Cost-efficient worst-case execution time analysis in industrial practice [1]

Jan Staschulat, Jörn C. Braam, Rolf Ernst
Technical University Braunschweig, Germany
Hans-Sommer-Str. 66, D-38106 Braunschweig
Email: staschulat|braam|ernst@ida.ing.tu-bs.de

Thomas Rambow, Rainer Schlör Rainer Busch
Ford Forschungszentrum Aachen GmbH, Germany
Süsterfeldstrasse 200 D-52072 Aachen
Email: trambow|rbusch1@ford.com,
Rainer_Schloer@gmx.de

***Abstract*— To guarantee real-time behavior of an embedded application, a schedulability analysis can be used. Such an analysis requires the worst case execution time (WCET) of the application. While several academic approaches to conservatively bound the WCET have been proposed in the last decade, common practice in industry remains simulation and software tests. One reason is that industrial requirements are not sufficiently addressed by academic approaches.**

In this paper we identify important industrial requirements for WCET-analysis tools. Then, we describe the methodolgy of a previously developed WCET-analysis approach and revise important aspects of its methodology and its implementation to address key industrial requirements. In a large-scale case study the WCET-analysis tool is applied to a safety-critical automotive control application to evaluate the applicability of the tool. Furthermore, the faced challenges and the re-targenting costs for a new processor are discussed.

I. Introduction

Timing requirements have become increasingly important in embedded system design. For example, software control applications continuously interact with their environment. To accurately implement a control function, these systems not only have to perform correctly but also have to provide the computed results within specified time bounds.

Schedulability analyses are used to guarantee that an application finishes within a given time bound, which is also called real-time behavior. Such schedulability analyses assume that the worst case execution time (WCET) of each application is given. Then, the worst case schedule is constructed including all interferences due to interrupts, task preemptions and blocking times.

A simple technique to estimate the WCET is software tracing and sampling. Industrial tools include RTA-Trace by ETAS [9], CodeTEST by Freescale [11], and a hybrid tracing and sampling approach by Nokia [18]. The concept is to insert measurement points at the beginning and at the end of each function to measure end-to-end execution times or around critical sections to monitor special events. The program is executed multiple times with different stimuli patterns, or input data. The main problem of such an approach is that it is difficult to guarantee that all execution paths are covered by the specified input data. For a full path coverage, an exponential number of input data would be necessary, which is strictly too time-consuming. Safe bounds of the WCET cannot be guaranteed if only a subset of all input patterns is considered.

Many static timing analysis approaches have been proposed in the last decade, e.g. [19] [1] [7] [15] [27] [10]. Some commercial tools for static WCET-analysis are available, e.g. aiT by AbsInt GmbH [1] and Bound-T by Tidorum Ltd [5] and industrial experience reports have been published [22] [6]. Most static analysis methods assume precise hardware models to compute the execution time of individual basic blocks. However, the main difficulty is that precise hardware models are not always available and would be too expensive to construct.

In measurement-based WCET analysis approaches, such as [4] [20] [26] [27] [16], a program is partitioned into program segments, whose execution time is then measured on real hardware or on processor simulators. The size of a program segment can be defined in several ways: by a single basic block [16]; by a measurement block [20], whose size is manually defined; or by a sequence of basic blocks, depending on the number of execution paths of the program segment [26], the structure of the control flow graph [4], and depending on the input data dependency of conditions in control structures [27]. The approach of probabilistic timing analysis by [4] is commercially available by Rapitime by Rapita Systems [21]. One drawback of this approach is that analysis precision depends on the quality of test patterns. Test data for just branch coverage are usually not sufficient to cover the dynamic timing behavior due to caches and pipelines, as the history of execution paths is not considered.

Measurement-based approaches have the advantage of being cost-efficient and easily to re-targeted to new microprocessors, as only evaluation boards are necessary but not formal model of the hardware. From the industrial perspective, these requirements are very important for a fast development of an embedded product [14].

In previous work we have developed a static timing analysis tool SymTA/P, which is based on measurements and static analysis. The main benefit is its easy retarget-ability to new processor architectures. The approach can be applied to many processors in a short time delivering safe bounds of the worst case execution time. Thus, we argue that SymTA/P is cost-efficient, because it is simple and not expensive to

[1]This work was partially funded by Ford University Research Program and by ARTIST2 - Network of Excellence on Embedded Systems Design.

