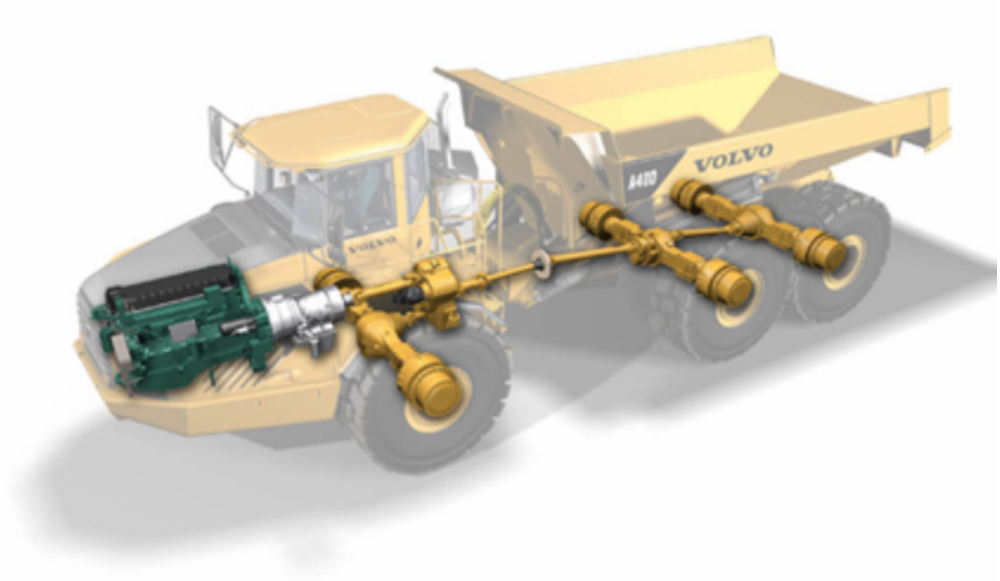
Static WCET Analysis of Task-Oriented Code for Construction Vehicles



Master's Thesis
Daniel Sehlberg

Dept. of Computer Science and Electronics
Mälardalen University, Västerås, Sweden

September 30, 2005

Abstract

When creating real time systems for embedded vehicle computers it is important to know how they will behave during the worst possible scenario. To know each task's WCET (Worst Case Execution Time) is a prerequisite for being able to know if their deadlines always will be met. A missed deadline in a real time system can lead to severe consequences.

Volvo CE is using the Rubus real time operating system for their vehicles. Before Rubus is creating the task schedule, the user has to specify the period time, offset time and WCET for each task. The estimated WCET for each task is today based on measures from automatic tests or by experiences. However, it can not be guaranteed that these automatic tests have found the longest possible path through each task. This can result in a too low estimate of the WCET, which means that the behaviour of the system is unreliable.

This thesis investigates if WCET analysis tool aiT can be used to find safe WCET estimates for the tasks in Volvo CE's applications, and how much user input is needed to get the result. Analysis results from aiT are compared to measured results and the WCET estimates used today.

We have found that the times set in Rubus are in many cases greatly over-estimated, and that Volvo CE would have good use of analyzing the tasks with aiT in order to get tighter WCETs. By giving annotations for the tool, the WCET estimates can become even tighter, but this also requires some manual work.

Acknowledgements

The work on this thesis was made at the department of electronic product development (TUE) at Volvo CE in Eskilstuna between March and August 2005. This is the final part of my education towards a Master of Science degree in Computer Science at Mälardalen University.

I would like to thank my mentor Björn Lisper and his colleague Andreas Ermedahl at Mälardalen University. Björn has constantly answered my questions and steered me in the right direction while Andreas has also been there to answer questions and coming up with ideas. It should also be mentioned that Björn and Andreas were the ones that initiated this project together with Kurt-Lennart Lundbäck at Arcticus Systems.

I would also like to thank everybody at Volvo CE who helped me in any way, especially Jesper Åström, Robert Larsson, and my second mentor Nils-Erik Bånkestad. Jesper was the application expert and helped me choose the functions to analyze. He also taught me how to use the debug environment. Robert gave me an introduction to Rubus and answered any questions about the platform that I had. Nils-Erik checked up with me daily with questions, answers and ideas. Andreas C Johansson and Robert Eriksson also deserves to be mentioned because of their help with setting up the emulator and getting it started, which was easier said than done.

Ola Eriksson, who was a couple of months ahead of me with his thesis, was kind enough to give me hints on working with aiT and we interchanged some thoughts and ideas about the tool.

Martin Sicks at AbsInt Angewandte Informatik has been showing a great deal of expertise. He has had very good answers to all my questions about aiT, and has spent much time helping me.

Contents

Chapter 1

Introduction

Most computers today are embedded as parts of a large variety of products, for example telephones, toys, airplanes and cars. Real time systems, which are common in vehicle computers, are systems where not only the computed result is of interest, but also at what time the result is delivered. Tasks in real time systems often have deadlines, and if those deadlines are not met it can have catastrophical effects. It could cause an airplane to malfunction or the airbag in a car to inflate too late or too soon. It is very important that these systems are predictable and products with unreliable behaviour should not be delivered from the developers.

One way to know if a task will be able to finish before its deadline is to know the WCET (Worst Case Execution Time) of the tasks in the system. The most common way to find WCET estimates today is to measure its execution time with a number of different inputs. A big problem with this approach is that with larger tasks it is very hard to know if the worst possible combination of inputs has been used. Another way of finding WCET estimates is to use something called static analysis, which relies on abstract models of the real systems.

AiT is a tool for static WCET analysis, created by German company AbsInt [1]. The tool makes an analysis of the binary file in order to find the longest possible path through a chosen code segment. After the analysis, aiT displays a combined call graph and flow graph with WCETs for each basic block, each routine, and for the whole segment.

The platform for Volvo CE's software applications is Rubus Operating System (hereafter Rubus OS), created by Arcticus Systems [2]. Rubus OS maintains real time tasks of different kinds. Time-triggered tasks that run periodically, interrupts, or low-priority, event-triggered tasks that run in the background.

This project was a collaboration between Mälardalen University [3], Arcticus Systems [2], and Volvo Construction Equipment (hereafter Volvo) [4]. The purpose was to evaluate if aiT could be of use for the time-triggered tasks in Volvo's articulated haulers. The tasks are written in plain C code with many if-statements, some loops, and no dynamic memory handling. Interesting aspects of this evaluation were how well the code at Volvo CE suited aiT and how much manual work was needed in order to get good results from the tool. The values from aiT was also compared to measured values.

1.1 Thesis Outline

This chapter will continue by introducing the reader to Volvo CE, real time systems and WCET analysis. The final section in this chapter will recommend some articles, theses and dissertations within the same area of research for any readers who wish to read more.

The second chapter will state the objectives of this thesis in detail, and describe methods on how to reach the results.

In the third chapter, the reader will be introduced to hardware and software used in this project. This includes information about the CPU (Central Processing Unit) and its pipeline, the CPU emulator, the operating system, and most importantly, the aiT WCET analysis tool.

In the fourth chapter, results are presented and analyzed. This chapter contains many graphs and tables used to draw conclusions for patterns between code sizes, call depths, loops, execution time etc. We will see if the Volvo CE code was well suited for static WCET analysis, how much user interaction was needed to get results and how good the results were.

The fifth chapter will sum up the conclusions, and chapter six will propose some future work within the area of WCET analysis.

1.2 Volvo CE

Volvo Construction Equipment (CE) is one of the world's leading manufacturers of construction equipment. Their product range encompasses backhoe loaders, wheel loaders (WLO), excavators, articulated haulers (ART) and motor graders (VMG).

In Eskilstuna, Sweden, where this project was performed, Volvo CE has their main office and development of engines, electronics, axles and transmission for WLO, ART and VMG.

The number of computers in the different vehicles varies, but an articulated hauler contains five computers or, to be more precise, five ECUs (Electronic Control Units).

- Cabin ECU
- Engine ECU
- Instrument ECU
- Transmission ECU
- Vehicle ECU

These computers 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. This thesis treats thirteen tasks in the Transmission ECU for articulated haulers.

Figure 1.1: Motor graders, wheel loaders, excavators and articulated haulers are some of the products developed by Volvo CE.

1.3 Real Time Systems

A real time system is a computer system where not only the results are important, but also the time at which they are delivered. Usually, real time systems consist of many tasks. A task can consist of one or more functions, but usually does not concern itself with more than one assignment. For example, one task can control oil temperature while another task has the assignment to change gears. The tasks can be time-triggered or event-triggered.

1.3.1 Time-Triggered Tasks

Time-triggered tasks usually have a period time, a release time, a WCET and a deadline as attributes. In many cases, the release time for a task is the beginning of its period and the deadline is the end of its period. Even though many tasks may have the same release time, it does not mean that they execute at the same time. A scheduler is used to decide which task is executing when, i.e. it creates the task schedule. For every new period the schedule starts over.

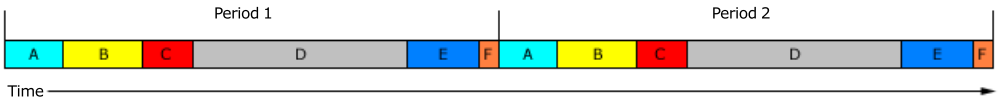


Figure 1.2: A task schedule with six tasks named A-F.

1.3.2 Event-Triggered Tasks

Event-triggered tasks can wait for some spare time in the schedule before running or, if it is a critical task, it can preempt[1] the task currently running. Examples of their attributes can be priority or stack size.

[1] preempt = to take the place of; displace (source: www.dictionary.com)

1.3.3 Scheduling Paradigms

The scheduler uses some paradigm in order to decide which tasks have higher priority than the others. In some systems, each task simply has an attribute called priority, and in other systems the task with the shortest period might be the task to run first. The latter paradigm is called Rate Monotonic (RM) and is commonly used. A more flexible paradigm is where the task closest to its deadline runs with the highest priority and this task might even preempt another task already running. This is called Earliest Deadline First (EDF).

