Experiences from Applying WCET Analysis in Industrial Settings[*][†][♮]

Jan Gustafsson and Andreas Ermedahl
Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden
{jan.gustafsson,andreas.ermedahl}@mdh.se

Abstract

Knowing the program timing characteristics is fundamental to the successful design and execution of real-time systems. Today, measurement-based timing analysis tools such as in-circuit emulators, logic analyzers and oscilloscopes, are used in industry. A critical timing measure is the worst-case execution time (WCET) of a program. Recently, tools for deriving WCET estimates, mostly based on static program analysis, have reached the market.

In this article we summarize experiences from five different industrial case-studies. The studies were made on typical industrial systems, in close cooperation with the system developers, using both static and measurement-based tools. The primary purpose was to investigate the difficulties involved in applying current timing analysis methods to industrial code. We were also interested how WCET estimates can be derived by different methods, how labor-intensive the methods are, and the accuracy of obtained results.

As a result, we provide observations on the benefits and drawbacks of the different timing analysis methods used and specify general conditions when a particular method should be most beneficial. We also show the benefits of having several types of timing analysis tools available.

1 Introduction

A *program timing analysis* obtains information about the execution time characteristics of a program. The fundamental problem of such an analysis is that the execution time often varies, with different probability of occurrence, across a range of times. These variations occur due to variations in input data, as well as the characteristics of the software, the processor and the used computer system.

The worst-case execution time (WCET) of a program is a key timing measure. For example, it is an important component of schedulability analysis, it is used to ensure that interrupts have sufficiently short reaction times, that periodic processing is performed quickly enough, and that operating-system calls return to the user application within a specified time-bound. The simplest case is whether a piece of code will execute within its allocated time budget [16].

Reliable timing and WCET estimates are important when designing and verifying many types of embedded systems and real-time systems, especially when the system is used to control safety critical products such as vehicles, aircrafts, military equipment and industrial plants. Basically, only if each hard real-time component of such a system fulfills its timing requirements, the whole system could be shown to meet its timing requirements. For less safety-critical systems exact timing and WCET estimates are not always required. It is really a business issue, where the cost of a timing-related failure has to be weighed against the cost of various means of preventing or handling such a failure. In some applications, only limited parts of the system software are time critical and in need of timing analysis.

Timing and WCET analysis are today performed in a number of ways, using different tools. The two main methodologies employed are *measurements* and *static analyses* (see Section 2). In general, measurements are suitable for less time-critical software, where the average case behavior or an approximate WCET estimate is of interest. For time-critical software, where the WCET must be known, static analysis or some type of *hybrid method* is preferable.

In this article we summarize experiences drawn from five different industrial case studies on timing analysis. We believe that doing case studies, with careful evaluations, provides valuable input both for research and tool development. The studies were made on typical industrial embedded systems and in close cooperation with the developing companies. By the large variety of systems investigated and methods used, we believe that our experiences should be representative for a large class of industrial embedded systems.

All studies were performed as MSc theses works, meaning that the students spent about five months on their work and were initially no system experts. Thus, obtained results can be seen as typical for an analysis made by a well-educated but external person; the work should probably have taken less time for an expert or system programmer.

[*] Supported by KK-foundation (www.kks.se), grant 2005/0271.
[†] ARTIST2 European Network of Excellence provided travel support.
[♮] This article is an extended version of [13].

The primary purpose of the studies was to investigate the practical and methodological difficulties that arise when applying current timing analysis methods to industrial code. In particular, we were interested in how different methods could be used to obtain WCET estimates, how labor-intensive the methods are, and the accuracy of the obtained results. As a result, we provide general observations on the benefits and drawbacks of different timing analysis methods used. We also point out important areas for future research and development of timing analysis tools.

The rest of this article is organized as follows: Section 2 gives an overview and categorization of timing analysis methods. Section 3 shortly presents the different timing analysis tools used in the case studies. Sections 4–8 present, in chronological order, our five case studies. Section 9 presents related work, and in Section 10 we draw some conclusions and present ideas for future work.

2 Timing Analysis Overview

This section gives an overview and categorisation of timing analysis methods, and shortly discusses their benefits and drawbacks. A more detailed description is found in [12].

Measurements (dynamic timing analysis). The traditional, and still most common, method in industry to determine program timing is by *measurements*. Basically, the program is executed many times with different inputs and the execution time is measured for each test run. Measurements are often immediately at the disposal of a programmer, and are useful when the average case timing behavior or an approximate WCET value is of interest.

It should be noted that each measurement run exercises only *one* path. For programs with only one single path, or a few paths, measurements might be feasible. However, for most programs, the number of possible execution paths is too large to do exhaustive testing. This means that the measured times will in many cases underestimate the WCET. To compensate for this, it is common to add a safety margin to the worst-case measured timing, in the hope that the actual WCET lies below the resulting WCET estimate. However, if too much margin is added, resources will be wasted, and if the added margin is too small, the system could become unsafe. Measurements can be made for a subset of the possible input values, e.g., by giving potential "nasty" inputs, which are likely to provoke the WCET, based on some manual inspection of the code. Unfortunately, this approach is not guaranteed to give a safe WCET, especially when complex software and hardware is being analysed.

