On the Use of Context Information for Precise Measurement-Based Execution Time Estimation[*]

Stefan Stattelmann[1] and Florian Martin[2]

1 FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10–14, D-76131 Karlsruhe, Germany
stattelmann@fzi.de

2 AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany,
martin@absint.com

Abstract

The present paper investigates the influence of the execution history on the precision of measurement-based execution time estimates for embedded software. A new approach to timing analysis is presented which was designed to overcome the problems of existing static and dynamic methods. By partitioning the analyzed programs into easily traceable segments and by precisely controlling run-time measurements with on-chip tracing facilities, the new method is able to preserve information about the execution context of measured execution times. After an adequate number of measurements have been taken, this information can be used to precisely estimate the Worst-Case Execution Time of a program without being overly pessimistic.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.64

1 Introduction

Information about the execution time of programs in embedded systems must be available at several design stages. During the initial phases, a rough estimate of the execution times should be available so that components which fit the expected workload of the system can be chosen. In the final phase of a project, precise execution times must be known in order to verify that the system fulfils all its timing requirements. The increasing complexity of real-time systems makes reasoning about the execution time of embedded software more and more challenging. This particularly holds for the Worst-Case Execution Time (WCET) of a task since it might only occur under rare circumstances which are caused by a nontrivial interaction of system components.

Existing methods for WCET analysis can be divided into static and dynamic methods. Static timing analyses try to determine a safe upper bound for all possible executions of a given program. In contrast, dynamic methods use measurements taken during a finite number of actual executions to determine an estimate of the WCET. On current processor architectures, both methods do not always produce satisfying results. The interaction of performance enhancing features in modern processors makes it very unlikely to observe the worst-case execution during a few test runs. Hence measurement-based analyses might underestimate the WCET considerably. Furthermore, existing methods are often not able to

* This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under grant 01M3088C and by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) funded by the European Commission's 7th Framework Programme on Research, Technological Development and Demonstration under contract 215068.

represent the performance gain from caches (cache effects) precisely. If only the worst-case execution time is considered for each basic block in a loop body, a performance increase in following iterations due to caching cannot be represented. This can make dynamic estimates very pessimistic, too. Static analyses must use (safe) approximations for the potential states of the analyzed system as the state space can grow very large for sophisticated architectures. These approximations are necessary to make the computation of WCET estimates feasible, but the increase of the reported bounds and the resulting imprecision can restrict their practical use.

This paper presents a context-sensitive analysis of accurate instruction-level measurements which can provide precise worst-case execution time estimates. The notion of context-sensitivity is a well-known concept from static program analysis. It has been shown that the precision of an analysis can be improved significantly if the execution environment is considered. This especially holds if the analysis does not only consider different call sites but also distinct iterations of loops (see [12]), as this allows the consideration of cache effects. Up to now, context information is mainly used in static timing analysis. The results presented in this work suggest that the precision of measurement-based timing analyses, too, can be increased considerably by incorporating context information.

Recent developments in debug hardware technology allow the creation of cycle-accurate traces with a fully programmable on-chip event logic [13]. The increasing availability of these tools for instruction-level measurements and the precise timing information they provide motivate the use of methods from static timing analysis for measurement-based approaches. As dynamic methods for WCET estimation can be adapted to new processor architectures much more easily than the models used in static analyses, this would reduce the initial investment necessary for performing exact timing analyses.

The influence of context information on the precision of execution time estimates is not only interesting for WCET analysis. All forms of execution time inspection on complex architectures might be improved by incorporating context information, even if the worst-case is not (yet) of interest, like during design space exploration in an early development phase.

The remainder of this work will be organized as follows. The next section will list some related approaches for dynamic WCET analysis. Section 3 introduces a new context-sensitive method for measurement-based execution time analysis. In the fourth section, experimental results with this method will be presented. The last section gives a summary of the work and the impact of the results.

