Evaluation of Static Time Analysis for CC Systems

Ola Eriksson
oen99001@student.mdh.se
2005-08-13

Department of Computer Science and Electronics
Mälardalen University, Västerås
And
CC Systems, Västerås

Supervisor at Mälardalen University: Andreas Ermedahl
Supervisor at CC Systems: Mattias Lång
Contact at CC Systems: Stefan Rönning
Examiner at Mälardalen University: Björn Lisper

Abstract

Most processors today are embedded in products like mobile phones, microwave owens, welding machines etc and are not used in PC's as many believe. Since some of these embedded computers are used in time-critical or safety-critical systems it is very important that the behaviour of these systems are well known. One part of that is to know the Worst Case Execution Time (WCET) of the different tasks in the embedded system. The traditional method of finding the WCET of a task is by measuring the execution time of the task when it is running on the target system. There are several problems with this approach. It is hard to check that the time you have got is actually the WCET.

Today there is another possibility to find the WCET. You can use static timing analysis that does not execute the program in order to find the WCET; instead it uses a model of the target hardware and uses static methods to calculate the WCET. There are a few commercial static WCET tools on the market and a few more academic ones, but so far no industry has started to use these on a regular basis.

This work was done at CC-Systems (CCS) in Västerås [1]. CCS develops and delivers electronic solutions and software for machines and vehicles in tough environments. This means that some of their code is time-critical. The purpose of this thesis work was to find out if it is possible to integrate the static WCET tool in CCS development tool chain and also to see if CCS would benefit from it. If it was possible to do so they also wanted to know if they must change their development process or make other changes to make it easier to analyse their code by a static tool. CCS was also interested to know if the static tool could be of use in other areas than WCET analysis, for example giving their simulation technique a correct timing behaviour.

To be able to compare the results of the static WCET analysis considering effort and accuracy, dynamic WCET analysis was also performed on the same code snippets as with the static tool. The dynamic measurements was a part of another thesis [2] on CCS performed by Yina Zhang. The purpose of her thesis was to evaluate different dynamic WCET analysis methods and to see if any of the methods is well suited for CCS. She also evaluated what CCS could do to make it easier to analyse their code with the selected WCET analysis method and if they could use the WCET analysis methods for other things than just WCET analysis. One part of her thesis was also to evaluate the currently used methods of finding WCET values on CCS.

The overall conclusion of this thesis is that to be able to get both tight and safe WCET values static methods should preferably be used. Dynamic methods can however help the static method giving tighter WCET values since they can be used to give input to the static analysis, for example give information about memory accesses. If general timing behaviour of the system is wanted dynamic methods are prefered since today's static methods can't provide this kind of information.

Keywords: WCET, Static Timing Analysis, Dynamic Timing Analysis, Method Comparison

Acknowledgement

The work of this thesis was entirely made at CC-Systems in Västerås during the time January 2005 to June 2005. The work is the final part to receive a Master of Science in Computer Engineering at Mälardalen University.

I want to start off by thanking my supervisor at the University, Andreas Ermedahl, for first of all choosing me to do this thesis and also for all the help and advice during the time I worked on this thesis. My supervisor at CC-Systems, Mattias Lång, also deserves a big thank you. Despite the fact that he has been away from the office in Västerås a lot of the time I couldn't have made this thesis without his help. Stefan Rönning, another employee at CC-Systems, has also been a great support.

The employees at AbsInt Angewandte Informatik GmbH also deserves my gratitude for letting me use their tool for static WCET analysis (aiT) and also for giving me a course in how to use it correctly that saved me a lot of time and effort. I especially want to thank Martin Sicks at AbsInt for answering all my tricky questions about aiT and for giving the course about aiT together with Florian Martin. I want to thank ARTIST2 [27] for providing the financial means of the course.

Another person I have to thank is Yina Zhang who also did her thesis at CC-System at the same time as me. Her thesis was about dynamic timing analysis. Therefore making it possible to compare static methods with dynamic ones. We could also help each other since some things were easier done with dynamic methods than static ones and vice versa. The biggest advantage of doing the thesis at the same time as Yina was that you always had someone that you could discuss things with and that helped a lot.

Finally I want to thank my friend Samuel Bjurhager for telling me that CC-Systems and Mälardalen University was going to cooperate and that it therefore was an opportunity to make a MSc thesis work at CC-Systems. It was this information that led me to contact Andreas Ermedahl and gave me the chance to write my thesis at CC-Systems.

This research has been supported by the Advanced Software Technology Center (ASTEC) in Uppsala [3]. ASTEC is a Vinnova [4] (Swedish Agency for Innovation Systems) initiative.

Västerås
2005-08-13
Ola Eriksson

1 Introduction

1.1 Background

Today most things contain some kind of embedded computer. They exist in cell-phones, microwave owens, cars, toys and a lot of other electronic products.

Many things formerly controlled entirely by hardware are today controlled by software in different kinds of distributed real-time systems. These kind of distributed real-time systems have also replaced old mechanical solutions with for example fly-by-wire, drive-by-wire and so on. A real-time system is a system with deadlines, i.e. everything has to be done before a specific time. Sometimes it is also important not to be done too early. If you take an airbag for example, it is equally important not to inflate it too soon compared to inflating it too late. A distributed system is a system consisting of several nodes working together performing a task and communicating through some kind of network.

Real-time systems have to be predictable in the sense that you know the maximum time for something to be done and sometimes you also have to know the minimum time. This makes it necessary to know the execution time of different parts of the software in the real-time system. It is especially important to know the Worst Case Execution Time (WCET) since this can have big effects on the rest of the system. The problem is to find the WCET for different code snippets. The most commonly used way to find the WCET is by using dynamic methods, i.e. making measurements on the code executing on the target. An easy way of doing this is by using built in time-function to measure execution time, but it requires an operating system (OS) and small embedded systems usually aren't equipped with an OS. Then you have to rely on different kind of hardware for making these kinds of measurements like for example oscilloscopes, logic analysers and emulators. If the system have built-in timers they can be used instead of hardware and for some processors there are also cycle-accurate simulators available that can be used to obtain executions times. It is often very hard to guarantee that the path generating the WCET actually has been found. It can also be very hard to connect all parts needed to make the measurements and simulate a correct behaviour. You are often forced to change the code to be able to make measurements and this can affect the system in different ways (known as probe effect).

An alternative to dynamic timing analysis for obtaining the WCET is static timing analysis. Static timing analysis uses a model of the target hardware and the program that runs on the target and static methods to estimate WCET for a certain code snippet. There is no need to execute the code when static timing analysis is used since the execution time is calculated and not measured. This can make it easier to estimate the WCET and it also makes it possible to estimate the WCET for systems that aren't finished yet or were the hardware isn't available.

CC-Systems (CCS) [1] is a company that develops and delivers electronic solutions and software for machines and vehicles in tough environments. The company was founded 1991 and have now grown into a company with 130 co-worker situated at offices in Alfta; Sweden, Uppsala; Sweden, Västerås; Sweden, Örnsköldsvik; Sweden and Tammerfors; Finland. Some of their customers are Timberjack, John Deere, BAE Land Systems Hägglunds, ESAB, Atlas Copco, Bromma Conquip and Rolls-Royce Marine. CCS wanted to know if it was possible to

integrate the static WCET tool in their current tool chain to be able to calculate WCET and also to improve their simulating technology.

The code that is analysed is code that is used in welding machines and some component-based code. The company ESAB provides the code used in welding machines. The component-based code is an Adaptive Cruise Control and it is part of a test to automatically generate tasks from different components. The ESAB-code that is tested consists of two interrupts and since the component-based code was much smaller all its tasks and components were analysed. All the tested code was time-critical, however there were no explicit deadlines for the interrupts in the ESAB-code.

1.2 Purpose

The purpose of this thesis was to find out if CC Systems (CCS) can benefit from using static WCET analysis methods in their development process. Static WCET analysis was also compared with dynamic WCET analysis methods to find out which method was best suited for CCS. CCS was also interested in other uses for static WCET analysis. For example if it was possible to get timing of code parts that could make their existing simulation technique more time accurate. If the conclusion of the thesis was that static WCET analysis can be integrated in CCS development process and that it would also be of great use to do so, CCS wanted to find out how they can make it easier to integrate this method. One way of simplifying the integration process could be to structure the source code in certain ways that would make the static WCET analysis much easier.

Several calculations of WCET have been performed on CCS's code in order to find out how accurate and time consuming static WCET analysis is. To have something to compare the WCET values to, dynamic measurements have also been performed on the same code snippets. The measurements have been performed by Yina Zhang [2] who concurrently made her MSc work at CCS, focusing on methods for dynamic timing analysis.

1.3 Worst Case Execution Time

The Worst Case Execution Time (WCET) has many definitions, the basic idea is however that the WCET is the longest time a program will execute during normal operation. One problem can be to determine what normal operation means. In this thesis normal operation means operation without fatal errors such as hardware failure and other failures that would make the system crash. Errors that won't affect the operation of the system are often integrated in the WCET.

The reason why WCET is important to know is that it is a parameter used when the resources needed for the system are calculated. It is also very important to know the WCET for tasks when the system is being scheduled since it represent the execution time needed if the system is going to be guaranteed to be schedulable. In order for a WCET value to be of good use it have to be both safe and tight. Safe means that the WCET value obtained is equal or greater than the actual WCET. Static WCET tools obtain a safe WCET value by always assume the worst case scenario when in doubt. Tight means that the WCET value should be as close to the actual WCET as possible. This can be a problem for static WCET tools since they use overestimations when in doubt. In order to assure good WCET estimations the tools and

models used to calculate WCET have to generate values that are safe and tight [5]. It is however always easier to produce safe values than tight ones.

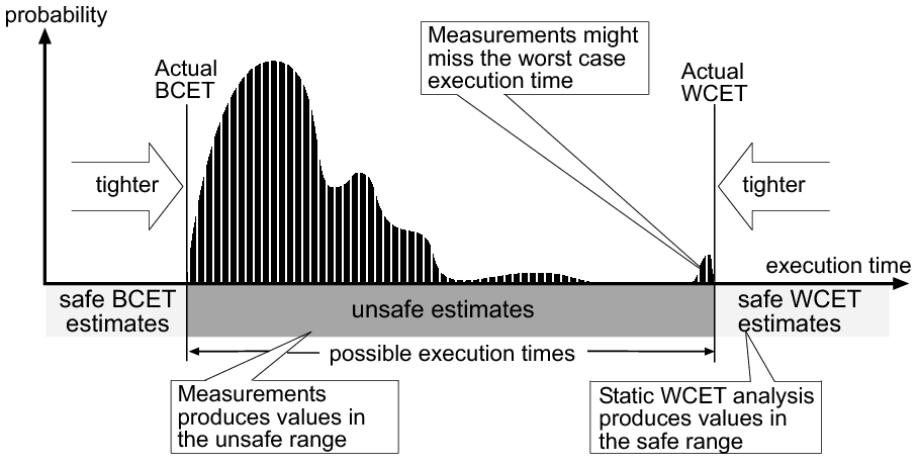

Figure 1.1: Execution time distribution

Figure 1.1 above gives a good illustration why it is hard to find WCET values using measurements. The values that are measured are the ones called possible execution times in the figure. It is often hard to find the WCET by measurements since the probability that it occurs is very low. Static WCET analysis on the other hand guarantees to generate a value that is greater than or equal to the actual WCET value; the problem is that sometimes the overestimation can be quite large. If not only a WCET value but also the execution time distribution is wanted dynamic methods can give such information. A big trouble of finding the WCET using dynamic methods is that a very large set of inputs has to be tried in order to be able to guarantee that the WCET have been found and this is often virtually impossible.

The software behaviour of a program could make the execution time of the system variable but also the hardware upon which the program runs can have that effect. Modern hardware is gets more and more advanced with pipelines, caches, branch prediction and out of order execution. All these features are there to lower the Average-Case Execution Time (ACET), however they often increase the WCET or at least make it hard to find the WCET since the system becomes much more complex. Static WCET analysis uses a hardware model of the target hardware to mimic its timing behaviour. If the target system is very advanced the task of making a model is almost impossible. In order to get a safe WCET value overestimations are often used when the behaviour of advanced hardware is modelled. These overestimations can in turn lead to a less tight but safe WCET value.

1.4 Related Works

This section will give some information about other studies of static WCET analysis tools. For more information about static WCET analysis and the tools available see Section 2.4.

Susanna Byhlin has performed a MSc thesis [6] similar to this one. Her job was performed at the company Volcano Communications Technologies AB (VCT). The purpose of her thesis work was to examine if a static WCET analysis tool could be integrated into VCT's development environment. This would enable VCT to calculate WCET values and also to reduce the development time and therefore also the development cost. The conclusions drawn

by the author is that the static WCET tool used (aiT) had both advantages and drawbacks compared to dynamic WCET analysis. A problem brought up by the author is the fact that VCT uses a lot of different processors and compilers and static WCET tools for these processors and compilers may not be developed in time. Another problem with static WCET analysis according to the author is that static WCET analysis is still an area of research and this can make is hard to assure tight and safe execution times. This thesis is also available as a smaller article [7].

Daniel Sandell also used aiT to calculate WCET values in his MSc thesis [8]. The purpose of his thesis was to find out if a WCET analysis tool could be used to derive WCET estimates on the OSE operating system code. The second purpose was to see what improvements have to be made in order to make WCET analysis tools more applicable to the demands of the industry. The code snippets tested were system calls and disable interrupt regions. The result from the WCET analysis is compared with values from the ARMulator, which is a hardware model of an ARM-processor that can be used to calculate execution times. The conclusions drawn by the author are that the code could be analysed with a WCET analysis tool and that disable interrupt regions were well suited for these type of static WCET analyses. System calls could also be analysed but demanded more work from the user. A summary of this thesis is available as an article [9].

The MSc thesis by Samuel Peterson [10] is about porting a static WCET tool to the Renesas H8/3292 processor. This processor is used in Lego Mindstorms, a kit used for building simple lego robots, and is commonly used in education for real-time courses. The goal of the thesis was to bring the WCET analysis technique into the education by introducing the ported version of the Bound-T WCET analysis tool [11] in real-time system courses at Mälardalen University. The conclusion drawn by the author is that is requires a lot of work to port a static WCET analysis tool to a new host platform. Although this processor didn't have any caches or pipelines it had a quite large instruction set and a lot of addressing modes, which made the implementation non-trivial.

Jakob Engblom, Andreas Ermedahl and Friedhelm Stappert have written a technical report about validating a WCET analysis method for embedded systems [5]. The report is a manual how to verify a WCET analysis method for an embedded processor. The author focuses on the safety of the analysis method but the tightness is also considered. In the report they verify their own WCET analysis method. The WCET analysis is composed of several components and each of these are tested separately. In order to verify the components, code with known WCET values were used to test the WCET analysis method on. To test one component in isolation all other components have to be made constant in some way. This is done by selecting certain input values that produce the WCET and then using them for all analyses. This will lead to a constant execution path, which makes the verification easier. The results they get from the components are compared to the known WCET's and the traces they have gotten from running the code on a PC. The authors claim that after verifying their WCET analysis tool their calculation method is correct and their pipeline analysis is both tight and safe.