Measurement-based methods can be divided into *hardware-based* and *software-based* methods. Hardware-based methods include oscilloscopes, logic analyzers, and in-circuit emulators. Software-based methods can be timing functions provided by operating systems, cycle-accurate simulators, or programs provided by tool vendors and designed specifically for execution time measurement.

Most types of measurements have the advantage of being performed on the actual hardware, which avoids the need to construct a hardware model. On the other hand, measurements require that hardware is available, which might not be the case for systems where hardware is developed in parallel with software. In many cases it might also be problematic to set up an environment which acts like the final system. Some measurement methods also introduce the problem of intrusiveness, i.e., the measurements themselves add to the execution time of the analysed program. This problem can be reduced, e.g., by using hardware measurement tools with no or very small intrusiveness, or by simply letting the added measurement code (and thus the extra execution time) remain in the final program. When doing measurements, possible disturbances, e.g., interrupts, also have to be identified and compensated for.

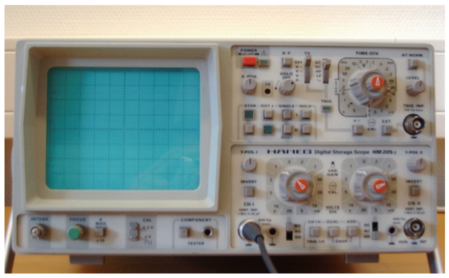
Static WCET analysis. An alternative technique to determine WCET estimates is *static WCET analysis*. Instead of running the program, it derives a WCET estimate by statically analysing the timing properties of the program. Given that inputs and analyses are correct, such a tool will derive a *safe* estimate, i.e., that is larger than or equal to the WCET.

Static WCET analysis is usually divided into three phases: a *flow analysis* where information about the possible program execution paths is derived, a *low-level analysis* where the execution time for atomic parts of the code (e.g., instructions, basic blocks or larger code sections) is decided from a model of the target architecture, and a final *calculation* phase where the derived flow and timing information are combined into a resulting WCET estimate.

Flow analysis research has mostly focussed on *loop bound* analysis [17, 18], since bounds on loop iterations must be known in order to derive WCET estimates. Automatic methods to find these exist, but for many WCET tools, some loop bounds must be provided manually. Flow analysis can also identify *infeasible paths*, i.e., paths which are executable according to the control-flow graph (CFG) structure, but not feasible when considering the semantics of the program and possible input data values [17].

The main issue for the low-level analysis is the complex behaviour of modern hardware, with features like pipelines, caches, and out-of-order execution. Models, e.g., simulators, of the hardware are used in the low-level analysis analysis. This eliminates the need of having the actual hardware available, but a (safe) timing model of the hardware must be developed to be used during the analysis, something which can be very complicated.

Due to the complexity of today's software and hardware, both flow- and low-level analysis may result in over-approximations, e.g., reporting too many paths as feasible or too large timings to instructions. Thus, the calculation


(a) Oscilloscope

(b) Logic analyzer

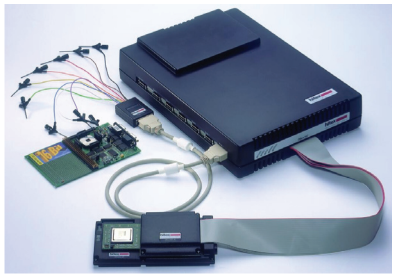

(c) In-circuit emulator

Figure 1. Tools for dynamic timing analysis

will give a safe but potentially pessimistic WCET value.

Today, static WCET tools are commercially available, including aiT [1] and Bound-T [32]. There also exists several research prototypes, including Chronos [8], the Florida State University tool [18], Heptane [19], and SWEET [17].

Hybrid WCET analysis. Hybrid analysis techniques combine measurements and static analyses techniques. The tools use measurements to extract timing for smaller program parts, and static analysis to deduce the final program WCET estimate from the program part timings. Example tools include Rapitime [25] and MTime [33]. No hybrid WCET analysis tool was used in the case studies.

3 Used Timing Analysis Tools

The following is a short summary of the timing analysis tools used in the presented case studies.

SWEET. SWEET is a static WCET analysis research tool developed at Mälardalen University [23]. SWEET consists of three main parts; a flow analysis, a low-level analysis, and a final WCET calculation. The flow analysis analyze intermediate code produced by a research compiler. Current focus is to develop automatic flow analysis methods [17].

aiT. The aiT tool is a commercial static WCET analysis tool from AbsInt GmbH [1]. In contrast to SWEET, aiT does not rely on a specific compiler, but analyses executable binaries, and has support for a larger number of targets hardwares. It performs many types of analyses, most of them based on abstract interpretation [30]. The tool includes an automatic loop bound analysis, which catches simple cases.

Oscilloscope. Using an oscilloscope often involves adding a bit-flip on an accessible pin of the processor at some program points, and then observing the resulting waveform to find the periodicity and thus the execution time.

