Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems

Daniel Sehlberg*, Andreas Ermedahl*, Jan Gustafsson*,
Björn Lisper*, and Steffen Wiegratz♮

*Dept. of Computer Science and Electronics, Mälardalen University, Västerås, Sweden
♮AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany

Abstract. Methods for Worst-Case Execution Time (WCET) analysis have been known for some time, and recently commercial tools have emerged. This technique is gradually being entered into industry to analyse real production codes. This article presents a case study where the aiT WCET analysis tool was used to find upper time bounds for task-oriented vehicular control code. The main purpose was to investigate the practical difficulties that arise when applying the current WCET analysis methods to this particular kind of code. In particular, we were interested in how labor-intense the analysis becomes, measured by the number of manual annotations necessary for calculating a WCET estimate. We were also interested how much tighter WCET estimates will become by manually adding extra annotations, and how much additional work that is needed to give these annotations. We also made some systematic comparisons between calculated and measured WCET estimates for the analysed system.

1 Introduction

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the worst possible execution time of a computer program. Reliable WCET estimates are a key component when designing and verifying real-time systems, especially when real-time systems are used to control safety-critical systems like vehicles, military equipment and industrial power plants. WCET estimates are needed in hard real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times [1].

Any WCET analysis must deal with the fact that a computer program typically has no fixed execution time. *Variations* in the execution time occur due to the characteristics of the software, as well as of the computer upon which the program is run. Thus, both the properties of the software and the hardware must be considered in order to understand and predict the WCET of a program.

The traditional way to determine the timing of a program is by measurements, also known as *dynamic timing analysis*. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [2, 3]. This is labor-intensive and error-prone work. Even worse, it cannot guarantee that the true WCET has been found, since in general it is not possible to perform exhaustive testing.

Static timing analyses estimate the WCET of a program without actually running it. The analyses avoid the need to run the program by simultaneously considering the effects of all possible inputs, including possible system states, together with the program's interaction with the hardware. The analyses rely on mathematical models of the software and hardware involved. Given that the models are accurate enough, the result is a *safe* timing estimate that is greater than or equal to the actual WCET.

Since a few years commercial WCET analysis tools have reached the market, including both static tools such as aiT [4] and Bound-T [5], and more measurement-based tools such as RapiTime [6]. However, practical experience of WCET analysis in industry has so far been rather limited, see Section 2.

In this case study we report from experiences when using the aiT tool to analyze code used in vehicles manufactured by Volvo Construction Equipment (Volvo CE) [7]. The code is for tasks managed by the Rubus Real-Time operating system [8], which is a commercial implementation of the BASEMENT real-time system architecture [9]. The article is based on the Master's thesis by Daniel Sehlberg [10].

The contributions of this article are:

– we estimate how much manual work is needed to perform a basic WCET analysis using a state-of-the-art WCET analysis tool,
– we evaluate how to tighten the WCET estimate using additional annotations, and the amount of effort needed,
– and we make comparisons between calculated and measured WCET estimates.

The rest of this article is organized as follows. In Section 2, we give a brief introduction to WCET analysis and related work in the area. In Section 3, we describe the experimental setup: the system where the code is run, and the analysis tool. Section 4 describes the analyses made, and gives the results. Finally, in Section 5, we draw some conclusions and give directions for further research.

2 WCET Analysis Overview and Related Work

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

The purpose of the flow analysis phase is to extract the dynamic behaviour of the program. This includes information on which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. Since the flow analysis does not know the execution path which corresponds to the longest execution time, the information must be a safe (over)approximation including *all* possible program executions. The information can be obtained by *manual annotations* (integrated in the programming language [11] or provided separately [12, 13]), or by *automatic flow analysis* methods [14–16].

Fig. 1. Motor graders, wheel loaders, excavators and articulated haulers are some of the products developed by Volvo CE

The purpose of processor behavior analysis is to determine the timing behaviour of instructions given the architectural features of the target system. For modern processors it is especially important to study the effects of various performance enhancing features, like caches, branch predictors and pipelines [17–20].