978-0-7695-3071-0/07 $25.00 © 2007 IEEE
DOI 10.1109/ISoLA.2006.64

apply the WCET-analysis to a new processor architecture. However, several assumptions of the tool limit the applicability to industrial projects.

The contributions of this paper are the following. We identify important industrial requirements for WCET tools as a result of a discussion with an automotive manufacturer. As a second contribution, we describe which revisions were necessary such that the previously developed WCET-analysis tool meets these requirements. As a third contribution, we present an industrial case study in which we demonstrate the applicability of the WCET-analysis method and describe the faced challenges.

The remainder of this paper is structured as follows. In section II we present important industrial requirements for WCET-analysis tools. In section III we give an overview of our previously developed WCET-analysis tool and in section IV we present the revised analysis framework. In section V we present the industrial case study, before we conclude in section VI.

II. Industrial requirements

Although many methods for WCET analysis have been proposed in academia in the last decade, very few approaches are being used in industrial projects. One reason is that industrial requirements differ from research goals in academia. The following list is based on [14] and is a result of a discussion with a automotive manufacturer who supplied the case study, as presented in section V.

- **Safe bounds.** The results of the analysis must be safe. This means that the execution time bound is always larger than any execution time of the task.
- **Integration.** The method has to be integrated in the standard development process. In industry, software is often developed with model-based tools, like Matlab/Simulink [17] and source code is automatically generated, e.g. with TargetLink by dSpace [25] or with ASCET by ETAS [3].
- **Easy usage.** The tool must work with minimal user interaction.
- **Cost-efficiency.** Cost-efficiency is crucial in mass production products, like automotive and telecommunication market. Typically, several processor alternatives are explored and thus the WCET-method has to be customizable to a new processor.

III. Earlier work on SymTA/P

SymTA/P is an acronym for SYMbolic Timing analysis of Processes. In this context, a process is a software task, or application, that is uninterruptedly executed on a single processor. In this section we give an overview of the functionality, the employed methodologies and the limitations of the WCET-analysis tool that has been developed in earlier work. [27], [28]

A. Functionality of the tool

The tool calculates a safe upper bound of the WCET and a safe lower bound of the best case execution time (BCET) of a task written in the programming language C. The method supports processors with pipelining as well as instruction and data caches. The SymTA/P method is cost-efficient and easy to re-target to new processors because it is based on measurements on real hardware.

As an additional feature, a safe bound of cache-related preemption delays for fixed priority preemptive scheduling is calculated [23]. This is important for a tight schedulability analysis in case several tasks share the same instruction cache.

B. Employed Methodologies

The SymTA/P method is structured in several modules: Frontend, Backend, Cache analysis, and ILP solver, as show in figure 1.

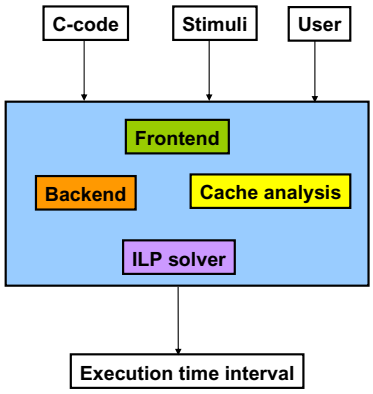

Fig. 1. Overview of SymTA/P tool.

In the *Frontend*, the control flow graph is constructed from source code and is partitioned into program segments. These program segments are as large as possible but containing only a single feasible path (SFP). A SFP is a sequence of basic blocks whose control flow is independent of input data. For example, a `for`-loop containing several if-then-else statements whose conditions only depend on loop-iteration variables is classified as a SFP. A FIRFilter (finite response filter) and FFT (fast fourier transformation) are further examples. If these conditions were dependent on input data then, the loop would be classified as a multiple feasible path, where each program segment has the size of a basic block.

In the *Backend*, the program is instrumented with instrumentation points and the execution time is measured for each program segment. Measurements can be done on real-hardware, e.g. an evaluation board, or on a cycle-accurate processor simulator. Sufficient test patterns have to be provided such that each program segment is executed at least once. The amount of input data for this coverage criteria is comparable to the branch coverage, which means that each condition is executed taken and not-taken.