2 Related Work

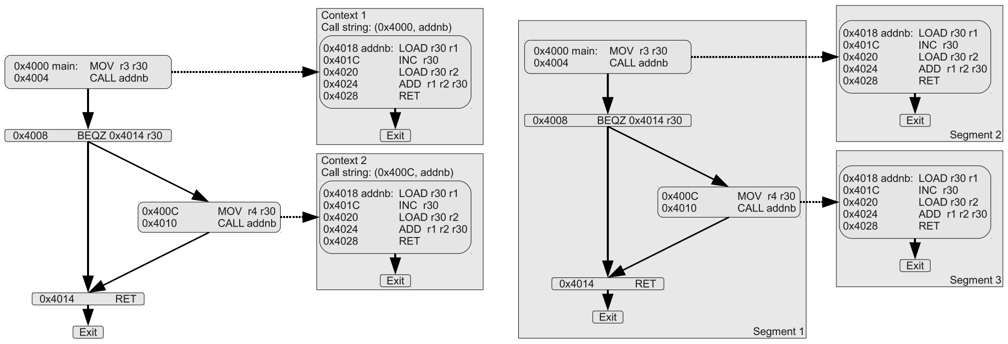
A complete overview of existing methods for WCET analysis can be found in [20]. This section will focus on measurement-based methods. One of the first attempts to consider the execution context of execution times was the *structure-based approach*, a technique for static timing analysis proposed in [16], but it only aimed at a more precise combination of execution times from individual program parts. Similar techniques are still used for measurement-based WCET analysis, e.g. in [5], but they lack the ability to reflect the interaction of individual program parts which is mandatory to represent the influence of the execution history for example due to cache effects. An extension to the structure-based approach which can distinguish execution times of loop iterations is described in [6], but there is no practical evaluation of the effects on the execution time estimates.

To overcome the problem of exhaustive measurements, several solutions have been developed (e.g. [7] and [18]) that try to partition a program into parts which can be measured more easily. These approaches usually assume that the system can be brought into a worst-case state before taking measurements, e.g. by clearing the cache. This assumption may hold on simpler processors, but it is hard to fulfil in complex systems as it might not be clear what this state looks like due to a complex interaction of system components [9]. Modifying the system state can also make the execution time estimates very pessimistic, for example if the cache is cleared too often during the measurements. Further solutions include the automatic generation of input data to enforce a worst-case execution [19, 18] or the probabilistic analysis of measurements [5]. The use of results from a cache analysis to guarantee that a sufficient number of measurements have been performed was described in [14]. An approach based on game theory which can represent varying execution times for different loop iterations is presented in [15]. It is also based on program partitioning, an automatic generation of input data and requires that all loops in the analyzed program can be unrolled completely. In [11] constraint logic programming is used to model context-sensitive execution times based on constraints that are derived from an execution time dependency analysis of program traces.

Although only an incomplete overview of methods for measurement-based timing analysis can be given here, the main concern of existing approaches seems to be to enforce the observation of a worst-case execution. Furthermore, many techniques require that the analyzed program is changed since instrumentation code must be added. In contrast to this, the following section will introduce an approach which aims at the precise combination of a large number of measurements. By using evaluation boards which provide hardware support for controlling the collection of trace data, probe effects are avoided since code instrumentation is not necessary. Furthermore, steering measurements precisely during runtime allows increasing the precision of execution time estimates since cache effects can be represented by distinguishing the execution history of an observed code region.

3 Proposed Method

This section presents a new concept for measurement-based timing analysis. The method works on the interprocedural control flow graph (ICFG) of a program executable and requires measurement hardware that can be controlled by complex trigger conditions. The development of the approach was motivated by the limited size of trace buffer memory which is available in current hardware for on-chip execution time measurements. Measurement hardware which stores traces in an external memory overcomes this problem by sacrificing accuracy. Due to bandwidth constraints these traces only store certain instructions, for example taken branches. Additionally, timestamps for these instructions are often only created when a partial trace is transfered from a small on-chip buffer to the large external memory. Hence deriving the execution time of every single instruction is hardly possible. As a consequence of these limitations, it is not feasible to determine context-sensitive execution times from end-to-end measurements, since it is not possible to create cycle-accurate end-to-end traces for programs of realistic size, i.e. traces containing a timestamp for *every* executed instruction. Instead of using traces of complete program runs, this work investigates the use of the programmable trigger logic in state-of-the-art evaluation boards for embedded processors to create context-sensitive program measurements. Current tracing technology, like the Infineon Multi-Core Debug Solution [13], allows considering the execution history of a program before starting a measurement run. This is achieved by dedicated event logic in the actual hardware which can be used to encode state machines to model the program