The purpose of the calculation phase is to calculate the WCET estimate for a program, combining the flow and timing information derived in the previous phases. A calculation method frequently used is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program flow and low-level execution times [12, 21, 16]. IPET calculations normally rely on integer linear programming to solve the generated constraint system.

Studies of WCET analysis of industrial code are not common. There are some reports on application of commercial WCET tools to analyze code for space applications [16, 22, 23], and in avionics industry [24, 25]. The experiences from some recent case studies are compiled in [26]. One of these case studies concerns WCET analysis for communication software in cars [27].

An investigation of industrial embedded code has been done by Engblom [28]. He collected statistics of the number of occurrences of certain code features that may be problematic for a WCET analysis, like recursion, unstructured flow graphs, function pointers and function pointer calls, data pointers, deeply nested loops, multiple loop exits, deeply nested decision nests, and non-terminating loops and functions. In a more recent study [29], industrial code is investigated with respect to how amenable it is to a syntactical flow analysis, a method which detects certain loop patterns where iteration bounds can be calculated immediately from the pattern.

3 Experimental Setup

We now describe the experimental setup: the kind of system being controlled by the analyzed real-time software, the hardware, the operating system, the WCET analysis tool, and the emulator that was used for estimating timing measurements.

3.1 The Volvo CE System

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 number of ECU's in the different vehicles varies, but an articulated hauler contains five: the Cabin, Engine, Instrument, Transmission, and Vehicle ECU. The ECU's currently use the Infineon C167CS processor.

The ECU's are connected with two CAN (Controller Area Network) buses and one J1587 bus. The CAN buses are the primary buses used for most of the data exchange, while the slower J1587 bus is mainly used for diagnostic service tools. The study described here considers 13 hard real-time tasks in the Transmission ECU for articulated haulers.

3.2 The Rubus Real-Time Operating System

The software for the vehicle systems at Volvo CE uses the Rubus real-time operating system from Arcticus Systems.

The Rubus OS is task-oriented, and supports *green*, *red*, and *blue* tasks. Green tasks are interrupt tasks. They have the highest priority in the system and preempt any other tasks when released. Red tasks are hard real time tasks triggered by the system clock and scheduled offline. Possible attributes for the red tasks are release time, period time, deadline, precedence and WCET. Blue tasks are event-triggered soft tasks which are scheduled online and only execute when there are no red or green tasks running.

Rubus contains a mechanism to measure the execution time of tasks: some timing code is then inlined with the task. This mechanism can be used to do high-water-marking of the execution time, which then 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.

Since red tasks needs WCET estimates as attributes, the codes for such tasks are prime targets to analyze with a static WCET analysis tool. The current common practice is to estimate the WCET by measurements, typically through the Rubus task timing facility, and then add a safety margin, since the measured WCET value may be underestimated. The inherent problem with this approach is that the correct size of such a margin is unknown.

If a static WCET analysis tool is used, which is based on a correct timing model, and if correct annotations are given, then the WCET estimate obtained will be safe, which means the static schedule generated by Rubus also will be safe. Since no safety margins then have to be added, the WCET estimate will also often be tighter. Thus, static WCET analysis fits very well in a task-oriented development model like the one supported by the Rubus OS and its supporting development tools.

3.3 The Infineon C167CS Processor

The software studied in this article is executed on Infineon C167CS [30], which is a 16-bit, single-chip microcontroller with a four stage pipeline, but no cache. The C167CS incorporates 32 KBytes of on-chip mask-programmable ROM. 3 KBytes of on-chip internal RAM and 8 KBytes of on-chip extension RAM are provided to store user data or code. It is also possible to use off-chip memory of up to 16 Megabytes. The different memory types have have different access times; this may affect the WCET calculations. The memory space of the C167CS is configured as a von Neumann architecture, which means that code memory, data memory, registers and I/O ports are organized within the same linear address space. The processor has a frequency of 33 MHz.

