from the module's configuration is used. The resulting event stream is then fed into the continuous aggregation module.

Continuous Context-Sensitive Aggregation To achieve precise results, it is important that the aggregation module accounts for cache effects. Typically, the first iteration of a loop needs more time than the subsequent iterations because the instruction cache is not yet filled. Simply aggregating all loop iterations in the same record would thus most probably overestimate the time spend in all iterations but the first. For well-formed loops, we thus compute two statistical records for each edge belonging to a loop, one that aggregates the execution times in the first iteration and another that aggregates the execution times in all subsequent iterations, i.e. we take the execution context into account. This resembles some kind of virtual loop unrolling. If a basic block is part of nested loops, we only discriminate the iterations of the innermost loop, due to limited storage for the statistical records.

The ID of an event is used as input for the loop automata cluster. Each automaton in the cluster performs one step. Then, their state is used to decide whether a loop is executed and if it is the first iteration of the loop or already a later one.

The timestamp of an event is used to measure the execution time of the code snippet represented by the waypoint edge. Various statistics (minimal observed execution time, maximal observed execution time, count of executions) are updated each time an edge event is processed. The ID together with the execution context computed in the loop automata cluster form the index in the memory of statistical records.

Post-Processing and Path Analysis After the program has finished (or the test engineer has collected enough data), the post-processing phase is started by downloading the statistics from the FPGA's memory. Subsequently, the WPG together with the edge timing statistics are used to construct a maximisation problem encoded as an integer linear program (ILP). Solving this ILP gives a path with maximal execution time (and consequently, an estimate of the worst-case execution time).

Finally, the computed path is visualised for the user. An edge is marked infeasible if no statistics have been created for it. This information can be used to detect dead code. Moreover, the WCET contribution of the individual parts of the program is visualised. That way, the test engineer can see where in the program the hot spots are. This is particularly useful if the program is the target of performance optimisations.

4 Embedded Trace Units

The measurability of the execution time of instruction or basic blocks is a precondition for the proposed hybrid WCET measurement approach.

The traditional software instrumentation methodology with its obvious change of the systems behaviour (application blow up, execution slow down) is inappropriate for this measurement. In lieu thereof ETUs are implemented in the SoC. An ETU observes the SoC internal states, compresses and outputs this information via a dedicated high-bandwidth trace port. There are several ETU implementations available, which differ in the type of trace information and the compression efficiency (see Table 1). The most important ETU implementations are Nexus 5001™ [11] (for instance within the NXP Qorriva/QorIQ devices [10]) and the ARM CoreSight™ architecture [4].

ETU type	Nexus 5001™ [11]		ARM CoreSight™			
Implementation	Traditional branch messages	Branch history messages	ETMv3 [3]		ETMv4 [5]	PFT [2]
Program Flow Observation Level	Branch	Branch	Instruction		Branch	Branch
Cycle count	Yes	No	No	Yes	Yes	Yes
Applicable for hybrid WCET measurement	Yes	No	No	Yes	Yes	Yes

Table 1 ETU overview and its applicability for hybrid WCET measurement

The processor can possibly generate more trace data than the SoC's trace port can output at a given time. Therefore, the ETU includes a FIFO to buffer trace messages. The trace processing unit has to be able to handle the overflow of the ETU FIFO if a large volume of trace messages is generated (e.g. at narrow loops with high branch frequency).

5 Revised Method

This section presents a revised version of our approach for hybrid measurement-based timing analysis [8]. The original version of this approach was based on basic blocks, therefore the trace extraction unit had to emit basic block events. These events were also used to determine the execution context of the measured basic blocks, and to compute statistics over these blocks. By considering the execution context of the basic blocks, two statistics per basic block were computed: one containing the execution times of the basic block during the first iteration of its innermost surrounding loop (cold cache) and one containing all subsequent iterations (hot cache).

It turned out that there are processor architectures on the market for which we cannot reconstruct the basic blocks because not enough information is available in the stream of trace data. So we had to revise our architecture to use waypoint edge events instead of basic block events. The module that determined the context of executed instructions based on basic block events had to be replaced by new a one that uses waypoint edge events. This new module is called loop automata cluster and the central point of our revised work. It determines the context of each instruction based on a set of finite state machines and will be further described in this section. Only a few changes had to be made to the original statistics module to be compatible with the new loop automata cluster.