1.3.4 Hard and Soft

There are two kinds of real time systems: hard and soft. In a hard real time system, it is very important to know the WCET of a task so that it can be guaranteed to meet its deadline. A missed deadline for a hard real time task can have fatal consequences, like for example if an airbag does not inflate at the correct time. Soft real time systems are those that do not cause big danger if malfunctioning, i.e. their deadlines are not as important to meet.

1.4 WCET Analysis

When constructing a real time system, some method of finding out the WCET for the tasks must be used. It is not unusual that the WCET estimate is just based on experience from earlier measurements, but with a large overapproximation added. These overestimations can be of a magnitude many times larger than the real WCET of the task and as long as there is room for all tasks in the schedule this is not a problem. If the schedule is getting full and the developers wants to add more tasks however, it is of interest to have tighter WCETs.

1.4.1 Dynamic Methods

Other methods include measuring the time for a number of different scenarios or calculate the times according to models. By measuring the time, a so called dynamic method is used. Common tools for measuring times are logical analyzers, oscilloscopes or some clock supplied by the OS. Many different measurements have to be done on many different inputs. The highest time measured might be the actual WCET, but unless all possible combination of inputs have been measured, there is no guarantee that the actual WCET have been found. When using this method, a safety margin is often added to the measured WCET.

1.4.2 Static Methods

The opposite of the dynamic methods are static methods. They rely on models of the CPU and might be a better choice to get a safe WCET estimate. When using a static method, the goal is to find the longest possible path through the task. If the models are correct, the time it takes to execute that path is definitely a safe WCET estimate, i.e. a value larger than or equal to the actual

WCET. The drawback with these methods is that they might find a path that is practically infeasible, and therefore produces an overestimation.

1.4.3 Differences Between the Methods

As Figure 1.3 shows, the measured values are less than or equal to the actual WCET assuming that there is no overhead in the measurements. The analyzed values are more than or equal to the actual WCET. If reliable dynamic and static methods would produce the same result, the actual WCET would be found.

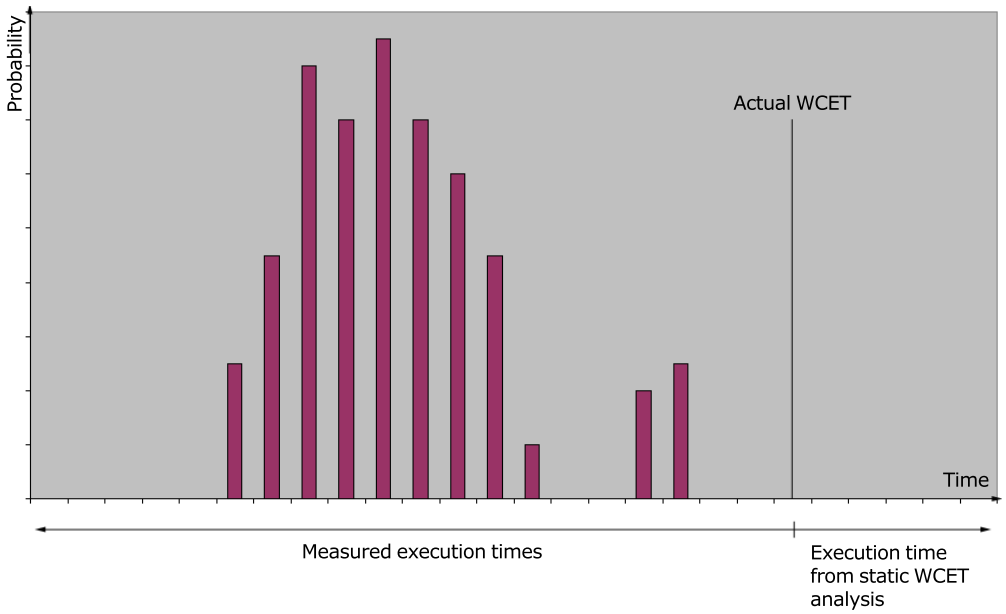


Figure 1.3: Results from dynamic methods are always lower or equal to the actual WCET, while results from static methods always are higher or equal.

1.5 Tools for WCET Analysis

Developed by German company AbsInt [1], aiT is the only commercial tool that has support for Motorola and Infineon processors. We will get more into detail about aiT later[2] since this is the tool used in this project. Except for aiT, there are a few other tools on the commercial market. Even though the biggest commercial tools, aiT and Bound-T, uses different types of static analysis methods and could therefore be rivals, they are also supporting different processors. This makes it an easy choice for which tool to use.

Bound-T is a WCET tool from Finish company Tidorum Ltd [5]. It was started as a project for the Europan Space Agency, and was then further developed by Space Systems Finland. Bound-T supports ERC32/SPARC V7, Hitachi

[2]aiT is treated in Section 3.3.

H8/300, ADSP-21020, and some processors from the Intel-8051 series. The tool is source code independent.

WCET tools still in the research stage are for example Heptane (Hades Embedded Processor Timing ANalyzEr) [6], which is published under the GNU GPL licence and downloadable for free at the website. It has support for Pentium I, MIPS and Hitachi H8/300.

Another prototype WCET tool is SWEET (SWEdish Execution time Tool), presented by Andreas Ermedahl, Jan Gustafsson, Björn Lisper and Christer Sandberg from Mälardalen University [7, 8, 9]. Today, their research concerns flow analysis on intermediate code in order to be platform-independent. The tool is divided into modules, so that it will be easy to change separate parts of the analysis. Supported processors are ARM9 and NEC V850E.

1.6 Related Work

In February 2003, Toni Riutta and Kaj Hänninen wrote their masters thesis entitled Optimal Design [10] at Volvo CE. The main objective was to find out if some of the hard offline scheduled tasks could be changed to soft online scheduled tasks. This report treats the development process at TUE and gives an introduction to Rubus. The main part is a huge literature study that discusses many articles about how to optimize the CPU utilization for a real time system. Their presented solution is a weakly hard real time system with some modifications.

Daniel Sandell evaluated in his masters thesis [11] how well suited the OSE operating system code was for static WCET analysis with aiT. System calls and interrupts were analyzed and compared to results from an ARM CPU emulator called ARMulator. This thesis gives a very good introduction to flow analysis, pipeline analysis and low-level analysis. He finds that the interrupts are well suited for static analysis and that the system calls also can be analyzed, but with a little more manual work. A shorter report on this project was also written [12].

Another student who has used aiT is Susanna Byhlin who wrote her masters thesis [13] at Volcano Communications Technologies AB. The objective was to evaluate if Volcano could save time and money by integrating static WCET analysis into their current tool chain. Her report gives good introductions to CAN and LIN (Local Interconnect Network). The conclusions were that most of the workload was to find loop bounds, and that aiT would be well suited for Volcano. This project is also documented by a shorter report [14].

Ola Eriksson and Yina Zhang were working on their masters theses at CC Systems [15] almost during the same time period as this project was performed at Volvo CE. They both analyzed the same code snippets, but Ola used static analysis with aiT [16] while Yina performed dynamic analysis with oscilloscopes and logical analysers [17]. Their conclusions are that the oscilloscope is not a suitable instrument for WCET analysis, but a combination of aiT and a logical analyzer would be optimal. Ola proposes that a person performing static analysis should have very good knowledge of the code or at least good contact

with the one who has written it.

For deeper information about static WCET analysis, Stephan Thesing has written a doctor's dissertation about modelling hardware [18]. The emphasis on the dissertation lies within abstract interpretation and pipeline models. He cooperated with people at AbsInt, the company behind aiT, and Airbus France during his work.

Another deep look into static WCET analysis can be found in Andreas Ermedahl's dissertation [7]. He proposes that the analysis is divided into modules with well-defined interfaces along with calculation methods and execution representation. Written in 2003, this was the beginning to the research WCET tool SWEET.

Chapter 2

Problem Description and Method

The objectives of this project were as follows:

- Evaluate how well suited the Volvo CE code is for static WCET analysis
- Investigate how much user interaction is needed when using aiT
- Investigate how exact the aiT results are
- Compare aiT results with WCET estimations used today
- Compare complexity and size properties of the code to WCET estimations and workload
- Investigate the use of batch jobs

The project started by analyzing some simple tasks and then proceeded with more complex tasks. The first step for every task was to analyze it with as few annotations[1] as possible. After that, the aiSee[2] graph was used to determine if aiT has chosen paths that are practically infeasible. If this was the case, these paths was excluded from the next analysis by giving manual annotations. Then the input arguments used to produce the WCET path was determined and used for doing a measurement in a hardware environment. It was of interest to see if the measurements would match the analysis results, and also how much the first analysis results (using as few annotations as possible) differed from the final analysis result.

During this whole process, notes was taken on how much labor that was needed to produce the first results as well as the final results. Different measures of size and complexity of the tasks were looked into trying to find some patterns between complexity and the number of annotations as well as between complexity and WCET.

The final part of this project was to see how command-line runs of aiT could be of use for running batch jobs.

[1] Annotations are instructions for aiT, written by the user. More about them in Section 3.3.2.

[2] AiSee is AbsInt's graph viewer. More about aiSee in Section 3.3.4.

2.1 Comparing WCET Estimates

As mentioned above, when all annotations were done, the input arguments that produced that path were determined by studying the graph. These arguments were then used in the hardware environment when running the same function. The Trace32Fire emulator was used as hardware since a run can not be traced on the real CPU. Trace32Fire provides tools for measuring the execution time of the run.

The scheduler in Rubus needs an approximation of the WCET in order to be able to schedule the system. Volvo has measured them with Rubus' own timer during automatic testing and then added a large margin to be sure to have safe WCET estimates. The times provided by aiT, with and without annotations, were compared to the times measured in the emulator and the WCET estimate given to Rubus.

If the measured time happens to exceed the aiT time, it means that the WCET estimate given by aiT is not safe. If the Rubus WCET is lower than the measured, it is an indication that Volvo might have assumed a too low WCET. However, it does not necessarily mean that the system can crash, because the measured value can represent a combination of inputs that never are possible during a regular run of the system. We can for example provoke heating of oil that is in critical need of cooling even though the system never would have let the oil reach such a high temperature in real life.

Chapter 3

Project Setup

This chapter will introduce the hardware and software used during this project. We will begin by looking at the CPU and try to understand how a pipeline works. Then we will look at the real time operating system and its accompanying development tool. Section 3.3 will give a description of aiT. Finally, we will take a quick look at the Trace32Fire emulator.