3.4 The aiT WCET Analysis Tool

The aiT tool is a commercial WCET analysis tool from AbsInt GmbH [4]. The aiT tool analyses executable binaries, and it supports a number of target architectures including the C167CS. aiT performs the following steps in its analysis (see [31], Chapter 6):

- a *reconstruction of the control flow graph* from the executable code,
- an analysis to *bound loop iterations*, based on a combination of an interval-based abstract interpretation and pattern-matching tuned to the compiler that generated the analyzed code,
- a *value analysis* to determine the range of values in registers,
- a *cache analysis* that classifies accesses to main memory w.r.t. hits and misses, if the processor has a cache,
- a *pipeline analysis*, where a model of the pipeline behavior is used to determine the execution time of basic blocks, and finally
- a *calculation* where an IPET calculation is made to determine the WCET.

In essence, the aiT WCET analysis conforms to the general scheme presented in Section 2. Several of the analyses in the chain are based on abstract interpretation [32], such as the value analysis and the cache analysis [21].

The aiT tool analyses executables stored in ELF format. This format contains information about the code, like symbol tables, which is used by aiT. The information present in the ELF file and the executable itself is typically not sufficient to yield a good WCET bound for the analyzed code. In particular, information about program flow, such as bounds to loop iteration counts not caught by the loop bounds analysis, and knowledge of infeasible paths, has to be provided by the user. Therefore, aiT supports a set of *user annotations* to provide external information to the analysis [13]. Some of the more important annotations constrain the possible program flows: *loop bounds*, *maximal recursion depth*, *dead code*, (static) *outcome of conditions*, and *possible values of registers*. In addition, there are other types annotations, e.g., to constrain the address range of memory accesses (allowing aiT to find better bounds for access times), and to provide hardware-related information such as clock rate and address mapping to different kinds of memories. There are also annotations for controlling the *context-sensitivity* of the analysis: this concerns the interprocedural part of the analysis, and controls for which contexts a function will be analyzed separately and for which contexts a single, joint analysis is to be made. Thus, these annotations can be used to control the tradeoff between analysis time and precision.

Since aiT analyzes the binary code, the annotations have to be given on this level, which is cumbersome. For some compilers, aiT can use symbol tables to map annotations given for the source code to the binary level: however, the user must be aware that the control flow of the code might be different for the compiled code, which means that annotations valid for the source code might not be valid for the binary code. In our study, annotations were made on the binary level.

3.5 The Trace32Fire Emulator

We used the Trace32Fire emulator to estimate the measured WCET, see Section 4.4. This is an emulator from Lauterbach Datentechnik GmbH [33]. It has full support for the whole C166 family, which includes the C167CS. The emulator works just like the real CPU, but has the ability to log every instruction along with register values and execution time. The trace can be folded or unfolded into three different levels. The highest level displays only the source code while the third level displays source code, assembler instructions and register values.

4 Analyses and Results

In this chapter we present and analyse the results from our WCET analyses and measurements. Section 4.1 describes the results from an initial and very simplified analysis made on 24 tasks. Section 4.2 gives some properties of 13 tasks selected for a closer study. Section 4.3 describes the user input needed in the form of annotations for these 13 tasks, and the relation between the number of annotations and the corresponding WCET estimate. In Section 4.4 we compare the initial aiT result, using a minimal number of annotations to obtain a WCET estimate at all, the tightened aiT result, obtained by adding annotations constraining the program flow harder, the measured result, and the current WCET estimate set in Rubus for the selected tasks. All aiT analyses were done with maximal context-sensitivity, in order to obtain maximal precision in this respect. In Section 4.5, finally, we discuss the labor effort involved for performing the different analyses.

4.1 Minimum Effort Analysis

We first tested if it was possible to derive WCET estimates with a minimal amount of effort. This was made by running aiT on an initial selection of 24 tasks. For each task a project file was created specifying the hardware configuration and the start address of the task. Except these analysis tool inputs, no additional annotations were given. The setup of the project files took about two hours, and to run the analyses on an AMD Athlon XP1900+ (1.61 GHz) with 512 Mb of RAM took in total about 70 minutes.