Loop Automata Cluster The loop automata cluster has the purpose to determine the context of each executed instruction, so that the statistics module can compute context-sensitive statistics. We define the context of an instruction by the context of its innermost surrounding loop. The context of each loop of an application can be determined by interpreting the waypoint edge event stream emitted by the trace extraction module [2]. For this interpretation WPG information is required because the event stream contains only the waypoint ID and the cycle count.

We model each loop of an application by two finite state machines (FSM) and four comparator trees. Figure 3 illustrates one set of four comparator trees that are used to translate the waypoint edge IDs of the event stream into loop specific context change events, namely `enter` (the loop has been entered), `reenter` (the loop has been iterated), `exit` (the

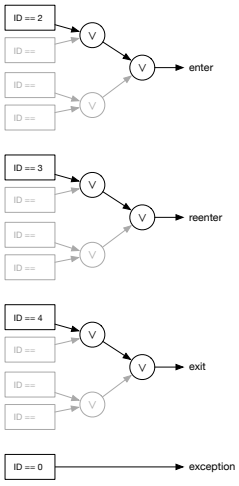

Figure 3 The instantiated comparator trees for the loop in Figure 1.

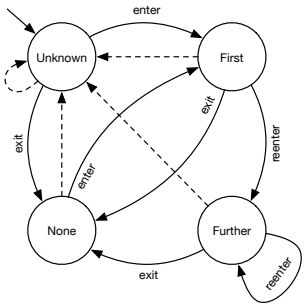

Figure 4 Finite state machine that reflects the different loop contexts.

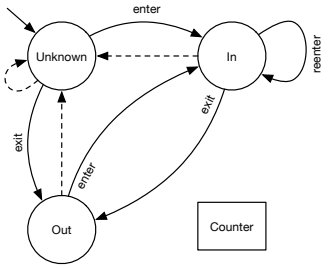

Figure 5 Finite state machine that counts the iterations of a loop.

loop has been exited), and `exception` (knowledge about the loop's context have been lost). The compare values of these loop specific comparator tree sets can be extracted from the WPG of the application.

Besides the comparator trees we use FSMs to store loop information. The first FSM gives information about the context of the loop and is illustrated in Figure 4. Its states reflect the different contexts of a loop, namely `None` (the loop is not executed), `First` (the loop is in its first iteration), `Further` (the loop is at least in its second iteration), and `Unknown` (no knowledge whether the loop is executed or not). If the FSM is in state `First`, the statistics for the first iteration of the waypoint are updated. If the FSM is in state `Further`, the statistics for all subsequent iterations are updated. If waypoint edge events have been lost during the trace extraction, e.g. because trace buffers within the processor have been overflowed, it can not be determined whether the loop is executed in the first or further iterations or not. In this case the FSM is in state `Unknown` and both statistics of a waypoint are updated to further maximize its WCET.

During program execution, several loops can be in their first or further iteration, due to nested loops. In this case, the context of the innermost loop determines the context of the waypoint edge events. For this, we use a stack to track the innermost loop during runtime.

The second FSM gives information about the iteration count of the loop and is depicted in Figure 5. It consists of tree states, namely `Out` (the loop is not being executed), `In` (the loop is being executed), and `Unknown` (it is not known if the loop is being executed or not). If the loop is not executed, the FSM is in state `Out` and the iteration counter is zero. Once the loop is executed the state changes to `In` and the counter is set to one because we count the executions of the loop header. Each time the FSM is in state `In` and a `reenter` event occurs the counter is incremented by one. As soon as the machine changes it state from `In` to `Out` the counter value is considered as performed loop iterations and the loop bounds statistics for this loop are updated.

It is possible that a trace analysis starts after the program execution has been stated. Consequently, there is a lack of loop context information at the beginning of the analysis. Therefore the initial state of each FSM is `Unknown`.

6 Evaluation

We evaluated our approach on a set of benchmarks. However, parts of the prototypical implementation have been simulated in software due to the changes we had to implement compared to our initial approach. We plan to have a full hardware implementation at the time of the workshop.