The problem of measurement-based approaches is to enforce the worst-case initial hardware state. In practice it is very difficult to specify them at the beginning of each program segment. In the probabilistic WCET approach by [21], each

program segment is measured including the effects of the execution history. Thus, all execution paths are necessary to detect the worst case execution time of a program segment because all possibilities of the the execution history have to be considered. However, a full path coverage is too time-consuming for large industrial projects. To tackle this problem in SymTA/P, a conservative overhead is added to the measured execution time of each program segment to account for the worst case initial hardware state. The overestimation of the WCET for a task scales with the number of program segments on the worst case path.

A static *cache analysis* for instruction as well as data caches is based on data flow analysis techniques. These analyses provide a tight and safe bound for worst case cache behavior. The analysis supports associative instruction caches assuming LRU replacement strategy and direct mapped data caches with write through and no-write allocate write miss strategy. We further assume a constant cache miss penalty.

The WCET bound is calculated by implicit path enumeration using the technique of integer linear programming (ILP). The measured execution time plus the time overhead for the initial worst case hardware state of each program segment has been determined in the backend and cache analysis module. The longest execution time of the task can be calculated by an *ILP solver* by searching for the longest path (in terms of execution time) in the control flow graph of the task.

C. Limitations

A research prototype has been developed, however, several limitations exist.

1) The ability to partition control intensive software applications into logn SFP program segments is very limited, e.g. in automotive ECUs. Automatically generated C source code, e.g. by TargetLink [25] or ASCET-SD [3], contains long `switch-case` statements with frequent function calls in `if`-conditions. In this case, a SFP program segment has the size of a basic block. This leads to many instrumentation points and thus to a higher total overestimation.
2) Program path analysis to identify SFPs is based on symbolic simulation and is restricted to a subset of the C programming language. Most importantly, sub-function calls, `switch-case` statements, pre-compiler statements and `break`-statements are not supported.
3) Global WCET calculation in the ILP solver module has been limited to single functions. Sub-function calls are not supported.
4) The WCET-tool requires Linux operating system. However in industry, any sophisticated tool has to be provided for the Windows operating system.

In summary, the WCET-tool has to overcome limitations due to its methodology, e.g. the partitioning in program segments (1), and several infrastructure restrictions, e.g. support of full C syntax and semantics (2), support of sub-function calls (3) and support for Windows operating system (4).

IV. Revised SymTA/P analysis framework

This section describes the necessary changes and enhancements of the SymTA/P analysis tool to meet the industrial requirements (section IV-A) and describes the design of the analysis framework in detail (section IV-B to section IV-F).

A. Satisfying the requirements

The realized changes address technical issues, e.g. the supported C language syntax, as well as theoretic issues, e.g. the placement of instrumentation points to reduce the measurement overhead.

Placement of instrumentation points

First, the placement of instrumentation points is revised. In a novel instrumentation methodology the number of instrumentation points is minimized while requiring only test patterns for full branch coverage. The instrumentation method inserts instrumentation points not at each basic block, which would be necessary if no SFPs are found, but rather around larger program segments [24].

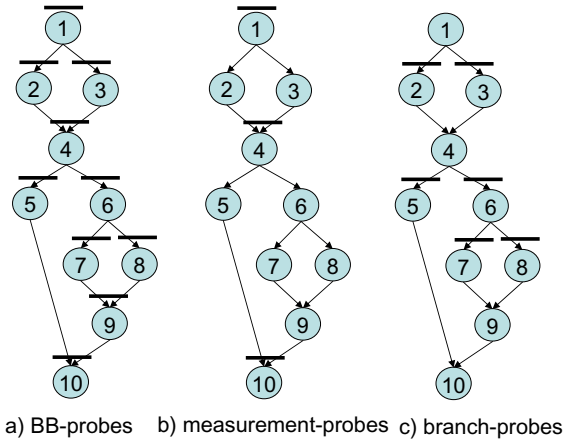

Fig. 2. Control flow graph with different instrumentation points.

The idea is to partition the program into program segments, which are longer than a single basic block, but short enough that test patterns for full branch coverage are sufficient. For this purpose, the instrumentation methodology partitions the instrumentation points into *measurement probes* and *branch probes*. The term *instrumentation point* denotes the place in the control flow graph. The terms *measurement probe* and *branch probe* are concrete implementations for a given hardware architecture that correspond to an instrumentation point.