For seven of the tasks aiT could not derive any WCET estimate, mainly due to loops which could not be bounded. (Without loop bounds, any instruction occuring in a loop might be taken an infinite number of times, leading to an unbounded WCET estimate).

For the remaining 17 tasks aiT was able to derive a WCET estimate. When comparing these calculated WCET estimates with the WCET parameters set

Table 1. The 13 tasks selected for more detailed analyses.

Task	Source code (kb)	Lines of code	Nodes in call graph	Depth of call graph
1	3.6	55	1	1
2	4.5	56	1	1
3	4.7	58	3	2
4	5.2	72	1	1
5	6.3	86	2	2
6	4.9	43	6	3
7	9.3	123	5	3
8	8.2	119	5	3
9	5.5	49	5	3
10	10.9	195	3	3
11	8.8	188	5	3
12	40.7	707	20	5
13	565.0	12765	115	8

using Rubus timing mechanism, we got an average time reduction of 59%[1]. Thus it is possible, with a very limited effort, to use aiT to calculate WCET estimates for many tasks in the system, and the calculated values are tighter than the ones derived from measurements.

4.2 Tasks Selected for More Detailed Analyses

As mentioned in Section 4, we selected 13 red Rubus tasks, in the Transmission ECU for articulated haulers, for a closer study (including some, but not all, of the 24 tasks used in the initial experiment, as described in Section 4.1). Table 1 shows a number of complexity measures for the 13 selected tasks. The 'Lines of code' column gives the number of C-code rows with comments and blank rows removed. The column 'Nodes in call graph' includes both functions and loops (aiT treats loops as function calls). Tasks 1 – 11 are quite small, whereas task 12 and especially task 13 are substantially larger.

The source code for the tasks is written in C. All tasks have a simple code structure, with almost exclusively if-statements, just a few loops, one nested loop, no recursion, and no dynamic calls or memory allocations. Switch statements are common. The larger tasks contain a large number of function calls, and some functions are used many times in many different contexts.

4.3 Annotations and WCET Estimates

We first investigated the case where only a minimal number of flow constraints, in order to obtain a WCET estimate at all, was given. This means to bound the

[1] It was not investigated in detail why the calculated WCET estimates got smaller than the Rubus values. Probable causes include measurement probe effects and too large safety margins (see Section 3.2) and that some measurements were made using an old CPU version (see Section 4.4).

Table 2. The effect of extra annotations.

Task	**Loops**	**Necessary annotations**	**WCET (aiT1)**	**Extra annotations**	**WCET (aiT2)**	**Reduction (%)**
1	0	0	5697	2	4788	16
2	0	0	7516	1	5879	22
3	0	0	13516	1	12122	10
4	0	0	19304	3	13394	31
5	0	0	25243	6	22364	11
6	1	1	28637	2	27243	5
7	1	1	34546	6	29061	16
8	0	0	39425	5	30455	23
9	0	0	41940	4	36000	14
10	0	0	73576	8	55091	25
11	0	0	78758	4	70273	11
12	0	0	199612	18	143606	28
13	18	11	1827000	103	1447000	21

number of iterations for all loops in the code, but not more. Only three tasks contained loops. aiT managed to find some loop bounds in one task, but the other loop bounds had to be given manually. To figure out bounds for these loops was not hard. Columns three and four, respectively, in Table 2, show the number of given annotations and the corresponding aiT WCET estimate in nanoseconds.

Next, we investigated if the WCET estimate could be made tighter by reducing the set of possible execution paths. The studied code contains quite a few if-statements where the conditions are or may be mutually exclusive, as exemplified by the following artificial code snippet:

```
if(x == 0) y = 1;
if(x == 1) y = 0;
```

This kind of code gives rise to infeasible paths, i.e., paths which are executable according to the control-flow graph structure, but not when considering the semantics of the program and possible input data values. In the example, both assignments cannot be executed in the same path. The current version of aiT does not detect this, and will consider a path executing both the assignments when calculating the WCET estimate.

