Context Sensitive Performance Analysis of Automotive Applications

Jan Staschulat, Rolf Ernst
Institute of Computer and Communication Network Engineering
Hans Sommer Str. 66, D-38106 Braunschweig, Germany
{staschulat|ernst}@ida.ing.tu-bs.de

Andreas Schulze, Fabian Wolf
Powertrain Electronics Software Systems, Volkswagen AG
Letter box 1687, D-38436 Wolfsburg, Germany
{andreas.schulze|fabian.wolf}@volkswagen.de

Abstract

Accurate timing analysis is key to efficient embedded system synthesis and integration. While industrial control software systems are developed using graphical models, such as Matlab/Simulink or ASCET/SD, exhaustive simulation is not suitable for verifying functional and timing behavior. Formal performance analysis is an alternative but can lead to wide timing intervals because of input data dependency and complex target architectures. Hence a designer might want to restrict the formal performance analysis to parts of the software system, called context or process modes.

In this paper, we describe how to define and characterize such context information from graphical models. Further, we extend the formal performance analysis to consider contexts. Results from an automotive application demonstrate the applicability of our approach.

1. Introduction and Motivation

Accurate timing analysis is key to efficient embedded system synthesis and integration. In general, imprecise estimation of software execution costs increases design risks or leads to inefficient designs [16].

Increasing functionality and software complexity in automotive systems requires concepts for distributed development between car manufacturers and control unit suppliers. These concepts need well-defined and established development processes as well as safe and certifiable software integration approaches [7] [9]. Functional and timing behavior is often specified in graphical environments, such as Matlab/Simulink or ASCET/SD. For timing validation the state-of-the-art in industry is still simulation or profiling. Since the automatically generated code from such models heavily depends on input data a great many test cases would be required for a full path coverage. The exponential number of input data makes exhaustive simulation impracticable for performance validations of industrial applications. To make testing feasible, system complexity should be broken down to the software module. Input dependency is restricted to this module and function calls and hence input data dependency to other modules can be ignored.

As a consequence, software simulation in general can only partially test a software system and only a lower bound of the worst case execution time (WCET) can be found. As an alternative to simulation, many approaches to static timing analysis [13] [18] [6] [2] have been proposed during the last decade. Formal analysis provides reliable upper and lower bounds of the execution time. Despite the theoretic advances in academic research there has been hardly any impact on the industrial practice of timing analysis [10].

None of the existing formal approaches can be applied to software modules because input data dependencies cannot be defined systematically. Context dependent analysis can be used to improve the analysis precision [17]. For example, a context for a cruise control software might be keeping the speed while driving uphill or stepping on the brake to disable speed control. Context information are also used in [8] for scheduling analysis to obtain tighter analysis results. However, for complex embedded applications it is often difficult to find these contexts.

This paper describes how to define and characterize contexts using graphical modeling environments. Here we propose a generic XML format to specify contexts which is useful for timing analysis approaches. Then we extend our static timing analysis to consider context dependency.

This paper is structured as follows. Related work is reviewed in Section 2. In Section 3 we describe our static timing analysis Symta/P. The context sensitive analysis is

presented in Section 4, and in Section 5 we show the applicability of our approach in several experiments before we conclude in Section 6.

2. Related Work

During the last decade many research approaches have been proposed for static performance analysis. In [13] implicit path enumeration was introduced. Here, the user provides linear equations to define feasible and false paths. To evaluate these equations, Li and Malik map the upper and lower bound identification to two optimization problems. The work in [6] is based on this kind of interaction while abstract interpretation is used for the static prediction of execution time including cache and pipeline behavior.

In [2] the notion of probabilistic hard real time system is introduced, as a system which has to meet all deadlines but for which a probabilistical guarantee is sufficient. The work in [5] presents a WCET framework with several low-level timing analysis techniques, while restricting input programs to a subset of the language C and requiring the user to specify all input data. In [12] a parametric WCET analysis is proposed, where parameters may represent values of input parameters, or maximal iteration counts for loops. No results from experiments have been reported. In [11] a hybrid framework using runtime measurements together with static program analysis is presented. Traditional WCET analysis is extended by automatic generation of test data using model checking and constraint-based analysis.