An example of an instrumented control flow graph is shown in Figure 2 in three variations. Figure 2 a) shows a traditional instrumentation at each basic block. Figure 2 b) shows the placement of measurement probes and Figure 2 c) shows the placement of branch probes. Instead of separately measuring 10 basic blocks we insert only 3 measurement probes. The timing analysis constructs an ILP based on a program segment graph. A program segment is constructed by all possible paths between two measurement probes. Due to space limitations,

the program segment graph is not shown here. A detailed description is given in [24].

This method has two benefits. The first benefit is a reduced number of time-critical measurement probes for which a time-overhead has to be accounted for in the WCET analysis. The purpose of the branch probes is to monitor which branch is taken during program execution. The execution of of branch probes is not time-critical, because only the c-line number is monitored during program execution. The second benefit is that test patterns for full branch coverage are sufficient.

Technical issues

Support of full C syntax and semantics. The Frontend has been considerably revised to support the full C syntax and semanctics, because the previous prototype was limited to a small subset of C language. The program path analysis to identify SFPs has been extended to more complex control structures like nested loops, sub-function calls, `switch-case` statements, pre-compiler statements and `break`-statements.

Sub-function calls. The WCET-calculation in the ILP solver module has been extended to support sub-function calls, which is essential for larger software applications. A bottom-up approach is used by constructing a function call tree. First, the WCET of functions are calculated that do not depend on other sub-functions (e.g. at the leaf nodes). Then, the WCET of all functions on the next higher level on the call tree are computed. We assume non-recursive functions or that recursive functions are bounded by the depth-level as the loop bound for loops. Recursive function calls are not supported. Function calls from different contexts are (not yet) supported. Each function is analyzed once regarding its best case and worst case timing behavior. Then, this timing characterization is used in every callee. This analysis is not context-sensitive.

Windows operating system. The analysis framework has been ported to Windows platform using the Cygwin [8] environment. Cygwin is a software package which allows Linux programs to be run on a Windows platform. Thus, all programs have been re-compiled under Cygwin while the graphical user interface, which is written in Java, is supported natively by Windows.

B. Overview of modular framework

An overview of the framework is shown in Figure 1.

The hybrid approach consists of a path analysis module (Frontend), an execution time measurement module (Backend), a cache analysis module, and a WCET calculation module (ILP solver). In the following subsection, these modules are described in more detail.

C. Program path analysis (Frontend)

The program path analysis module corresponds to the Frontend in Figure 1. Based on the C source-code, the program path classification and the control flow graph of a software task are created by a Frontend parser, as shown in Figure 3. This Frontend parser reads the abstract syntax tree, which is generated by GNU gcc compiler, and performs a symbolic simulation on the abstract syntax tree to determine input data dependency. As mentioned above, this Frontend supports the full C syntax and the identification of SFP was accordingly extended.

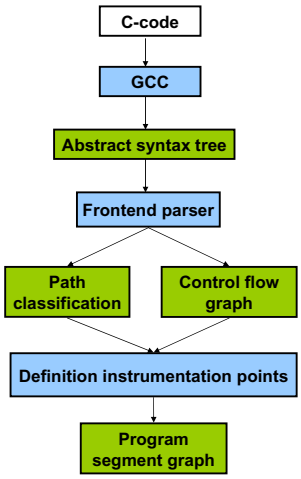

Fig. 3. Frontend.

Based on the control flow graph and the path classification, the placement of measurement and branch probes is implemented in the `Definition instrumentation points` phase.

D. Execution time measurement (Backend)

The execution time measurement, called *Backend*, is shown in Figure 4. Cycle-accurate processor simulators or cost-efficient evaluation boards can be used to measure the execution time of program segments.

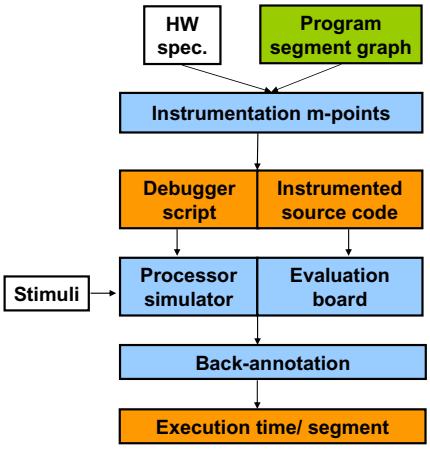

Fig. 4. Backend.