A. Colin and I. Puaut have written a report about a static WCET analysis of the RTEMS Real-Time Operating System [12]. The actual analysis has been performed by a student named Philippe Selo. The Heptane WCET analysis tool was used to analyse 12 system calls in the

RTEMS Real-Time Operating System. All the systems calls were dealing with task management and synchronization. The following properties were found in the code of RTEMS: small number of loops and no loop nesting, absence of recursion, small number of dynamic calls and high degree of reuse of functions. The main problem with the analyses was to find correct loop bounds. Only 25% of the loop bounds were trivial to find, the rest demanded deep studies of the source code of RTEMS. The loop bounds found were often very pessimistic since they depended upon the number of task in the system which had an upper bound that was much higher than the number of task that usually were in the system. Most of the dynamic calls could be replaced by static ones to simplify the analysis, but according to the authors it was hard to find the function that was actually called. The authors summarises the report by saying that they think RTEMS is well suited for WCET analysis.

An article [13] about testing the Debie software with the Bound-T static WCET analysis tool is written by N. Holsti, T. Långbacka and S. Saarinen. The Debie software is an ESA instrument that monitors space debris and micrometeoroids. The software consists of many different tasks and was already verified using measurements when the analysis with Bound-T was performed. This made it possible to compare the results of the different methods. The code had to be changed slightly in order for Bound-T to analyse it. Bound-T couldn't for example analyse assignments of struct-values and the floating-point division including an irreducible loop structure. A total of 68 lines of code were changed. The number of annotations needed were 47 and 38 of them were for loops. Many of the loop annotations will be found by Bound-T automatically when it is completely developed. The result showed that the statically calculated values were slightly higher than the measured ones. Bound-T found several things that could lead to deadline misses, one of those was a poor implemented change in the code that increased interrupt blocking by 413 μs. The conclusion drawn by the authors is that Bound-T would have been very useful during the development of the Debie software since it would have been able to measure times early in the developing process. Moreover, recomputing WCET values would also have been much easier with Bound-T than with measurements says the authors.

The aiT tool has been used on Airbus and the results have been published as an article [14] for the International Performance and Dependability Symposium 2003. The article was written by S. Thesing, H. Heckman, J. Souyris, F. Randimbivololona, M. Langenbach, R. Wilhelm and C. Ferdinand. The different phases used by aiT to determine the WCET are described together with the information on how the pipeline used by MCF 5307 is constructed. Some information on how the result can be safe is also given by describing abstract interpretation, the technique used in many phases of aiT. The WCET tool was ues in the DAEDALUS project. The things analysed were the precision and usability of the tool. A benchmark was used so that the results of the tool and the results from Airbus' method[1] could be compared. The overestimation was larger with Airbus' method than with aiT. The result from the tool could also be proven to be safe since it was compared with real executions of the program. A strong point of the tool is that it can be used in the development process when hardware isn't available yet. The conclusions drawn by the authors is that the aiT tool provide safe and tight WCET bounds and that it also can be applied to realistically sized programs an environments.

[1] Which method that was used by Airbus is never explained in the article but it is probably some sort of dynamic method.

1.5 Thesis Outline

Chapter 1 gives an introduction to the thesis including background, purpose, WCET, and related works. Chapter 2 includes information that is needed for a good understanding of the rest of the thesis such as the development process (Section 2.1) and compiler/linker (Section 2.2) used by CC Systems (CCS). It also contains information about the target system analysed both concerning hardware (Section 2.3) and software (Section 2.7). Sections 2.4 and 2.5 give some general information about static and dynamic WCET analysis. The technique to make CCS simulations time accurate is described in Section 2.6. Chapter 3 talks about the problem (Section 3.1) and the methods (Section 3.2) used to solve them. Chapter 4 will give information about how the static tool aiT (Section 4.1) and dynamic analysis methods (Section 4.2) can be used to obtain WCET estimates. This chapter is meant to be used as a manual for these types of analyses. The code structure and characteristics of the code that is analysed is described in chapter 5. Chapter 6 goes into the actual analyses with aiT in this thesis and chapter 7 do the same but with the dynamic methods. In chapter 8 the results of the thesis are divided up into the three parts comparing methods (Section 8.1), comparing results (Section 8.2) and describing different usages of WCET analysis (Section 8.3). The results are based on the Chapters 6 and 7. The conclusions drawn in this thesis are presented in Chapter 9 and finally future work is given in Chapter 10.

2 Relevant technologies

This section will go through information that is needed to get a good understanding of the rest of the thesis. Section 2.1 will talk about the development process for software used by CC Systems (CCS). Section 2.2 give some information about compilers and linkers and especially about the compiler/linker used by CCS called Tasking. Target hardware of embedded systems is described in Section 2.3 and Section 2.3.1 and 2.3.2 give some deeper information about the circuit board used by the ESAB welding system and the microcontroller used in all the WCET analyses. Sections 2.4 and 2.5 are about static and dynamic WCET analysis and describes tools and hardware typically used in such analyses. In Section 2.6 the technique used to make CCS's simulations system time accurate is described.

2.1 The Embedded System Software Development Process

Most companies that develops software use a special process to help the programmers and to ensure that the software live up to the demands of the buyer. The development process used by CC Systems (CCS) is called PUP (Process för Utveckling i Projekt[2]). The process is based on RUP (Rational Unified Process) and the different phases of PUP and RUP can be seen in figure 2.1. The following purposes of the phases are used in PUP. The purpose of the inception phase is to set the bounds of the project; gather the demands; create a first sketch of the system architecture; create a plan and a cost estimate of the project and find the risks of the project. The purpose with the Elaboration phase is to establish an architecture; reduce the risks; prepare tests for the entire project; create a plan for the Construction phase and adjust the cost estimates. The Construction phase's purpose is to step by step create and deliver a system with more and more functionality; evaluate and use experiences from system evaluations; test, evaluate and estimate the quality of the system and prepare the users for receiving the system. The purpose of the final phase, the Transition phase, is to carry out final users tests and education; document experiences; correct the system and finish the project. The phases are divided in iterations and each iteration consists of a number of steps. The steps are gather requirements; analyze and design; implement and test. The number of iterations per phase can vary depending on the project. Each iteration is ended with an iteration evaluation. The reason for using iterations is that problems can be taken care of more quickly and the work of the testers is more evenly spread out throughout the project.

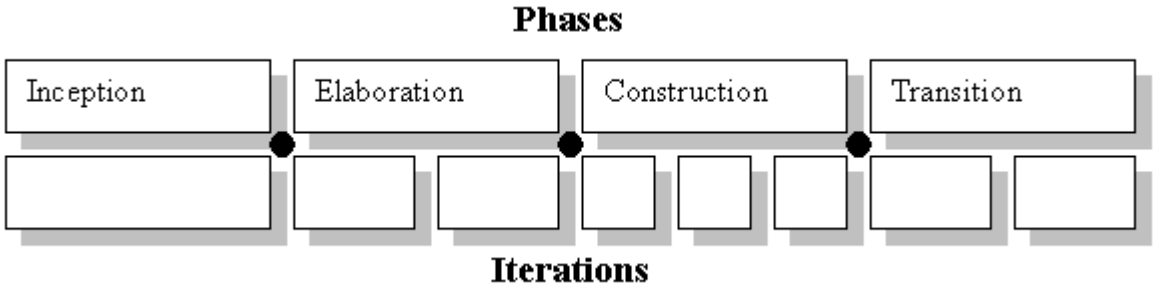

Figure 2.1: The development phases used in PUP and RUP

[2] This is a Swedish name and can be translated to "process for development in project"

2.2 TASKING C166/ST10

To make the source-code understandable by the processor, the source-files have to be compiled and then linked together into an executable. To compile the source-code a compiler, who can translate the source-code into instructions that the processor can understand, have to be used. After the compilation the linker takes the object-files created by the compiler and turns them into an executable. Different compilers and linkers produce different executables and to be able to make a precise WCET estimation as possible the static tool must know which compiler and linker that was used and it also has to support it. The reason for this is that different compilers and linkers generate different information in the executable and this information is by the analysis tool.

The compiler/linker used by CC Systems (CCS) to compile code for the Infineon C167CS-LM processor is TASKING C166/ST10. This compiler/linker is the only one supported by aiT for the C167CS-LM processor used by CCS on the ESAB welding system (Section 2.7). TASKING supports a lot of processors but the version of aiT used only supports processors from Infineon with the C166 and ST10 CPU cores. TASKING C166/ST10 has a lot of compilator optimisations that can be used. There are 18 optimisations that can be individually turned on or off. To make it easier for the user there are a number of predefined groups of optimisations to choose from. They are: no optimisation, default optimisation, optimise for speed, optimise for size and custom optimisation. It is only in the custom optimisation category that the user can choose freely between the different optimisations. The optimisation group commonly used by CCS is the default optimisation group. The reason for CCS to choose this optimisation group is that it doesn't affect the general structure of the code much and therefore makes it easier to test the functionality of the code.

There are eight output file formats to choose from in TASKING. The one used for the analysed code is Motorola S records for EPROM programmers (.sre). This executable is later downloaded to the external EPROM-memory on the circuit board.

2.3 The Target Hardware

The hardware in embedded systems is often much simpler and have much less features than hardware in a PC. The reason for this is that an embedded system doesn't demand the same processing power as a PC. Embedded systems also have a greater need to be predictable and basically the more features there are in the system with caches, out-of-order execution, branch prediction etc the less predictable the system becomes. The cost is also of great importance in embedded systems and that is another reason for using simpler hardware. Section 2.3.1 contains information about the circuit board used the ESAB part of this thesis and Section 2.3.2 contains information about the microcontroller used in this thesis.

2.3.1 The ESAB Circuit Boards

The circuit board used for the ESAB welding system is designed by ESAB. The microcontroller used is the Infineon SAK-C167CS-LM (see Section 2.3.2). The microcontroller itself has a small internal memory but since it is not enough for the welding system, extra memory has been added for the code and data. The external memory for program code is a 1 MB flash memory and the external memory for data is a 32 KB RAM.

The built-in frequency of the microcontroller is 25 MHz but ESAB have put a crystal on the circuit board to lower the frequency to 20 MHz instead. Depending on the use of the circuit board different peripherals can be attached. The peripherals can be displays, buttons, controls for valves etc. One of the ESAB Circuit Boards can be seen in Figure 2.2.

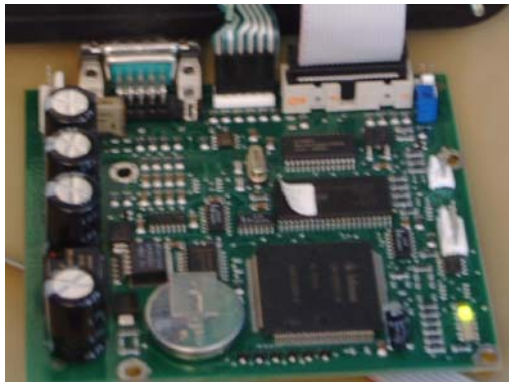

Figure 2.2: One of the ESAB Circuit Boards (the WDS node)

2.3.2 The C167CS-LM Microcontroller

The microcontroller used in this thesis is the Infineon SAK-C167CS-LM, which is a part of the C166 microcontroller family. Below is a listing of some of the components in the microcontroller:

- 16-bit CPU with 4-Stage Pipeline
- CPU clock speed 25MHz
- Two on-chip CAN modules version 2.0B active
- Peripheral Event Controller (PEC)
- Capture Compare Unit (2x16 channels)
- 4-channel PWM unit
- 24-channel 10Bit A/D Converter
- Idle, Sleep and Power Down Mode with Flexible Power Management
- Two Multi-Functional general purpose timer units with five 16-bit timers
- Watchdog Timer and Oscillator Watchdog
- Up to 111 General purpose I/O lines
- Full Automotive Temperature Range: -40°C to +125°C

Up to 16 MB of external RAM and/or ROM can also be connected to the microcontroller. The microcontroller have a feature called a jump cache, which allows the execution time of repeatedly performed jumps in a loop to be reduced from 2 cycles to 1 cycle. It also has a separate multiply and divide unit to speed up execution.

Most instructions on the microcontroller take one machine cycle[3] to execute but there are exceptions that take longer time. A multiplication of two 16-bit numbers takes 5 machine cycles. Indirect addressing also causes longer execution times.

[3] One machine cycle is equal to two clock cycles i.e. one machine cycle is 80 ns on this microcontroller when it is running at 25 MHz.

2.4 Static Worst-Case Execution Time Analysis

Static worst-case execution time analysis is a way to calculate the worst-case execution time without measuring it. In order to calculate the WCET for a code snippet, all properties of the code like upper loop bounds, infeasible paths, recursion depths etc. have to be known. A model of the hardware must also be used so that the timing behaviour of the target system can be mimicked. The hardware model can be very advanced if the system uses performance-enhancing features like instruction caches, data caches, pipelines, branch prediction and out of order execution. There has been a lot of work on how these things affect static WCET analysis, for example [15] and [16]. Fortunately, most embedded system uses simple processors without caches, branch prediction and out-of-order execution. Almost all processors today have pipelines and so do embedded ones, but they usually only consists of a few steps compared to the many step processors of modern PC's.

Static WCET analysis is generally divided into different steps. There are different approaches to how the analysis is divided, which steps there are and how many. In [17] the analysis is divided into three steps and these steps will now be explained. The first step in the analysis is called *flow analysis*. This step is responsible for finding the flows of the system, i.e. all possible executions paths. It is also here flow constraints like loop bounds, infeasible paths, function calls, recursion depth is found automatically or given as input from the user. The program flow can be extracted from three different types of code; source code, intermediate code and object code. If the flow analysis is performed on source code there is a problem of knowing how the compiler will change the code when it is compiled, therefore it's not very common to use source for the flow analysis. Intermediate code is used by some tools since it contains more information from the compiler that can make the analysis more precise but the drawback is that it is very compiler dependent. Commercial tools often use object code for their flow analysis since it is independent of which compiler is used. This makes it is easy to adapt the flow analysis to other compilers and hardware.

The second step of static WCET analysis is *low-level analysis*. It is in this step the execution time for each basic block[4] or instruction in the program is calculated. The low-level analysis is often divided into two parts called global low-level analysis and local low-level analysis. Local low-level analysis considers effect between instructions like pipeline overlap and also takes memory access speed into account. Global low-level analysis considers effects between basic blocks like instruction caches, data caches, branch prediction and translation look aside buffers (TLB's). Global low-level analysis isn't used to generate execution times but is instead used to capture effects of hardware features that needs to be modelled over the whole program. If the processor used is very complicated it might be hard to separate the global and local low-level analysis since for example data caches effects both the local and global part of the low-level analysis.