Despite the theoretic advances in academic research in WCET and the commercial availability of analysis tools, such as [1] [3] [4], there has been hardly any impact on the industrial practice of timing analysis [10]. One reason is that performing pure runtime measurements with exhaustive search over the value space of the input data is in general not feasible. In approaches with abstract interpretation a new abstract model has to be written for every new processor which is time consuming and makes it difficult to re-target the analysis.

Input data generation becomes feasible for programs with few input-data dependent control flows [10]. In previous work [17] we have shown that contexts lead to a reduced number of input data and more precise execution time intervals. However, for complex embedded applications, it is often difficult to find these contexts. Therefore, we describe in Section 4 how to define and characterize contexts in a general and flexible framework and extend our static timing analysis to context dependency in Section 3.

3. Symta/P - a Static Timing Analysis Tool

Symta/P is a tool for static timing analysis of embedded applications [14] [15]. Best case and worst case execution time bounds of C programs for complex target architectures are computed. Path analysis based on an abstract syntax tree identifies input data independent program segments while the execution time of program segments can be estimated by a cycle accurate off-the-shelf processor simulator or by direct measurement. For the determination of execution times of program segments no interrupts are allowed. A cache analysis determines the number of cache hits and misses for best and worst case.

3.1. Program Path Analysis

For path analysis techniques [13], a program is typically divided into basic blocks. Then, the program structure is represented as a directed program flow graph with basic blocks as nodes. For each basic block the execution time is determined. The longest and shortest path analysis on the program flow graph is used to identify a global execution time interval. This procedure does not provide sufficient accuracy. For acceptable analysis precision feasible paths of a program have to be identified. A feasible program path is a path in this flow graph corresponding to a possible sequence of basic blocks when the program is executed. A program segment is a sequence of nodes in a program flow graph.

Program properties can be exploited to simplify path analysis for the determination of the execution time through basic block sequences [18]. Large parts of typical embedded system programs have a single program path only. An FIR filter is a simple and a Fast Fourier Transform is a more complex one. There is only one path executed for any input pattern, even though this path may wrap around many loops, conditional statements and even function calls. A program segment has a single feasible path (SFP) when paths through this segment are not depending on input data. Most practical systems also contain non-SFP parts. These parts have multiple feasible paths (MFP).

For the identification of SFP and MFP segments, the input program is mapped to its syntax graph. A depth first search algorithm on the syntax graph can be used to determine input data dependencies of conditions while storing the data in a symbol table. Every control structure which does not contain an input data dependent condition is a SFP. If conditions depend on input data, the syntax tree nodes are classified as MFP.

3.2. Execution Time Model

The execution time model in [13] is established as a standard model for static approaches which is called sum-of-basic-blocks model. In previous work [18] we have extended this concept to program segments. Let a program consist of N program segments with x_i the execution count

of program segment s_i and c_i the execution time. Then the program execution time is determined by: $C = \sum_i^N c_i \cdot x_i$. For the execution counts x_i, the designer provides functional constraints, such as loop bounds or implicit description of possible paths by means of linear equations. Structural constraints define another set of equations: The execution count inflow of a segment equals its execution count outflow. These equations for the upper and lower execution bound are mapped to two integer linear optimization problems (ILP).

3.3. Architecture Modeling

The execution time of a program segment can be determined by instruction time addition or program segment simulation [18]. Instruction time addition adds up the execution time for each instruction. Host tracing is used while execution times are taken from a table. This approach only works for simple architectures without pipelines and caches where instruction execution is independent. A second approach is program segment simulation. Program segments are simulated on a cycle accurate processor simulator using known input data or the segments are executed directly on the target architecture.

4. Context Sensitive Analysis

Often the designer is interested in a context dependent process behavior. Here, context is defined to be a subset of input data and/or a subset of possible process states, often called process modes.

In each context, only a subset of paths through a program segment can be executed. This potentially means reduced time bounds which could be exploited for analysis. Global process representation models [19] can support process modes, such that the distinguishable contexts are considered in scheduling analysis [8].

The next subsection describes how to define context information and Section 4.2 describes how to specify them in a generic way to be useful for a static timing analysis. In Section 4.3 we describe the extensions of Symta/P to analyze context information, and give an overview of the general tool flow in Section 4.4.