If an evaluation board is used, the C source code is instrumented once with measurement probes and once with branch probes. Each probe is a function call that is defined later in the linking phase by processor dependent assembly instructions.

Then, both instrumented programs are executed with the same set of input data. The first one delivers execution time results for program segments, and the second identifies which execution path was taken. With both information, the measured execution time can be annotated back to the corresponding program segment. For a complete WCET-analysis, sufficient input data for full branch coverage has to be supplied.

If a cycle accurate processor simulator is used, a debugger script is automatically generated that starts and stops the execution of the program and traces the taken branches. The output is also back annotated to the program segment.

E. Cache Analysis

The cache analysis in SymTA/P is shown in Figure 5.

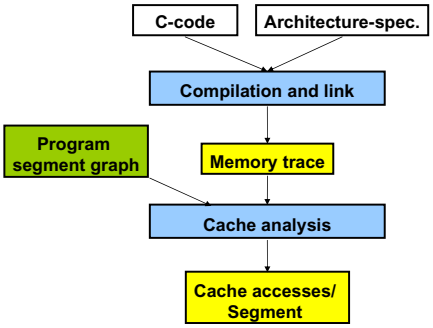

Fig. 5. Cache analysis of SymTA/P.

The measured execution time and the cache behavior are estimated in two phases. First, the measurement is done by assuming each cache access is a *cache hit* and in a second step the cache analysis converts some cache hits to cache misses to account for the worst case timing behavior. Based on the program segment graph and the memory trace files, a static cache analysis [28] is applied.

Practically, the cache hit assumption can be enforced by storing the program code in the cache, e.g. the cache is large enough, by storing the program code in equivalently fast memory, such as Scratch-pad RAM on TriCore processor [13], or by using a processor simulator where memory access times can be configured, such as RealView Development Suite for ARM cores [2].

F. WCET Computation

The longest and shortest paths in the control flow graph are found using integer linear programming (ILP). This module is shown in Figure 6. The measured execution time and the cache access behavior are added and used as execution time costs for each program segment. The ILP is constructed using structural constrains, which are automatically generated from the program segment graph. Loop bounds have to be specified by the user to bound the maximum number of loop iterations.

If for some program segments no execution time was assigned, the ILP solver issues a warning. In this case, the user

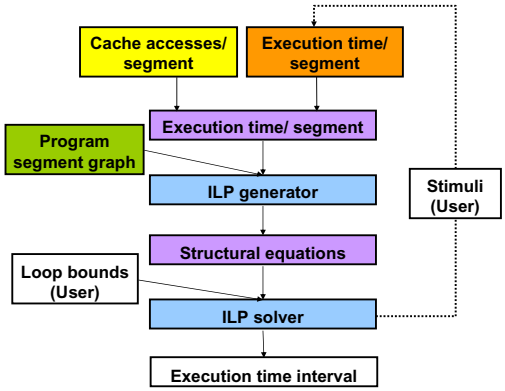

Fig. 6. WCET-calculation with ILP solver.

can specify additional test patterns (stimuli) and can re-run the timing measurement (Backend). Finally, the total best-case and worst-case execution time become available.

V. Case Study

In this case study we apply SymTA/P to a industrial project of an real-time application. First we describe in section V-A the software and hardware setup. Then, we describe in section V-B preliminary steps and in section V-C the application of SymTA/P. In section V-D we present the results of the WCET analysis and in section V-E we list the faced challenges. In section V-F we list which steps are necessary for a customization and give an outlook in section V-G.

A. Software and hardware setup

The WCET-analysis is applied to a safety critical automotive control application. It is part of an automatic start-stop function for a micro-hybrid vehicle. It consists of a C-source file with about 5800 lines of code. The largest function has about 4200 lines. Totally, there are 11 functions in the application.

The measurement framework consists of a C167CR evaluation board, a C compiler (Tasking), bootloader (I+ME Actia) and a tailor-made measurement methodology for the C167CR board. The C167CR evaluation board is equipped with 2kB internal (on-chip) RAM for registers and stack and 256kB static RAM for code. The board does not provide ROM. Communication is only possible over serial bus; JTAG or other debug interfaces are not available.

Now, we describe each step of the process of timing analysis with SymTA/P. We have structured the process into preliminary steps and application of SymTA/P.

B. Preliminary steps

Before we can apply SymTA/P the following preliminary steps must be undertaken. The module of SymTA/P, in which this feature is implemented, is denoted in parentheses.