(a) ICFG with virtual inlining **(b)** Partitioning into segments

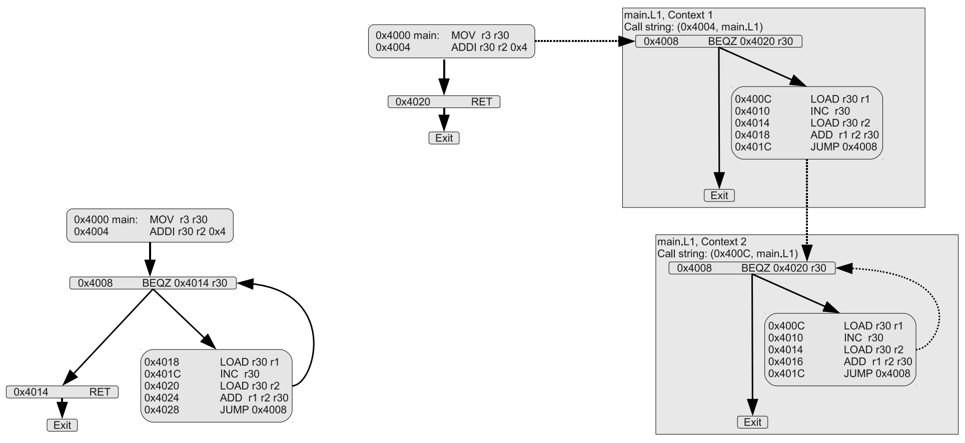
Figure 1 Execution contexts and program partitioning

state. These possibilities motivated the development of an analysis which makes use of this additional logic to generate context-sensitive traces despite the limitations of the trace buffer size. The analysis is divided into several phases:

- Initially, the ICFG is created and partitioned into *program segments* in such a way that every possible run through the segments can be measured with the available trace buffer memory.
- The information gathered during the partitioning phase is used to generate trace automata that will control the measurements.
- Taking measurements requires a sufficiently large number of actual executions of the analyzed program on the target hardware.
- After the measurements have been taken, the context-sensitive timing information for each basic block of the program can be extracted and annotated to the ICFG. Further computations then yield the worst-case path through the ICFG and an estimate of the worst-case execution time of the program.

3.1 Control Flow Extraction

Initially, the control flow is extracted from the program executable and its ICFG is constructed. To represent the execution context of program parts precisely, the concept of *virtual inlining, virtual unrolling* (VIVU) is used [12]. VIVU applies function inlining and loop unrolling on the level of the ICFG. Thus the ICFG can contain several copies of the same basic block for which different execution times can be annotated. To consider the execution history of these duplicates, the control flow graph is extended with additional information that represents the execution context. A *call string* is used to model a routine's execution history. Call strings can be seen as an abstraction of the call stack that would occur during an execution of the program. In this work, a call string will be represented as a sequence of *call string elements*. A Call string element (b, r) is a pair of a basic block b and a routine r which can be called from b. Only *valid* call string elements will be allowed, meaning it must be possible that the last instruction of the basic block b is executed immediately before the first instruction of routine r. For the entry routine of the analyzed task (e.g. *main* in a standard C program) there is no execution history as the execution is started by calling the respective routine. This context is described by the empty call string ϵ. It will be omitted in the following examples. The intuition behind this representation of an execution context is that whenever a routine is called, the call string is extended with another element to



(a) Original CFG **(b)** First iteration unrolled

Figure 2 Extension of the CFG by virtual unrolling

describe the context of the function body. Therefore extending the call string works similar to extending the call stack during program execution. Since the execution history of a routine can be very complex, its call string representation might become very long. In order to achieve a more compact representation of execution contexts, the maximal length of call strings will be bounded by a constant $k \in \mathbb{N}_0$. For call strings which describe a valid execution but exceed the maximal length, only the last k call string elements will be used to describe the context. In the following examples a call string length of one will be used.

Figure 1a depicts an example of virtual inlining in an ICFG by duplicating routine bodies for every call site. Intraprocedural edges are drawn with solid lines, while the edges describing a function call are represented by dashed lines. Routines are not explicitly highlighted in the ICFG, but every routine is assumed to have a unique entry node, which is the target of the call edges, and an artificial exit node through which the routine must be left. The effect of virtual unrolling is shown in in Figure 2. Virtual unrolling also extends the ICFG by duplicating nodes. Additional precision is gained by extracting loops from their parent routine and treating them like recursive routines. This allows a more precise classification of the execution history than a simple calling context when searching for the WCET path through the program, since varying execution times in different loop iterations can be represented independently from the parent routine.