Thus, we decided to investigate the effect of eliminating such infeasible paths. After analysing the code, we were able to add a number of manual annotations that removed many infeasible paths, and reduced the WCET estimate. Columns six and seven in Table 2 show the effect on the WCET estimates. We obtain reductions in the range 5 - 31 %. The column 'Extra annotations' shows the number of annotations used to exclude infeasible paths from the analysis.

The time used by aiT to analyze the tasks was between 2 and 6 minutes for all but the largest task, for which the analysis times varied between 15 and 100 minutes depending on which annotations that were used.

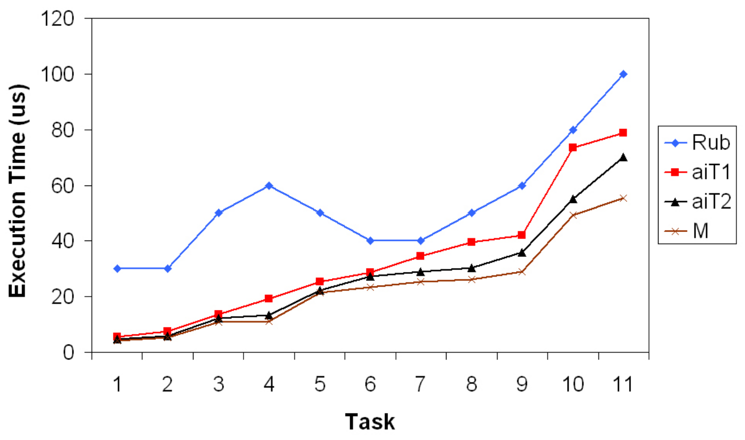

Fig. 2. WCET values for the first 11 tasks.

4.4 WCET Comparisons

We also compared the statically estimated WCET values with measurements. We decided *not* to use the Rubus timing mechanism for this, partly due to the inexactness of this mechanism, but mostly because it is hard to have control over the actually taken path when running on the real hardware, where the possibilities for monitoring are limited. Therefore, it is hard to know that a measured value actually corresponds to the longest path, and thus is the WCET.

Instead, we decided to use the Trace32Fire emulator, see Section 3.5, for the measurements. This emulator is 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 real hardware from the Rubus timing mechanism. They showed consistently that the overhead from the Rubus timing mechanism is around 3.5 microseconds, which strengthens our belief that the emulator indeed has a quite accurate timing model for the Infineon C167CS.

The real advantage of using the emulator is that it gives full control over the execution, as well as full monitoring capability. Thus, we could, for all tasks, force the emulator to execute the same path that aiT considered would yield the WCET path.

The results are illustrated in Figures 2, 3, and 4, and the data and some interesting ratios are tabulated in Table 3. Since task 12 and 13 have WCET's of different magnitudes than the others, Fig. 2 shows only the figures for the first 11 tasks, Fig. 3 for task 1 – 12, and Fig. 4 for all 13 tasks. For each task, we compare four values: the Rubus WCET estimate used in the current system, as given from the Rubus timing mechanism, with an added safety margin (Rub), the aiT estimate with a minimal number of annotations (aiT1), the aiT estimate with infeasible path-reducing annotations (aiT2), and times measured with the emulator, with the same execution path as aiT2 (M).

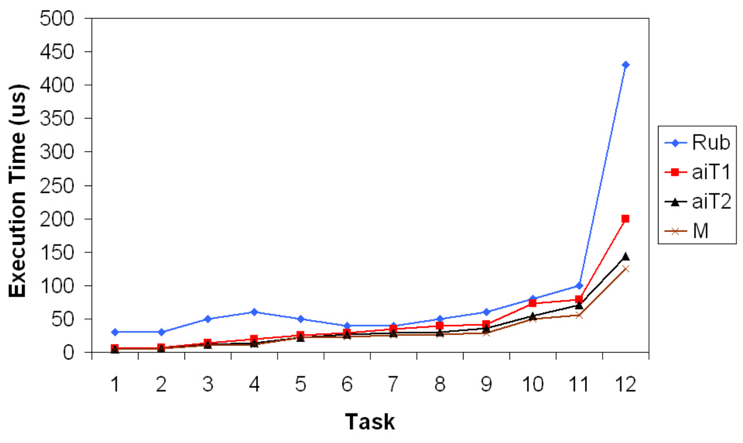

Fig. 3. WCET values for the first 12 tasks.