- **Generation of C code for application (Frontend)** The application is developed by Matlab/Simulink. The

C source code without assembly instructions was automatically generated.

- **Communication with microcontroller (Backend).** With a bootloader the program was loaded onto the evaluation board. Since the C167CR board has only serial interface, an additional protocol had to be developed to write information back from the board to the host PC. We have used a bootloader by IME+ACTIA and a terminal program, which records all communication in a text file.
- **Create test harness (Backend).** Test patterns are manually defined using the Matlab/Simulink environment [17]. The tests are hierarchically structured in test sets and test cases. Each test set consists of several test cases. A test set corresponds to a sequence of input signals (test case). A test case corresponds to a set of input signals for a single simulation step of the application. In total, 65 test sets are specified, with about 4,185 test cases for each test set; in total, 272,022 test cases.
- **Test pattern harness (Backend).** The application reacts to external signals, thus we wrote a test pattern harness which controls the values of these signals and activates the software application. Because of memory space restrictions and the high number of test cases, we developed a method which compresses the total number of input signals.
- **Worst-case state overhead (Backend).** For a conservative timing analysis in SymTA/P, the worst case system state at the beginning of each program segment has to be analyzed. The C167CR processor consists of a 4-stage instruction pipeline [12]. The clock frequency is 20 MHz (50ns) and one CPU cycle needs two clock cycles. (100ns). To simplify the following discussion we denote an instruction cycle (I-cycle) as two clock cycles. Figure 7 shows the instruction pipeline with the stages: instruction fetch (IF), instruction decode (ID), execute (EXE) and write back (WB). The vertical line denotes the I-cycles.
  Assume we could exactly measure the execution time at each pipeline stage. For the simple C167CR processor without cache and branch prediction, it is conservative to assume an empty pipeline as worst case system state. The total execution time of the program segment is the time between the first instruction enters the pipeline (IF stage) and the last instruction leaves the pipeline (WB stage): In I-cycle 1 the first instruction is about to enter the pipeline and in I-cycle 7 the last instruction is finishing in the WB stage (dashed circles). Thus, the total execution time for these instructions is $7 - 1 = 6$ I-cycles.
  Assume now that we insert an start-timer instruction (A) before and a stop-timer instruction (B) at the end of the program segment. The execution time is recorded when the instruction is in the EXE stage. Then, the measured execution time using start-timer and stop-timer instructions is $7 - 3 = 4$ I-cycles (solid circles). We assume no additional overhead for the completion of the timer operation (a special function register `T6CON` is set at the C167CR processor). Thus, the underestimation would be 2 I-cycles compared with exact measurement.
- **Measurement overhead (Backend).** To implement the start-timer and stop-timer, we use the GPT2 timer of the C167CR processor. It runs with one fourth of the CPU clock (200ns) and thus has a maximum resolution of two instructions (since one CPU-cycle takes two clock cycles). If the program segment finishes in an un-even number of I-cycles, then the timer would report one I-cycle less then the correct value. Thus, to be safe, we add one I-cycle as measurement overhead. In this analytic discussion we assume no clock skew and jitter.
- **Total added overhead for each program segment (ILP solver).** To summarize, we have to account two I-cycles for the WC state and one I-cycle for the measurement-overhead which amounts to three I-cyles. This overhead of three I-cycles (300ns) is added to the measured execution time of each program segment in the ILP-solver phase.

I-cycle	IF	DE	EXE	WB
1	A			
2	1	A		
3	2	1	A	
4	3	2	1	A
5	B	3	2	1
6		B	3	2
7			B	3
8				B

Fig. 7. C167CR Pipeline with start (A) and stop (B) timer instruction.

C. Application of SymTA/P

We apply SymTA/P to the case study in the following steps: Frontend, Backend and ILP solver. Because the C167 does not have a cache, no cache analysis is performed.

- **Frontend.** In the path analysis module, the control flow graph is automatically constructed for each function and the code is instrumented with measurement and branch probes. The control flow graph (for basic block instrumentation) consists of 1102 nodes and the program segment graph contains 893 program segments. These numbers are totals, e.g. for all 11 functions of the application. In this case study, 354 measurement probes and 511 branch probes were inserted in the source code. If each basic block were instrumented, 865 time-critical measurement probes would be necessary. Thus the new instrumentation method reduces the number of time-critical measurement probes by 59% compared to instrumenting each basic block.
- **Backend.** The implementation of the probes for the C167CR processor is shown in Figure 8. The measurement probe is a macro which outputs a system timer value and the branch probe outputs only the c-line. Once the execution time has been measured on the C167 board, they are annotated back to the program segment graph.