3.1 Infineon C167CS

We have been working with a project that uses Infineon C167CS [19]. This is a 16-bit, single-chip microcontroller with a four stage pipeline. The memory space of the C167CS is configured as a Von Neumann architecture. That means that code memory, data memory, registers and I/O ports are organized within the same linear address space which in this case covers up to 16 Megabytes.

CPUs in embedded systems are used for simpler purposes than those for a PC. They are usually simpler and do not have the same processing power. For example, this processor has a frequency of 33 MHz, and the processor in a PC today has about 3 GHz.

3.1.1 Pipeline

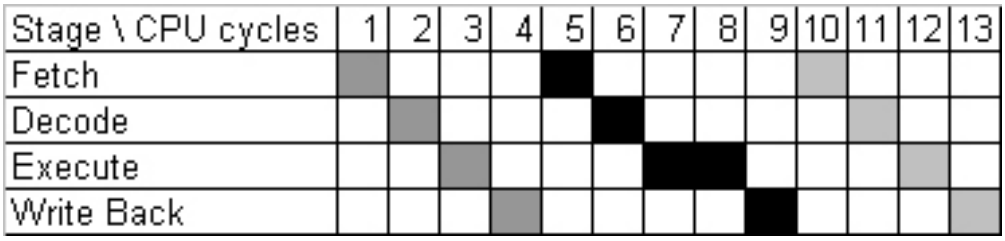


Figure 3.1: This picture shows three instructions executed without a pipeline. In this case, the CPU finishes each whole instruction before starting with the next.

Pipelining instructions can be compared to manufacturing parts on an assembly line [20]. The purpose in both cases is to keep every stage of the process as busy as possible. Assume, for example, a factory that is cutting, drilling, polishing and packaging a product, it might be a good idea for them to use four stations for these processes. In that case, the station cutting the product does not have to wait for each product to be packaged before starting to cut the next. The more stages, the less has to be done at each stage and the more can be done at the same time. However, there is also a bigger chance of dependencies between the stages.

Pipelining is used to speed up the CPU by overlapping different execution stages of instructions [11]. Under ideal conditions, a pipeline can perform a speedup equal to the number of stages, but resource and instruction dependencies usually cause a lower speedup. For example, instructions can have to wait for data that is not yet produced by an earlier instruction, or some instruction can need more than one cycle in some stage, which causes the next instruction to be stalled until that stage of the pipeline is available. The pipeline of the Infineon C167CS has four stages. These are:

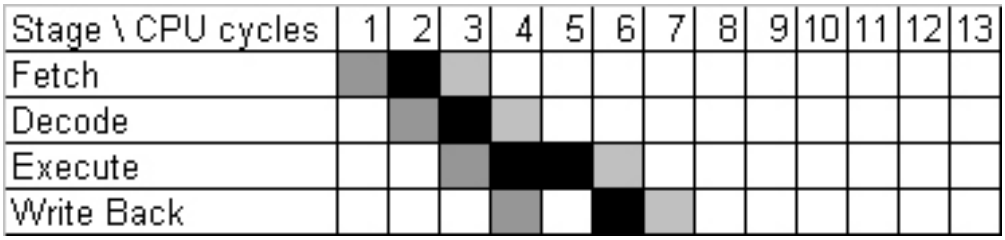


Figure 3.2: The same instructions executed in a pipelined CPU. Note that the third instruction is stalled in the decoding stage for one cycle because the second instruction takes two cycles in the execution stage.

1. **Fetch**
   The instruction to be executed is fetched from internal ROM (Read-Only Memory), internal RAM (Random Access Memory), or external memory.

2. **Decode**
   The instruction is decoded and, if necessary, operand addresses are calculated and respective operands are fetched. The stack pointer (SP) register can be decremented or incremented. For branch instructions the instruction pointer and code segment pointer can be updated.

3. **Execute**
   The operation is performed on the fetched operands. Auto-increment or auto-decrement writes used as indirect address pointers are also performed.

4. **Write back**
   External operands and remaining internal operands are written back to a specified location.

If comparing Figure 3.1 and Figure 3.2, it shows that the effectiveness is greatly improved if a pipeline is used.

3.2 Rubus

The platform for the vehicle systems at Volvo is the Rubus real time operating system, which is created by the Swedish company Arcticus Systems [2]. Rubus OS implements three types of tasks called green, red and blue. Green tasks are interrupts. They have the highest priority 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 executes when there are no red or green tasks running. They are scheduled firstly according to a priority attribute and secondly by the First-In-First-Out paradigm. The other attribute for blue tasks is stack size. Since preemption is very rare between the red tasks, they share the same stack. The blue tasks allocate their own stacks.

Priority	Name in Rubus	Type of tasks
High	Green	Interrupts
Medium	Red	Time-triggered hard tasks
Low	Blue	Event-triggered soft tasks

The Rubus OS supports different modes[1] that can have different schedules. When switching mode, the schedule is also switched.

3.2.1 Rubus VS

Rubus Visual Studio (VS) is a graphical development tool used when creating or editing the system. The user can easily create components and connect them with ports and communication paths.

3.3 aiT

The German company AbsInt [1] has, apart from cache analyzers, stack analyzers etc., created a static WCET analysis tool called aiT. The tool was created as part of the DAEDALUS project and by specifications from Airbus France.

The tool supports Texas Intruments TMS320C33, ARM7, HCS12/STAR12, PowerPC MPC555, PowerPC MPC565, PowerPC MPC755, ColdFire MCF5307, Infineon C167CS-LM, ST10F269 and ST10F276. In this study, Infineon C167CS was the CPU used. The company has sold its product to Airbus France, Bosch, DaimlerChrysler and Ford to mention a few. They were awarded the European IST award in 2004.

[1] Startup mode, drive mode, shutdown mode etc.

The analysis is done on a binary file in the IEEE-695 format (.abs), which in this project was created by the Tasking C166/ST10 compiler. WCET determination consists of several analysis phases. First, the binary is read, then the loop bound analysis tries to determine bounds on the number of loop iterations, the value analysis tries to statically determine register values, the pipeline analysis computes WCETs of the basic blocks, and the path analysis derive an overall WCET based on the block WCETs. The pipeline analysis might also be called low-level analysis. If no errors were encountered during these analysis phases, a combined call graph and flow graph is constructed and opened in aiSee where it can be interactively explored.

Figure 3.3 shows the aiT workspace. In the upper left window, it is declared which binary to analyze, which program point to start the analysis at, which annotation file[2] to read, where to save the report file[3], and where to save the graph file[4]. The window below displays info, warnings and errors during the analysis. The window to the right is the disassembly window, which displays the entire assembler code analysed. Inclusion of source code in the disassembly window is optional.

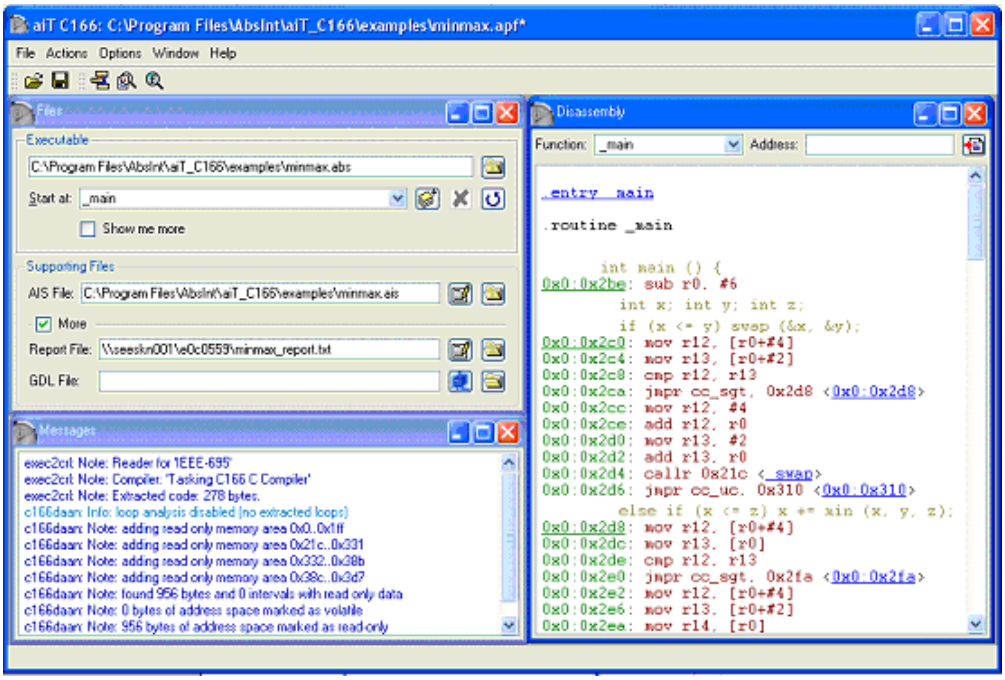


Figure 3.3: The aiT workspace. The annotation file is referred to as AIS file and the graph file is referred to as GLD file. These letters are the respective file extensions.

[2]More about annotations in chapter 3.3.2
[3]More about report files in chapter 3.3.3
[4]More about the generated graphs in chapter 3.3.4

3.3.1 Assembler

Since the analysis in aiT is made on the binary, it means that it operates on the assembler instructions. These instructions are also presented to us when viewing a flow graph or the code in the disassembly window. Luckily, there is information in the binary specifying which rows in which source files that certain instructions correspond to. AiT could therefore display the source code above the corresponding assembler instructions. Sometimes however, the source code can be erroneous or missing. This could be caused by either wrong information in the binary or by faulty interpretation in aiT. Those times, some knowledge of the assembler code can be of good help. Below are three rows of simple c code and their corresponding seven rows of assembler code. This is how it looks in the disassembly window.

```
x ++;
0x14:0xb106: mov r9, 0x3502
0x14:0xb10a: add r9, #1
0x14:0xb10c: mov 0x3502, r9

if (x > 49)
0x14:0xb110: cmp r9, #0x31
0x14:0xb114: jmpr cc_sle, 0xb11e <0x14:0xb11e>

x = 49;
0x14:0xb116: mov r12, #0x31
0x14:0xb11a: mov 0x3502, r12
```

Before each instruction, the page number (e.g. 0x14) and address (e.g. 0xb106) to that instruction is specified. The assembler instructions above are read as:

1. write the value located at address 0x3502 to register 9
2. add 1 to the value of register 9
3. write back the value of register 9 to memory address 0x3502
4. compare the value in register 9 to hex number 31 (decimal number 49)
5. jump to instruction 0x14:0xb11e depending on above comparison
6. write hex number 31 (decimal number 49) to register 12
7. write the value of register 12 to memory address 0x3502

3.3.2 Annotations

In order to help aiT in many different ways, annotations can be written. These can either be written in a specific annotation file or directly in the source code. During this project, we have chosen to write them in the specific file. The

annotations brought up in this section are only the ones used in this project and there are several other kinds of annotations in the aiT manual. The other types of annotations can for example specify an ending point of the analysis, help aiT with unresolved calls, specify for each instruction which memory address is accessed, exclude parts from the analysis and set a maximum recursion depth.