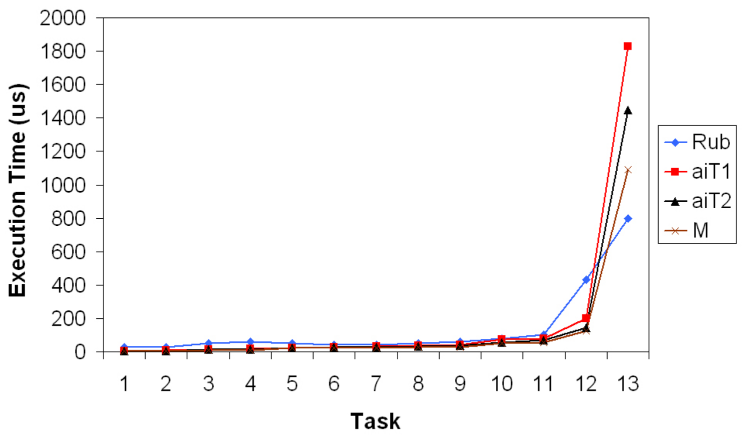

Fig. 4. All 13 tasks.

We believe M, in most cases, to be an overestimation of the true WCET, for the following reason: since aiT may overestimate the set of possible execution paths, which may happen if the program-flow constraining annotations given are not tight, there is a chance that the path chosen in aiT2 is an infeasible path. This path, used in the emulator to measure M, would then yield an overestimation. This is because in our measurements, the emulator was used to force the evaluation to follow the path chosen by aiT even if that path in reality was infeasible.

However, M may in some cases be an underestimation: even if aiT produces safe estimates and the emulator has a correct timing model, it is still not certain that the measured time for this path overestimates the WCET. It might be that the processor behavior analysis in aiT made a large overestimation for precisely this path, which implies that the execution time for this path actually is an underestimation to the true WCET. However, for a relatively simple processor such as C167CS, there is little reason to believe that the processor behavior analysis of aiT gives dramatically different precision for different paths. Since we also believe the emulator to have a quite precise timing, we believe that M does not in general underestimate the WCET by more than a small fraction.

The graphs show that, for all tasks, both aiT1 and aiT2 yield WCET estimates which are larger than the measured value M. The aiT2 estimate consistently improves on the aiT1 estimate, with improvements in the range 5 – 31%. The aiT2 estimate in turn overestimates the measured value M, with overestimations in the range 4 – 33%. As argued above, M probably overapproximates the real WCET in most cases: for those cases, this is a lower bound to the real overapproximation.

For all tasks but one, the currently set WCET values in Rubus are larger than the both the statically derived and the measured WCET estimates. These values were set using the measurement facilities of Rubus plus adding a safety margin (see Section 3.2), which explains why they are larger. Sometimes they are much larger than the measured value, up to 624%. One reason for this is that some, but not all, of the Rubus values were derived using measurements from an old, slower version of the CPU. Unfortunately, Volvo does not have any documentation of which tasks have been remeasured since the CPU upgrade. Nevertheless, the currently set WCET values are substantially larger (at least 38%) than the aiT2 estimate for these tasks, so there should be quite some room to set tighter WCET values for the tasks in Rubus based on the aiT2 estimates. This would leave more room for other activities in the system.

For task 13, the WCET value set in Rubus is lower than both the aiT estimates and the measured value! However, it is not totally clear that this undershoots the real WCET, since both the measured and aiT estimates might have been obtained for a infeasible path. A closer examination of the code for the task would be required to determine whether this is the case or not. Nevertherless, the static analysis has been helpful here to spot a possible problem with the current system setup.

4.5 Labor Efforts

It took the first author, who did the analyses as part of his M.Sc. project, about one week to become sufficently acquainted with aiT to be able to give non-trivial annotations. For the studied 13 tasks, it was then a matter of hours to provide the minimal annotations that would yield a WCET estimate at all: however, this estimate was quite inexact for most tasks.