- **ILP solver.** Finally, the longest execution path is calculated by the ILP solver. The ILP was automatically constructed and no additional user interaction was necessary because the software did not contain loops.

D. Results

The static timing analysis results were compared to simulation results. The entire application was instrumented by only two measurement-probes: one after the first line of code and one before the last line of code. Then, the total execution time was measured between these measurement-probes. The distribution of the execution times for all test cases is shown in Figure 9. The longest execution time was 491 micro seconds (μs). The range of execution time is between 326 and 491 μs.

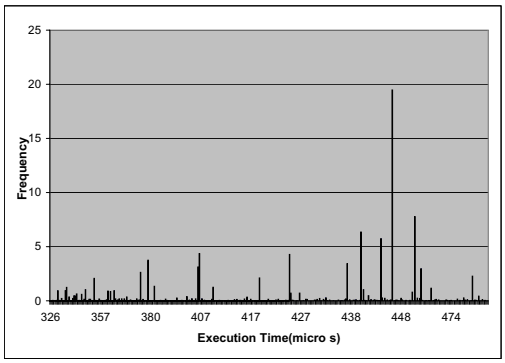

Fig. 9. Execution time distribution obtained from simulation.

The static timing analysis of SymTA/P determines the WCET of 797 μs, which is about 62% longer than the tracing result. However, this cannot be considered as an overestimation, because it cannot be guaranteed that the worst case path was actually measured during tracing.

The time spent in each phase is summarized in table I. The total time was about 54 hours for the entire case study.

Phase	Time
Program path analysis (CFG)	6min
Instrumentation of measurement points (Board)	1min
Configuration and compilation in Tasking EDE	5min
Execution on C167 board (measurementProbes)	22h
Execution on C167 board (branchProbes)	20h
Back-annotation of timing results	11h
WCET calculation (ILP Solver)	1min
Total	54 h

TABLE I

TIME REQUIREMENT FOR EACH ANALYSIS PHASE.

E. Practical challenges in case study

The long running time for the static timing analysis is unacceptable. The main reason is the execution time measurement on the evaluation board and not the running time of the SymTA/P analysis framework itself. The time for *execution on C167 board* in Table I includes the communication via serial line. Most of the time is spent for communication, which is very slow. The running time could be significantly reduced by using a more sophisticated measurement methodology, e.g. JTAG interface or by using a evaluation board with sufficient on-board memory. The bottleneck was the communication between C167 board and the host PC. The back-annotation software is a prototype and has not yet been optimized, which is rather a technical issue.

F. Customization for new application

The Backend provides a framework that is easy to retarget to new processors. In general the preliminary steps in section V-B must be carried out. Most importantly, this includes

- Test patterns have to be specified for full branch coverage. Usually these test patterns are already available from previous functional test phases in industrial projects. SymTA/P does not provide a test pattern generator. However, the WCET-calculation module highlights the source-code lines that were not measured.
- The implementation of the measurement and branch probes has to be defined for the new processor.
- The measurement framework has to be adapted to the new microcontroller. This includes the communication to load the program to the board, to start and stop the program, to access system timers, and to write back timing results to the host PC.
- The time overhead for the worst case initial hardware state for each program segment has to be calculated.

G. Outlook

Traditional WCET-analysis approaches focus only on a single task execution, but complex interactions with other tasks have to be considered when caches are used. This includes cache related preemption delays which are due to task preemptions in preemptive real-time scheduling and, as well as the cache contents at the beginning of a task activation, because some cache lines might be availabe from a previous activation.

Furthermore, latencies to background memory have been assumed to be constant. However assuming a constant cache miss penalty might result in pessimistic overestimations for realistic applications.

VI. CONCLUSION

Performance validation is a key issue for real-time embedded systems. In this paper we have presented industrial requirements for WCET-analysis tools and revised an earlier prototype to meet these requirements. Then, we described the measurement-based WCET analysis methodology in detail and applied it to a industrial case study. We have discussed the faced challenges and pointed out open issues.