Logic analyzer. A logic analyzer looks at the data- or address bus of the system to see when certain instructions are being fetched. However, it requires that relevant memory transactions reach the bus, which not always the case on systems with a cache.

In-circuit emulator. An in-circuit emulator behaves like a particular processor, but with better debug and inspection capabilities. Provided that they do match the target processor, they can provide very detailed data. Today, emulators are being replaced with hardware trace facilities, since they are too hard to construct for current processors.

4 *First Study:* Static Analysis for Disable-Interrupt Regions in a RTOS (2001)

Enea OSE is a real-time operating system (OS) used in many type of embedded applications, e.g., in mobile phones and aircrafts. The purpose of the first case study was to see if static WCET analysis could be used to bound the execution time of Disable Interrupts (DI) regions in the delta kernel (ARM9 version) of the Enea OSE [10]. DI regions should preferably have short execution times, since the execution of these regions can potentially delay other activities in the system. The study is described in more detail in [5, 6].

We had access to the Enea OSE source and object code. The object code was generated by ARM tools, containing symbol tables which were helpful to relate object code constructs to its source code counterparts.

The case study was performed with the low-level and calculation parts of an early version of SWEET. During the case study, a timing model for ARM9 was implemented. We also made prototype tools to extract the DI regions and to build the CFG for the extracted regions. Flow constraints for the CFGs had to be manually provided. Of particular importance was to provide loop bounds. We thought that these should be simply deducible from the source code, but in practice this was hard (see below). Finally, SWEET was used to calculate a WCET estimate for each DI region.

We identified 612 DI regions in the kernel. Most of these were very simple. We selected ten DI regions for a more detailed investigation. These regions had a more complex control structure than the others, and several contained loops.

Experiences and conclusions.

- An OS is often run in *modes* which may affect loop bounds, making the WCET typically mode-dependent.

- The usefulness of WCET analysis increases with the level of automation. To construct tools for identifying DI regions and to extract CFGs required a lot of work. However, when done, the analysis task got simplified.

5 *Second Study:* Static Analysis for DI Regions and System Calls in a RTOS (2003)

The Enea OSE was also analyzed in our second study. Some of the tools developed in the first study were re-used, such as the DI region extractor tool. The CPU chosen was the ARM7TDMI and the static WCET analysis tool used was aiT [1]. The study is described in detail in [27, 28].

The aiT ARM7 tool analyses executables generated by the ARM C compiler. The information derivable from executable is often not sufficient to yield information about program flow, such as loop bounds and knowledge of infeasible paths, so these have to be provided by the user. Therefore, aiT supports a set of *user annotations* [15]. Some of the more important annotations used in this study were *loop bounds*, *dead code*, and (static) *outcome of conditions*.

We used the ARMulator simulator to do some rudimentary measurements for timing. The ARMulator is not guaranteed to be cycle-accurate. However, since ARM7TDMI is not a very complex processor without a cache, we expect the ARMulator to be rather timing accurate.

We made a series of experiments. First we analyzed 180 of the previously extracted DI regions, as well as four system calls, using the aiT tool. Secondly, we investigated the influence of code optimization on WCET analysis. We then compiled some standard benchmark programs with different levels of optimization, and performed WCET analyses on the resulting binaries, again using the aiT tool.

To get some justification of the quality of calculated WCET estimates, we compared timing estimates from aiT and the ARMulator for a number of benchmarks. Since both methods rely on software models of the hardware we cannot say if one timing estimate is more correct than the other. The benchmarks contained features like system calls, loops and branches, but had only a single execution path through the program. By keeping track of the number of times each basic block was taken during a simulator run, we were able, by annotations, to provide exact bounds on the executions of each basic block for the aiT WCET calculation. Thus, the resulting timing discrepancies were not due to incorrect flow information, but only to differences in the hardware timing models. The experiments showed that the aiT WCET estimates were on average about 5% larger than the times obtained using the ARMulator. For none of the benchmarks aiT gave a WCET lower than the timing from the ARMulator.

Experiences and conclusions. We discovered that the execution time of the system calls depended on many parameters. A global WCET bound, valid for all possible parameter values, could become very poor for actual configurations and standard running modes. We dealt with this problem in our experiments by assuming some "typical" scenarios for parameters affecting the WCET (after correspondence with the OSE designers). We also excluded uninteresting execution paths from the analysis by manual annotations.

We made the following general observations:

- Many annotations were required for each system call; for routines of sizes between 78 and 143 instructions, the number of annotations were between 10 and 33.
- The exclusion of error handling code in the OSE system calls yielded significantly smaller code to analyze.
- Code optimizations do not seem to affect the precision of the static WCET analysis.
- Many loops in the OSE kernel were dependent on dynamic data structures. As a consequence, the aiT loop bound analysis did not perform well for these loops.
- Providing upper bounds manually for these loops required a deep understanding of the code and discussions with the OSE designers. The analysis became therefore quite labor-consuming, even if the codes were small.
- A simulator like the ARMulator gives timing estimates that seem to be fairly accurate (i.e., within a few %).