Setting The target SoC for our prototype is a Xilinx Zynq featuring a dual-core ARM Cortex-A9 running at 667 MHz. The memory subsystem of this SoC constists of 32 kilobytes of L1 instruction cache, 512 kilobytes of L2 cache and 1 gigabyte of DDR main memory. We deactivated the L2 cache and the dynamic branch prediction in order to focus on L1 instruction cache effects. We used the TACLeBench benchmark collection [9] for the evaluation. We started with the evaluation before version 2.0 of the benchmark collection was finalized and had some problems with some of the tests. In particular, the benchmark `sha` could not be compiled with the C++ compiler provided with the Xilinx SDK 2014.4 that we used. We ran the triplet of a benchmark's `init`, `main` and `return` functions ten times in a row, except for `powerwindow`, which has been run only once as it contains a sligthly different structure than the other benchmarks. Unfortunately, this setting led to runtime errors in some of the benchmarks such that we could not use them for the evaluation.

Results The results of our evaluation are shown in table 2. We performed two runs of measurements, one with the L1 instruction cache enabled and one with disabled L1 instruction cache.

For the measurements performed with activated L1 instruction cache, we give the maximal observed end-to-end execution times of executing the benchmark's `main` function, the result of our analysis when the execution context is ignored, the result of our context-sensitive analysis, the improvement ratio between the later two and the overestimation of the context-sensitive analysis compared to the end-to-end measurements. A smaller ratio denotes a better improvement of the estimated execution time bound when the loop iteration has been taken into account as typical cache behaviour is exploited.

For the measurements performed with disabled L1 instruction cache, we give the maximal observed end-to-end execution time and the result of the context-insensitive analysis. Moreover, we compared them with with the results when the L1 instruction cache is enabled to see what impact the L1 cache has one the execution time of the benchmarks.

On average, an improvement ratio of 0.94 has been reached, i.e. the estimated execution time bound was decreased by 6% when the execution context has been taken into accout. Some benchmarks, like `md5` and `prime` showed much better bound reductions with 33% and 49%, respectively. Other showed almost no improvement at all. On closer inspection, it turned out that those benchmarks had little variance in the observed waypoint execution times. We suspect that the prefetching mechanism of the Cortex-A9 pipeline is able to prevent long delays during instruction fetch.

Most benchmarks showed reasonable overestimation when comparing the end-to-end execution times and the estimated context-sensitive bounds, with a median of 1.90. Exceptions are `bitonic`, `fft` and `quicksort` which contain data dependant loops and recursions. Since we used the maximal observed loop bounds as bounds for our ILP, we get huge overestimations.

For two benchmarks, we where not able to perform a full evaluation. One is `md5`, where we encountered a trace buffer overflow. We could thus not measure any end-to-end time, but our approach worked nonetheless, as we observed enough small snippets to estimate an

	L1 instruction cache activated					L1 instrcution cache deactivated			
program	end-to-end	context-insensitive	context-sensitive	improvement	overestimation	end-to-end	context-insensitive	end-to-end ratio	bound ratio
adpcm_dec	2099113	4292573	3585664	0,84	1,71	8269445	30367476	3,94	7,07
adpcm_enc	52417	90387	88641	0,98	1,69	82542	154810	1,57	1,71
basicmath	19497183	44583249	44138972	0,99	2,26	40107369	168812436	2,06	3,79
binarysearch	1726	2499	2292	0,92	1,33	3237	5739	1,88	2,30
bitcount	149719	466712	463553	0,99	3,10	273131	1599420	1,82	3,43
bitonic	162046	33378706	33361129	1,00	205,87	333355	60556151	2,06	1,81
bsort	2912025	9847181	9829328	1,00	3,38	4470758	15486611	1,54	1,57
complex_updates	7948	11066	10997	0,99	1,38	10735	14086	1,35	1,27
countnegative	98709	256863	255445	0,99	2,59	150774	420206	1,53	1,64
crc	23728	42511	41035	0,97	1,73	1010312	4941180	42,58	116,23
fac	4567	19492	19141	0,98	4,19	10892	54133	2,38	2,78
fft	4967772	2060545070	2060447457	1,00	414,76	7638856	2386497145	1,54	1,16
filterbank	52757627	56175367	56000579	1,00	1,06	130534265	214822229	2,47	3,82
fir2dim	45900	87390	81835	0,94	1,78	91914	174316	2,00	1,99
iir	1779	2401	2227	0,93	1,25	2431	3286	1,37	1,37
insertsort	29397	68977	68291	0,99	2,32	40268	83702	1,37	1,21
jfdctint	37358	39795	39657	1,00	1,06	44769	46902	1,20	1,18
lift	10329419	17278328	13925216	0,81	1,35	20087468	38399195	1,94	2,22
lms	5209670	12704581	12478923	0,98	2,40	8672208	25151604	1,66	1,98
ludcmp	38448	154678	146569	0,95	3,81	65963	256881	1,72	1,66
matrix1	136086	351749	350891	1,00	2,58	275566	537144	2,02	1,53
md5[a]	154573496	615340440	410888084	0,67	2,66	–	1596703270	–	2,59
minver	25746	41736	40152	0,96	1,56	61579	149386	2,39	3,58
pm	178339194	349576605	348784220	1,00	1,96	273680209	503009838	1,53	1,44
powerwindow[b]	8204	–	–	–	–	14096	30889	1,72	–
prime	135944	485956	249622	0,51	1,84	682279	2939746	5,02	6,05
quicksort	56159097	138962109974	135157270647	0,97	2406,69	82844689	206979278894	1,48	1,49
recursion	25991	35506	35506	1,00	1,37	85317	233528	3,28	6,58
st	1726823	2682337	2663482	0,99	1,54	2893966	5864534	1,68	2,19