The SymTA/P analysis framework is cost-efficient and easy to re-target to new processors because it is based on measurements of program segments. In addition, the method is safe, and provides a conservative upper bound of the worst case execution time.

```
#define measurementProbe(l) T6CON &= ~(0x0040);\
    printf("timing %u %lu00\n", (l),(( (unsigned long int) \
  (T5) << 16 ) | T6)* (TICK_NS/100) ); T6CON |= 0x0040;
#define branchProbe(l) printf("branch %u\n", (unsigned int) l);
```

Fig. 8. Implementation of probe functions for C167 evaluation board.

VII. Acknowledgments

This work was partly funded by Ford University Research Program and by ARTIST2 - Network of Excellence on Embedded Systems Design. We would like to thank Jonas Diemer for his valuable work regarding the hardware setup of the C167 evaluation board.

References

[1] Absint GmbH, www.absint.de.
[2] ARM: Realview development suite. http://www.arm.com.
[3] ASCET-SD embedded control development systems for automotive solutions. ETAS, August 2002.
[4] A. Betts and G. Bernat. Issues using nexus interface for measurement-based wcet analysis. In *WCET Workshop*, Palma de Mallorca, Spain, 2005.
[5] Bound-t execution time analyser. www.bound-t.com.
[6] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static wcet analysis to automotive communication software. In *Euromicro conference on real-time systems (ECRTS)*, pages 249–258, 2005.
[7] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, 18(2-3):249–274, 2000.
[8] Cygwin Homepage. www.cygwin.com.
[9] ETAS. *RTA-TRACE Datasheet*, June 2004.
[10] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system c programs. In *IEEE International workshop on object-oriented real-time dependable systems (WORDS)*, Sedona, USA, 2005.
[11] Nat Hillary. *Beyond Profiling Gaining Control of Software Performance*. Freescale Simiconductor, Nov. 2005.
[12] Infineon Technologies AG. *C167CR Derivatives, 16-Bit Single-Chip Microcontroller*, March 2003.
[13] Infineon Technologies AG. *TC1796 User's Manual, V1.0, 32 Bit Single-Chip Microcontroller: System Units*, June 2005.
[14] Raimund Kirner, Peter Puschner, and Ingomar 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.
[15] M. Lindgren, H. Hansson, and H. Thane. Using measurements to derive the worst-case execution time. In *RTCSA*, 2000.
[16] T. Maier-Komor, A. von Bülow, and G. Färber. Metac and its use for automated source code instrumentation of c programs for real-time analysis. In *Euromicro Conference on Real-Time Systems (ECRTS), Work in Progress*, 2005.
[17] The Mathworks Inc.: Matlab/Simulink. http://www.mathworks.com/.
[18] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance data collection using a hybrid approach. In *CM SIGSOFT Software Engineering Notes, Proceedings of the 10th European software engineering conference*, volume 30(5), 2005.
[19] A. Mok. Evaluating tight execution time bounds of programs by annoations. In *Workshop on Real Time Operating Systems and Software*, Pittsburgh, USA, 1989.
[20] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Conference on Real-Time Computing Systems and Applications (RTCSA)*, Dec. 1999.
[21] Rapita-Systems Ltd. www.rapitasystems.com.
[22] Jean Souyris, Erwan le Pavec, Guillaume Himbert, Victor Jegu, Guillaume Borios, and Reinhold Heckmann. Computing the wcet of an avionics program by abstract interpretation. In *Worshop on worst-case execution time analysis*, July 2005.
[23] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *EUROMICRO Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.
[24] Jan Staschulat. *Instruction and data cache timing behavior analysis in fixed priority preemptive real-time systems*. PhD thesis, Technical University Braunschweig, to appear, 2006.
[25] dSpace: TargetLink, automatic code generator. http://www.dspace.com/.
[26] Ingomar Wenzel, Bernhard Rieder, Raimung Kirner, and Peter Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Conference on Design, Automation and Test in Europe (DATE)*, 2005.
[27] F. Wolf, R. Ernst, and W. Ye. Path clustering in software timing analysis. *IEEE Transactions on VLSI Systems*, 9(6):773–782, 2001.
[28] F. Wolf, J. Staschulat, and R. Ernst. Hybrid cache analysis in running time verification of embedded software. *Journal of Design Automation for Embedded Systems*, 7(3):271–295, 2002.