We conclude that the usefulness of WCET analysis would improve with a higher level of automation and support from the tool. Especially, it should be important to develop better loop bound analyses. Furthermore, absolute WCET bounds are not always appropriate for real-time operating system code. The reason is, as mentioned above, that the WCET often depends on dynamic system parameters.

6 *Third Study:* Static Analysis for Automotive Communication Code (2004)

This case study targeted automotive code, namely the Volcano Tool Suite for design and implementation of in-vehicle communication over CAN and/or LIN networks. The company Volcano Communications Technologies AB (VCT) [34] provides tools for embedded network systems, principally used within the car industry. The Volcano LIN Target package (LTP) was selected as a suitable part of the Volcano LIN tool suite to analyse. The microcontroller used was a MC9S12DP256 from Motorola, which includes a 16-bit Star12 CPU of the MC68HC12 family. The aiT HC12 tool was used for static WCET analysis. The work is described in closer detail in [3, 4].

We selected nine different LIN API functions, and were able to obtain WCET values for all of them with the aiT HC12 tool. The WCET values were often not a constant single value, but instead dependent on some system parameters. Moreover, for many code parts it was hard to directly see how system parameter values affected the execution of the code. We therefore analyzed each function in a number of specific *cases*. Each case gave a WCET for the function under some specific conditions. All functions needed manual annotations to be analysed. The number of annotations ranged between 6 – 14 for codes of sizes between 2 – 14 kb.

Experiences and conclusions. As for the OSE code, the WCET for the studied LIN functions were often dependent on some specific system configuration parameters and modes. A mode- and input-sensitive WCET analysis would give better resource utilization.

For many parts of the LIN API it was possible to manually create parametrical WCET formulas. This allowed us to test how certain parameter values affected the WCET and gave a good understanding of the execution behavior of the analyzed code. It seems interesting to develop methods to automatically derive these parametrical formulas.

Much work was required to set annotations manually and this required a detailed understanding of the code. Ways to automate this would be beneficial, e.g., better support for loop bound analysis would avoid much manual work.

After discussions with the VCT employees it turned out that not only the WCET, but also the jitter of a piece of code, is of large interest. Jitter is the difference between the best-case execution time (BCET) and the WCET. Thus, better support for BCET analysis would be useful.

7 *Fourth Study:* Static Analysis and Measurements for Welding Systems Code (2005)

In this case study, several timing analysis methods, i.e., oscilloscope, logical analyzer and static analysis, were used to analyse the timing for time-critical code in products from CC-Systems AB (CCS) [7]. The case study was performed as two cooperating Master's theses and are described in two reports: [36] (measurements) and [11] (static analysis).

CCS develops and delivers electronic solutions and software for tough environments, including forestry machines, construction equipments, trucks, marine vessels, industrial automation, railway vehicles and military vehicles.

The program code in the analyzed system was written in C++ and run on an Infineon SAK-C167CS-LM microcontroller. The analysis was performed for a number of interrupt routines. By using several types of analysis methods upon the same code we could compare the work required to obtain a result, as well as the quality of the timing estimates obtained by each method.

There was no OS in the system, so we could not get any help from time-functions that an OS provides. To get as non-intrusive measurements as possible, an oscilloscope and a logic analyzer was selected as measurement tools.

Measurements using an oscilloscope. There was a number of issues concerning setup that had to be solved before the measurements could start; the report [36] describes this in detail. Once measurements were possible, a number of methodological issues were raised. One problem was to find out the execution path that corresponded to a measured value. This was very difficult to do, since the only help was three LEDs on the microcontroller. There were many if-else- and switch-statements in the interrupt functions, and to figure out the path executed through those statements was almost impossible. Due to the limited visibility, it was hard to guarantee that the WCET had been measured.

To get closer to the WCET, the following method was used: some branch conditions were modified to always give an outcome according to the worst case path, as calculated by the static analysis (see below). After the program code was changed and directed into a longer path, a longer execution time was obtained. However, these code modifications also caused some unexpected system behaviours.

We made the following general observations:

- It was hard to set up the complete system.
- Only a few measurement points were possible.
- It was hard to identify the execution path taken.
- It was difficult to detect if other interrupts had disturbed the measurement of the current interrupt.
- It was hard to trigger all possible executions (only a few scenarios could be tested).
- It is dangerous to modify code to force WCET execution since it might affect the system in unpredictable manner. Furthermore, we do not know if the modified scenario could occur in reality, giving that the we might make an overestimation of the WCET.
- Some errors could not occur in the testing environment. Could they occur in reality?
- A small probe effect was introduced due to inserted measurement points.

Measurements using a logic analyser. These measurements were done in a non-intrusive way by listening to the address-bus. A whole trace of accessed addresses could be observed and stored with their corresponding access times. The execution time for the interrupt was calculated as the time difference between the times corresponding to the start and return addresses of the interrupt-routine.

To find out what happened during the execution, these addresses had to be matched with program code to find out the actual execution path through the program. This meant a lot of detailed level work, see [36] for more info.