For adding the extra annotations, the required labor varied quite a lot. The smaller tasks were annotated in about an hour each, but the two largest tasks took together around seven weeks. This also includes the work to find the proper restrictions on input values. Since these restrictions can affect when contradicting conditions give rise to infeasible paths, it is hard to split the time between the work needed to restrict input values and find infeasible path annotations. The annotation language of aiT currently does not have any good means to specify that the execution of two instructions is mutually exclusive. Thus, such infeasible

Table 3. Data derived from Rubus, aiT and measurements.

Task	**Rub**	**aiT1**	**aiT2**	**M**	$\frac{\textbf{Rub}}{\textbf{aiT2}}$	$\frac{\textbf{aiT1}}{\textbf{aiT2}}$	$\frac{\textbf{M}}{\textbf{aiT2}}$
1	30000	5697	4788	4138	6.27	1.19	0.86
2	30000	7516	5879	5340	5.10	1.28	0.91
3	50000	13516	12122	10840	4.12	1.11	0.89
4	60000	19304	13394	11240	4.48	1.44	0.84
5	50000	25243	22364	21560	2.24	1.13	0.96
6	40000	28637	27243	23480	1.47	1.05	0.86
7	40000	34546	29061	25400	1.38	1.19	0.87
8	50000	39425	30455	26235	1.64	1.29	0.86
9	60000	41940	36000	28880	1.67	1.17	0.80
10	80000	73576	55091	49335	1.45	1.34	0.90
11	100000	78758	70273	55418	1.42	1.12	0.79
12	430000	199612	143606	124800	2.99	1.39	0.87
13	800000	1827000	1447000	1090000	0.55	1.26	0.75
AVG	140000	184170	145940	113590	2.68	1.23	0.86
MIN	30000	5697	4788	4138	0.55	1.05	0.75
MAX	800000	1827000	1447000	1090000	6.27	1.44	0.96

path information had to be expressed by condition annotations explicitly stating the outcome of certain branches as taken or not taken. This added to the labor time.

5 Conclusions and Future Work

A number of conclusions can be drawn from the study. Clearly, static WCET analysis tools seem useful for the analysis of task-oriented real-time code like the one studied here, and they fit well into a tool chain like Rubus where the WCET's of tasks are explicit parameters. This kind of hard real-time code often has 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. For the studied code, we have shown that it is possible with a very small effort to get a safe WCET value for the tasks, and that already this basic WCET in almost all cases is tighter than currently used WCET estimates based on measurements with a safety margin added.

However, if tighter WCET estimates are desired, then substantially more work may be needed. 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 as many of these paths as possible 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, and the aiT annotation language currently does not provide the proper constructs for expressing mutual exclusiveness of instruction execution. However, such constraints are readily expressed as linear constraints on execution counters, and for instance the *flow fact format* [12] provides them. Another intriguing possibility is to derive these constraints automatically: initial experiments in this direction using our prototype tool SWEET [34] have shown promising results [35].

We can conclude that there are cases where the use of a tool like aiT can improve the utilization of the system 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.

Future work includes tests to see if the infeasible paths found in the Volvo CE code can be found automatically using the SWEET tool, and to further develop existing techniques to identify infeasible paths in real-time code.

Acknowledgements

This work was performed within the ASTEC competence center[2] under support by VINNOVA[3], as well as by the KK-foundation[4] under grant 2005/0271. Travel support has been provided by the ARTIST2 European Network of Excellence. We want to thank AbsInt GmbH [4] for giving us access to the aiT tool, as well as Volvo Construction Equipment [7] for giving us access to production code to analyze.