There are two different groups of annotations - mandatory and optional. The mandatory annotations are those needed to help aiT with, for example, unresolved loops, while the optional annotations are used to tighten the result from the analysis. Every annotation is started with a keyword and ended with a semicolon. Comments in the annotation file are started with #.

Some of the annotation examples in this section are written in two different ways. All program points can be described with either absolute or relative addressing, and the closing paragraphs will describe the differences between these two more in detail.

Compiler and Clock Rate

Since we in this project have used the Tasking C166 compiler and analyzed tasks for a CPU with a clock rate of 33 MHz, the following annotations have been used for every analysis:

```
clock exactly 33 MHz;
compiler "c166-tasking";
```

Loop Bounds

After running an analysis with only these two conditions, aiT might fail because of unknown loop bounds. The error message looks like this:

```
pathan2: Warning: In "?\\loop.c":
        In routine '_main.L1' at address 0x0:0x252:
        Loop bound missing at start node.
lp_solve: This problem is unbounded
Execution of 'C:\Program Files\aiT_C166\bin\lp_solve' failed
```

_main.L1 in the warning represents the first loop in function main. If there were more loops in the same function, the second one would be referred to as routine _main.L2 and so on. All functions and all loops are routines in aiT.

To help aiT with this problem, the user must manually find the maximum loop bound to the first loop in function main. By clicking on the warning, aiT opens up the source code and the line where the loop is located. Let us say for example that the user find the maximum number of iterations in this loop to be 5, and that the comparison steering the loop is located at the beginning[5]. The user can now write the annotation in two different ways, either using absolute addressing or relative addressing:

[5]For-loops and while-loops have the comparison at the beginning. Do-while-loops have the comparison at the end.

```
loop 0x0:0x252 max 5 begin;
```

or

```
loop "_main" + 1 loop max 5 begin;
```

Context Specification

Every routine is analyzed for each context it appears in, i.e. for every time it is called from another routine. In a very complex system there can be a huge amount of different contexts that increases the analysis time. If the user wants to limit the number of contexts analyzed, an annotation can be written for that. The annotation is called interproc and has two calculation methods - limited and flexible. In order to activate the automatic loop bound detection the flexible method must be chosen, hence the limited method is not discussed here.

```
interproc flexible;
```

The attribute max-length is what limits the length of the call strings[6] for each routine. A max length of 1 will result in a less specific result, but will also shorten the analysis time a little. To analyze each context separately, an infinite max length is used. The longer analysis time is definitely a worthy tradeoff for the better precision, so we have in this project therefore used the infinite max length.

```
interproc flexible, max-length = inf;
```

There are also attributes for specifying a maximum loop bound for the automatic loop detection and a maximum recursion depth, but those are not used in this thesis.

Memory Accesses and Known Register Values

It might happen that aiT is unsure of which memory address to access, and it will therefore assume an access to the slowest possible memory. These accesses can be seen in the report file as a totally unknown access to

```
phys([?])
```

or an access to somewhere within an interval, like for example

```
phys([0x100000..0x10ffff])
```

For these uncertain accesses, aiT can be helped in a couple of ways. Either memory addresses can be defined, or accesses can be limited to certain address spans. This can be done either for a whole task, or for each program point. An annotation can look like this:

```
instruction "_main" + 1 write accesses 0x10cfd2 .. 0x10cfe6.
```

[6] A call string is a list of routines that are called before reaching the present routine.

Instead of declaring specific accesses, register values can be declared at some suitable program point. The values for the registers can be fetched from the debug environment by setting a breakpoint at a suitable line in the task and reading the register values after stopping there. What we have done in this project is to give aiT the values of some important registers as they are at the entry point of the task.

```
instruction "_main" is entered with r12=0x4363, r13=0x0041;
```

In order to remove all uncertain accesses, aiT can be given the register values or accesses can be declared for every instruction, but this is very time demanding.

Conditions

The condition-annotation can be used to force a branch in one direction. If an expression is supposed to be false even though aiT seems to choose the true-path, aiT must be told which way to go. Take the following program snippet as an example:

```
if (x == y) z = x;
```

The corresponding assembler code might look like this:

```
0x0:0x250: cmp r3, r4
0x0:0x252: jmpr cc_sgt, 0x256
0x0:0x254: mov r5, r3
```

The first instruction does a comparison between the content in register 3 (x) and register 4 (y). The second instruction is a branch that makes a jump to address 0x256 if the comparison gets evaluated to be true. The third instruction assigns the content in register 3 (x) to register 5 (z).

If we could guarantee that x and y never have the same values, and z should not be assigned the value of x, the analysis of the branch should be forced to jump over the assignment and land on the instruction after (0x256). The branch must therefore be set to be always true. This can be written as:

```
condition 0x0:0x252 is always true;
```

or

```
condition "_main" + 2 branches is always true;
```

The second choice assumes that this snippet is located in the routine _main and that this is the second branch in that routine.

It is often the case with branches that if they look like they should be false judging from the source code, the corresponding assembler condition should be true and vice versa. Look at the assembler snippet above for example. All annotations concerning program points are written for the assembler instructions. The source code is only there to help us understand the instructions.

Flow

If the user does not want to manually figure out to know which way to go in a branch, aiT can choose the correct path if it is annotated which instructions always execute together. Take the following code snippet as an example:

```
if(a == 1) {
        b = 1;
        d = 1;
}
else
        b = 0;

if(a == 1)
        c = 1;
else {
        c = 0;
        d = 0;
}
```

Are there some parts of this program that always are executed together? Yes, for example the assignments of b=1 and c=1 or the assignments b=0 and c=0.

AiT will choose the path that assigns b=1, d=1, c=0, d=0 since that is the longest path, but since that is impossible, aiT needs information about that. Assume that the assignment b=1 corresponds to instruction 0x0:0x1e40 and that the assignment c=1 corresponds to instruction 0x0:0x1e52. It can then be written:

```
flow each 0x0:0x1e40 / 0x0:0x1e52 is exactly 1;
```

This annotation means that if 0x0:0x1e40 is executed n number of times during each run of this function, then so is 0x0:0x1e52. Now aiT will choose either the path that assigns b=1, d=1, c=1 or the path that assigns b=0, c=0, d=0. For this small segment, the predicted WCET has probably been shortened with about 2 CPU cycles.

There are more ways to use this annotation, but this is the only way it is used in this project. Another example is:

```
flow sum 0x0:0x1e40 / "_main" + 2 calls is max 4;
```

This means that the instruction at 0x0:0x1e40 will be executed at most four times as many as the second call in routine _main during all combined runs of this routine. The flow annotations can only be used for two program points within the same routine, so the only way that the annotation above would work is if 0x0:0x1e40 is located in the routine _main.

If the user would have been able to annotate two program points that are not executed together, this type of annotation would have been even more useful. Take the following code snippet:

```
if(a == 1) b = 1;

if(a == 0) b = 0;
```

The user sees that only one of these two assignments will be executed, but the flow annotation can not be used to tell aiT. If there would have been an else-statement to just one of these, the annotation could have been used in order to say that the else-statement would be executed just as many times as the other if-statement. What the user has to do here is to read the graph to see if any of the assignments take more time than the other. Then the condition annotation can be used for steering one of them.

For the simple examples used in this report, it is not much of a problem to manually solve which path to choose. The real code can be much more complicated with function calls, multiplication, division and more if-statements. This is when the flow annotation really comes to good use.

Absolute Addressing

The easiest way to write an annotation is to use the address to the instruction involved since these addresses are easily found in the flow graph or in possible warnings. An absolute address looks like 0x0:0x252.

Relative Addressing

Relative addressing is a little more complicated to find and to write, but is much more explaining than an absolute address. A relative address consists of a name to the routine followed by +/- n parameters. The parameters can be, for example, loops, calls, branches, writes, reads or instructions. Every parameter type except loops can be combined. If, for example, the user wants to declare that the condition in the third branch after the fourth call in routine _main is always false, it would look like this:

```
condition "_main" + 4 calls + 3 branches is always false;
```

AbsInt have constructed this annotation language so that it will be very easy to understand what is written, and it even let the user choose between singular and plural when declaring a number of parameters. For example, 3 branches have the same meaning as 3 branch.

If the project is rebuilt, an absolute address is most likely incorrect and the instruction must therefore be relocated and the annotation rewritten. A relative address might still be correct, but since the compiler can build the project differently each time this is not a certainty.

3.3.3 Report File

The report file created by aiT contains detailed information about the whole analysis. It mentions the analysis start time, which files are used, the addresses of the memory pointers, loop bounds, memory addresses which are considered

read-only and memory accesses for each instruction. The memory accesses are displayed like this:

```
instruction 0x12:0xaca4 writes to phys([0x10f73e]):2
instruction 0x12:0xaca8 reads from phys([0x10f73e]):2
instruction 0x12:0xacb0 [step 1/2] writes to phys([0xfbd6]):2
instruction 0x12:0xacb0 [step 2/2] writes to phys([0xfbd4]):2
instruction 0x12:0xacb6 reads from phys([0x10f73e]):2
```

For some instructions, aiT is unsure of what address to access and therefore assumes the slowest possible access. This is usual for indirect accesses, i.e. use of pointers. In some cases aiT knows at least in which memory interval to access and then assumes the slowest possible access within that interval. The uncertain accesses are displayed in two ways, either with an interval or with a question mark.

```
instruction 0x13:0x3766 writes to phys([0x10cfd2..0x10cfe6]):2
instruction 0x13:0x378a reads from phys([?]):2
```

There are annotations to help aiT with these accesses, and many of them could be helped by letting aiT know the values of some registers as they are when entering the function. To specify each access means rather much work though.