4.1. Context Definition

Automotive control systems are typically designed in graphical models, such as the development environment ASCET/SD or Matlab/Simulink. The source code for the software is automatically generated from hierarchical block diagrams and state machines. Simulation is state-of-the-art to test functional behavior. A complete path coverage is not possible for complex software because of the exponential number of test data. Therefore testing is prioritized by functional importance: for critical paths more intensive testing is required than for less critical ones.

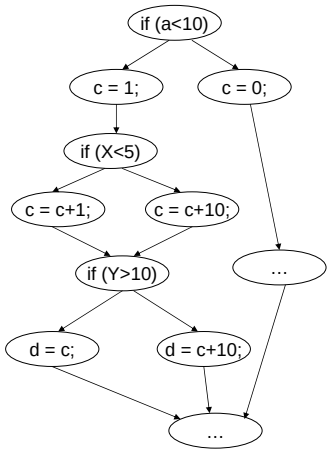


Figure 1. Example control flow graph

The application is analyzed on each hierarchical level. Consider a software module m_l on level l which calls several functions on higher hierarchy and is called from lower levels. The top level function level is 1 and the level increases by one for every hierarchy top down. Then a functional software test requires input data to reach module m_l through all hierarchical levels smaller than l. For a function call to a higher level, not every path is tested, instead only one arbitrary path is chosen. This functional testing strategy can be adapted for static timing analysis and the test pattern can be used for a context definition.

A *context* C is defined as a subset of all input data variables I which have predefined values. Variables of the set C are called *context* variables and all other variables $I \setminus C$ are called *free* variables. The static timing analysis is simplified because several paths are excluded, and consequently, the total number of input data is reduced.

Fig. 1 shows an example control flow graph with three if-conditions where a, X, Y are (unknown) input variables. Suppose the designer wants to test the left branch of the first if-condition. Then a is a context variable and X, Y are free variables.

To define contexts from graphical models, variables of some hierarchy level l are useful for context definition. In this paper we assume that one context corresponds to one hierarchical level l as a first approach. All variables on higher and lower levels $i \neq l$ are classified as context variables and all variables of module m_l are free variables. If a context spans over several hierarchies the input variables from different levels have to be considered also. Note, that different contexts can share code segments.

The next subsection describes how to specify a context from an abstract specification model and how to provide context information as an interface to a static timing analysis, such as Symta/P.

4.2. Context Specification

From the graphical specification environment ASCET/SD a logical context is defined on the functional level. For a cruise control a context might be keeping the speed while driving uphill or accelerating to a given speed. This is translated into a table of input data variables with the corresponding values for every context. To specify context and free variables in a flexible way the language XML is used. Further on, the precise input data values for the measurement need to be specified.

In Table 1 the contents of the XML structure is given with context variable a and free variables X and Y. Two combinations of test patterns for the free variables are sufficient to execute all feasible branches of the program of Fig. 1 for this context.

Field	Value
contextVar	a
freeVar	X Y
testCase 1	a=1; X=1; Y=12;
testCase 2	a=1; X=6; Y= 1;

Table 1. Example of XML Specification

4.3. Extentions of Symta/P to Contexts

Current approaches for static timing analysis assume that input data is unknown and additional information has to be specified by the user, such as loop bounds [13] [6]. Context information, as described in Section 4.2, are specified by the predefined variables. The path analysis is extended by classifying context variables as input data independent.

This is done by extracting the names of the context variables from the XML structure. In the initialization step, a symbol table is filled with these variables and classified as input data independent. Whenever a context variable is used in other expressions the input data classification is simply propagated. As a result, the program is structured into larger segments according to the context specification. No changes are necessary for architecture modeling and ILP formulation because context dependent control flow is independent of the target architecture.

4.4. Overall Tool Flow

Fig. 2 shows the tool flow for the context sensitive performance analysis. Rectangular shapes denote tools or user actions and circular shapes denote input, intermediate or output files. The analysis starts with the c file and the context specification in the upper left corner. From the abstract model context definitions are extracted and specified in `Context` using the XML interface, as described in Section 4.2. Together with the source file `f.c` the path analysis identifies input data dependent control flows, classifies the source code into segments `f.seg` and outputs the control flow graph `f.cfg`. According to architecture specific measurement instructions `Syntax` and the segment classification the source code is instrumented for measurement `f.path.c` and for `f.instrum.c` branch identification.