3.2 Program Partitioning

To cope with the limited memory for trace data, the ICFG is partitioned into *program segments*. These segments consist of a start and an end node in the graph which must fulfil the condition that the longest path in terms of executed instructions (not execution time) between them is smaller than or equal to the number of instructions for which timestamps can be stored in the trace buffer. Additionally, both nodes must be part of the same execution context. Segments can either include all nodes which lie on the interprocedural paths between the start and the end node or they can be restricted to the nodes on the intraprocedural paths. By excluding calls to other routines, the size of a program segment can be reduced, but information about the execution context can be preserved in the traces. After this partitioning, each node of the ICFG is covered by at least one program segment and it suffices to perform measurements for individual segments to determine context-sensitive execution times for every basic block.

The program from Figure 1a will be used to illustrate the concept of program segments. Assume the program is to be traced with a trace buffer which can hold timestamps for at most 6 instructions. In order to extract cycle-accurate and context-sensitive timing information, at least 3 program segments are necessary. Each of these segments is measured individually and the results are combined during a post-processing phase. Figure 1b illustrates one possible partitioning. In this example, a separate segment is created for the body of the routine *addnb* at each call site. Additionally, the segment for the top-level routine *main* is assumed to be measured without the routine it calls. This assumption makes it possible to handle the limited trace buffer. However, to fulfil this assumption during an actual measurement run, it must be possible to trigger the measurement hardware precisely.

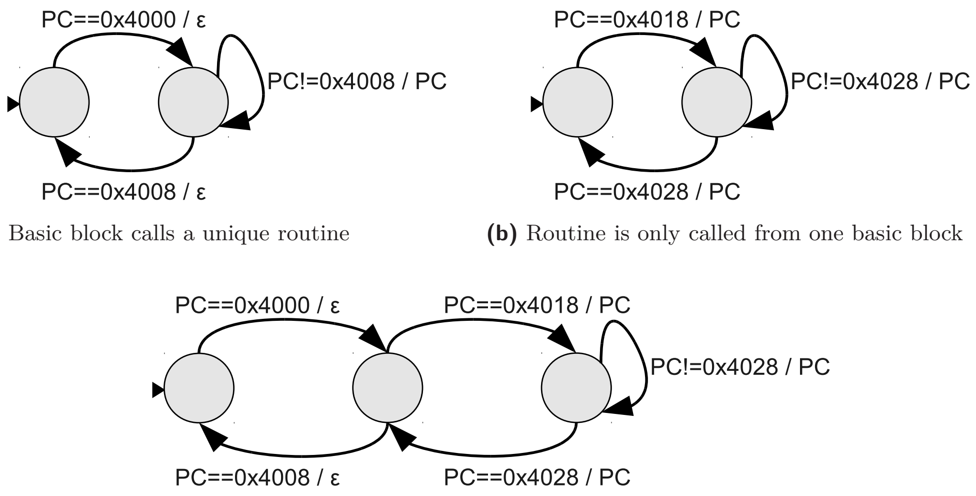
3.3 Trace Automata

For each program segment, a *trace automaton* is generated to control the measurement runs of the respective segment. These state machines encode the conditions which describe the execution history of each segment, i.e. which instructions must have been executed before the execution context of the observed program run matches the program segment. Monitoring the execution of the program before starting a measurement allows preserving information about the execution context even if the trace does not contain the complete execution history of the analyzed code regions. The automata are constructed from the execution history of the respective segments, which is described by a sequence of call sites and loop entries (call string). This abstract description of trace preconditions is translated to the event logic of the evaluation board using a software debugger [4] which is then used to collect measurement data.

For the description of an execution context, each element of a call string describes two conditions in terms of executed instructions: the call instruction in the call block must be executed immediately before the first instruction of the called routine. Additionally, the sequence of the elements constrains the order of these conditions, i.e. the order of the calls. In principle, they can be directly translated to a trace automaton which changes its state depending on whether the correct routines are called at the appropriate call sites. But since most call sites call exactly one routine, the automata created by this strategy are not minimal. On the other hand, there might be program segments which have a common call string, but lie in a different instruction address range (e.g. if a routine gets partitioned into two segments). Hence it is not sufficient to consider only the context description when constructing trace automata.