At the end of the report file the predicted WCET is shown for the whole task and for each of the routines involved. This might look something like this:

```
Computed Worst-Case Execution Time: 8256 cycles = 0.250 ms

Predicted Worst-Case Execution Time Contribution(s):
_Task_Main_Function: 2559 cycles = 77.546 us
__mul: 3843 cycles = 0.117 ms
_FunctionA: 84 cycles = 2.546 us
_FunctionB: 0
_FunctionC: 0
_FunctionA.L1 (loop): 1764 cycles = 53.455 us
_FunctionB.L1 (loop): 0
_FunctionB.L2 (loop): 0
```

In this example, *FunctionB* and *FunctionC* are not parts of the WCET path, hence their WCET contributions are 0 cycles.

3.3.4 aiSee

AiSee is the graph viewer from AbsInt. After all steps of the analysis are done, aiSee draws the combined call graph and flow graph. When opened, only the call graph is visible. Each function is displayed as a node in the graph, and all loops are also presented as nodes. Red lines between nodes indicate the WCET path. Above the graph, the total predicted WCET is displayed.

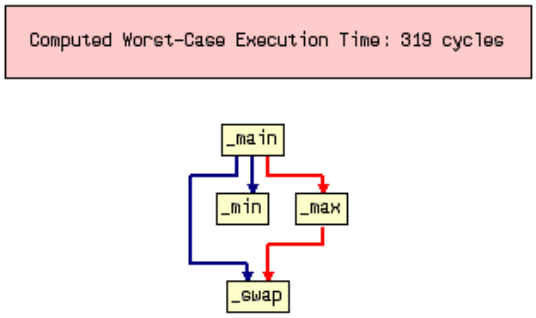


Figure 3.4: A simple call graph.

For every routine, there is a flow graph describing the execution flow through it. When unfolding a routine, the flow graph will be visible. The base blocks[7] in the routine will be represented as nodes in the flow graph. If chosen in the aiT visualization settings, the source code will be displayed in these nodes. This setting is recommended to turn on, otherwise it might be tough to understand the flow graphs.

Below every basic block is a white box displaying two numbers. The first number is the maximum number of times this path has been visited in one context. Except for in loops, these numbers are 1 or 0 depending on if that path is part of the WCET path or not. The second number represents the maximum number of CPU cycles used for one context.

If a base block is unfolded, it will display every assembler instruction that represent that base block. This is the lowest level graph in aiSee.

There are a number of features in this program. The list below mentions some of the most used ones during this project. The quick commands for them are displayed in the parentheses.

- zoom in (+)
- zoom out (-)
- maximum zoom to fit in window (m)
- redraw graph (g)
- fold routine or instruction (f)
- unfold routine or instruction (u)
- unfold routine or instruction into a box (x)
- unfold routine to the lowest level (y)
- unfold entire graph to the lowest level (z)

[7] A base block is a snippet of program code that always executes as whole (i.e. a snippet without branches or calls).

- second info field (j)
- follow edge (e)

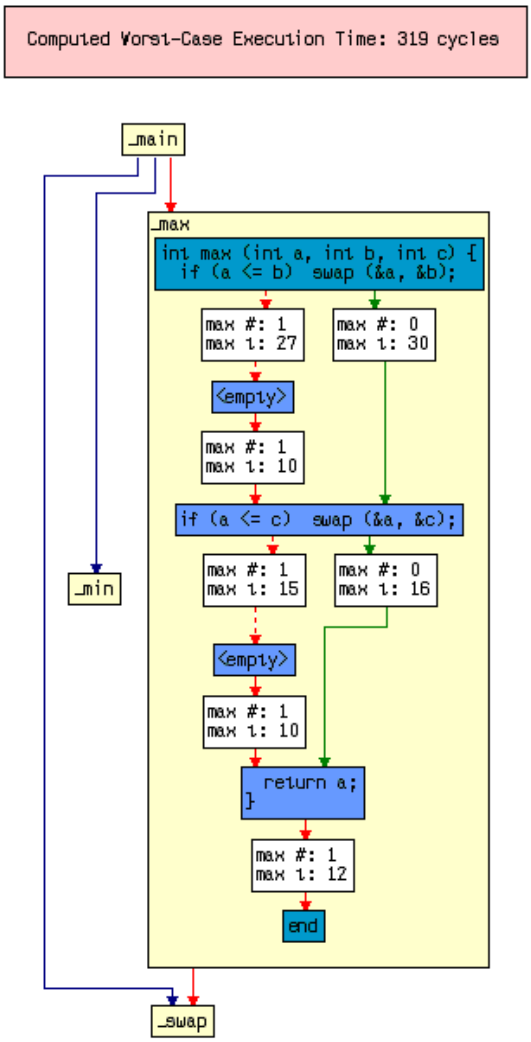


Figure 3.5: One routine unfolded to display a flow graph.

To view the WCET contribution of one specific routine, the routine can be selected and j can be pressed on the keyboard. This is the quick command for the second information field, which pops up as a layer on top of the routine. It displays the predicted WCET contribution and a list of the contexts for which this routine is analyzed. When used on instructions, this info field displays the addresses to the instructions. There are also the first and the third information fields available, but those have not been used at all during this thesis and are therefore not described here.

3.4 Trace32Fire

Trace32Fire is an emulator from Lauterbach Datentechnik GmbH [21]. 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 different levels. We have been looking mostly at level one and three. The highest level displays only the source code while the third level displays source code, assembler instructions and register values.

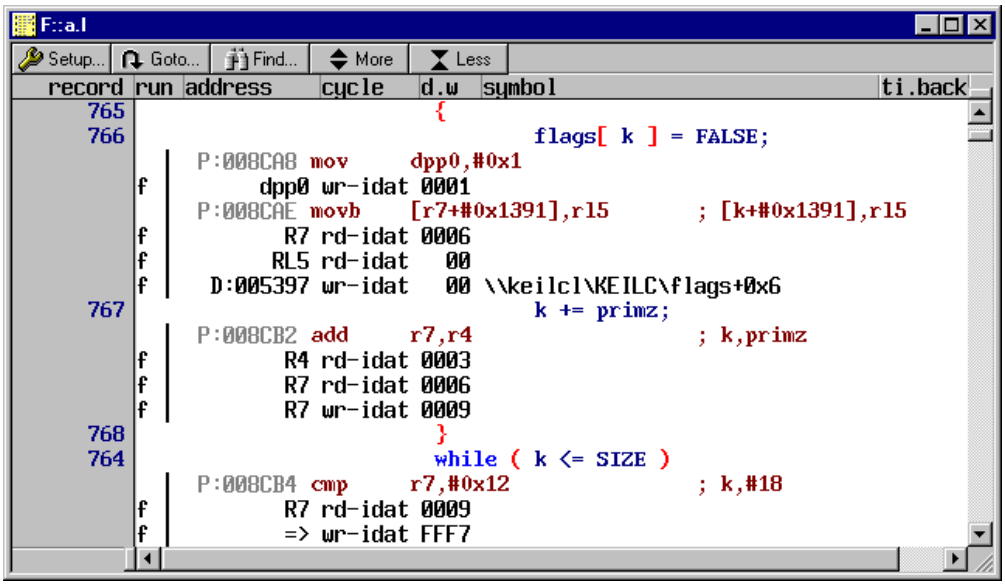


Figure 3.6: The trace window, unfolded to the third level.

A trace can also be viewed as a graph with each function name on one axis and the running time on the other axis. By selecting two points on the graph, the time difference between them is shown. Interrupts had been disabled, so they did not cause a problem.

The first 12 tasks of this project were measured on the real hardware using the Rubus clock, but since we could not trace the measured run we decided to use the emulator instead. All these tasks were remeasured in the emulator, and the differences between the old and new times were between 3.4 and 3.6 microseconds. We were able to establish this as the timing overhead in Rubus, and that the times from the emulator are accurate.

Chapter 4

Analysis Results

In this chapter we will present and analyze the results from our aiT runs and measurements. Section 4.1 describes how much user input that was needed, and Section 4.2 investigates how well suited the code which had been selected was for static WCET analysis. In Section 4.3 we make a comparison between the initial aiT result, the tightened aiT result, the measured result and the WCET set in Rubus.

When terms like 'analysis values', 'analysis times', or 'analysis results' are mentioned they refer to the WCET derived from aiT. Terms like 'measures', 'measurement values' etc. refer to WCET obtained from the CPU emulator. When talking about large and small tasks, we refer to their aiT WCET.

Some abbreviations will be introduced for this chapter:

- WCET written in Rubus (Rub)
- The first aiT WCET, with as few annotations as possible (aiT1)
- The last aiT WCET, with as many annotations as possible (aiT2)
- Time measured in the emulator, with the same execution path as aiT2 (M)

4.1 User Interaction

AiT needed some inputs before we could get started, such as the CPU used, XRAM settings, buscon settings and register settings. The XRAM settings and the buscon settings was found in the initiation file for the system. The values for the context pointer (CP), system stack pointer (SP), user stack pointer (R0) and the four data pointers (DPP0-DPP3) might be different for each task. To find these, we ran the project in the hardware environment and stopped at breakpoints on the first row for every task that were going to be analyzed. When the debugger had stopped, we could view the values of these registers. To make sure they were constant for each task, we repeated this action approximately ten times and watched the values every time.

Out of thirteen analyzed tasks, only three needed annotations before we could get the first results. Finding the information for these annotations did not

cause much work, but on the other hand, the WCETs presented were in many cases just theoretical values that would not be possible in the real environment. Table 4.1 shows the number of annotations used. What we call the *necessary annotations* are loop bounds that aiT did not manage to find automatically. The so called *possible annotations* are used to exclude infeasible paths from the analysis. Much more time was needed to find these.