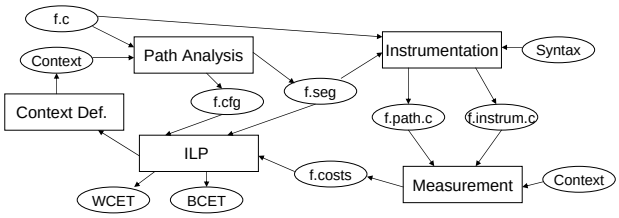


Figure 2. Overall Tool Flow of Symta/P

The instrumentation syntax contains instructions specific for target architectures to measure the execution time or other properties like power consumption of a program segment. The syntax of the instrumentation depends on the measurement strategy. When processor simulators are used an instrumentation consists of debugging commands, such as stopping the execution and reading the system timer. For direct measurement on target architecture of automotive software, an instrumentation might be a CAN message.

For input dependent control structures the branch identification is used to determine which branch is taken during the execution of a given set of input data. This is also necessary for back annotation of the execution time. The instrumentation for branch identification also depends on the architecture modeling strategy. For example, the line numbers of the executed branch could be recorded in a local variable or written out by a CAN message.

Given the `Context` specification and the instrumented source code, the execution time is measured either by cycle accurate processor simulator or by direct measurement on an evaluation board. Using the branch identification the times for every segment are back annotated. The `Context` file in the upper left and lower right corner have the same content and are shown twice for notational convenience.

Finally the ILP problem is formulated using the control flow graph `f.cfg` the segment classification `f.seg`

and the execution time file `f.costs` and solved by an ILP solver. The solutions for the WCET and the BCET are now available for the software if all program segments have been reached. If not, the user has to supply additional test cases for this context in the `Context` XML file and the analysis continues.

5. Experiments

The context sensitive analysis described in Section 4 has been applied to modules of a cruise control software for a TriCore architecture with 75 MHz clock frequency.

5.1. Experimental Setup

The cruise control consists of several moduls and was designed with ASCET/SD's graphical environment and the source code was generated automatically from it. Then, Symta/P has been used to analyze the source code and to extract the control flow graph (CFG). From the ASCET/SD model and from the CFG the contexts have been defined by specifying the free and context variables as XML format. Table 2 shows the size of the software modules in C lines without considering the size of sub-functions and the number of free and context variables. The real names of the modules had to be removed because of non disclosure agreements.

Context.	#cLines	#freeVar	#contextVar
A	57	4	9
B	33	7	12
C	27	4	12
D	34	6	12
E	33	5	12

Table 2. Context Specification

The testing-environment is a Triboard, equipped with a variety of peripherals for connecting the environment and a TriCore microcontroller. The TriCore is a 32-bit microcontroller-DSP architecture for real-time embedded systems. Binaries are created with GNU Cross-Compiler toolchain for TriCore. Hard real-time debugging requires close interaction with the processor. JTAG OCDS (On-Chip Debug Support) offers direct access to microcontrollers with an OCDS module. It provides a direct serial interface to the controller-internal functional units (registers, busses, control unit etc.) to make accurate measurements of execution time and path identification feasible in real time. The JTAG OCDS is connected via flat ribbon with the UAD (Universal Access Device). UAD is a high-speed communication hardware for interconnection between microcontroller-boards and desktop PCs. On PC-side the UDE (Universal Debug Engine), a debugging-software for microcontrollers, is used. For reproductional purposes and to achieve the best cycle performance using the TriCore non-cached memory, both the program code and data are allocated in the internal scratch pad memory. When the instrumented program segment starts and stops, the cycle information of the actual timestamp and the c-line number is saved in an array. This array is allocated in the non-cached internal memory. Start time is the time stamp when the segment starts and stop time is the time stamp when the segment ends. Thus, the difference between these two time stamps represents the time taken by the run of the program segment.

5.2. Results

Several test cases were specified for a complete branch coverage for every context. Table 3 shows the number of test cases for each context. The number of test cases ranged from 14 for small modules to 47 for larger ones.