One interrupt was studied both at the system start-up phase, and during normal operation. The result showed a significant difference. The traces of the execution paths were compared, and the reason to the difference was found.

Other interrupts were studied in detail during normal operation. Typically, each interrupt showed a set of different possible execution times. The path for the longest was analysed to find out the reason for the longer time.

Some of the interrupts under study had large execution time variations. By detailed study of the extracted execution traces, we discovered that other interrupts sometimes occurred in the middle of the interrupt under investigation. These interrupts took some time to handle, and should be removed from the timing of the investigated interrupt.

We made the following general observations:

- The logic analyzer gave non-intrusive measurements. There was not a single line of program code changed or inserted for probing.
- The logic analyzer provided better means to identify the execution paths taken in comparision to the oscilloscope.
- Much time was needed to investigate address traces to find the execution path that was taken. With the aiSee tool from aiT, which presents the CFGs graphically, the address traces became much more useful.
- We experienced large variation in measured times, both due to different paths taken and to interfering interrupts.
- Sometimes the interrupt handling was disabled and an occurred interrupt was handled at a later time. This made it hard to measure the time of just the interrupt, since it did not always start exactly when it occurred.
- The interrupts often had a longer execution time during system start-up than during normal operation.

Static Analysis. The aiT C167 WCET analysis tool [1] was used for the static WCET analysis. The code for which the oscilloscope and logic analyzer measurements was made was also analysed using aiT. To do WCET calculations using aiT, a set of mandatory annotations had to be given, see [11], e.g., to specify the clock frequency, the compiler used, and the hardware system configuration.

aiT was able to find loop bounds for many of the loops in the analyzed routines. For the remaining loops, loop bounds had to be given manually. When all loop bounds were given, and aiT had managed to calculate a WCET value, it was important to check that the execution path was correct and didn't include infeasible paths. If so, more annotations to restrict the flow of the program had to be added. When the execution path that yielded the WCET was found, there were two choices, either to accept the WCET value obtained or to add extra annotations, e.g., on valid addresses for memory-accesses, to get a tighter WCET value.

Eriksson [11] describes in detail, for each analysed code, which problems that came up and how they were solved. During the handling of an interrupt up to six CAN messages can be received. One problem was to make aiT calculate the time to receive just one message. Other parts of the code required other annotations, e.g., to turn error handling off.

The same codes were analyzed using different compiler optimisations. As a result, some loops bounds needed to be changed, and all flow constraints that were given using absolute addresses had to be modified (aiT supports relative addressing of flow constraints to avoid this).

Result comparison. To be able to compare the results of different methods, the same codes were analysed using both dynamic and static WCET analysis methods. This was done to see how much overestimation aiT introduced, and also to see if the dynamic methods could find the WCET path. The term overestimation is only valid if the path that leads to the WCET is measured, and this wasn't always the case here. But the time difference can give a hint about the overestimations of aiT if the same execution path is analysed with both static and dynamic methods.

We first compared WCET estimates derived using oscilloscope and aiT. As mentioned above, it was hard to identify the execution path(s) taken when using the oscilloscope. However, since the code only contained a limited set of paths, the measured values should be comparable with the ones obtained by the static method. The difference was 4 – 8%, which can be considered as acceptable.

The time differences of another more complex code, for handling a special CAN status message, was 19 – 117%, which was not considered as acceptable. The reason for these large time differences was hard to deduce, since the measured paths were not completely known.

Secondly, we compared estimates derived using the logic analyser with estimates from aiT. The execution paths of the measurements were extracted from the traces, then aiT was forced to execute the same path so execution times could be compared. The execution path in aiT was changed with flow annotations. The difference between the values was 3 – 8%, which was considered as acceptable. The difference was reduced to 1 – 5% when additional memory annotations were added to the aiT analysis.

We also tried to force measurement executions to take the path aiT chose as the WCET path. This required a lot of modification of the code. Moreover, this affected system behaviour and it was not certain that the path measured could be taken in reality.

In fact, what we can say for certain (under the assumption that we have removed all interfering interrupts in the measurements and that all annotations given to aiT are correct) is that: $\mathrm{WCET_m} \leq \mathrm{WCET_a} \leq \mathrm{WCET_c}$ where $\mathrm{WCET_m}$ is the worst measured timing, $\mathrm{WCET_a}$ is the WCET of the program and $\mathrm{WCET_c}$ is the estimate calculated by aiT.

General conclusions of the case study.

- An oscilloscope has limited amount of measuring points and limited granularity. It is difficult to find the exact execution path using the oscilloscope. The measured results have rather coarse time resolution and cannot be guaranteed to be the WCET.
- A logic analyzer is a better option for a detailed timing analysis. It can observe several measuring points simultaneously, and give results on a level of single instructions. Furthermore, the measurements can be done in a totally non-intrusive way. A shortcoming is that the

measured result cannot be guaranteed to be the WCET.