Task	Loops	Necessary annotations	Possible annotations
1	0	0	2
2	0	0	1
3	0	0	1
4	0	0	3
5	0	0	6
6	1	1	2
7	1	1	6
8	0	0	5
9	0	0	4
10	0	0	8
11	0	0	4
12	0	0	18
13	18	11	103

Table 4.1: The number of annotations written for each task.

After the annotations have been given, there was still a possibility to give more annotations. There were still some uncertain memory accesses for almost all of the tasks, but finding and annotating them would be very time consuming.

4.2 Complexity

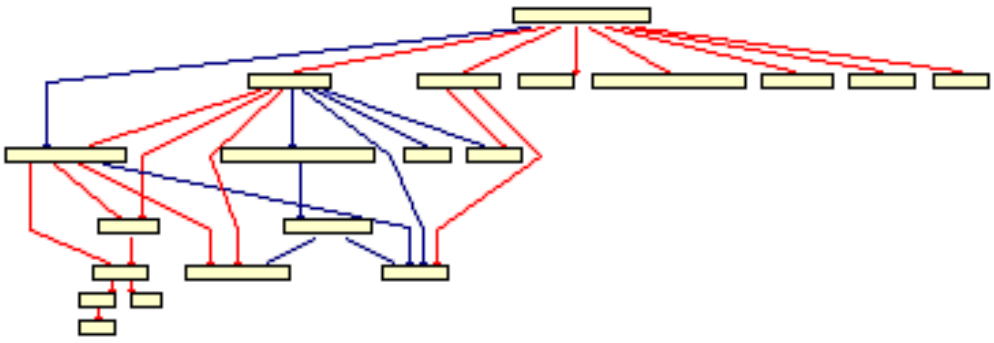


Figure 4.1: The 20-node call graph for task 12.

Different tasks have different source code sizes, contain different number of functions and have different call depths. These are some ways of measuring

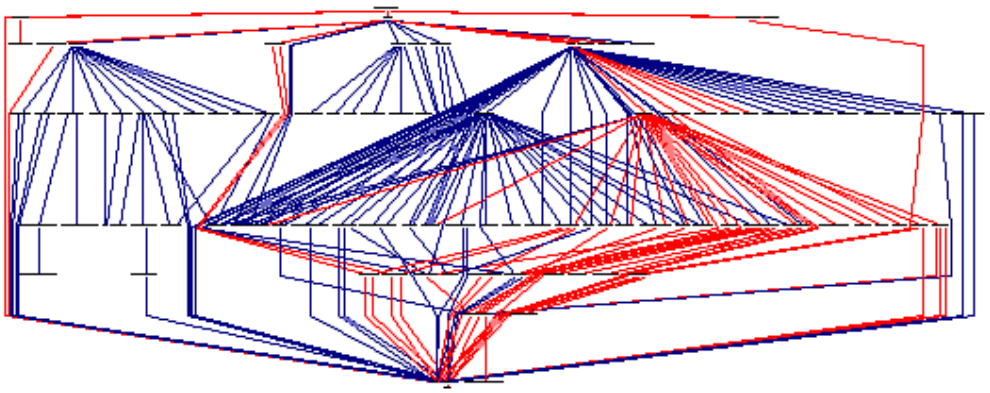


Figure 4.2: The 115-node call graph for task 13.

complexity. The code size is the easiest to look up, but should also be considered a rather bad measure of complexity since it can contain many comments that do not affect the complexity at all or just very long symbol names. This could be called the practical size of the task. To find a logical size using the source code, we should count something that tell more about how much executable code there is. One way would be to count everything but comment rows and blank rows. What we get then should be the number of executable rows. Another way can be to count the semicolons, since each statement in C ends with such.

By looking at the aiSee call graphs, we see that it is possible to count the number of nodes[1] and the call depth. This is a different complexity measure.

Table 4.2 shows a number of complexity measures for the tasks. The tasks are ordered according to their WCET, so what is interesting here is to see if there are any patterns in these complexity numbers.

Except for 4-5 tasks, the size of the source code, the number of executable rows or the number of semicolons follow the same pattern as the WCET. Counting the number of nodes in the aiSee graph, there are three tasks that do not match the pattern, and there is only one task that differs when counting the call levels. However, the number of nodes or number of call levels do not show much indication of an incremention, so these could not be used to tell which tasks have higher WCETs than others, except for the last two tasks that are much larger in every aspect than the others.

We also looked up task sizes in the .map-file created by the compiler, but we only found them for eleven of the thirteen tasks. The sizes found followed the pattern of the source code sizes, and therefore they would not contribute any more information. It was decided to leave them out.

In the big picture, all these complexity measures are some kind of indicator for the WCET, but neither of them can be used to approximate the execution time nor to compare WCET between tasks of fairly equal size.

Table 4.3 is sorted by the difference between aiT1 and aiT2, and we can directly verify that there is no pattern at all between the complexity of the

[1]Nodes in aiSee represent all functions and loops involved in the analyzed segment.

Task	WCET (aiT2)	Size of source code	Exec. Rows	Semicolons	Nodes in graph	Call levels
1	4788	3,6 kb	55	11	1	1
2	5879	4,5 kb	56	17	1	1
3	12122	4,7 kb	58	21	3	2
4	13394	5,2 kb	72	18	1	1
5	22364	6,3 kb	86	35	2	2
6	27243	4,9 kb	43	11	6	3
7	29061	9,3 kb	123	70	5	3
8	30455	8,2 kb	119	55	5	3
9	36000	5,5 kb	49	23	5	3
10	55091	10,9 kb	195	83	3	3
11	70273	8,8 kb	188	73	5	3
12	143606	40,7 kb	707	360	20	5
13	1447000	565,0 kb	12765	4143	115	8

Table 4.2: WCET and complexity measures.

tasks and how much we were able to tighten the WCET estimate from the first aiT result.

The time used by aiT to analyze the tasks was for all but one task between 2 and 6 minutes, but for the largest task the analysis times varied between 15 and 100 minutes depending on which annotations that were used. Apparently, and not unexpected, flow-annotations increased the analysis time while condition-annotations decreased the time.

4.3 Code Properties

The source code in this project is written in C. For the thirteen tasks analyzed, the code structure is very simple 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, and according to aiT they take less execution time than nested if-statements. The complexity of the larger tasks is caused only by a large number of calls, and some functions were used many times in many different contexts.

More paths can be excluded by applying more else-statements instead of lining up lots of single if-statements that contradict each other. Since the biggest overestimation by aiT is the inclusion of infeasible paths, else-statements would decrease the WCET estimate from the analysis without having to use annotations. To make this point clearer, consider the following C code segment:

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

Task	$\frac{\text{aiT1}}{\text{aiT2}}$	Size of source code	Exec. Rows	Semicolons	Nodes in graph	Call levels
4	1,44	5,2 kb	72	18	1	1
12	1,39	40,7 kb	707	360	20	5
10	1,34	10,9 kb	195	83	3	3
8	1,29	8,2 kb	119	55	5	3
2	1,28	4,5 kb	56	17	1	1
13	1,26	565,0 kb	12765	4143	115	8
1	1,19	3,6 kb	55	11	1	1
7	1,19	9,3 kb	123	70	5	3
9	1,17	5,5 kb	49	23	5	3
5	1,13	6,3 kb	86	35	2	2
11	1,12	8,8 kb	188	73	5	3
3	1,11	4,7 kb	58	21	3	2
6	1,05	4,9 kb	43	11	6	3

Table 4.3: Level of tightening and complexity measures.

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

AiT would consider both of these expressions to be true when calculating the WCET, something that any programmer can see is impossible. The variable x can not have the value 1 and 0 at the same time. If the code would have been written as below, aiT would have been forced to choose only one of the expressions to be true.

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

This is a recurring cause of overestimations, but rarely found in this simple form.

4.4 WCET Comparisons

In the beginning of the project, the real hardware was used to measure the tasks. The problem with this was that it did not show what path that was measured and that it was some overhead from starting and stopping the timer. When the emulator was put into work, everything was much simpler. The path was logged and the overhead could be left out of the measurement. Therefore, the only measured times mentioned in this report are the ones from the emulator, and comparing these to early measurements done on the real hardware we could establish the fact that they are accurate. The overhead for the timers is approximately 3.5 microseconds, which for the shortest task was as much as 85% of its measured WCET. For the longest task, the overhead was only 0,32%.

As mentioned in the introduction, the sources for the Rubus WCET estimates are measurements done while running automatic testing plus a safety margin. The size of the margin can vary, but is roughly based on the complexity of the tasks. Since the Transmission ECU was upgraded with a new CPU, there are many tasks that have not been remeasured. This can be the answer to why some of the WCETs are greatly overestimated. Volvo does unfortunately not have any documentation of which tasks have been remeasured since the CPU upgrade.

4.4.1 A Look at the Numbers

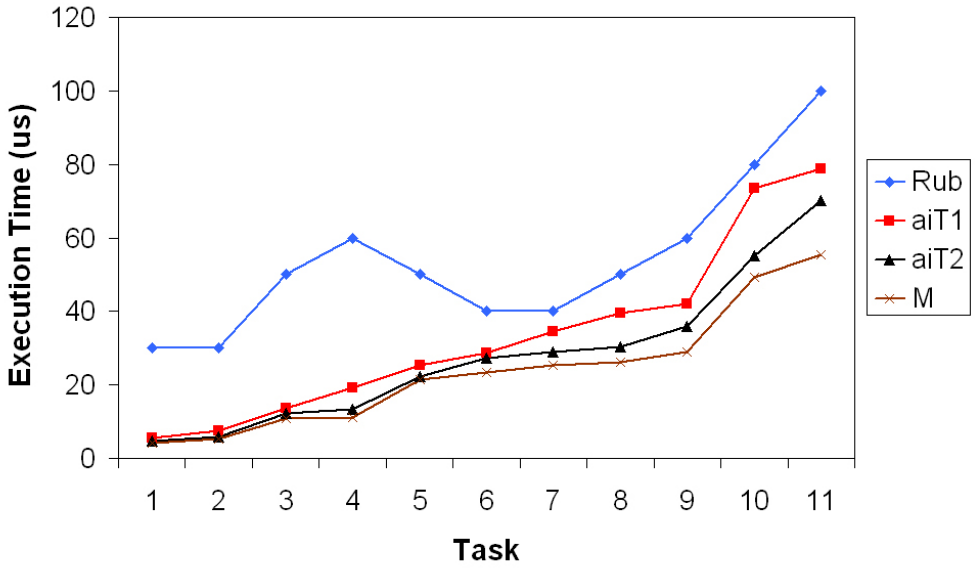


Figure 4.3: The first eleven tasks.