The *calculation* step is responsible for calculating the overall WCET of the program by combining the results from the flow analysis step and the low-level analysis step. To be able to find the WCET the path that generates WCET must be found and there are different ways of finding this path. There are three methods commonly used and they are: path based calculation, tree based calculation and Implicit Path Enumeration Technique (IPET). The path

[4] A basic block is a group of sequential instructions where instructions are executed if the first one of them is executed.

based calculation method simply calculates the execution time for all paths through the program and selects the one with the highest execution time as the WCET. A less time consuming approach is the tree-based calculation that uses a tree representation of the program that it traverses bottom-up, calculating the execution time of a node with the help of the execution time of its child nodes. The calculation method that is most commonly used is IPET since it can handle complex flow constraints. The idea is that each basic block has an execution time and a count variable. By setting different dependencies between count variables various flow constraints can be expressed. To find WCET the maximum value of the sum of all count variables multiplied with their execution time, considering dependencies, is calculated. To perform the actual calculation Integer Linear Programming (ILP) or constraint solving can be used. ILP is the most commonly used.

To learn more about static WCET analysis read [16], [17] and [18].

2.4.1 Static Worst-Case Tools

There aren't a lot of static WCET tools available today. The ones that exist are either research prototypes or commercial tools. The things that separate the tools are the CPU's supported and calculation method used. Some of the tools are also compiler dependent.

One of the commercial WCET analysis tools now available is aiT Worst-Case Execution Time Analyser (aiT) from the German company AbsInt Angewandte Informatik GmbH [19]. AiT supports the processors ARM7, HCS12/STAR12, PowerPC MPC555, PowerPC MPC565, PowerPC MPC755, ColdFire MCF5307, Texas Instruments TMS320C33, Infineon C166 (only C167CS-LM) and Infineon ST10 (only ST10F269 and ST10F276).

Bound-T is another commercial WCET analysis tool and is a product from the Finnish company Tidorum Ltd [11]. Bound-T was first developed at Space Systems Finland Ltd with support from the European Space Agency (ESA/ESTEC). The Bound-T supports several processors, including the Intel-8051 series, the ERC32 processor in the SPARC series, the DSP Analog Device processor ADSP-21020 and the Renesas H8/300 processor.

The last commercial WCET analysis tool is RapiTime from Rapita Systems Ltd [20] and it is based on the research prototype pWCET. More information about pWCET can be found in [21]. RapiTime isn't a strict static WCET analysis tool since it relies on dynamic methods to get the actual execution times. It doesn't only produce a WCET value it also give a execution time distribution of the analysed code snippet. It uses traces from either a cycle accurate processor simulator or by executing the code on the target system to obtain execution time distributions for smaller code snippets and then combines them using static WCET analysis techniques. RapiTime isn't restricted to be used on certain processors.

One of the research prototypes is the WCET tool SWEET (SWEdish Execution time Tool) [22], [23], [17], presented by Andreas Ermedahl, Björn Lisper, Christer Sandberg and Jan Gustafsson. Jakob Engblom and Friedhelm Stappert have also been in involved in the low-level analysis and calculation part of the tool. The tool is very modular to make it easier to change different parts of the analysis. They have implemented three different calculation methods that they can choose from; they are IPET, path based calculation and clustered. The supported processors are ARM9 and NEC V850E. The part of the tool that is currently being developed is a flow-analysis working on compiler-generated intermediate code. Thereby, the

flow-analysis becomes target independent. The result of the flow analysis is given as input to the low-level analysis and calculation steps of the analysis.

Another research prototype is the Heptane (Hades Embedded Processor Timing AnalyzEr) WCET tool [24]. Heptane is published under the GNU GPL license, and is downloadable free of charge. It supports the processors Pentium I, MIPS and Renesas H8/300. At the moment Heptane works on C source code but work is on the way to make it work on disassembled binary. The people behind Heptane are Antoine Colin and Isabelle Puaut. More information about this tool can be found in [25].

2.5 Dynamic Worst-Case Execution Time Analysis

Dynamic worst-case execution time analysis is the traditional way of finding WCET of a program. There are a lot of methods for obtaining the WCET dynamically, and the thing they all have in common is that the time is measured on a program when it is executing. There are both benefits and drawbacks with that. One benefit is that all execution paths measured are possible paths through the program, which is something that cannot always be guaranteed with static analysis. The problem is to find the path that results in the WCET. It is impossible to force the execution to take a certain path without affecting the system, since the code have to be added or the original code have to be changed. Therefore theoretically all possible inputs have to be tried to find the WCET path. The problem here is that the number of different inputs often is too large to test within a reasonable time. The method usually practiced is to add a safety margin on the value measured to compensate for not finding the actual WCET. This isn't a very good approach since the WCET on some system can be just a little bigger than the measured value and sometimes many times bigger. There is also the problem with interrupts pre-empting the code that is measured leading to incorrect values.

The first problem encountered when doing dynamic measurements is to get the target system connected so that it works like intended. This often much trickier than it seems since it can be hard to get a hold of all hardware required. Most embedded systems interact with the environment and that can be hard to simulate correctly. A problem that can occur during development of the system is that all the code needed for running the system may not even be available since its not ready yet or it is being developed by a different company. The hardware can also be under development. Once the target system is correctly connected it is rather easy to do different measurements. It is also possible to record execution times of a code snippet over a longer period of time and then get statistics of the execution time so that things like jitter and average execution time can be calculated but it is still no guarantee that the WCET has been found. Instead of just measuring execution time, dynamic timing analysis can also be used to get different kind of traces by measuring the address bus or data bus. This is very helpful way of finding the execution path.

The problem with dynamic WCET analysis is to be able to guarantee that the WCET value obtained is safe. As can be seen in Figure 1.1 all the values that can be measured are less than or equal to the actual WCET which means that a safety margin probably has to be added to the WCET value measured to make sure the value is safe. It is however hard to estimate how big the safety margin should be. Since the correctness of the WCET estimate is hard to validate it is also hard to get a tight value. If the system is very small and simple and all inputs can be tested, then dynamic measurement should come up with a value that is exactly equal to the WCET. This means it is both safe and tight but these kinds of system is very rare if not non-existent.

2.5.1 Dynamic Worst-Case Execution Time Hardware

The most common way of finding WCET dynamically is by using an oscilloscope or a logic analyser. The oscilloscope is the cheapest options but it is also the most limited tool. The easiest way of finding WCET with an oscilloscope is by using a pin on the processor, for example setting it high when the execution starts and then setting it low when it ends and measuring the time in between. This method makes it impossible to see which path has been

executed and it can also affect the system slightly. If instead a logic analyser is used the same type of measuring can be made but there is also other possibilities. The entire address bus can be measured so that the execution time can be found by just calculating the time between the address of the first instruction of the code measured and the last one. A trace of the execution can also be saved so that the execution path can be found.

An emulator can also be used to measure execution time if it's time accurate. Most emulators are able to produce traces so that the execution path can be found in the same way as with the logic analyser. The problem with emulators is that they can only emulate the behaviour of one or a couple of processors and they are rather expensive to purchase. They can also, depending on the actual emulator, need some modification of the circuit board. This makes it hard to validate that its timing behaviour corresponds to the timing of the real board.

If the execution time is expected to be several minutes or even hours a simple stopwatch could actually be used to measure the execution time. This kind of long execution time is however very unusual in embedded systems. Pictures of different dynamic measurement hardware can be seen in Figure 2.3

Figure 2.3: A logic analyser, an emulator and an oscilloscope

2.5.2 Dynamic Worst-Case Execution Time Software

If the embedded system uses an operating system (OS), built in system calls for getting the current time can be used to measure execution time. There are also systems without an OS that can have accessible counters but they might be a bit trickier to use. A function call to get the current time can be added in the beginning of the code that is going to be tested. The same is then done in the end of the code. The execution time can then easily be found by subtracting the start time from the end time. This method has a few drawbacks. It is hard to know if the code is pre-empted by other programs, the added code affects the system and the system call to get the time may not have a granularity fine enough to measure the code correctly. It may have a lot of drawbacks but it is probably the fastest way to get an execution time for a program and if the program isn't very time critical this method is probably accurate enough too.

2.6 Time Accurate Simulation

Time Accurate Simulation is the result of a Master thesis at CC-Systems (CCS) 2001 [26]. It is based on the existing simulation technology at CCS.

Time Accurate Simulation is used to extend an existing simulation technique to be able to handle time accurate simulations and not just function accurate simulations. To be able to simulate a system on a PC the hardware dependent code has been replaced with code that

simulates the hardware on the PC. In order to get a time accurate behaviour, breakpoints are added in the code to slow down the execution to the speed of the target system. The breakpoints takes the target time elapsed from the last breakpoint as an in-parameter. The time is used to synchronize the execution of the nodes of the system and can also be used to slow down the execution of the system to the speed of the target system.

There are more advantages of this technique than just a time accurate behaviour. It can be used to slow down or speed up the execution of the system by simple multiplying the system clock with a constant. If the constant is less than one the system is running in slow motion and if the constant is bigger than one the system is running faster than on target. It is also possible to step through the execution of the system one breakpoint at a time. This can be very helpful if you want to study some part of the execution of the system extra carefully. If the only thing of interest is the synchronisation it is possible to run the system in full speed, i.e. only compare times between the nodes and ignore the system clock.

It is possible to simulate different system nodes and each node can also consists of several threads and/or interrupts. The problem with several threads or interrupts in a node is that they use the same CPU and therefore competes for CPU time. When one thread on a node executes the target system time for all the threads/interrupts on that node increases. To be able to implement this the first registered process on a node is called the parent and all other processes on that node have to name that process as their parent. This enables all the processes on the same node to have the same target system time. But since the synchronisation is built upon the fact that the process with the smallest target system time get to execute a problem comes up. In order to select which process on the node is going to execute all processes have priorities. The process with the highest priority gets to execute first when several processes have the same smallest target system time. If there are interrupts in the system, they are simply given a higher priority than the main process to ensure that they get to run before the main process.

To add Time Accurate Simulation to an existing simulation technique, the DLL-file has to be added to the project and then there are just a few extra function calls needed. All the processes of the system have to be registered with a call to the function TimeSyncRegisterUnit, which takes a string with the name of the process and a priority as in-parameters. To start the synchronisation with the other processes one of the functions TimeSyncStart or TimeSyncStartTime must be called. They take the handle returned from the register function call and the target system time of the process as in-parameters. If the target system time isn't given the global system time is used instead. To be able to get the time accurate behaviour you then add breakpoint in the code with the target time elapsed from the last breakpoint and the handle as in-parameters. The breakpoint function is called Tsbreak. When a process is terminated or wants to leave the synchronisation it have to be unregistered or else it will cause a deadlock in the entire system. To unregister the handle to the process is sent as an in-parameter to the function TSunregister.

2.7 The ESAB Welding System

The ESAB welding system is a modular system consisting of up to four different types of permanent nodes and one service node connected via CAN[5]. Depending on the kind of

[5] Controller Area Network. The most commonly used communication bus for embedded systems.

welding system different nodes are used and sometimes there are several nodes of the same type present in the system. One of the two types of permanent nodes always present in a welding system is the Man Machine Communication (MMC). The MMC is the master node in the system and holds information about the entire system and it is also responsible for all interaction with the user through the display and buttons. The second permanent node type that always is present is the PowerSource A (PSA) node, where A is the version. It is responsible for the power supply to the welding process. The first of the two types of permanent nodes that isn't always present is the Remote Control (RC) node; it is used to replace the MMC node when the welder is far away from the actual welding unit. The last type of permanent node is the Wire Feeder (WF) node. The WF node is responsible for feeding wire during the welding phase if the current welding method uses wire. If there are several nodes of the same type in the system it usually is several WF nodes present responsible for different kind of wire (different materials or dimensions). The fifth and final node type is the ESAT node. It is used for services or for upgrading the software in the system. The node isn't actually a separate piece of hardware but software running on a PC and communicating with the other nodes through a CAN-card in the computer. The structure of the system can be seen in figure 2.4.

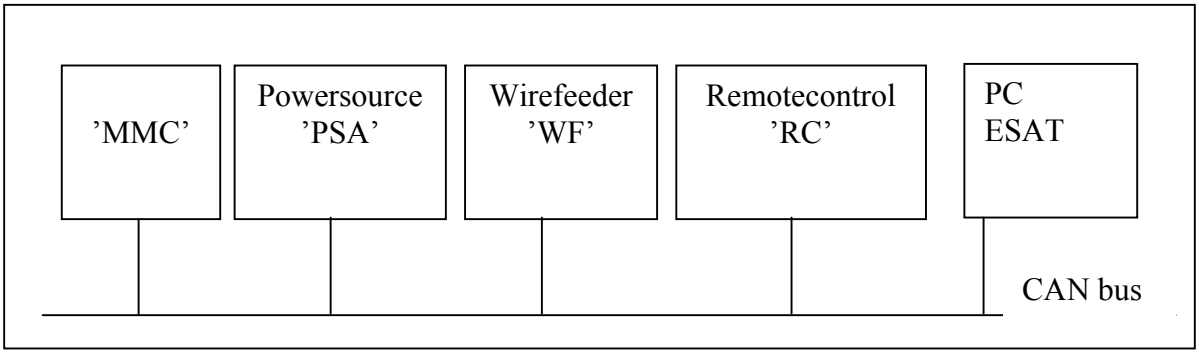

Figure 2.4: The nodes of the ESAB Welding System

Each permanent node has basically the same basic structure. They all use the C167CS processor and they have no operating system (OS). The code is object oriented (C++) and is constructed like a big loop and interrupts. One of the interrupts is responsible for receiving CAN-messages and putting them into a buffer. There are also timers that can be used to trigger interrupts. For example a time triggered interrupt is used on the PSA node to control the power to the welding process. The main loop is a bit different depending on which type of node it is. There are some things that all nodes do in their main loop. First they check their incoming buffer if there are any new CAN-messages, if there are any they send them to the correct class to be processed. Then the outgoing buffer is checked for CAN-messages, if there are any they are sent over the CAN-network. The outgoing CAN-messages are often responses to incoming CAN-messages or they are notifications of changing states possibly due to input from the welder.

The thing that is unique for the MMC node in the loop is that it starts the loop by checking if the welder have pushed a button or turned a knob. If that is the case it lets the current state handle it. The PSA node also has some extra tasks to perform. It has to control a water pump and if the hardware supports a trigger that also has to be handled. These are just some of the things preformed by the nodes but they are the most important ones.

3 Problem description and method

The question that CC Systems wants an answer to is if they could benefit from using dynamic or static WCET analysis in their development process. This chapter will go into the problem and the methods used to solve it. It will also give a description of the aiT Worst-Case Execution Time Analyser tool and the dynamic methods used.