To generate a trace automaton for measuring a program segment, the first step is to create states and transitions which correspond to the constraints described by the call string representation of the segment's execution context. After that, states must be added to express which instructions on the paths through the segment should be traced. The complete approach proceeds as follows:

Initially, the automaton has a single state, no transitions and generates no output. Then, at least one state for each element of the call string is added to the automaton. How many states are added depends on the properties of the call site described by the string element. If the call described by the element has only one possible destination, it suffices to use the address of the call block as condition for the transition to the next state. Similarly, if this is not the case but the destination routine is only called at this call site, it is enough to add a single state which is entered as soon as the entry address of the respective routine is



(a) Basic block calls a unique routine **(b)** Routine is only called from one basic block

(c) Several callers and callees

Figure 3 Translation of a call string element

encountered. For call string elements which fulfil neither of the conditions, both states have to be added to the trace automaton to model the requirements for the execution history described by the call string elements. These three cases are illustrated in Figure 3 for the one-element call string (*0x4000*, *addnb*) and the routine *addnb* from Figure 1a which starts at the memory location *0x4018* and returns at address *0x4028*.

Finally, the state for actually storing trace data is added. For program segments which cover all routines that are called on the paths through the segment, this can be done by adding a single state which is entered as soon as the start block is entered and left when the end block is left. In this state, all instructions which will be executed will also be stored in the trace buffer. For segments which exclude called routines, things are slightly more complicated and an additional state gets necessary. The tracing state is constructed as before, but the additional state is entered when calls are executed within the segment (i.e. when the address interval for the segment is left) and no trace data will be generated while the automaton is in this state. Note that no extra state for storing trace data is necessary if the program segments covers a complete routine and all its calls. In this case, tracing can start as soon as the addresses for all call string elements of the execution context have been processed by the automaton. This is also illustrated in Figure 3.

3.4 Trace Data Generation

Taking measurements requires a sufficiently large number of actual executions of the analyzed program on the target hardware. The approach relies on the assumption that all worst-case execution times of each basic block in every execution context were observed during the measurements to produce a safe estimate. As the program under consideration is not modified in any way, measurements should be taken under realistic conditions to produce execution time estimates that match the expected workload. Using typical inputs during a large number of measurements should result in estimates close to the actual WCET. Controlling the generation of trace data with state machines offers the advantage that the measurement logic can wait for code region which are executed rarely before triggering the trace generation. Since this process can be automated, achieving a sufficient level of code coverage is facilitated, but not guaranteed.

3.5 Timing Extraction

After a set of traces has been generated for each of the segments into which the program of interest was partitioned, the maximal execution time for each basic block is extracted from the measurements. As the traces are assumed to be cycle-accurate, this is a straightforward process since every instruction which gets executed during a measurement run must also be contained in the respective trace. Additionally, the traces must contain a (relative) timestamp for each instruction. Since tracing is controlled by (an implementation of) a trace automaton, the precise execution context of the trace data is known. Hence the execution time for each basic block can be extracted from a trace by simply going through the trace and the ICFG in parallel. Whenever a new basic block is entered in the trace, the respective node must be found in the ICFG. Depending on the type of program segment which is annotated, this search for a successor must be carried out on the whole ICFG or just within the current routine, i.e. without following call edges. The execution time of a basic block is determined by subtracting the timestamp of its first instruction from the timestamp of the first instruction of its successor block. As the context in which a trace is generated is preserved while creating the measurements, basic block execution times from the trace are only annotated to those nodes with matching context. In case of virtual inlining, this means that execution times are only annotated to those nodes in the ICFG at the correct call site, but not to the nodes in other contexts (although these nodes represent the same basic blocks on assembly level). Depending on the level of unrolling, the execution times of nodes within a loop can also be assigned to distinct iterations. Hence the worst-case execution time of nodes in the first iteration of a loop, which might generate many misses in the instruction cache, can be easily separated from the remaining iterations. Further iterations are usually not expected to suffer the same performance penalty from cache misses. By duplicating these nodes, the WCET estimates get more precise compared to approaches which cannot make this distinction. In contrast to the method presented in [11], no additional processing of the traces has to be performed to derive these dependencies between basic block execution times. Additionally, the worst-case execution time of the whole program can still be computed by implicit path enumeration [10] and there is no need to resolve additional execution time constraints using constraint logic programming.

For each node in the ICFG which was covered by a trace, this provides the execution time for this particular run. Under the assumption that all local worst-cases were observed during the measurements, meaning that the worst-case execution time of each node is covered by at least one of the traces, the maximum from all of the execution times equals the worst-case execution time. All nodes in the ICFG which never occurred in one of the traces are assumed to be never executed during *any* execution of the program. If a *sufficiently large* number of measurements has been taken under *realistic conditions*, taking the maximum of the measured execution times for each node is likely to provide the worst-case execution time or at least a *realistic estimate* of it. Nevertheless, the presented work provides no means to enforce these conditions.