From the complexity section, we know that the last two tasks in this project are much more complex than the others. We therefore begin by looking at a graph showing only the eleven first tasks (Figure 4.3), consisting of four lines. We revisit the abbreviations introduced in this chapter when saying that these lines represent, from top to bottom:

- WCET given to Rubus (Rub)
- The first aiT WCET, with as few annotations as possible (aiT1)
- The last aiT WCET, with as many annotations as possible (aiT2)
- Time measured in the emulator with the same execution path as aiT2 (M)

This shows that both Rub and aiT1 are safe. On the other hand, when looking at this graph, the Rubus values seem to be unnecessarily pessimistic,

especially for the shorter tasks. The most probable cause to this is the CPU upgrade. We also see that there is an unexpected gap between aiT2 and M. Since these two have the same execution path, it was expected that they would show almost identical times. The source to this overestimation by aiT is probably most due to uncertain memory accesses. As mentioned earlier, it takes very long time to annotate every memory access and this is why we chose to only annotate the values of the two most influencial register pointers.

Since aiT makes overapproximations, it might be the case that the tool finds a WCET path that is very close to, but not the actual WCET path. The path found by aiT can be the longest due to the overapproximations. If this would be the case, we can not guarantee that M represent the actual WCET. The aiT2 value is a little larger than M and the margin should be enough to call aiT2 a safe WCET.

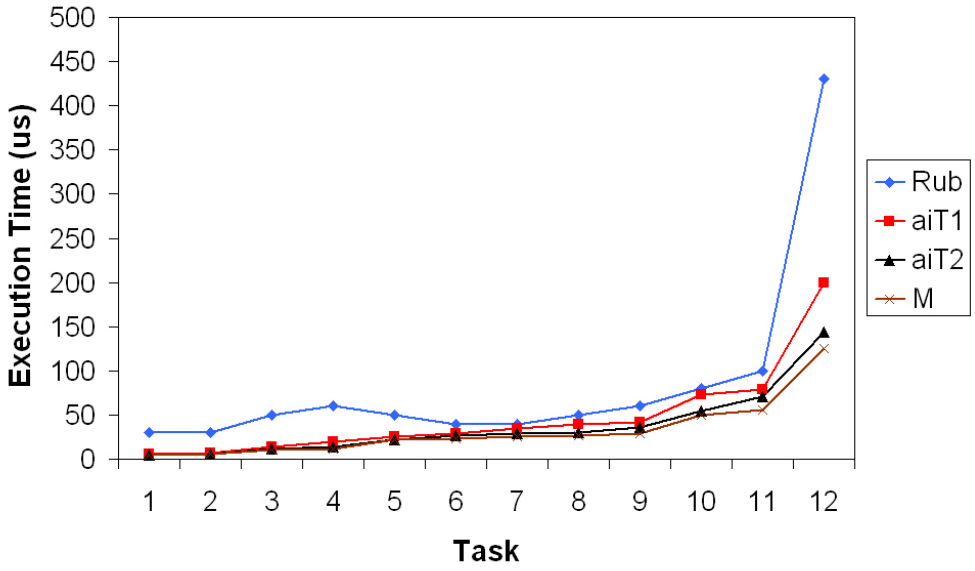


Figure 4.4: The first twelve tasks.

We now take a look at figure 4.4, which is the same graph as above but with one more task added. The reason we excluded this task from the previous graph is that it is so much larger than the others that we would get a poor resolution of the graph.

We can draw the same conclusions as above, but add that Rub for task 12 is greatly overestimated. Although it does not have as high percentage of overestimation as tasks 1-4 it definitely books more of the total period time.

When adding the largest task to the graph (Figure 4.5), we directly see that this task is almost ten times as large as task 12. This is the most complex task in the articulated haulers today.

As we can see here, we get some different results than earlier. The Rubus line is crossed, and that means that the Rubus WCET might be unsafe. However, the

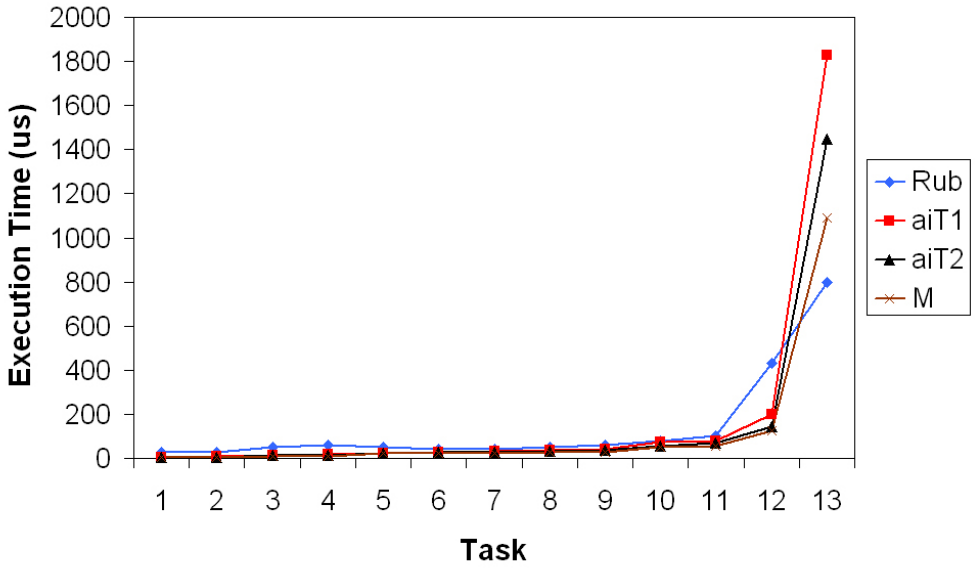


Figure 4.5: All thirteen tasks.

path found might still be practically infeasible depending on the values of the input arguments. When the tasks were run in the debugger, every argument was set manually. In the real environment, this combination of inputs might never happen. Nevertheless, it should be noted that there exists a path that indicates a large underestimation in Rubus.

4.4.2 Another Way of Comparing

The graph in figure 4.6 is different from the others since it shows M, aiT1 and Rub in relation to aiT2. The lines represent, from top to bottom:

1. Rub / aiT2
2. aiT1 / aiT2
3. M / aiT2

We can not draw any new conclusions from this graph since it is just another way of looking at the data presented by the other graphs, but it might be easier to see some properties. We can easily see that Rub is more than twice needed for tasks 1-5 and task 12, and that it underestimates the last task. We also see that M always is less than aiT2 and that aiT1 always is larger. The main point of this graph is to get a better look at the sizes of overestimations and the accuracy of aiT. Causes for the large overestimations of tasks 1-5 are probably the CPU upgrade and that interrupts can affect the execution times of the smaller tasks to a higher percentage than for the larger.

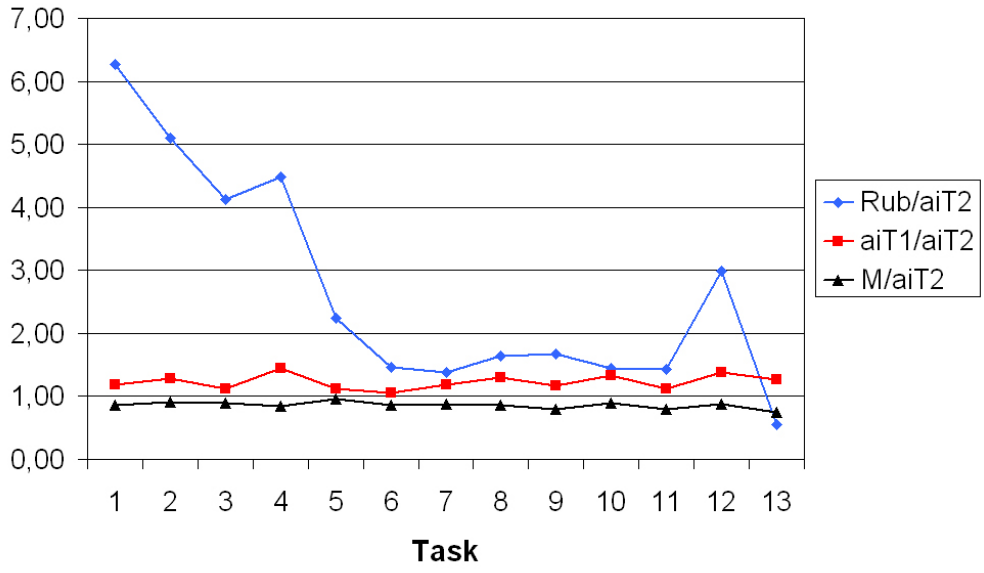


Figure 4.6: Rub, aiT1 and M in relation to aiT2.

Task	Rub	aiT1	aiT2	M	$\frac{\text{Rub}}{\text{aiT2}}$	$\frac{\text{aiT1}}{\text{aiT2}}$	$\frac{\text{M}}{\text{aiT2}}$
1	30	5,697	4,788	4,138	6,27	1,19	0,86
2	30	7,516	5,879	5,34	5,10	1,28	0,91
3	50	13,516	12,122	10,84	4,12	1,11	0,89
4	60	19,304	13,394	11,24	4,48	1,44	0,84
5	50	25,243	22,364	21,56	2,24	1,13	0,96
6	40	28,637	27,243	23,48	1,47	1,05	0,86
7	40	34,546	29,061	25,4	1,38	1,19	0,87
8	50	39,425	30,455	26,235	1,64	1,29	0,86
9	60	41,94	36	28,88	1,67	1,17	0,80
10	80	73,576	55,091	49,335	1,45	1,34	0,90
11	100	78,758	70,273	55,418	1,42	1,12	0,79
12	430	199	143,606	124,8	2,99	1,39	0,87
13	800	1827	1447	1090	0,55	1,26	0,75
AVG	140	184,17	145,94	113,59	2,68	1,23	0,86
MIN	30	5,70	4,79	4,14	0,55	1,05	0,75
MAX	800	1827,00	1447,00	1090,00	6,27	1,44	0,96

Table 4.4: Data derived from Rubus, aiT and measurements. Source data for the graphs in this chapter. Times in microseconds.