3.1 The Problem

As mentioned above the purpose of this thesis is to find out if CC systems (CCS) could benefit from using dynamic or static WCET analysis in their development process. The purpose of this thesis was to use static WCET analysis methods to test how easy to use and how precise answers they give. Another thesis performed by Yina Zhang [2], also at CCS, looked into the use of dynamic WCET measurements. These methods were then compared to see which one that were best suited to be used by CCS in their development process. CCS also wants to know if the could write their code in a certain way to make it easier to use these methods. They are also interested in a way of making their current simulation technique more time accurate.

To evaluate static and dynamic WCET analysis CCS provided time-critical code to do tests on. The code that was chosen was interrupts from the ESAB welding system. These interrupts were chosen because they were time-critical and of a suitable size for this kind of evaluation. Some component-based code was also tested to be able to see if this kind of code would be different from testing ordinary code.

3.2 Methods

There are only a few commercial WCET tools available today (see Section 2.4.1). This is a reason why static WCET analysis isn't more widely used in the industry. The reason for choosing the aiT Worst-Case Execution Time Analyser (aiT) for this thesis is that it supports processors used by many of the projects at CC Systems (CCS), including the infineon C167CS-LM processor used in the nodes in the ESAB welding system. Another reason for choosing aiT is that the company behind aiT, AbsInt, work within the ASTEC [3] project together with Mälardalen University.

The same codes, which was analysed using aiT, was also analysed using dynamic methods as a MSc thesis work made by Yina Zhang [2]. This made it possible to compare different WCET methods both concerning accuracy and the work effort required to obtain WCET values.

3.2.1 aiT Worst-Case Execution Time Analyser

The aiT Worst-Case Execution Time Analyser (aiT) was initially developed by Saarland University and AbsInt. The research related to WCET analysis at Saarland University started in 1995. AbsInt Angewandte Informatik GmbH was founded in February 1998. It is a spin-off from the Department of Compiler Construction and Programming Languages at Saarland University, Germany. AiT is pronounced I T. AiT supports the processors ARM7, HCS12/STAR12, PowerPC MPC555, PowerPC MPC565, PowerPC MPC755, ColdFire MCF5307, Texas Instruments TMS320C33, Infineon C166 (only C167CS-LM) and Infineon ST10 (only ST10F269 and ST10F276). AiT has also been awarded a 2004 European

Information Society Technology (IST) Prize. Some of the companies that have used aiT are: Airbus, Bosch, DaimlerChrysler and Ford.

AiT uses object code to produce WCET estimates, but AiT doesn't only produce WCET-values. It also has a good graphical interface, which can be used to see the structure of the code from the call-graph (see figure 3.1) from functions down to single instructions. The program that displays the graphs is named aiSee (pronounced I See). The graph can be produced with and without execution times and the graph is a basic call-tree when it is opened. Each function can then be opened to reveal the basic blocks[6] of the function. The basic blocks can in turn be opened so that the individual instructions can be viewed; this enables the user to get both a good overview and also to get down on a very detailed level. If the user requires, each basic block can be viewed in a such way that all the pipelines stages the instructions go through are visible. In the call-graph the loops are also displayed as functions. Instead of seeing a loop as a bit of code running several times in a function it is represented as a function that calls itself recursively (see figures 3.4 and 3.5).

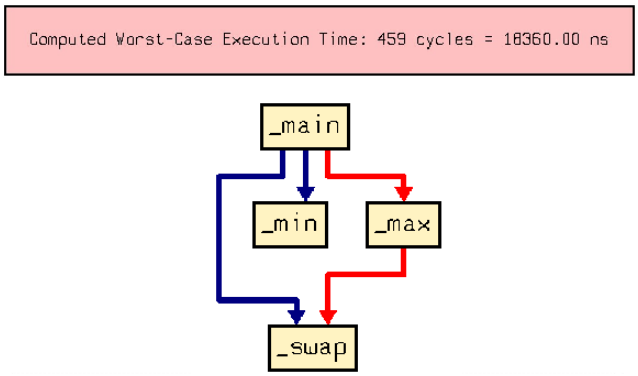

Figure 3.1: Call-graph with WCET displayed

A nice feature of aiT is that source code can be added to the graph, so the user can see which basic block represent which line of source code. Figures 3.2 and 3.3 shows basic blocks without and with showing source code. Displaying the source code in the graph makes the job of interpreting the graph much easier. Some source code lines become several basic blocks when compiled and then only the first basic block is labelled with the source code line and the others are marked empty, this is done so that the user isn't confused by the same source code appearing several times.

[6] A basic block is a group of sequential instructions where all instructions are executed if the first one of them is executed.

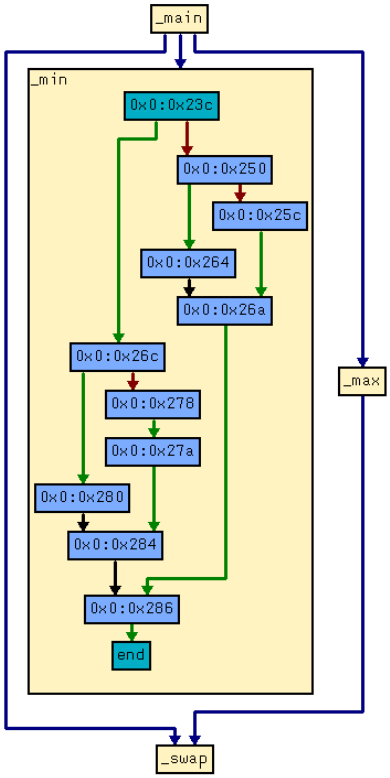

Figure 3.2: Basic block labelled with addresses

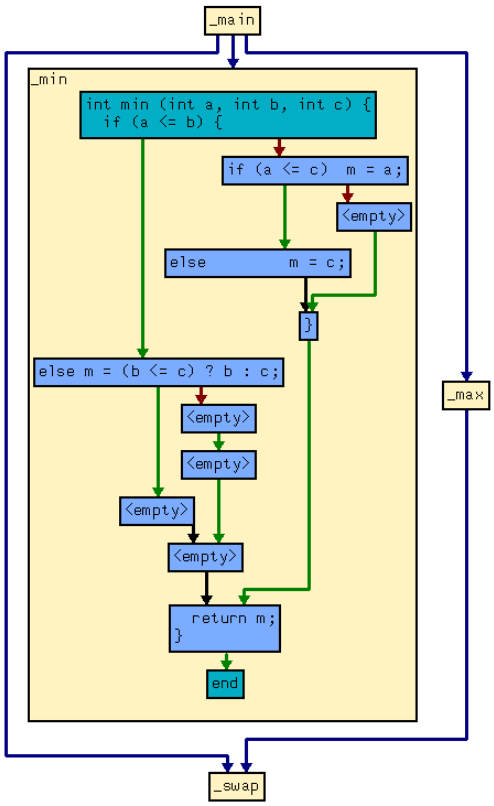

Figure 3.3: Basic blocks labelled with source code

To separate the different types of basic blocks the basic blocks are coloured. The first basic block of each function is called the entry block and is coloured green. The last block in the function is labelled end and is called the exit block and is also green. All ordinary blocks in the graph are blue. Loop calls are represented by a block that is a sort of greenish blue and labelled with the name of the loop. All these types of nodes can be viewed in Figure 3.4. Edges also have different colours so that they can be distinguished. The most important edges for the user are the true and false edges, true edges are green and false edges are dark red. The colours of these edges is the only way of knowing if jump-instructions are performed on not. The colours of the edges represent the condition in the object code. As illustrated in Figure 3.3 this is often the opposite of the condition in the source-code. If the graph displays WCET, the WCET path marked by red edges as illustrated in Figure 3.1. The user can customize the colour scheme if the default colours aren't satisfactory. The WCET graph also displays the WCET contributions of functions together with the execution time of each basic block. All these features makes the graphical interface very useful, not only to show the WCET, but also to help the user understand the structure of the code, and to visualize the timing contributions of different code parts. More details on how aiT is used to analyse the WCET of different programs are given in Section 4.1.

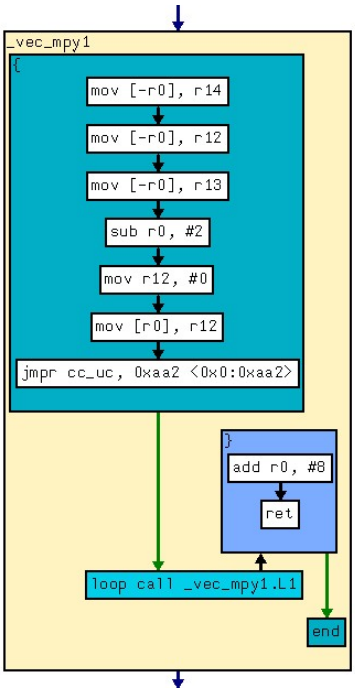

Figure 3.4: Basic blocks with instructions visible

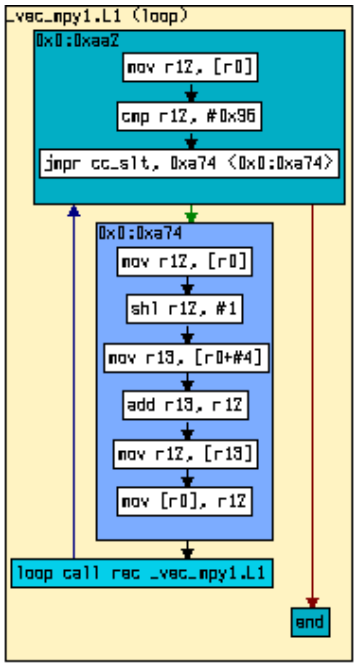

Figure 3.5: Loop function with instructions visible

3.2.2 Dynamic Methods

As mentioned earlier, another master thesis was also performed at CC Systems (CCS) by Yina Zhang [2] with the goal of finding and testing different methods of dynamic WCET analysis. This made it possbiel to compare her results and the results in this thesis in order to find which method or methods that are best suited to be used by CC Systems. This section will details on the methods she looked into and the methods selected for the comparison.

The easiest way of doing dynamic WCET analysis is by using the internal clock in the CPU by calling time-functions in the operating system (OS). Unfortunately the ESAB welding system doesn't have an OS so therefore these methods could be excluded right away. There was also a possibility to use a CPU emulator to be able to a get time-stamped trace of the execution. The emulators Yina found that supported the C167CS-LM processor were however too expensive for CCS to invest in. Some other methods were examined, but were classified as too advanced for this purpose. The two methods finally selected were using an oscilloscope and a logic analyser. These are two of the most commonly used methods to acquire the WCET of a program.

The oscilloscope is very limited since it can only measure one or maybe two things simultaneously. This makes it hard to do any advanced measurements with several measuring points or to measure an entire bus. Basically the only way of measuring execution time with an oscilloscope is to set a bit high in the code where the measurement should start and then set the bit low when the measurement should end. The execution time is the period during which the pin is high and that time can easily be read on the display of the oscilloscope. This method makes it very hard to see which path the execution has followed.

When a logic analyser is used the possibilities are much greater. It is often possible to measure more then a hundred different points on the circuit board simultaneously. This makes it possible to monitor an entire bus, for example the address bus. It is also possible to measure several measuring points in the code simultaneously, allowing the user to keep track of several things at once. This the user to get information not only about the total execution time, but also execution times of smaller blocks inside the code. This can be useful if you want to find places in the code that take long time to execute and maybe optimise that code so that it runs faster.

The methods finally chosen by Yina were to use an oscilloscope and a logic analyser. The oscilloscope was used to measure execution time by setting a bit high in code where the measurement was started and setting the bit low in the code where the measurement was ended and then measure the time in between. When the logic analyser was used to measure execution time it was connected to the address-bus and then traces of the execution could be extracted and analysed. This made it possible not only to get execution times but also to see which path that was executed. Information on how the actual measurements were performed can be found in Section 4.2.

4 Solution

The solution that was selected for this thesis to calculate the WCET was the aiT Worst-Case Execution Time Analyser (aiT) from AbsInt. How to use aiT to calculate the WCET is described in Section 4.1 below. The methods chosen by Yina Zhang in [2] to dynamically measure WCET were using an oscilloscope and a logic analyzer, described in Sections 4.2.2 and 4.2.3 respectively. Section 4.2.1 gives some brief information on how to set up the target system was set up for the dynamic measurements.

4.1 Analysing Executables with aiT

The aiT tool is used for calculating WCET for a piece of code, but in order to do so it requires input from the user. Different inputs are required, basically because of the differences in the hardware supported by the different aiT versions. The version of aiT used in this thesis supports microcontrollers with an Infineon C166 or a ST Microelectronics ST10 processor core. The hardware specific input needed for this version of aiT is for example which processor core is used, which X-Bus features are enabled/disabled and the memory configuration.

The user could also give software inputs, basically annotations used to control, restrict or direct the flow of the program. These annotations are described in Section 4.1.1. The executable also has to be added to aiT in the correct format. The executable formats supported by this version of aiT are the IEEE-695, the ELF, the COFF and the XCOFF file formats. The format that was used in this thesis was the IEEE-695 format, since it was the only supported formats that could be produced by TASKING. It is also important to use the same version of TASKING that was used during the development of the code; otherwise it is often impossible to compile the program. Another important thing was to set the correct addresses for the Context Pointer, System Stack Pointer, User Stack Pointer and the Data Page Pointers. They can have a big impact on the WCET value. The data page pointers are used by the Infineon processor to access memory. The memory is divided into different pages and depending on the amount of memory used these pointers can have fixed or changing values.

The first thing to do when calculating WCET for a code snippet is to learn the code so that annotations for loop bounds, recursion depth, infeasible paths etc can be correctly given. One way of learning how the code is meant to work is to create a combined call-graph and control-flow graph in aiT using the “Compute CFG” feature. This can give a better overview of the code compared to examining it in the compiler. To start the actual WCET calculations in aiT all the hardware-inputs have to be correctly set. When TASKING is used most of the necessary hardware-settings can be found under the EDE-menu in TASKING. The values of the different pointers should also be set. It is not necessary to set them manually and sometimes aiT finds their values itself. However, if aiT can’t find them it can result in overestimations of the WCET if they aren’t set manually. The values of these pointers can often be found in the map-file created when the program is compiled and linked together.

The next step is to create the ais-file that contains all the annotations. There are a couple of annotations most frequently used when calculating the WCET. These are annotations for the clock frequency, the compiler used and the annotation for the context specification. None of them are really necessary but very useful. When these annotations have been added to the ais-file it is basically a matter of trial-and-error to find out which annotations that have to be used.

The user can try to specify all the annotations needed from the beginning but since aiT can find many of these itself it is better to let it do so. When it is time to let aiT try to calculate the WCET there are two possibilities to do so, one is to select "Analyze" and one is to select "Visualize". The Analyze choice calculates the WCET and creates a combined call-graph and control-flow graph. The WCET for the entire code snippet is displayed in a separate box as can be seen in Figure 3.1. The WCET contribution of the functions can also be seen as well as the execution time for basic blocks with different contexts. The Visualize choice does the same as the Analyse option but it also provides access to the pipeline states of the basic blocks.