- The aiT tool can do WCET analysis without the target system and the result is guaranteed to be a safe WCET estimate. But it requires the user to have a good understanding of both the aiT tool and the program code; otherwise the quality of the estimated results can be seriously affected and result in large overestimations.

8 *Fifth Study:* Static Analysis and Measurements for Articulated Haulers Code (2005)

In this case study we used both measurements and static analysis to analyze code in articulated haulers developed by Volvo CE [35]. The study is described in detail in [29].

Volvo CE is one of the world's leading manufacturers of construction equipment. Their product range encompasses backhoe loaders, wheel loaders, excavators, articulated haulers and motor graders. The vehicles are controlled by a distributed, computerized control system, consisting of a set of networked ECU's (Electronic Control Units). The ECU's currently used in the articulated haulers are based on the Infineon C167CS processor.

The software for the vehicle systems at Volvo CE uses the Rubus real-time operating system from Arcticus Systems [2]. Rubus forces the designer to structure the system in a task-oriented way. Rubus also contains a mechanism to measure the execution time of tasks during run-time, which can be used to do high-water-marking of the execution time, which bounds the WCET from below. However, since the measurements are done in software, there is a probe effect, which is hard to compensate for in a precise way.

We used the Trace32Fire in-circuit emulator (from Lauterbach Datentechnik GmbH [22]) for the timing measurements. The aiT C167 WCET analysis tool [1] was used for the static WCET analysis.

Static analysis. We first tested if it was possible to statically derive WCET estimates with a minimal amount of effort, i.e., only providing hardware configuration and starting points for tasks to the aiT tool, for an initial selection of 24 tasks. For 17 of the tasks aiT was able to derive a WCET estimate. For the remaining seven tasks no WCET estimate could be derived automatically, mainly due to unbounded loops. When comparing these calculated WCET estimates with the WCET parameters set using Rubus timing mechanism, we got an average time reduction of 59%[1]. We consequently concluded that it was possible, with a very limited effort, to use aiT to calculate WCET estimates for many tasks in the system, and that aiT gives tighter estimates than the ones based on measurements.

For the remaining experiments we selected 13 Rubus tasks for a closer study (including some, but not all, of the 24 tasks used in the initial experiment). For these 13 tasks we first investigated how few annotations, in order to obtain a WCET estimate at all, that must be manually given. Thus, we made sure that all loops in the code got upper bounded (either by aiT flow analysis or by manual annotations). Only three of the 13 selected tasks contained loops. In total, 12 out of 19 loops had to be manually bounded.

Next, we investigated how much the calculated WCET estimates could be tightened by extra manual annotations. The studied code contained some if-statements where the conditions are mutually exclusive. This kind of code gives rise to infeasible paths. After analysing the code, we were able to add a number of (183) manual annotations that removed many infeasible paths. This reduced the calculated WCET estimates for the 13 tasks between 5 – 31%.

Measurements using an in-circuit emulator. The Rubus timing mechanism was not used for the measurements. This was partly due to the inexactness of this mechanism, but mostly due to limited monitoring possibilities on the real hardware, meaning that it was hard to identify the actual path(s) taken during measurements.

We used the Trace32Fire in-circuit emulator for the measurements instead, which was believed to be quite cycle accurate. In order to obtain some evidence for this, we compared timing values from the emulator with measured values on the hardware obtained by the Rubus timing mechanism. They showed consistently that the overhead from the Rubus timing mechanism was about 3.5 microseconds.

The main advantage of using the in-circuit emulator is that it gives full control over the execution and full monitoring capability. Thus, we could force the emulator to execute the same path that aiT derived as the WCET path. Since the emulator might be forced to execute a non-feasible path, the result might overapproximate the WCET.

For all tasks, both the first static analysis (with minimal number of flow annotations) and the second static analysis (with extra flow constraints) yielded WCET estimates larger than emulator values. The second static analysis overestimated the emulator with 4 – 33%.

For all tasks except one, the original WCET values, set by the Rubus timing mechanism, substantially larger (at least 38%) than the best aiT estimate. Consequently, using aiT values should improve schedulability, and leave room for more activities in the system.

For one task, the WCET value derived by Rubus timing mechanism was lower than both the aiT and emulator estimates. However, it cannot be guaranteed that the Rubus value is below the WCET, since both the aiT and the emulator estimates might have been obtained for an infeasible path. A closer examination of the code would be needed to determine whether this is the case or not.

[1]These values were set using the measurement facilities of Rubus plus adding a safety margin which explains why they are larger.

Conclusions drawn from the case study. Static WCET analysis tools seem to be useful for the type of task-oriented real-time code studied. Such tools are suitable for Rubus, and other similar real-time operating systems, which require that the WCET's of tasks are explicitly given.

The code studied had a fairly simple control structure, with few loops, no recursion, and few if any dynamic features. For such code, it is easy to provide the minimal information needed to get a WCET analysis tool to produce a WCET estimate. Furthermore, the simply calculated WCET estimates are in almost all cases tighter than WCET estimates based on measurements with a safety margin added. However, if tighter WCET estimates are desired, then substantially more work is required. The code under study happened to be written in a style with many if-statements with more or less exclusive conditions. We don't know how common such code is, but it gives rise to many infeasible paths. A tight WCET estimate requires that most of these paths are pruned away in the analysis, even if the mutual exclusion occurs between instructions in different functions. To do this conveniently by hand requires good support from the annotation language, to allow such constraints to be easily expressed.