3.6 Cache Analysis

The design of the proposed method allows an easy integration of static analyses to make the measurement phase more efficient. To demonstrate this, the cache analysis described in [8]

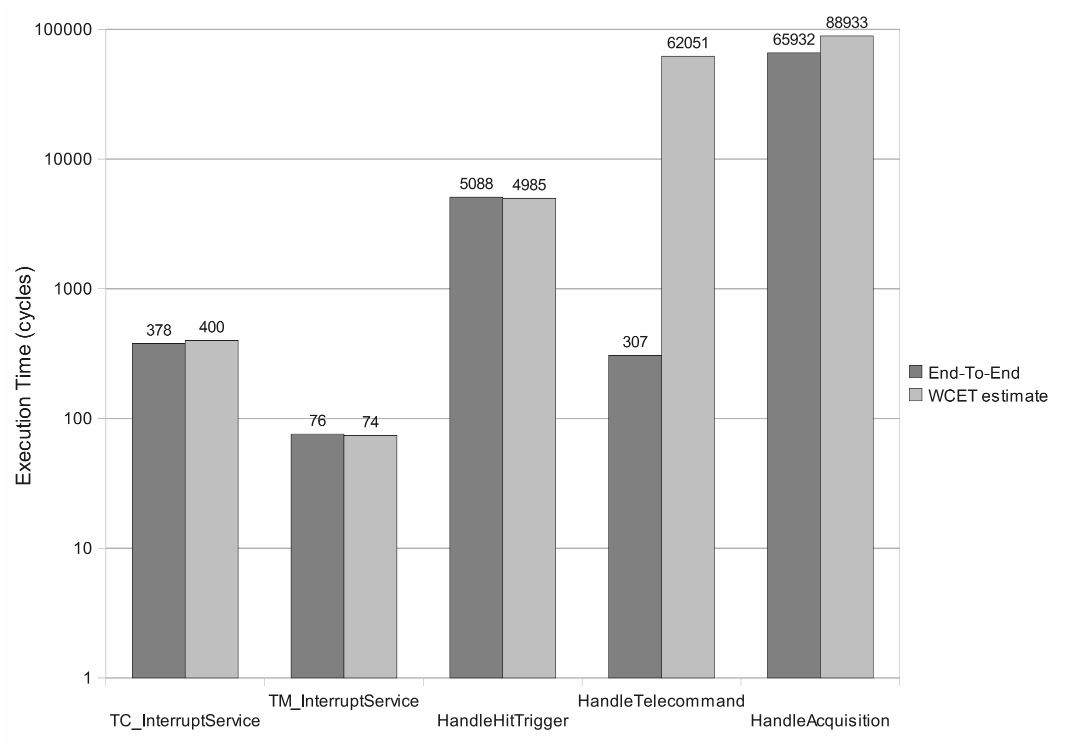


Figure 4 Comparison of context-sensitive and end-to-end measurements

has been adapted to classify the instruction cache behavior of different execution contexts. This classification is achieved by comparing the number of potential and guaranteed cache hits and misses for different execution contexts of a basic block. If two program segments from the same code region, but with a different execution history, will exhibit a (roughly) identical cache behavior, they will not be distinguished during the measurements. By joining some of the execution contexts generated by the VIVU approach, the number of measurements can be reduced without influencing the precision of the WCET estimates. A detailed description of this optimization is beyond the scope of this paper, but further details can be found in [17].

4 Experimental Results

The proposed method was implemented as an extension of the AbsInt aiT WCET Analyzer [1] and tested with an Infineon TriCore TC1797ED evaluation board. Several common WCET benchmarks could be successfully analyzed, in particular programs from the Mälardalen WCET Benchmark Suite [3] and the DEBIE-1 benchmark [2], which is an adapted satellite control application. No changes to the hardware state or the analyzed software were performed during the experiments. The programs were simply loaded into the flash memory of the evaluation board and then measured several times successively. Input data for the programs did not have to be generated as this was already handled by the benchmarks.