4.5 Work Time

Since every WCET estimate above aiT2 is safe, it would definitely be a tightening of the times if we applied aiT1 in Rubus instead of the values used today. For the last task we would get a safer WCET estimate. Of course, it would be even better if we could apply the values in aiT2, or even a value close above M. The question is if the work to find these values is worth the improvement. Well, that depends on how full the schedule is, i.e. how tight values we need. The labor needed to get aiT1 down to aiT2 varied from task to task, but the two largest tasks took together around seven weeks. The easiest tasks were annotated and solved in about an hour each. Since the input arguments affect the path and vice versa, it is not possible to divide the work time into the time used for annotating and the time used to solve the input arguments. There are many things that need to be considered when manually cutting paths. These are a few:

- If two paths contradict each other, which one is the longest?
- If path A assigns a value to a variable that controls path B, is it worthwhile to enter path A in order to be able to enter path B? What is the tradeoff?
- What variables are changed in each path, and how do they control the following execution of the task?

If aiT would be able to consider how assignments of variables affect the flow further along the task, we would get a much better WCET estimate directly and we would not need to put so much labor into manually steering aiT. We hope to see a WCET tool in the near future that both calculates execution times and finds a feasible flow depending on the variables controlling the if-statements (i.e. a combination of good flow analysis and good calculation methods). The flow analysis in aiT today does not do much to exclude infeasible paths.

We had never used aiT before this project, and can now establish the fact that it can take about a week to get used to the program. The first analysis can perhaps be run within an hour, but in order to annotate correctly and to produce tighter results, more work time is needed. It is our intention that this report should be of help when getting started. After having read this report, read the aiT manual, and spent a week with aiT, any user should be able to get a decent analysis result for almost any of the tasks in this project within a day. To rerun an analysis after a code revision or a CPU upgrade could be done in an hour if relative addressing have been used in the annotations and if the code revision is fairly small. The user should always save the project files and the annotation files so that all work does not have to be redone in case of another analysis.

4.6 Batch Jobs

Since it can take a number of minutes to run an analysis, it might be good if we are able to run a batch of tasks in sequence without having to manually

start the analysis of every task. If the project files are created beforehand, they can all be run from a script. The script might be an executable file entitled batch.bat that includes an arbitrary number of lines like these:

```
ait c166 -b -z -m task1.apf
ait c166 -b -z -m task2.apf
```

The following flags can be used in the commands:

- -b: batch mode - starts the analysis directly
- -x: terminates aiT if the analysis is successful
- -z: terminates aiT whether the analysis is successful or not
- -m: aiT starts with the window minimized

When running this test, we used an older version of aiT than the version that exists today. A slight problem with batch jobs in that version was that the graphs were discarded upon termination of aiT, so only the report files was saved. If we wanted to save the graphs, we should not have used the flags -x or -z, and we would have still needed to sit in front of the computer to save the graphs and to terminate aiT before the next analysis could start, and this contradicted the purpose of a batch job. A newer version of aiT was however released during the final weeks of this project, where we could choose to save the graph.

We tried to run a batch job for the 24 first red tasks alphabetically. 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 about 70 minutes.

Many of the tasks contained unresolved loops and one task even contained a memory access that aiT classified as erroneous. The table below shows the results of the analyses. The unbounded tasks may be rather easy to annotate the loop bounds for, but we will not do that now because the test with this batch was to see how we could get many results in very short time.

A much better use of the batch job is to run it on tasks that have been analyzed before, i.e. when all tasks are to be reanalyzed after code revisions or a CPU upgrade. Remember that annotations might need to be updated if the project have been rebuilt. Annotated register values or memory accesses might have changed for all tasks.

Task	Rub	aiT1	$\frac{\text{Rub}}{\text{aiT1}}$
A	1000000	unbounded	
B	1400000	unbounded	
C	50000	11728	4,26
D	200000	15304	13,07
E	100000	23667	4,23
F	60000	19304	3,11
G	30000	19607	1,53
H	100000	unbounded	
I	40000	39819	1
J	50000	23910	2,09
K	100000	39788	2,51
L	200000	13455	14,86
M	30000	unbounded	
N	100000	unbounded	
O	200000	memory error	
P	50000	unbounded	
Q	100000	3182	31,43
R	25000	10697	2,34
S	50000	30849	1,62
T	60000	44273	1,36
U	50000	21637	2,31
V	50000	32667	1,53
X	70000	28607	2,45
Y	30000	3879	7,73

Table 4.5: The results from the batch job.

Chapter 5

Conclusions

AiT would be of good use for Volvo if there is need to make more room in the red task schedule, but the tool still has a long way to go before it is perfect. The measurement results were at average as low as 86% of the aiT results produced from the same path.

Even though aiT overestimates the WCET because inclusion of infeasible paths in the analysis, the values derived with minimal number of annotations are much tighter than the values used today. Since the work for tightening the aiT results further is quite time consuming, it might not be worth it. AiT1 were at average 23% larger than aiT2. The Rubus overestimations compared to aiT's varies between 527% and 42%[1] with an average of 168%. Future versions of aiT will most probably give tighter results with less user interaction. This thesis is giving a good indication that the aiT value is safe, and no measurement should be needed if aiT is used in future work. If the tool is used in this way, it does not consume much time and gives a decent result.

Three pros of using aiT with only the mandatory annotations:

- Calculates fast
- Guaranteed safe WCETs
- Tighter WCETs than used today

It is quite hard to find some really strong arguments against using aiT, but the licence fee and the time needed to get started with the program are of course two negative aspects. Another one is that the overestimation increases with the complexity of the tasks, so aiT may give very high results for complex tasks.

Today, the code analyzed is well suited for static WCET analysis but could be even better. If-statements that contradict each other should be written as if-else-statements when possible. In the case that Volvo decides to include aiT in their development process, annotations could be written in the source code. Since it can be hard to predict which branches that not are parts of the WCET path, the only annotations suited for source code should be loop bounds.

To use batch jobs is a good idea if many tasks, or perhaps the whole system, is to be analyzed. Batch script, project files and annotation files should be saved

[1] Except for the last task that had an underestimation of 45%

so that they can be reused. Some memory pointers might be changing every time the project is built, so they should be checked in the debug environment before running analyses on a new binary.

Chapter 6

Future Work

Since we only have analyzed thirteen tasks out of hundreds, analyses for the others could be done. We saw by the batch job test that we can get results quickly if we are satisfied with the first aiT value.

The main focus for AbsInt, and other WCET researchers as well, should lie on getting the value analysis to exclude more infeasible paths, even between routines. Since the automatic loop bound detection only manages to find the very same bounds that can be found manually, it is not of much use except for speeding up the process a little. It is probably a long way before the difficult loop bounds can be found, those that even are hard to find manually. The flow-annotations in aiT should also be able to specify program points that are not executed together.

Bibliography

[1] AbsInt Angewandte Informatik GmbH company website. URL: http://www.absint.com

[2] Arcticus Systems company website. URL: http://www.arcticus-systems.com

[3] Mälardalen University website. URL: http://www.mdh.se

[4] Volvo Construction Equipment company website. URL: http://www.volvoce.com

[5] Tidorum Ltd company website. URL: http://www.tidorum.fi

[6] Heptane WCET Tool website.
URL: http://www.irisa.fr/aces/work/heptane-demo/heptane.html

[7] A. Ermedahl: *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* PhD dissertation, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden (2003). ISBN 91-554-5671-5

[8] N. Bermudo, J. Gustafsson, B. Lisper, C. Sandberg: *A Tool for Automatic Flow Analysis of C-programs for WCET Calculation.* In 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)(2003).

[9] A. Ermedahl, J. Gustafsson, B. Lisper: *Towards a Flow Analysis for Embedded System C Programs.* In 8th IEEE International Workshop on Object-oriented Realtime Dependable Systems (WORDS 2005)(Feb 2005).

[10] K. Hänninen, T. Riutta: *Optimal Design.* Master's Thesis. Dept. of Computer Science and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden (February 2003).

[11] D. Sandell: *Evaluating Static Worst-Case Execution-Time Analysis for a Commercial Real-Time Operating System.* Master's Thesis. Dept. of Computer Science and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden (2004). Also as technical report [12].

[12] D. Sandell, A. Ermedahl, J. Gustafsson, B. Lisper: *Static Timing Analysis of Real-Time Operating System Code.* In Proc 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA´04)(Oct 2004). Based on Master's Thesis [11].

[13] S. Byhlin: *Evaluation of Static Time Analysis for Volcano Communications Technologies AB.* Master's Thesis. Dept. of Computer Science and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden (2004). Also as technical report [14].

[14] S. Byhlin, A. Ermedahl, J. Gustafsson, B. Lisper: *Applying Static WCET Analysis to Automotive Communication Software.* Euromicro Conference of Real-Time Systems (ECRTS'05)(July 2005). Based on Master's Thesis [13].

[15] CC-Systems company website. URL: http://www.cc-systems.com

[16] O. Eriksson: *Evaluation of Static Time Analysis for CC Systems.* Master's Thesis. Dept. of Computer Science and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden (June 2005).

[17] Y. Zhang: *Evaluation of Methods for Dynamic Time Analysis for CC Systems AB.* Master's Thesis. Dept. of Computer Science and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden (June 2005).

[18] S. Thesing: *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD dissertation, Universität des Saarlandes, Saarbrücken Germany (2004). ISBN 3-937436-00-6

[19] Infineon company website. URL: http://www.infineon.com

[20] V.P. Heuring, H.F. Jordan: *Computer Systems Design and Architecture.* Prentice Hall, Upper Saddle River, NJ 07458 (1997). ISBN 0-8053-4330-X.

[21] Lauterbach Datentechnik GmbH company website. URL: http://www.lauterbach.com