A tool such as aiT can improve the system utilization as well as detect potential sources of timing errors. For the system under study, this is clearly the case, since the WCET estimates obtained by aiT are substantially more precise than the current WCET parameters set in Rubus. The use of a WCET tool also gives a possibility to remove the use of high-water measurements. This may reduce system load, especially in systems with many small tasks.

9 Related Work

There have been a number of studies of WCET analysis of industrial code. There are some reports on using commercial, static analysis WCET tools to analyze code for space applications [20, 21, 26], and in avionics industry [14, 24, 31]. Colin et al. [9] analyzed operating system functions of RTEMS, a small, open-source real-time kernel.

The conclusions from these studies confirm our observations. Much work is connected with finding loop bounds and infeasible paths. Montag et al. [24] reports another issue; imprecision due to conservative assumptions in the WCET tool during floating-point calculations.

10 Conclusions, Discussion and Future Work

The case studies, presented in Sections 4 to 8, provided a number of detailed results, experiences, and conclusions. Some common conclusions can be drawn from these.

What timing value is really wanted? It is important to understand that the method used should be selected according to the type of system and the needs of the system designer. Some of the issues are described below.

- Is a safe upper bound required? If so, then static analysis probably is the right way to go.
- Is an approximate bound (yielding either over- or under-estimation) ok? That may be the case, if the system is a soft real-time system, or if the system can tolerate occasional overruns. In that case measurement-based timing values can be sufficient.
- What part of the code should be analysed? Is it the whole task, or part of it? Do we differ between, e.g., the start-up phase of the system and the system during normal operation? Should error handling be considered? Prepare for more work the more detailed the restrictions are.
- What type of timing values do you want, a single value or, e.g., parametrical values? Be aware of that today's timing analysis methods basically support the calculation of one value only.
- The structure of the code of a system has significant impact on the analyzability of a system. Systems with many small and well-defined tasks, scheduled by a strict priority RTOS or a time-triggered schedule, is easier to analyze than monolithic interrupt-driven programs based on an infinite main loop.

Static WCET analysis. It is possible to apply static WCET analysis to code with properties similar to the analysed ones. The tools used performs well and produces safe WCET estimates, once the necessary preparatory work, such as providing annotations, has been done. However, there are some issues that have to be considered:

- The WCET analysis process is not automated on a 'one-click-analysis' basis. Much manual intervention, and detailed knowledge of the analyzed code, is required to perform the analysis. For example, loop bounds have to be provided for many, especially complex, loops.
- The WCET estimates might be untight. By adding more annotations the estimates can become tighter, but to the cost of a lot of work, especially for complex code.
- A graphical interface is beneficial, giving an overview of the analysed code and illustrating its WCET execution.
- A high degree of support from the tool, for example with automatic loop bounds calculation, is desirable.
- Absolute WCET bounds are not always sufficient. Support for some type of parametrical or mode-sensitive WCET calculation is sometimes needed.

Measurements. Measurements are a possibility, assuming the hardware and the software of the system is available. The main drawback of measurements is that it is very hard to find the real WCET, especially for complex code and/or systems with complex hardware features. To force the program to take the worst-case path requires a lot of knowledge of the code and the hardware. It is often hard to force the program to take a certain path by just using different inputs. Furthermore, using measurements to derive WCET estimates is only suitable for program with a limited set of possible execution paths.

A summary of the measurement methods applied in the presented studies are:

- Oscilloscope: Intrusive method, since measuring code has to be inserted. It is hard to see what happens, for example to follow the path, or if interrupts has disturbed the measurement. The hardware limits the number of measuring points, and also offers poor resolution. Only suitable for codes with a very limited set of execution paths and not too time critical code.
- Logical analyzer and in-circuit emulator: The measurements are non-intrusive. They give a better possibility to see what happened during measurements as compared to the oscilloscope. However, a lot of work is required to see where the trace goes. It offers good resolution of timing values. The timing can be disturbed by interrupts.

Discussion. It seems like a good way to do the WCET analysis is to combine measurements and static analysis so that the two types of methods are able to support and compensate each other. From the static method, we get a complete view of the code of interest, together with the possible longest execution path. Using dynamic methods we can find out if this path is feasible or not, and the required conditions/inputs for the program for taking this longest path. If we can, somehow, make sure that these required conditions are fulfilled, and afterwards measure the execution time of this path by a dynamic measuring tool, e.g., a logic analyzer, then we will be able to say, with high confidence, that we have found the actual WCET of the program. On the other hand, the dynamic measurement can provide useful information, such as addresses of memory accesses, to get tighter static analysis results.