If aiT can't find for example a loop bound it will fail to calculate WCET and display the message "This problem is unbounded" and then the user has to give an annotation for that loop. If the user has given contradicting annotations aiT will display the message "This problem is infeasible" and the user has to remove the contradicting annotation. When aiT has managed to calculate a WCET value it is important to check that the execution path is correct and doesn't include infeasible paths or other things that can't happen when the software is running on the target. If the path leading to the WCET isn't possible, more annotations to restrict the flow of the program have to be added. When the execution path that yields the WCET is correct there are two choices, either accept the WCET value obtained or use extra annotations, for example memory-accesses, to get a more tight WCET value. It is a good idea to save the graphs and the ais-files so that the calculations can be saved for future use. This make it possible to reuse most of the annotations if the code is changed and a new WCET value needs to be calculated.

4.1.1 User annotations

There are many user annotations in aiT. All of the annotations are used to make the calculation of WCET possible and to give WCET that are values as tight as possible. Some annotations can be given directly in the source-code, but the most common way of giving annotations is by writing them in ais-file. The annotations in the ais-file are written after each other and ended by a semicolon. The following list contains some of the annotations that can be specified in the ais-file. All of the annotations used in the thesis are described below.

Clock rate: This annotation allows aiT to convert the WCET value from clock-cycles to units of time. This annotation is optional but it saves the user from having to calculate the WCET from clock-cycles to real time. The following example sets the clock rate to 20 MHz.

```
clock exactly 20 MHz;
```

Compiler: This annotation is used to specify which compiler is used. Knowing which compiler is used makes the job of the executable reader easier. This annotation should always be present. The following example selects TASKING as the compiler.

```
compiler "c166-tasking";
```

Context specification: This is one of the most important annotations. It is used to limit the number and depth of the calling contexts[7] used. This is done since the number of contexts can

[7] Calling contexts are the calling history of functions i.e. the calls from the start of the analysis to the point when the function was called.

otherwise be almost infinite and this will make the WCET calculation to take very long time. First there are two kinds of context calculation, called limited and flexible. Flexible enables loop bounds to be automatically detected while limited only accepts loop bounds given by the user. Since automatic loop bound analysis is often desired only the flexible version will be explained here. There are three attributes to the `interproc flexible` annotation and two of them are optional. The only attribute that has to be given is the max-length one. It restricts the maximum length of the call string i.e. how many calls backwards that are saved. The optional attributes are default-unroll and max-unroll. Default-unroll is used to limit the automatic loop bound analysis. If the a loop bound cannot be found by aiT this attribute will make it stop trying when is has reached the value set by default-unroll. The higher this value is the longer time the analysis may take but the higher loop bounds may be found automatically. Max-unroll is a way of limiting the calculation of execution time of loop iterations dependent on contexts. The value set by max-unroll is the number of different execution times that will be calculated for the iterations of a loop. This is to reduce the required analysis time. If this value is low the WCET estimation will be less tight than if the value is high, but this is mostly true if caches are used and this in not the case for the processor used in this thesis. Default-unroll and max-unroll doesn't affect the number of iterations of the loop, they only affect the number of loop contexts. All these attributes limit individual loops. The following example of a `interproc flexible` annotation will make aiT allow call strings that are 3 calls deep (max-length), automatically find loop bounds not bigger than 5 (default-unroll) and to calculate up to 4 different execution times for loop iterations (max-unroll).

```
interproc flexible, max-length = 3, max-unroll = 4, default-
unroll = 5;
```

Stop decoding: AiT normally ends the WCET analysis when the function containing the entry point is left. If for some reason the user wants to end the analysis somewhere else, this annotation can be used. The following example ends the analysis at page 2 (hex) and CPU address 5a3e (hex).

```
end 0x2:0x5a3e;
```

Targets of computed calls and branches: There are annotations to tell aiT the target of unresolved branches and calls. They can be useful when inheritance is used and aiT doesn't know which method to call. The example below tells aiT that the call at address 0x2:0x5a3e calls address 0x0:0x14c1.

```
instruction 0x2:0x5a3e calls 0x0:0x14c1;
```

Properties of calls and functions: Some calls and functions immediately return or never return. This can be expressed by the user using annotations. If it is a call that never return the annotation `instruction 0x2:0x5a3e never returns;` can be used and if it is a function that immediately return the annotation `snippet 0x0:0x14c1 immediately return;` can be used. The following examples are a function that doesn't return and a call that immediately return. Snippet is used to reference a bigger part of the code such as a function or a basic block.

```
snippet 0x0:0x14c1 never returns;
instruction 0x2:0x5a3e immediately return;
```

Memory accesses: In order to get tighter WCET values memory accesses that aiT can't find the correct address for can be specified by the user. Some processors have instructions that can do several memory accesses, in this case the user can specify in which step which address is used by adding for example "in step 2" after the annotation. Exact memory addresses aren't necessary since it is possible to give an interval or even an array as the target of a memory access. The following examples are one instruction that accesses memory 0x0:0x14c1 and another instruction that access memory in a range in step 2.

```
instruction 0x3:0x4a72 accesses 0x0:0x14c1;
instruction 0x3:0x781d accesses 0x0:0x1400 .. 0x0:0x1500 in
step 2;
```

Known register values: If register values are known when an instruction is executed this can be expressed for aiT in a way similar to the memory accesses annotations. This can be helpful when for example an enumeration is used to declare a state; then this annotation can be used to tell which state is active before a switch-statement depending on the state is executed. The example below is an instruction that has the value of r4 equal to 45 and the value of r7 between 12 and 99 when it starts its execution.

```
instruction 0x3:0x4a72 is entered with r4 = 45, r7 = 12 .. 99;
```

Not-analysed and external function: If a function isn't available in the executable its execution time can be given by the user. The user can also give execution time for functions that he knows the execution time for. The execution time can be specified in either cycles or time units. The following examples are an external function and an internal function that a user has specified the execution time for.

```
snippet "function1" is external and takes 455 ns;
snippet "function2" is not analysed and takes 2333 cycles;
```

Infeasible code: If there is code that never will be executed this annotation can be used. It can be very useful if the system has a special function for error handling and errors aren't supposed to be included in the analysis. Infeasibility is propagated through the control-flow graph; if a block is only reachable from an infeasible block that block is infeasible too. The following example will avoid errors being included in the analysis (if the function that handles errors is called errorhandler).

```
snippet "errorhandler" is never executed;
```

Values of conditions: This annotation tells aiT that a condition is always true or always false. The example is a condition that is always true.

```
condition 0x3:0x4a72 is always true;
```

Recursion depth: This annotation restricts recursion. The restriction bound can be a *max* and/or a *min* or an *exactly* value. The example is a function has a recursion depth of max 2.

```
recursion "Function2" max 2;
```

Loop bounds: There are ways of giving both global and local loop bounds but only the annotation for local ones will be mentioned in detail here. Global loop bounds are used to set

bounds on several loops at the same time (this can be rather risky) and local loop bounds are set for individual loops. The bound is a value telling aiT how many times the loop body is executed and a qualifier telling aiT if the loop test is at the beginning or end of the loop. Both the value and the qualifier refers to the executable and not to the source-code. The bound can be a *max* and/or a *min* or an *exactly* value. The example below is a loop that iterates between 3 and 6 times and the loop test is in the beginning.

Alt1: `loop 0x3:0x4a72 min 3 max 6 begin;`
Alt2: `loop 0x3:0x4a72 begin min 3 max 6;`

Relative execution counts: This annotation (flow) is useful to express known relations between the execution count of different basic blocks. It can be useful when setting loop bounds for loops with several entry points since they can't be bounded by the ordinary loop bound annotation. This annotation can however be used for other things as well. It can for example be used to tell aiT how many times a certain instruction or block is executed in total by giving a relation between it and the entry point of the analysis. The actual relation can be a *max* and/or a *min* or an *exactly* value. If the relation between execution counts applies to all contexts cumulatively the qualifier *sum* should be used and if it applies to each context individually the qualifier to use is *each*. If the relation is between two program points not in the same function the qualifier *sum* must be used. The example below tells aiT that instruction 0x3:0x4a72 runs twice as much as instruction 0x0:0x14c1 in every context.

```
Flow each 0x3:0x4a72 / 0x0:0x14c1 is exactly 2;
```

4.1.2 Relative addressing

Instead of using absolute addressing, a paged address containing a page number and a CPU address, relative addressing can also be used. Relative addressing can address instruction relative to the start point of a function. The relative address is given in the form *"Target + n Things"* or *"Target - n Things"*. Target can be for example a paged address or a function and Thing can refer to for example instructions, calls, branches, returns and bytes. So *"function1" + 3 instructions* refers to the third instruction in function1, therefore *"function1" + 1 instructions* is the same as just *"function1"*. Several things can also be added or subtracted, for example *"function1" + 3 calls – 2 instructions* is the instruction before the third call in function function1. It is best to use relative addressing since it requires less amount of annotation work if the code is changed. The relative address should still be valid if the executable becomes different but the absolute address is probably invalid.

4.2 Using Dynamic Methods to Measure WCET

This section gives a brief overview on how to measure execution time with an oscilloscope and a logic analyser. For more detailed information see [2].

4.2.1 Setting up the Target System for Dynamic Measurements

To be able to do dynamic measurements the target system must first be set up correctly. Connecting things like power supplies and communication cables is something that has to be done. To be able to get as close to the real thing as possible as much as possible of peripheral devices, such as for example sensors and actuators have to be connected. It is very important to make sure that system behaves like it would in its real environment or at least as close to it as possible. The last thing to do before measuring is to download the actual software that is going to be measured on the target system. It can also be a good idea to add monitoring on

things that can be of interest. For example a CAN-listener to monitor CAN-messages or multimeter to measure voltage or current can be useful. When all this is done it is a matter of connecting the actual measuring equipment and start the measurements. How to do this is explained in Sections 4.2.2 and 4.2.3.

4.2.2 Measuring Execution Time with an Oscilloscope

The oscilloscope used in this study was a Hameg Digital Storage Scope HM205-2. The method chosen for measuring with the oscilloscope was to set a pin on the processor high in the code where the measurement was going to start and then set the same bit low where the measurement should end and then measure the time in between. The problem was that it was impossible to measure on the processor since the pins were to small and it was a big risk of short-circuiting them. There are however 3 Light Emitting Diodes (LEDs) on the circuit board used for showing error messages etc. Each one of these could shine green and/or red or not at all. This means that there were 6 places to measure on since these LEDs weren't used in the code that was going to be measured. Start and stop-points are then inserted in the code as functions writing to the correct pin. The program then has to be recompiled and linked and then downloaded to the memory on the circuit board. It is then just a matter of running the code that is going to be measured. This can be a problem if the code to be measured is an interrupt and it is triggered by an external event. Then that event has to be triggered in some way for example by sending a CAN-message or by setting a bit high. The LEDs can also be used to indicate which path that was executed. A signal can be sent to make the LED shine green if an if-statement is true and red if it isn't. In this way one if-statement per LED can be tested for each compilation. It can be a bit problematic if the program is large to find all path-selections made.

The drawback of this method of measuring execution time is that it introduces a probe-effect. Probe-effect means that by inserting code for measurements the system itself is affected and in complex system this can give very strange effects like timing errors or other things that affect the system. The code inserted for actual measuring isn't very big and shouldn't affect the system much but if the user want to affect the execution path by changing values of variables or something similar this can have bigger consequences.

4.2.3 Measuring Execution Time on with a Logic Analyzer

The logic analyser used in this study is called Hewlett Packard 1670D. There are many ways of measuring execution times with a logic analyser. However for this study the logic analyser was connected to the address-bus. First the problem was to connect the logic analyser to the address bus on the circuit board. It wasn't possible to connect the logic analyser directly to the address-bus on the microcontroller since the distance between its pins was too small. There are adapters available to connect the logic analyzer directly to the microcontroller but these are rather expensive and hard to connect. It is also possible to custom make a circuit-board were all pins from the address-bus are connected to some kind of port that the logic analyzer can be connected to. It this case the easiest way of connecting the logic analyzer to the address-bus was by using the Flash-memory on the circuit board. There are adapters for this too but it is possible to use the clips that come with the logic analyzer and connect them individually to the pins of the memory corresponding to the address-bus. To find out which pins correspond to the address-bus on the Flash-memory chip the connecting diagram of the chip had to be examined. In this case the memory chip was an AM29F800BB chip with the connecting diagram illustrated in Figure 4.1. One thing to consider when listening to the

address-bus of the chip is that it can address either bytes or words. If the pin labelled BYTE# in Figure 4.1 is set at logic '1' the device is in word-configuration. In this case the word-configuration was used. This means that A0 on the Flash-chip is connected to A1 from the microcontroller since the lowest bit is of no interest when addressing words. When the logic analyzer is properly connected to the address-pins on the memory-chip the settings of the logic analyzer have to be set. The pins that are going to be monitored have to be set together with the trigger address to start the trace recording. The trigger address can be the start of the function that is going to be measured or the address in the interrupt-vector where the address to the actual interrupt-routine is. To be able to measure the execution time the address of the return-instruction can be searched by the logic analyzer. The actual trace can be downloaded on a floppy and then examined on a computer so that the execution path of the trace can be revealed. The addresses of the trace have to be translated in some way to the instruction of the object-code and this can be done with a control-flow graph of the program or by using a debugger. The control-flow graph gives a better overview of the program than the debugger.

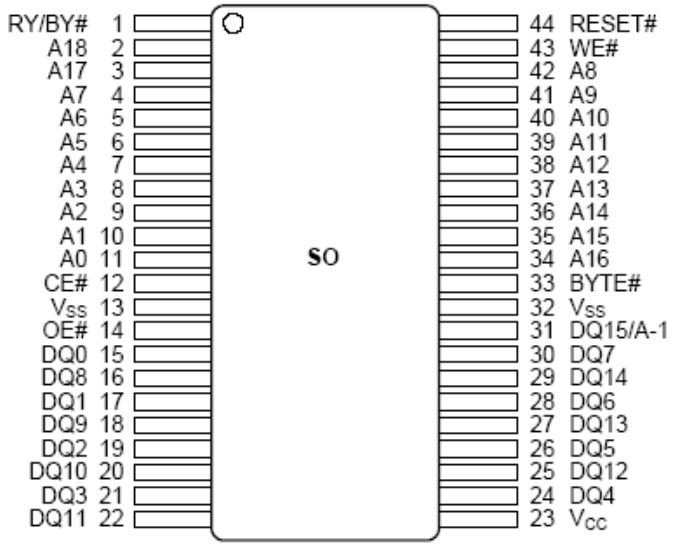

Figure 4.1: The connecting diagram of the AM29F800BB chip

The advantages of this method compared to using the oscilloscope is that no code has to be inserted to be able to measure execution time. The actual execution path can also be found and that gives a much clearer image of what has actually been measured and can reveal things like interrupts that preempts execution of the measured code. This wouldn't be detected by the oscilloscope and can therefore make the longest measured execution time actually not valid at all. If the user has to force a ceratin path by inserting new he might also introduce a probe-effect. However, knowing the execution path alows the user to only force the execution where he knows that it takes the "wrong" way.