References

1. Ganssle, J.: Really real-time systems. In: Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006). (2006)
2. Ive, A.: Runtime Performance Evaluation of Embedded Software. Presented at the 8th Nordic Workshop on Programming Enviroment Research (1998)
3. Stewart, D.B.: Measuring execution time and real-time performance. In: Proc. of the Embedded Systems Conference, San Francisco 2004 (ESCSF 2004). (2004)
4. AbsInt: aiT tool homepage (2006) `www.absint.com/ait`.
5. Tidorum: Bound-T tool homepage (2006) `www.tidorum.fi/bound-t/`.
6. Rapita: RapiTime WCET tool homepage (2006) `www.rapitasystems.com`.
7. Volvo: Volvo CE (construction equipment) homepage (2006) `www.volvo.com/constructionequipment`.
8. Arcticus: Rubus homepage (2006) `www.arcticus-systems.com/productsrubusos.php`.
9. Hansson, H., Lawson, H., Bridal, O., Eriksson, C., Larsson, S., Lönn, H., Strömberg, M.: BASEMENT: an architecture and methodology for distributed automotive real-time systems. IEEE Trans. Comput. **46** (1997) 1016–1027

[2] www.astec.uu.se
[3] www.vinnova.se
[4] www.kks.se

10. Sehlberg, D.: Static WCET analysis of task-oriented code for construction vehicles. Master's thesis, Mälardalen University, Västerås, Sweden (2005)
11. Kirner, R., Puschner, P.: Transformation of path information for WCET analysis during compilation. In: Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01), Delft, IEEE Computer Society Press (2001) 29–36
12. Ermedahl, A.: A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden (2003)
13. Ferdinand, C., Heckmann, R., Theiling, H.: Convenient user annotations for a WCET tool. In: Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003). (2003)
14. Gustafsson, J.: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden (2000)
15. Healy, C., Sjödin, M., Rustagi, V., Whalley, D.: Bounding Loop Iterations for Timing Analysis. In: Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98). (1998)
16. Holsti, N., Långbacka, T., Saarinen, S.: Worst-case execution-time analysis for digital signal processors. In: Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference). (2000)
17. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of WCET tools. IEEE Proceedings on Real-Time Systems (2003)
18. Engblom, J.: Analysis of the execution time unpredictability caused by dynamic branch prediction. In: Proc. 8th IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03). (2003)
19. Healy, C., Arnold, R., Müller, F., Whalley, D., Harmon, M.: Bounding pipeline and instruction cache performance. IEEE Transactions on Computers **48** (1999)
20. Engblom, J.: Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden (2002) ISBN 91-554-5228-0.
21. Ferdinand, C., Martin, F., Wilhelm, R.: Applying compiler techniques to cache behavior prediction. In: Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97). (1997)
22. Holsti, N., Långbacka, T., Saarinen, S.: 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). (2000)
23. Rodriguez, M., Silva, N., Esteves, J., Henriques, L., Costa, D., Holsti, N., Hjortnaes, K.: 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)
24. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Proc. 1st International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211. (2001)
25. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software. In: Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003). (2003)

26. Ermedahl, A., Gustafsson, J., Lisper, B.: Experiences from industrial WCET analysis case studies. In: Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005). (2005) 19–22
27. Byhlin, S., Ermedahl, A., Gustafsson, J., Lisper, B.: Applying static WCET analysis to automotive communication software. In: Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05). (2005) 249–258
28. Engblom, J.: Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In: Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99), IEEE Computer Society Press (1999)
29. Sandberg, C.: Inspection of industrial code for syntactical loop analysis. In: Proc. 4th International Workshop on Worst-Case Execution Time Analysis, (WCET'2004). (2004)
30. Infineon: Infineon Systems homepage (2006) `www.infineon.com`.
31. Thesing, S.: Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models. PhD thesis, Saarland University (2004)
32. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM Symposium on Principles of Programming Languages, Los Angeles (1977) 238–252
33. Lauterbach: Lauterbach datentechnik GmbH homepage (2006) `www.lauterbach.com`.
34. Mälardalen University: WCET project homepage (2006) `www.mrtc.mdh.se/projects/wcet`.
35. Gustafsson, J., Ermedahl, A., Lisper, B.: Algorithms for infeasible path calculation. In Mueller, F., ed.: Proc. 6th International Workshop on Worst-Case Execution Time Analysis, (WCET'2006), Dresden, Germany (2006)