[a] trace buffer overflow
[b] consistency check failed

Table 2 Results for the TACLeBench benchmark suite. The programs have been measured twice, once with L1 instruction cache enabled and once with L1 instruction cache disabled. The first column gives the measured end-to-end execution time in cycles. The second and third columns give the computed execution time estimates, once ignoring the execution context and once taking the loop iteration context into account. The forth column shows the ratio between context-sensitive and context-insensitive estimates (smaller is better). The fith column shows the overestimation comparing the end-to-end observations and the analysed context-sensitive bounds. The final four columns compare activated and deactivated L1 instruction cache.

overall bound. For benchmark `powerwindow`, we could not give any bound with activated L1 instruction cache because a consistency check in our prototype failed.

Deactivation of the L1 instruction cache leads to a slowdown factor of 3.29 on average. Ignoring the outlier `crc`, which benefits extremly from the L1 instruction cache, the average slowdown factor is 1.94.

Overall, our evaluation shows that the benchmarks benefit from the L1 instruction cache (as visible in the end-to-end measurements), but it is sometimes hard to capture the typical cache behaviour in a hybrid approach which aims for upper bounds.

7 Conclusion and Future Work

In this contribution, we have shown a method that is capable of estimating meaningful WCET of embedded software under the realistic conditions of modern SoCs. Even using the waypoints instead of basic blocks, a context sensitive aggregation of instruction execution times can be achieved. These execution times can be combined to form a WCET for the overall program (or larger portions of it). Using TACLeBench examples, we can show that the results are highly realistic.

Still many open questions remain. We are currently working on a method to gather a

much more detailed statistics of the execution times between waypoints. This would allow a better judgement of the gathered statistics. Also, we want to check, whether this approach can be used for other trace streams like Nexus 5001™. Ultimately, the question still has be answered whether slightly enhanced trace streams could give better results for the WCET estimation.

References

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

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

3 ARM Ltd. Embedded Trace Macrocell™ ETMv1.0 to ETMv3.5, 2011. ARM IHI 0014Q.

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

5 ARM Ltd. Embedded Trace Macrocell™ Architecture Specification ETMv4.0 to ETMv4.2, 2016. ARM IHI 0064D.

6 A. Betts and G. Bernat. Tree-based wcet analysis on instrumentation point graphs. In *9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*. IEEE Computer Society, April 2006.

7 A. Betts, N. Merriam, and G. Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 54–63. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

8 B. Dreyer, C. Hochberger, S. Wegener, and A. Weiss. Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 45–54, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

9 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume ?? of *OpenAccess Series in Informatics (OASIcs)*, pages ??–??, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Submitted for review.

10 Freescale Semiconductor, Inc. P4080 Advanced QorIQ Debug and Performance Monitoring Reference Manual, Rev. F, 2012.

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

12 K. Schmidt, D. Marx, J. Harnisch, and A. Mayer. Non-Intrusive Tracing at First Instruction. SAE Technical Paper 2015-01-0176.

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

14 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.