5 Code characteristics

This chapter will give a short description of the structure of the analysed code snippets and also give some characteristics of the code.

5.1 The WDS node

WDS stands for Weld Data unit Small. The WDS node is a MMC node and it is the master node in the system. It holds information about the entire system and it is also responsible for all interaction with the user through the display and buttons. For more information about the WDS node see Section 2.7.

The following code characteristics have been found for the WDS node:

- 126 C++ source-code files. Between 30 and 2400 line of code per file including comments. There is a file with 13000 lines but it contains only icons written as char-arrays.
- 133 header-files with between 20 and 600 lines each.
- About 1100 functions
- The code itself doesn't contain assembler routines but there are some library routines written in assembler, for example floating-point additions and type casts.
- Switch-statements are commonly used since many parts of the node can be considered as state-machines and the switch-statements are there to make sure that the correct state is run.
- Recursion doesn't seem to occur or are used scarcely.
- There are 119 for-loops in the node. 10 of them are nested and 2 levels deep. 15 loop tests are dependent on functions calls (many of the functions only return a register value). There are no for-loops tests dependent on pointers or triangular for-loops. At least 3 for-loops contain a switch-statement.
- 119 while-loops. No nested while-loops. 10 loop tests are dependent on functions calls. 7 pointer dependent loop tests. 57 of the while-loops were non-terminating loops designed to halt the execution of the system if a serious error occurs. The system would then be rebooted when a watchdog timer expires.

5.1.1 The CAN-interrupt

The basic call structure of the CAN-interrupt on the WDS-node can be seen in Figure 5.1. There are 3 different types of messages that can be received. They are: *STATUS*, *MESSAGE* and *MESSAGE15*. *MESSAGE* is an ordinary CAN-message, *STATUS* is message from the actual CAN-controller informing of an error and *MESSAGE15* is used for service messages from the ESAT node (see Section 2.7 for information about the ESAT node). The function `ReadInterruptIdentifier()` tells `GetMessageFromCanController()` which type of message that has been received and then `GetMessageFromCanController()` either checks which error that has occurred if it is of STATUS type or receives the message if it is of *MESSAGE* or *MESSAGE15* type. There are switch-statements in the functions `GetMessageFromCanController()` and `PutMsgOnBuffer()` that depend on which message that have been received. The function `Interrupt()` is also dependent on if the message is an ordinary CAN-message (*MESSAGE* or *MESSAGE15*) or not. The reason that there are two sets of functions named `CheckWarning()` and `CheckBusOff()` is that the set marked (C) contains methods in the ECan-class while the two marked (CM) are methods in the ECanMessage-class. The functions in the ECan-class read from the CAN-controller to

see if the bus is off or if there is a *warning*, while the functions in the ECanMessage-class just returns values of booleans. The `Instance()` functions are responsible for making sure that there exists just a single instance of a class and it also returns a handle to that instance. If an instance of the class doesn't exist it has to be created and this means a big call-tree with a lot of calls to library routines like malloc etc. that are very hard to analyse. The function `AddAndLogError()` is responsible for logging errors and acting upon them, it also has a large call-tree that isn't shown in the picture. The interrupt is designed to be able to receive up to 6 messages at the same time. This is implemented by a for-loop in the function `Interrupt()` calling the function `PutMesssageOnBuffer()` up to 6 times. The reason why several CAN-messages can be received during one interrupt is messages will not cause a new interrupt and therefore otherwise be lost. The loop is a for-loop that iterates 6 times but it also contains a return that is executed if an empty message or a *STATUS* message is received.

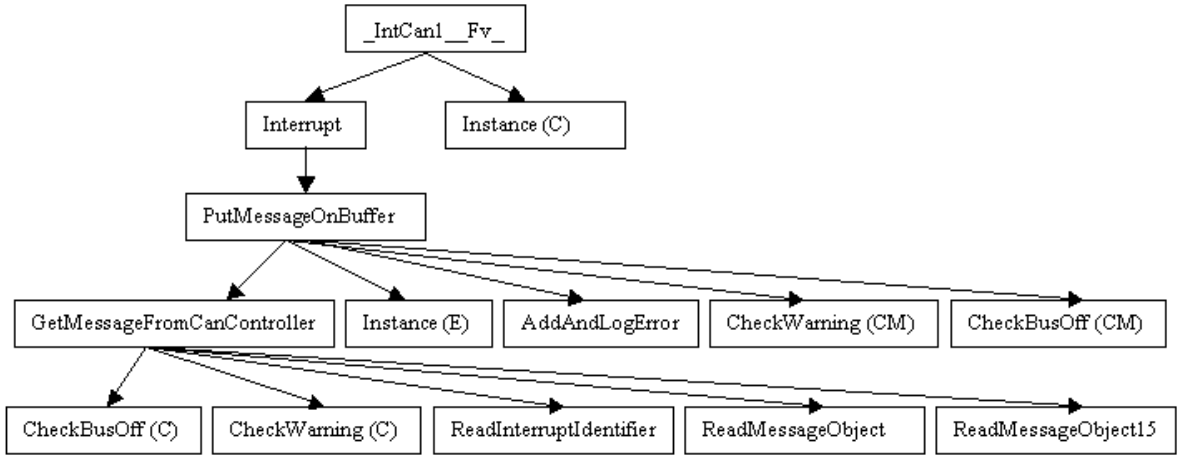

Figure 5.1: The call-tree of the CAN-interrupt on the WDS-node

The following code characteristics have been found for the CAN-interrupt on the WDS node[8]:

- 129 functions of which 42 are non-library functions. The call-depth[9] is 22.
- 65 loops of which 10 are in non-library functions. Most loops are very simple. No loop-tests are dependent on pointers or functions results. No nested loops. Two of the loops are often terminated by a break-instruction in the loop, which makes the loops a bit harder to analyse.
- No dynamic branches.
- No recursion.

5.2 The PSA node

This node is responsible for adjusting the power when welding. For more information about the PSA node see Section 2.7.

The following code characteristics have been found for the PSA node:

[8] Instantiation of static variables isn't included since it contain only library-code that isn't analysed.
[9] Call-depth means how many levels of functions are called i.e. how long the calling context is.

- 41 C++ source-code files. Between 40 and 2400 line of code per file including comments. Most of the files have less than 400 lines of code.
- 40 header-files up to 400 lines each. Most of them have less then 100 lines.
- There are about 500 functions.
- The code itself doesn't contain assembler routines but there are some library routines written in assembler that are used for example for floating-point additions and type casts.
- Switch-statements are commonly used, since many parts of the node can be considered as state-machines, and the switch-statements are there to make sure that the correct state is run.
- Recursion doesn't seem to occur very often. There is recursion in the library function fflush and the recursion depth in that function is up to 5. The recursion depth was found by studying the code.
- There are 64 for-loops in the node. Many for-loops are identical and placed in several cases in the same switch-statement. 3 of them are nested and 2 levels deep. Basically all loop tests were simple integer comparisons.
- 58 while-loops. No nested while-loops. Basically all loop tests were simple integer comparisons but some loop tests are however dependent on pointers. About half of the while-loops were designed to wait for something to be ready (while(busy)).

5.2.1 The CAN-interrupt

The basic call structure of the CAN-interrupt on the PSA-node can be seen in figure 5.2. The structure is a bit different from the one on the WDS-node but the basically the same functions are used. The same three types of CAN-messages as in WDS-node are used (see Section 5.1.1). The only difference is that *MESSAGE15* on the PSA-node is used for boot messages when in simulation mode. The function `Interrupt()` contains a loop in the same way as on the WDS-node but it is very different in structure since it is responsible for calling most of the other functions involved in reading CAN-messsages from the CAN-controller. The loop receiving CAN-messages is terminated when there are no more messages to receive (a empty message is received) or the loop limit is reached (6 iterations). The function `ReadInterruptIdentifier()` is called first in every iteration and returns the type of the message. A switch-statement is then used to receive the message if it is a *MESSAGE* or *MESSAGE15* and to check which error it is if it is a *STATUS*-message. The `Instance()` function has the same function as on the WDS node but here a static variable is used instead of a dynamically allocated variable. The code to create the variable is however as hard to analyse as on the WDS node.

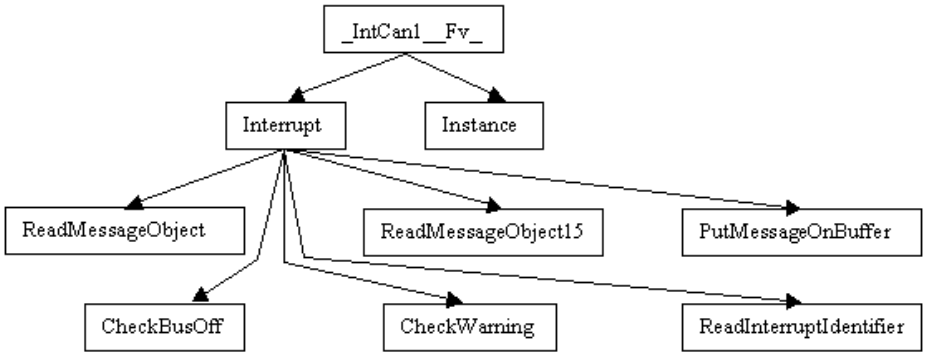

Figure 5.2: The call-tree of the CAN-interrupt on the PSA-node

The following code characteristics have been found for the CAN-interrupt on the PSA node[10]:

- 14 functions of which all are non-library functions. The call-depth is 5.
- 1 loop. It is a simple for-loop that iterates 6 times (for(i=0;i<6;i++)) but is also has a return statement inside the loop making it terminate when there are no more messages to receive.
- No dynamic branches.
- No recursion.

5.2.2 The Regulator-interrupt

The regulator-interrupt is a periodic interrupt responsible for adjusting the power to the welding process and it also starts and stops the welding-process depending on the inputs it gets. The actual structure of the interrupt depends on the welding-method currently used. Each welding method has it own `Regulator()` function, the regulator-function that is associated with the current welding-method is then called by the Regulator-interrupt. This is done by inheritance since all welding-method classes inherit from the class EMethod. All the different regulator-interrupts are constructed as state machines with different states depending on if the system is actually welding or not, if the system is overheated or if it is idle etc. The states that is used for the Mma-welding method are *IDLE*, *START1*, *START2*, *WELDING*, *STOP* and *OVERTEMP*. The welding method Tig has over 20 states. The basic structure of the regulator-interrupt when the Mma and Tig welding methods are used can be seen in Figures 5.3 and 5.4 respectively[11]. If Mma is used then the current- and voltage-values are first read and then a function for determining the state is executed. The current state and the values of current and voltage are considered when determining the new state. If the state is changed messages can be sent to inform the WDS node that the welding current will be turned on or off. When this is done the actual welding process is regulated in another function called `CalculateCurrent()`. This function uses the current state and the values of current and voltage read to do adjustments to the welding process by changing the current. The structure when Tig is used is a bit different. First a function called `PulseTimer()` is called. It is responsible for adjusting pulse welding if that is activated. Then the function `State()` is called and the first thing it does is to read the current- and voltage-values. It then both regulates the welding-process and changes state if necessary. Since Tig welding involves gas the gas-valve is also regulated in the `State()` function. Messages are sent to the WDS node to inform about changes including opening and closing the gas valve.

[10] Instantiation of static variables isn't included since it contains only library-code that isn't analysed.
[11] The dots in the figure represent other functions being called that aren't depicted since the graph would be too big.

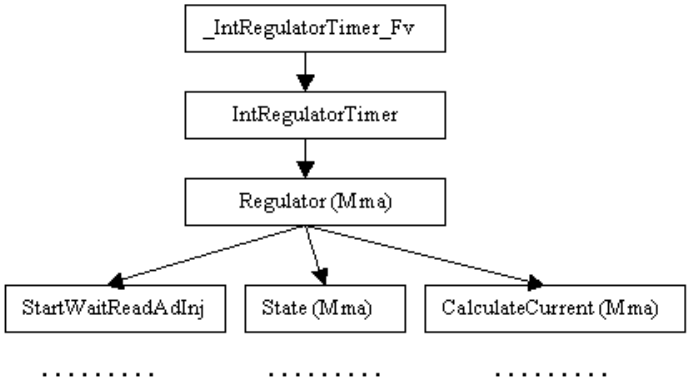

Figure 5.3: The call-tree of the Regulator-interrupt with Mma

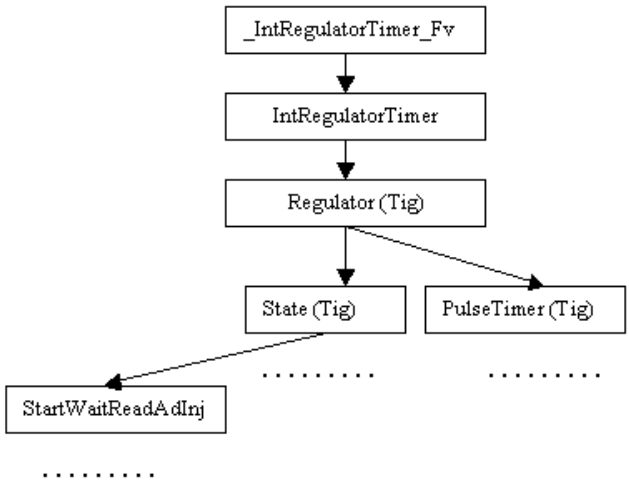

Figure 5.4: The call-tree of the Regulator-interrupt with Tig

The following code characteristics have been found for the Regulator-interrupt if Mma is used as the welding method:

- 30 functions of which 21 are non-library functions. The call-depth is 11.
- 7 loops of which 2 are in non-library functions. One of the non-library loops is very simple; it is just a for-loop copying data that iterates 8 times. The other loop is a bit trickier. It is a while-loop that iterates as long as the function `BusyAdInj()` is true but it also has an upper loop bound of 100000 times.
- No dynamic branches.
- No recursion.

5.3 Component-based Code

5.3.1 The ACC code

ACC stands for Adaptive Cruise Control and is a part of a project to build components in C-code and then automatically combine them into different tasks. The reason for testing this code with static WCET analysis is to see if there are any specific problems with testing just component-based software and also to see how much overhead there is in the tasks. Since the ACC code was just a way of testing if the conversion from components to tasks worked the code is very simple. The code has never been used as a real ACC but it has been downloaded to a target system for tests. In order to get some load on the system they have added some loops in some of the components that doesn't actually do any work. The tasks are formed by a number of components put together. Only components that have the same period or for other reason can execute after each other can be put together to form a task. The components are put together in tasks as illustrated in Figure 5.5.

This is the design of the system:
SpeedLimit: responsible for adapting the speed to current speed limit if there is one and also triggers *ObjectRecognition.*

ObjectRecognition: responsible for detecting if it is a obstacle in front of the vehicle and adapt the speed or even brake according to the relative speed to the obstacle. It also triggers Mode Switch and informs *Mode Switch* if *BrakeFunc* is needed or not.