For an initial test, the estimates provided by the implementation were compared to WCET estimates based on a number of simple end-to-end measurements. The result of this comparison is shown in Figure 4. Estimating the worst-case execution time of programs

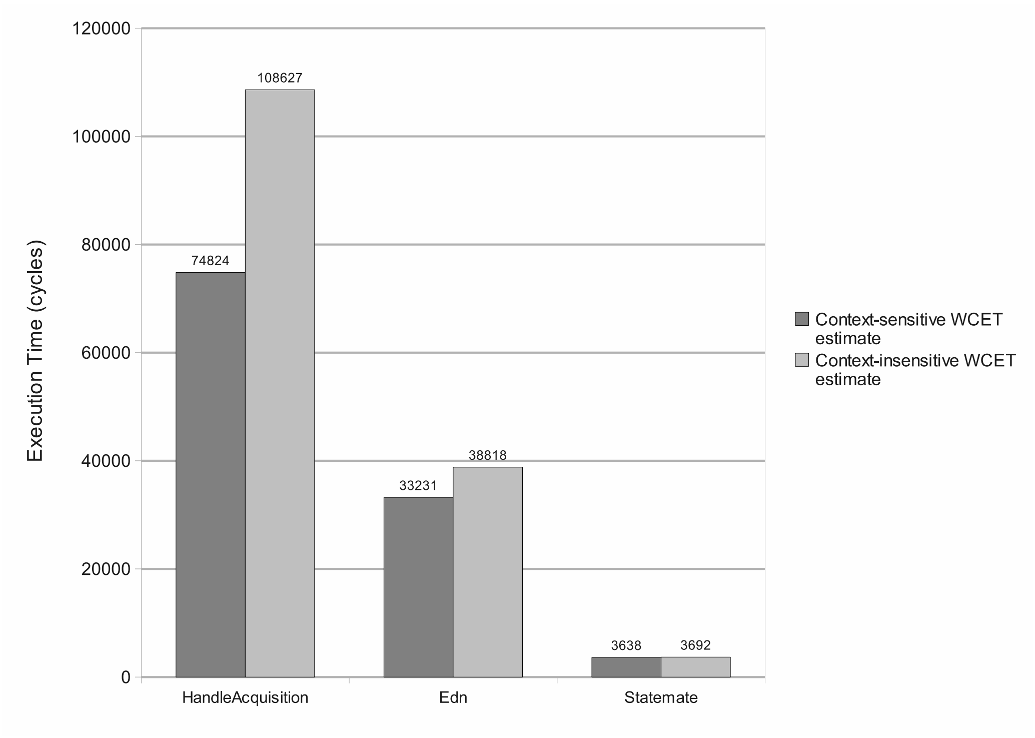


Figure 5 Improvement of WCET estimates through context information

based on measurements is problematic as it usually cannot be guaranteed that the worst-case has been covered. This was demonstrated by the test case *HandleTelecommand* from the DEBIE-1 benchmark: though a considerable effort was made for the measurements, the observed end-to-end execution times were considerably smaller than the context-sensitive WCET estimates. Manual examination of the traces showed that some routines which were on the WCET path reported by the context-sensitive approach were never executed during the end-to-end measurements. Hence this test case showed that for programs which rarely execute the routines which are responsible for the worst-case execution, the presented approach is superior to simpler methods. The program partitioning and the precise control over the measurement runs allows the measurement hardware to wait for these rarely executed program parts before starting the actual trace. Nonetheless, the prototype implementation reported some WCET estimates which were smaller than the maximal execution time observed during the end-to-end measurements. One reason for this is that the measurement hardware sometimes did not start the traces immediately after they were triggered. As a result of these delays, some basic blocks were never completely covered by the measurements and thus the execution time was underestimated. As the number of measurements which were taken during the experiments was relatively small, insufficient coverage of critical program parts is another potential cause of the underestimation.

The effect of context information for measurement-based execution time estimation was investigated by using the same set of trace data with and without the consideration of the execution history (Figure 5). A smaller number of measurements was performed for these experiments than for the previous ones as the focus was not on precisely estimating the WCET (i.e. covering *all* local worst-cases), but on investigating the effect of context information. For this reason, some results presented in Figure 5 differ slightly from previous estimates. The context-insensitive analysis uses the maximal execution time of each basic block found in the traces and annotates this value to every copy of the respective basic block in the ICFG. On the other hand, the context-sensitive analysis was able to annotate smaller execution times to some of the basic block instances since it is able to preserve information about the execution history, e.g. by distinguishing the first from all remaining iterations of a loop. For two out of three test cases, the context-sensitive approach seems to be able to represent cache effects more precisely. Hence, smaller WCET estimates are reported. This effect could not be observed for the smallest of the test cases, probably since the execution time of the program does not benefit from caches due to its linear structure. The results of this comparison suggest that the difference between a context-sensitive and a context-insensitive analysis can be substantial. By increasing the number of measurement runs, this effect can only be intensified, as for every increase in the context-sensitive estimate, the context-insensitive estimate must grow as well. Thus the execution context of execution time measurements should be preserved whenever possible. If this is not done, cache effects cannot be determined correctly, which is why a context-insensitive evaluation might introduce a severe amount of pessimism to the execution time estimates, which renders them less precise.