Then for these modules the execution time was determined by high level simulation using the existing functional testing equipment. In this experiment the complete module was tested by inserting an instrumentation point only at the beginning and at the end of the module. Table 3 summarizes the best case and worst case execution times. The execution

Context	#test cases	BCET [clk]	WCET [clk]	range [clk]
A	47	840	1152	312
B	20	324	336	12
C	18	342	450	108
D	14	126	156	30
E	14	102	156	54
no cont.	113	156	1152	996

Table 3. Context sensitive analysis using simulation

time interval without considering context is between 156 and 1152 cycles, a range of 996 cycles, whereas the largest range is 312 cycles for context A and the smallest is 12 cycles for context B.

For simulation, software modules were chosen which could easily be tested. As a second experiment, more complex software modules were statically analyzed by considering context information. Table 4 shows the execution time for the worst and best case. Each context corresponds to a module that contains several function calls. For example, context G consists of 14 function calls to other modules and has a total of 161 lines of code.

Context	BCET [clk]	WCET [clk]	range [clk]
F	72	794	722
G	72	320	248
H	72	1896	1824
I	72	1686	1614
J	156	156	0
K	72	616	544
no cont.	72	1896	1824

Table 4. Context sensitive analysis using the static timing analysis

In almost all cases the best case execution time is 72 cycles because the software module consists of a sequence of `if-then-else` structures and in the `then` branch there is a `return` statement. In the best case the function executes only the first `if` condition and exits. The range without context information is 1824 cycles that is the same range as for context H. However, the ranges for context F, G and K are much smaller (248 - 722). In context J no input dependent flow leads to the same execution time for best and worst case.

6. Summary and Conclusion

This paper has proposed a methodology to define and characterize context information to give a basis for a performance validation of software modules. Our formal analysis approach has been extended by context specification and a general tool flow with context sensitive analysis has been described.

The approach was applied to an industrial automotive control application. The experiments show that our approach is easy to use and flexible enough for complex applications. Timing bounds by simulation and by applying the static timing analysis lead to tighter timing intervals for each software module.

To conclude, context sensitive analysis makes formal performance analysis approaches feasible for software modules. Input data dependent control flow can systematically be restricted to software modules reducing the number of necessary input patterns. Further work will involve automatic generation of context information from functional specification documents and ASCET/SD models and the automation of the context sensitive timing analysis.

References

[1] Absint GmbH, www.absint.de.

[2] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *RTSS, Real-Time Systems Symposium*, Austin, TX, USA, December 2002.

[3] G. Bernat, A. Colin, and S. Petters. pwcet: a tool for probabilistic worst case execution time analysis of real–time systems. Technical report, University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr. 2003.

[4] Bound-t execution time analyser. www.bound-t.com.

[5] J. Engblom, A. Ermedahl, and F. Stappert. A worst case execution time analysis tool prototype for embedded real time systems. In *RTTOOLS Workshop*, Aalborg, Denmark, August 2001.

[6] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 1999.

[7] A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, editors. *Embedded Software for SoC*, chapter Formal Methods for Integration of Automotive Software, pages 11–24. Kluwer Academic Publishers, Aug. 2003.

[8] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded system design. In *Proceeding Design Automation and Test in Europe*, March 2004.

[9] M. Jersak, K. Richter, R. Ernst, J.-C. Braam, Z.-Y. Jiang, and F. Wolf. Formal methods for integration of automotive software. In *Designers Forum at Design, Automation and Test in Europe Conference*, pages 45–50, March 2003.

[10] R. Kirner and P. Puschner. Discussion of misconceptions about wcet analysis. In *WCET Workshop*, TODO, 2003.

[11] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th Euromicro International Workshop on WCET Analysis*, Catania, Italy, June 2004.

[12] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET Workshop*, pages 85–88, 2003.

[13] S. Malik and Y.-T. S. Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[14] Symta/p. www.ida.ing.tu-bs.de/research/projects/symta.

[15] F. Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

[16] F. Wolf. Integrationsverfahren fuer Softwaresysteme im Antriebsstrang. In *Electronic Systems for Vehicles*, Baden Baden, 25.-26. Sept. 2003. VDI-Berichte 1789.

[17] F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, pages 339–356, April 2001.

[18] W. Ye and R. Ernst. Embedded program timing analysis based on path clustering and architecture classificaton. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD '97) San Jose, USA*, pages 598–604, 1997.

[19] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Sixth International Workshop on Hardware/Software Co-Design*, pages 9–13, Seattle, 1998.