The address traces produced by the logic analyzer are very helpful because they give the processor activity list sorted by time. From this list one can easily see how and when the program code is executed, and whether the functions are executed exactly in the way that they are designed or planned to be. If there has been an error, the trace can also help us to find out why it happened and even how the error has affected other parts of the system. We can also supervise the memory accesses made using the logic analyzer. For example, we can see whether there has been an illegal access in a protected memory area.

If the timing analysis tool has a well-developed graphical interface, it can provide a clear and easy handled view of the whole system. For instance, from the graph we can see which function that is calling which function; we can also see all the possible execution paths in the program and the influence from the inputs on the selection of the paths. All this information is very valuable for the software developers and timing analysts.

Both static and measurement-based methods benefit from well-documented code and a nicely structured system architecture regarding timing aspects. Therefore, code should be written to facilitate timing analysis, by adhering to strict coding rules. Typically, you want all loops to be statically bounded, input-data dependence minimized, pointer use minimized, and code structure as simple and straightforward as possible.

Future work. We are currently initiating case studies where we will evaluate the flow analysis of SWEET [23] upon industrial code. This allows us to test if our own developed loop bound and infeasible path analyses [17] are applicable in these settings. We also plan to use the hybrid WCET analysis tool Rapitime [25] in industrial case studies. It should be very interesting to compare this type of tools to the ones already used. We will also try to analyze industrial code generated from modeling tools. Compared to traditional programming, code generated from modeling tools poses new challenges and possibilities as regards timing analysis and predictability.

References

[1] AbsInt. aiT tool homepage, 2006. `www.absint.com/ait`.

[2] Arcticus Systems homepage, 2006. `www.arcticus-systems.com`.

[3] Susanna Byhlin. Evaluation of Static Time Analysis for Volcano Communications Technologies AB. Master's thesis, Mälardalen University, Västerås, Sweden, Sept 2004. `www.mrtc.mdh.se/publications/0797.pdf`.

[4] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, July 2005.

[5] M. Carlsson. WCET Analysis, Case Study on Interrupt Latency, for the OSE Real-Time Operating System. Master's thesis, Kungliga Tekniska Högskolan, Sweden, December 2001. `www.e.kth.se/~e96_mca/WCETAnalysisForOSE.pdf`.

[6] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *Proc. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002)*, 2002.

[7] CC-Systems AB homepage, 2006. `www.cc-systems.com`.

[8] The Chronos WCET analysis tool homepage, 2006. `www.comp.nus.edu.sg/~rpembed/chronos`.

[9] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th*

Euromicro Conference of Real-Time Systems, (ECRTS'01), June 2001.

[10] Enea Embedded Technology homepage, 2006. `www.enea.com`.

[11] O. Eriksson. Evaluation of static time analysis for CC systems. Master's thesis, Mälardalen University, Västerås, Sweden, August 2005. 63 pages, `www.mrtc.mdh.se/publications/0978.pdf`.

[12] Andreas Ermedahl and Jakob Engblom. *Handbook of Real-Time and Embedded Systems*, chapter Execution Time Analysis for Embedded Real-Time Systems. CRC Press. Accepted for publication.

[13] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 19–22, July 2005.

[14] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. 1st International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211*, Oct 2001.

[15] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[16] Jack Ganssle. Really real-time systems. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, April 2006.

[17] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, December 2006.

[18] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[19] Homepage for the Heptane WCET analysis tool, 2006. `www.irisa.fr/aces/work/heptane-demo`.

[20] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[21] Niklas Holsti, T. Långbacka, and S. Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, September 2000.

[22] Lauterbach. Lauterbach datentechnik GmbH homepage, 2006. `www.lauterbach.com`.

[23] Mälardalen University. WCET project homepage, 2006. `www.mrtc.mdh.se/projects/wcet`.

[24] Pascal Montag, Steffen Goerzig, and Paul Levi. Challenges of timing verification tools in the automotive domain. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.

[25] RapiTime WCET tool homepage, 2006. `www.rapitasystems.com`.

[26] M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in calculating the WCET of a complex on-board satellite application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[27] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, October 2004.

[28] Daniel Sandell. Evaluating Static Worst Case Execution Time Analysis for a Commercial Real-Time Operating System. Master's thesis, Mälardalen University, Västerås, Sweden, June 2004. `www.mrtc.mdh.se/publications/0738.pdf`.

[29] Daniel Sehlberg, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Steffen Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.

[30] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[31] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks (DSN-2003)*, June 2003.

[32] Tidorum. Bound-T tool homepage, 2006. `www.tidorum.fi/bound-t`.

[33] Vienna real-time systems group homepage, 2006. `www.vmars.tuwien.ac.at`.

[34] Volcano Technologies Communications AB homepage, 2005. `www.volcanoautomotive.com`.

[35] Volvo CE (construction equipment) homepage, 2006. `www.volvo.com/constructionequipment`.

[36] Y. Zhang. Evaluation of Methods for Dynamic Time Analysis for CC-Systems AB. Master's thesis, Mälardalen University, August 2005. 72 pages, `www.mrtc.mdh.se/publications/0977.pdf`.