Mode Switch: is used to trigger the execution of the ACC Controller assembly and the Brake Assist component, based on the current system mode (ACC Enabled, Brake Pedal Used) and information from Object Recognition.

BrakeFunc: responsible for braking the vehicle hard if an obstacle may cause a collision.

LoggerOutput: responsible for all logging and display the ACC status for the driver.

ACC Controller: this assembly is responsible for calculating the throttle level. It consists of the two assemblies *Distance Controller* and *Speed Controller*.

Distance Controller: this assembly is responsible for adjusting the throttle level according to the distance to any obstacle in front of the vehicle. It consists of the two components *CalcDistOutput* and *UpdateDistState.*

Speed Controller: this assembly is responsible for adjusting the throttle level according to the speed limit. It consists of the two components *CalcSpeedOutput* and *UpdateSpeedState.*

CalcDistOutput: is responsible for adjusting the throttle level according to the distance to any obstacle in front of the vehicle.

UpdateDistState: responsible for updating the distance state.

CalcSpeedOutput: responsible for setting the speed to the selected speed or the maximum speed allowed if the selected speed is higher than the maximum speed allowed.

UpdateSpeedState: responsible for updating the speed state.

The ACC code is very simple in its structure so it hasn't been investigated for different kinds of characteristics.

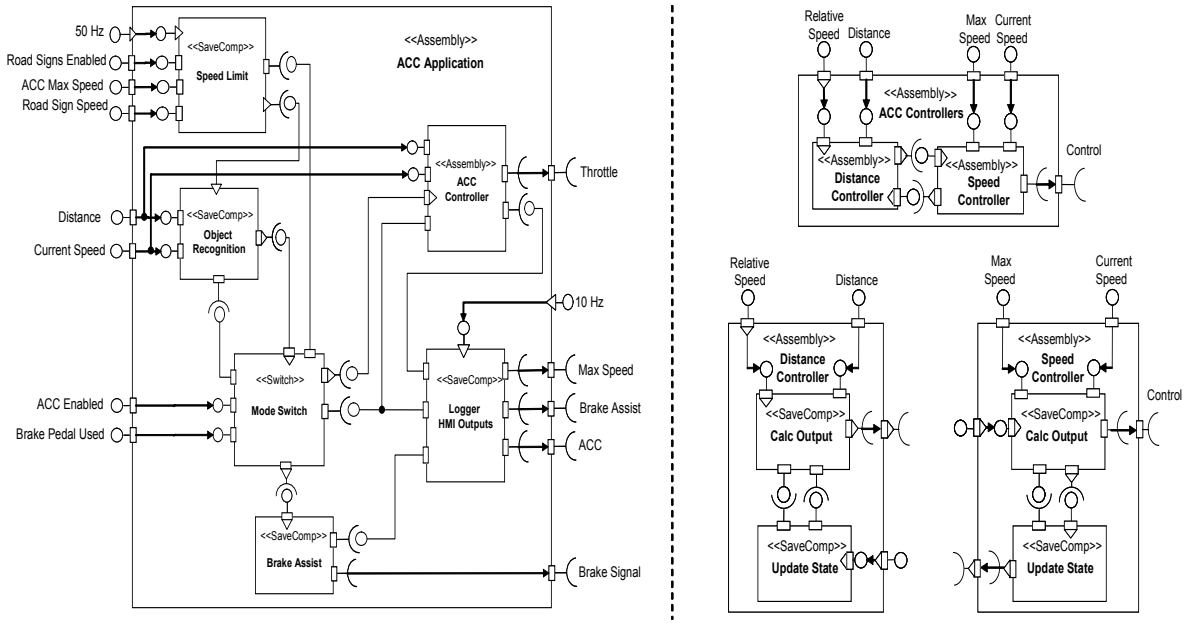

Figure 5.5: The ACC application implementation

6 Static analysis with aiT

This chapter is about the actual analysis performed using aiT. It will describe for each code snippet which problems that came up and how they were solved.

All the analyses below contains three basic annotations. One is to specify that the compiler TASKING is used, one is to set the clock rate to 20 MHz and the last is to set the context specifications. The three annotations are the ones written below. There can be some different parameters to the `interproc flexible` annotations but nothing that would affect the WCET value in a big way.

```
clock exactly 20 MHz;
compiler "c166-tasking";
interproc flexible, max-length = 3, max-unroll = 8, default-
unroll = 8;
```

The internal clock frequency of the microcontroller is 25 MHz but ESAB have put a crystal on the circuit board changing the clock frequency to 20 MHz instead. It is very important to have the correct clock frequency set because aiT recalculates the time from cycles to time by simply dividing the cycles with the clock frequency and aiT has no way of knowing if the clock rate is set correctly or not.

It is also important to tell aiT the addresses of the Context Pointer, System Stack Pointer, User Stack Pointer and the Data Page Pointers since it can have a big influence on the WCET value obtained. Several of these values can be found in the map-file that is created after the compilation and linking of the program. It isn't always important to know the exact address of the pointer since aiT uses this information to see how long time it takes to access these pointers and this is only affected by which memory the pointers are in and not the exact adress.

More information about the different annotations used can be found in Section 4.1.1.

6.1 CAN-interrupt

The CAN-interrupt is responsible for taking CAN-messages from the CAN-controller and placing them in an incoming buffer that can later be read by the main loop (see Section 2.7 for information about the ESAB Welding System).

6.1.1 CAN-interrupt on WDS

The structure of this code together with its characteristics of it is described in Section 5.1.1. This code was pretty hard to understand at first since it has an advanced structure. The executable file is quite large so each analysis takes about 24 minutes (the conversion from executable to flow information take most of this time); so it was very time consuming to find the correct annotations. As described in Section 5.1.1 there are three types of CAN-messages that can be received. They are *MESSAGE*, *MESSAGE15* and *STATUS*. The WCET have been calculated for the handling of all three types of messages. These three analyses will be described separately below. All analyses have also been performed on compiled code with different types of optimisations and the times can be seen in Table 6.1.

MESSAGE

The most important message type is *MESSAGE* since it is by far the most common type received. To be able to get a correct WCET value 4 loop bounds had to be set manually by annotations. One of them was on a malloc-function that isn't included in the final WCET path and the others are simple for-loops with fixed iteration counts. All errors are handled by the function `_AddAndLogError__12ErrorHandlerFUcN21()` and since errors shouldn't be included in the analysis the following annotation was used:

```
snippet "_AddAndLogError__12ErrorHandlerFUcN21" is never
executed;
```

This annotation means that all execution paths leading to the function `_AddAndLogError__12ErrorHandlerFUcN21()` being called are excluded from the analysis. This is a much better way than to manually for each error-check say that an error hasn't occurred. In the function `Instance()` in the class Communication an instance of the class Communication is created if there isn't already one created. The creation of the instance is pretty advanced since the command new is used which means dynamic memory with calls to malloc etc. Therefore it was assumed that the instance was already created and this was done with the following annotation saying that a jmp-instruction should always be performed that jumps over the code that creates the instance. The function `Instance()` is called from several places in the code and the variable is only created the first time so therefore it is acceptable to exclude this part from the analyses. Dynamic measurements also confirmed that this code isn't executed.

```
condition "_Instance__13CommunicationSFv" + 1 branch is always
true;
```

The hardest thing was to make aiT calculate how long it would take to receive just one message. The CAN-interrupt can receive up to 6 messages on one interrupt but since it almost every time only receives one message only that scenario was analysed. If no flow-annotations was made it calculated the worst case which of course is receiving 6 messages but since that probably never happens and since CC Systems was more interested in the normal case (receiving one message) flow annotations that forced aiT to only receive one message was made. Since the structure of the code was pretty advanced flow-annotations had to be made in three different functions. Most of the flow-annotations used were to set a relative execution count for a basic block compared to the first instruction in the interrupt, which means that for each time the interrupt is executed the basic block will be executed for the stated amount of times. The first two flow annotations had to be made in the `Interrupt()` function telling aiT that the loop in `Interrupt()` would iterate 2 times, one time to receive an ordinary message and the second receiving an empty message and therefore terminate the loop and the entire interrupt. In the function `PutMessageOnBuffer()` three annotations had to be made. Two of them were flow-annotations telling aiT to receive a CAN-message one time and to receive an empty message the other time `PutMessageOnBuffer()` was called. The third annotation was to force aiT not to think that the message received was a *STATUS* message, as it would do without the annotation since it represented the WCET path in `PutMessageOnBuffer()`. The third annotation simple said that the function `CheckBusOff()` wasn't called and since that call is just in the *STATUS* path that path was excluded from the WCET path and the *MESSAGE* path was selected by aiT instead. The final

function where flow-annotation had to be made was the function `GetMessageFromCanController()`. In this function three flow-annotations had to be used. The first flow-annotations were to make the function execute a basic block containing a jmp-instruction both times the function was called. The two other flow-annotations was to make the function select the path corresponding to receiving a *MESSAGE* one of the times the function get executed and also make the function receive an empty message the second time it is called. Since the basic annotations for compiler, clock frequency and context specification were used the total number of annotations to get a WCET value for receiving a single CAN-message of the type *MESSAGE* was 17. Different compiler optimisations were also analysed to see how much extra work it would create to change optimisations. The same annotations were used but the parameters in the annotations in form of addresses had to be changed. Since the structure of the code didn't change very much this was a relatively easy job. The times for all the analysis of receiving a CAN-message of the type *MESSAGE* with different optimisations can be seen in Table 6.1.

MESSAGE15

The message type *MESSAGE15* on the WDS node is used for receiving service messages from the ESAT node (see Section 2.7). The annotations used to get a WCET value on this type of message are pretty much the same as analysing a message of type *MESSAGE*. There is one address in one of the flow-annotations that is different. This annotation tells aiT to use the path that calls the function `ReadMessageObject15()` instead of `ReadMessageObject()`. The total number of annotations is therefore 17 just as in the case with the message type *MESSAGE*. When different compiler optimisations were tested the number of annotations didn't change but the addresses in the annotations changed. The optimisation 'speed' required most work since some loops and jumps changed quite a bit. The optimisation 'size' didn't require much job since the structure was pretty much the same of with default optimisation. The times for all the analysis of receiving a CAN-message of the type *MESSAGE15* with different optimisations can be seen in Table 6.1.

STATUS

This message type is used by the CAN-controller itself to send messages telling the system that something is wrong. There are two types of errors that can occur and they are *warning* and *bus off*. *Warning* means that there has been trouble in sending a message and *bus off* means that there have been so many warnings that there must be a more permanent fault on the bus. First a *STATUS* message with *warning* and *bus off* both being false was analysed despite the fact that this couldn't happen. This was done because the code to handle the errors was very complex to analyse. To simulate to aiT that a *STATUS* message was received lesser annotations was demanded since the loop in the function `Interrupt()` is immediately terminated when a *STATUS* message have been received. This means that the functions `PutMessageOnBuffer()` and `GetMessageFromCanController()` are only called once instead of twice when receiving *MESSAGE* and *MESSAGE15* type messages. There are only 4 flow-annotations here compared to 7 with the other two types. There were still flow-annotations in all three functions `Interrupt()`, `PutMessageOnBuffer()` and `GetMessageFromCanController()` but there was just one in each of the functions `PutMessageOnBuffer()` and `GetMessageFromCanController()`. To get a WCET value of the *STATUS* message when both *warning* and *bus off* is false required 13 annotations. To be able to get a correct WCET value the case with both *bus off* and *warning* being true was also analysed. This was much trickier since it involved much more code and

the code was also much more complex. The WCET path when *warning* and *bus off* were both false involved 12 different functions while the execution path when both *warning* and *bus off* were true involves 32 different functions. The reason for this amount of functions is that each error on the WDS-node is logged and acted upon. 5 annotations were added to avoid instantiation of different classes as described for the Communication class above. The biggest trouble was however to set loop bounds on a loop that was called two times with different loop bounds. AiT didn't have a way of giving context dependent loop bounds at that point so that was a big problem. After some consultation with AbsInt the problem was solved by giving register-values were this function was called using annotations and then aiT could calculate different loop bounds for the different executions of the function. This only works with library routines where a pattern was implemented[12] and since the function in question was `memcpy()` this approach could be used. The user must however know which register corresponds to which parameter and aiT also have to implement a pattern for just that function to make it work. When the case with both *warning* and *bus off* was tested correctly there were no problem testing when just one of them occurred. One extra annotation had to be added so that aiT would exclude either *warning* or *bus off* from the analysis. The number of annotations needed to analyse *STATUS* when both *warning* and *bus off* were true were 22. When different optimisations were used some loops were changed when the optimisation 'speed' was used and all addresses that were given with absolute addresses had to be changed. There were also some trouble with the loop in `Interrupt()` when the optimisation 'size' was used, but is was pretty easy to fix. The times for all the analysis of receiving a *STATUS* CAN-message with different optimisations can be seen in Table 6.1.

Message type	**WCET**		
	Default opt	*Speed opt*	*Size opt*
STATUS (warning = busoff = 0)	71,75 µs	71,25 µs	72,00 µs
STATUS (warning = busoff = 1)	21,089 ms	16,667 ms	21,080 ms
STATUS (warning =1, busoff = 0)	10,223 ms	—	—
STATUS (warning = 0, busoff = 1)	10,223 ms	—	—
MESSAGE (one message with 8 byte data)	146,75 µs	145,90 µs	147,20 µs
MESSAGE15 (one message with 8 byte data)	139,85 µs	131,50 µs	140,80 µs

Table 6.1: WCET for CAN-interrupt on WDS with different optimisations

6.1.2 CAN-interrupt on PSA

The structure of this code together with its characteristics is described in Section 5.2.1. This code was much easier to analyse than the CAN-interrupt on the WDS node since the code structure seem more carefully constructed. In this interrupt flow-annotations only had to be made in the `Interrupt()` function which simplifies the analysis a lot. As in the CAN-interrupt on the WDS there are three types of messages that can be received and each of them is analysed separately. These three analyses will be described below. All analyses have also been performed with different types of optimisations and the resulting times can be seen in Table 6.2.

[12] The pattern is constructed by absInt and they have constructed patterns for some of the more common library routines

MESSAGE
This is the most common message type. Three flow-annotations are used in the `Interrupt()` function to make aiT simulate the receiving of a CAN-message of type *MESSAGE*. There are 2 static variables that are created the first time they are used and this process is pretty complex. Since it only happens the first time the code is executed the variables are assumed to be already created. The annotations used for not creating the 2 static variables are 2 "condition is always true" annotations. The total number of annotations used for obtaining a WCET value with this type of CAN-message is 8. The creation of one of the static variables has also been analysed. This add 48 µs to WCET, which is almost a 50 % increase. To analyze this message type with the optimization 'size' the same annotations are used if relative addresses have been used. If absolute addresses have been used the addresses have changed slightly and the addresses in the annotations have to be changed. If the optimization 'speed' is used the structure of the code is changed and the flow-annotations have to be changed. When all annotations needed were written the analysis required 2 annotations less than with default optimization and optimization 'size'. The times for all the analysis of receiving a CAN-message of the type *MESSAGE* with different optimisations can be seen in Table 6.2.