5 Conclusion

This work proposed a new approach to measurement-based timing analysis which makes use of techniques from static program analysis. The results obtained during the experiments show that state-of-the-art measurement hardware can be used to determine WCET estimates automatically. To get precise results, a large number of measurements should be performed since the method relies on the assumption that the local worst-case for each basic block was observed during the measurements. Although the new approach seems to be more robust and more precise than existing methods for measurement-based timing analysis, it does not overcome their inherent problems, like the dependence on input data. However, controlling the collection of trace data precisely allows weakening the influence of these problems to the WCET estimate, e.g. because it is now possible to facilitate measurements within program parts or execution contexts which are executed very rarely. While the precise control of trace data generation makes it more likely that local worst-case executions can be observed, the use of context information allows the precise combination of partial execution times. This makes the calculated WCET estimates less pessimistic.

The outcome of the experiments also shows that only measuring each basic block often enough, which is the prevailing paradigm for measurement-based timing analysis, is not enough to determine precise execution time estimates as the execution history might have a significant influence on them. For static timing analysis this is a well-known fact, but the presented results suggest that all techniques for reasoning about software execution times on complex hardware can benefit from the use of context information. This includes measurement-based execution time analysis as well as techniques for execution time estimation in a design space exploration or simulation environment.

As a complex event logic for trace data generation is not always available, measurement-based methods for WCET analysis could try to reconstruct context information from trace data instead of controlling its creation. This should still improve the precision of the estimates, but would also work with simpler measurement facilities. Extracting a context-sensitive worst-case execution time for each basic block from the trace data has the additional benefit that only one value has to be stored for every execution context. As more traces are generated, these values are only updated if a longer execution time has been observed. All execution times which are below the worst-case can be discarded without influencing the final result. This allows the efficient parallelization of trace data generation and timing extraction. Additionally, this approach can also help to reduce the tremendous storage requirements which measurement-based methods for WCET analysis often have.

Acknowledgements The authors would like to thank Christian Ferdinand for fruitful discussions about the content of this paper.

References

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

2 Debie-1 WCET Benchmark. http://www.mrtc.mdh.se/projects/WCC08/doku.php.

3 Mälardalen WCET Benchmark Suite. http://www.mrtc.mdh.se/projects/wcet.

4 pls Development Tools Universal Debug Engine (UDE). http://www.pls-mc.com.

5 Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET: A Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. Technical report, Department of Computer Science, University of York, February 2003.

6 Adam Betts and Guillem Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 558–565. IEEE Computer Society, 2006.

7 Jean-François Deverge and Isabelle Puaut. Safe Measurement-Based WCET Estimation. In Reinhard Wilhelm, editor, *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

8 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

9 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

10 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 456–461, New York, NY, USA, 1995. ACM.

11 Amine Marref and Guillem Bernat. Towards Predicated WCET Analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

12 Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC '98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94, Berlin, 1998. Springer.

13 Albrecht Mayer and Frank Hellwig. System Performance Optimization Methodology for Infineon's 32-bit Automotive Microcontroller Architecture. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 962–966. ACM, 2008.

14 Stefan Schaefer, Bernhard Scholz, Stefan M. Petters, and Gernot Heiser. Static Analysis Support for Measurement-Based WCET Analysis. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Work-in-Progress Session*, 2006.

15 Sanjit A. Seshia and Alexander Rakhlin. Game-Theoretic Timing Analysis. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 575–582, Piscataway, NJ, USA, 2008. IEEE Press.

16 Alan Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15:875–889, 1989.

17 Stefan Stattelmann. Precise Measurement-Based Worst-Case Execution Time Estimation. Master's thesis, Saarland University, September 2009.

18 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-Based Timing Analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Oct. 2008.

19 Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proc. Conference on Design, Automation, and Test in Europe*, Mar. 2005.

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