MESSAGE15
This message type is only used as a boot message when the system is simulated and is therefore never used on the read target. The reason for analysing this type of message is to see how difficult it is to analyse the different types of messages. The annotations for this type of message are the same as for an ordinary CAN-message. The only thing that differs is an address in one of the annotations telling aiT to call function `ReadMessageObject15()` instead of `ReadMessageObject()`. When the optimisation is changed to optimise for 'size' all relative addresses have to be changed but apart from that the annotations are the same. If the optimisation 'speed' is used the flow-annotations have to be changed since the code structure is different. There is one flow-annotation less with optimisation 'speed' than with the other two optimisations. There was also a problem giving one of the flow-annotation a relative address since there was a hole in the code, therefore the address had to be given from the end of the function by using the following address:
"_IntCan1__Fv" / "_Interrupt__13CommunicationFv" + 1 return - 3 call.

The times for all the analysis of receiving a CAN-message of the type *MESSAGE15* with different optimisations can be seen in Table 6.2.

STATUS
It is much easier to analyse the *STATUS* message type on this node compared to analysing it on the WDS node. Much less code is involved here and there is also no error handler here which logs the error and tries to handle it in some way. The only thing done when there is a *warning* or *bus off STATUS* message is to set a variable to true. The only annotations required except the three standard ones are two flow-annotations and two condition-annotations. The condition-annotations are only there to exclude the creation of the two static variables mentioned above. So in total 7 annotations are needed to calculate the WCET for this code with the default optimisation set. To analyse the code with the optimisation 'size' the only thing that had to be changed were absolute addresses and that wasn't a very hard considering the few annotations needed. When the optimisation 'speed' is used the same type of changes

with the absolute addresses had to be made. The times for all the analyses when receiving a *STATUS* CAN-message with different optimisations can be seen in Table 6.2.

Message type	**WCET**		
	Default opt	*Speed opt*	*Size opt*
STATUS (warning = busoff = 1)	38,30 µs	37,35 µs	38,50 µs
MESSAGE (one message with 8 byte data)	73,55 µs	72,40 µs	73,75 µs
MESSAGE15 (one message with 8 byte data)	63,70 µs	62,55 µs	63,90 µs

Table 6.2: WCET for CAN-interrupt on PSA with different optimisations

6.2 Regulator-interrupt

The regulator-interrupt is responsible for controlling the welding process. It calculates the correct current and also controls gas-valves and other peripherals such as the Wire Feeder node. The regulator-interrupt is constructed as a state-machine with the state representing different welding-states such as overtemp and stop and start welding. Section 6.2.1 will go into the actual analysis of the regulator-interrupt on the PSA node.

6.2.1 Regulator-interrupt on PSA

The structure of this code along with its characteristics is described in Section 5.2.2. Another approach was used when analyzing this code compared to analyzing the code for the CAN-interrupt on the different nodes PSA and WDS nodes. An ais-file was constructed for each welding method that covered all test-cases and then the annotations not needed for that specific test-case were simply commented out. Analyses were made on the Tig and Mma welding methods and these analyses are described separately below.

To select which welding method to use, a subclass to the class EMethod is called in the code. There is one subclass for each welding method. The user has to specify which welding method to use by giving an annotation telling aiT which function is called in the function `IntRegulatorTimer()`. The annotation looks like this if the method Mma is used:

```
instruction "_IntRegulatorTimer__5PMainFv" + 1 call calls
"_Regulator__4EMmaFv";
```

Mma

This welding method was analysed deeply. Figure 5.3 describes the call-tree of the Regulator-interrupt with the Mma welding method. The functions `State` and `CalculateCurrent()` have both a switch-statement that runs different code depending on the current state. The function `State()` can also change the current state. The problem of this analysis was to make aiT choose the correct states. AiT didn't understand that if the state *STOP* was used in `State()` it should also be used in `CalculateCurrent()` so this had to be done manually. To achieve this an annotation that basically said *STOP* in `State()`, also means *STOP* in `CalculateCurrent()` had to be added. The annotations looked like this:

```
flow sum "_CalculateCurrent__4EMmaFv" + 0x356 bytes /
"_State__4EMmaFv" + 43 branch is exactly 1;
```

The annotation says that the code corresponding to the state *STOP* in `State()` will be executed equally often as the code corresponding to the state *STOP* in `CalculateCurrent()`. To make it a bit easier to understand it can be looked at as an equation where X/Y=1 equals X=Y. This had to be done for all 6 states in the Mma welding process. To be able to control if the state was changed in `State()` annotations were added to prevent the execution to enter if-statements that would change the state. These annotations could then easily be commented out if the analysis should include a change of state. If the state should change in `State()` an annotation had to be made in order to inform aiT of which state to use in `CalculateCurrent()`. This annotation could look like this:

```
flow sum "_IntRegulatorTimer__Fv" /
"_CalculateCurrent__4EMmaFv" + 1 call is exactly 1;
```

This annotation says that the code corresponding to the state *START1* in `CalculateCurrent()` should be executed as many times as the actual Regulator-interrupt function i.e. one time. The final information needed by aiT is which state should be used in `State()` and this is done in the same way as in the annotations above.

An example of an analysis could be to measure the WCET for the state *START1* used in both `State()` and `CalculateCurrent()`. To do this, an annotation had to be used to inform aiT which state was active in `State()`. The annotations that tell aiT that the state *START1* should be used in `CalculateCurrent()`, if it is used in `State(),` also had to be given. Two annotations telling aiT not to execute the if-statements changing the state from *START1* to *STOP* or *START2* in `State()` were also needed. The annotations looked as follows (the lines that start with a '#' are just comments):

```
##Use START1 in state()
flow sum "_IntRegulatorTimer__Fv" / "_State__4EMmaFv" + 17
branch is exactly 1;
##START1 in CalculateCurrent() if START1 in State()
flow sum "_CalculateCurrent__4EMmaFv" + 1 call /
"_State__4EMmaFv" + 17 branch is exactly 1;
##Not to change to state STOP
condition "_State__4EMmaFv" + 20 branch is always true;
##Not to change to state START2
condition "_State__4EMmaFv" + 23 branch is always true;
```

In addition to these types of annotations there were also annotations for 3 loop-bounds and 2 annotations for turning of error handling. There was also one annotation that was used to give the correct register value to one instruction. In total there were 35 annotations in the ais-file but on average only 14 of them were used in one analysis and the rest was commented out. 10 of the ones used were always present and was used for loop-bounds and other things that were constant between the different analyses.

When using the optimization 'speed' 12 annotations had to be changed and for all of them the reason was that the addresses changed because of changes to the code-structure. If the addresses had been absolute all addresses had to be changed. There were also a problem with holes in the code and that made it necessary to give 2 addresses relative to the end of the

functions instead of the beginning. The compiler itself also created a new loop and that loop had to be bounded by an annotation. The optimization 'size' didn't demand any changes of the annotations compared to using default optimization. The only thing that changed when the optimization 'size' was used was the addresses, they all changed by a fixed number of bytes.

The times for all the analysis of the regulator-interrupt on the PSA node using MMA with different optimisations can be seen in Table 6.3.

State in State()	State in Calculate()	WCET		
		Default opt	*Speed opt*	*Size opt*
IDLE	IDLE	118,95 µs	119,45 µs	119,15 µs
IDLE	START1	149,50 µs	149,75 µs	149,70 µs
START1	START1	138,05 µs	137,85 µs	—
START1	STOP	153,80 µs	—	—
START1	START2	186,85 µs	—	—
START2	START2	138,05 µs	137,85 µs	138,25 µs
START2	WELDING	326,40 µs	—	—
START2	STOP	153,80 µs	—	—
WELDING	WELDING	264,25 µs	260,05 µs	—
WELDING	STOP	155,80 µs	152,30 µs	—
STOP	STOP	114,35 µs	—	—
STOP	IDLE	116,25 µs	—	—
OVERTEMP	OVERTEMP	113,15 µs	—	—
OVERTEMP	IDLE	117,1 µs	—	—

Table 6.3: WCET for regulator-interrupt on PSA using MMA[3] with different optimisations

Tig

This welding method wasn't completely analysed. It was only analysed to be able to have values to compare with the results from the dynamic measurements in [2]. Figure 5.4 depicts the call-tree of the Regulator-interrupt with the Tig welding method. This welding method only has one function that is dependent on the current state and this made the analysis much easier. Compared to the Mma welding method, another approach has been used to select the current state. The state is represented by an enumeration which in turn depends on which code to be executed in the switch-statement. In order to force aiT to execute a certain state the register representing the enumeration before the switch-statement has to be set to the correct value. The annotation below forces aiT to execute state *START2* since it is represented by the value 10 in the enumeration and the enumeration is in register 7.

```
instruction "_State__4ETigFv" + 2 call + 7 instruction is
entered with r7 = 10;
```

Two calls had to be annotated to call the correct function. A constructor that contains complex code like malloc was given an execution time and was therefore not analysed further. Three loop bounds were set by annotations and three annotations for turning off error handling were used.

6.3 The Component-based Code

Component-based code was tested to see if there were any particular problem when analysing this kind of code compared to other code. The code that was chosen for this analysis is code for an adaptive cruise control. The structure of this code is described in Section 5.3.1.

6.3.1 ACC

This code was very easy to test since it wasn't constructed to be an advanced adaptive cruise control, but rather a part of a project to build components in C-code and then automatically combine them into different tasks. Both the individual components and the tasks have been analysed so that overhead can be measured. The only extra annotation needed to analyse the components, besides the three for clock rate, context specification and compiler, is an annotation for loop bound in some of the components. The only purpose of this loop is to simulate load on the system. When the task are analysed the loop bound-annotations for the included components have to be added. Which components that are located in which task can be seen in figure 6.5. An annotation to end the analysis also has to be added in each task analysis since they all contain a loop that never ends. A branch instruction has to be redirected so that all functions are included in the analysis. One of the task analyses also includes an extra loop bound-annotation and an annotation to inform aiT that a function never returns. The results of the analyses can be seen in Figures 6.4 and 6.5.

Component	WCET
objectRecognition	4806,0 µs
speedLimit	14,5 µs
BrakeFunc	4240,0 µs
loggerOutput	41,9 µs
calcDistOutput	4261,0 µs
updateDistState	9,3 µs
calcSpeedOutput	19,1 µs
updateSpeedState	11,0 µs

Table 6.4: WCET for the different components in ACC

	WCET
Task0:	4494,7 µs
updateSpeedState	11,0 µs
calcSpeedOutput	19,1 µs
updateDistState	9,3 µs
calcDistOutput	4261,0 µs
overhead	194,3 µs
Task2:	5316,8 µs
objectRecognition	4806,0 µs
speedLimit	14,5 µs
overhead	496,3 µs
Task3:	4354,8 µs
BrakeFunc	4240,0 µs
overhead	114,8 µs
Task4:	247,7 µs
loggerOutput	41,9 µs
Overhead	205,8 µs

Table 6.5: WCET for the different tasks in ACC

7 Dynamic measurements

Dynamic measurements were done on the same code as the static ones. The purpose was to compare both the actual methods and the results. The dynamic measurements were performed by Yina Zhang in another MSc thesis work. This chapter will only give brief information on the made measurements. For more detailed information about the measurements, see Yinas thesis [2].

7.1 CAN-interrupt

The CAN-interrupt is responsible for taking CAN-messages from the CAN-controller and placing them in a queue. The CAN-interrupts on both the WDS and PSA node was analysed with dynamic methods. CC Systems has developed an own CAN-card that is connected to the computer and that can monitor the traffic on a CAN-bus and also send CAN-messages on the same bus. The software tool that let the user send CAN-messages and monitor CAN-traffic has been developed by CC Systems and is called CanTool.

7.1.1 CAN-interrupt on WDS

An oscilloscope was chosen to analyse the CAN-interrupt on the WDS node. The code characteristics and structure of this interrupt can be seen in Section 5.1.1. The WDS-node is responsible for the communication with the welder through the display and with buttons etc. It also has control over the system state. When this node was analysed it wasn't connected to any other nodes in the system so CanTool (se Section 7.1) was used to send CAN-messages to trigger the CAN-interrupt. The power to this node usually comes from the PSA-node but since it wasn't connected to the system a power supply have to be connected. The power supply was set to 12 V DC and was connected to the WDS node through the CAN-interface since it is where the power from the PSA comes. A machine-ID also have to be set for the node so that it knew which type of display and which buttons that was used. The card must first be reset by pressing down a couple of the buttons at the same time. Then a CAN-message was sent to set the machine-ID and this was done using CanTool.

To be able to measure execution times on the node the software has to be downloaded to it. Before this could be done extra code was added to turn on and off Light Emitting Diodes (LEDs). When the CAN-interrupt was called a red LED was lit and when the interrupt was finished a green LED was lit. This means that if the green LED was lit at least one CAN-message has been received. After adding these codepieces the entire code was compiled and linked. The resulting code was then downloaded on the memory of the circuit board through a serial port on the circuit board connected to a serial port on the computer. Now the system was ready to be analysed. A probe was attached to the LED that was lit when the interrupt starts and then a CAN-message was sent using CanTool to trigger the oscilloscope. Then the time was measured on the screen of the oscilloscope when the signal was high (the LED is lit). The precision is down to microseconds and this is enough for these measurements. All three types of CAN-messages were analysed. To be able to analyse a *STATUS* message the code had to be somewhat changed to simulate that s STATUS message has been received. For information about the actual execution times see [2]. There were also some attempts to use the LEDs to indicate which execution path that was taken but it proved to be pretty hard since there weren't many LEDs compared to possible execution paths.

7.1.2 CAN-interrupt on PSA

This interrupt was analysed with a logic analyser connected to the address bus. The code characteristics and structure of this interrupt can be seen in Section 5.2.1. The PSA node is responsible for regulating the actual welding process by regulating the current and turning gas valves on and off etc. The PSA node can't boot without the WDS node being present since it rely on the WDS node to answer a CAN-message it send during the boot process. The WDS and PSA nodes were therefore connected to each other via the CAN-interfaces, as they would be in the real system. The voltage needed by the PSA node is 42 V AC and this was supplied from a special power supply connected to a port on the PSA node. The power supply that was used when analysing the WDS node wasn't needed anymore since it now got its voltage through the CAN-interface from the PSA-node. There was some trouble when the logic analyser was connected to the address bus. Firstly, the only pins big enough to connect the pins on were on the memory chip. But since the memory chip was addressed word-wise and not byte-wise, the clip from the logic analyser that represented bit 0 of the address bus wasn't connected at all and the second clip from the logic analyser was connected to address pin 0 on the memory chip. This was done to be able to get the correct addresses.

Since the address bus was monitored the code didn't have to be changed before it was compiled, linked and downloaded. The software was downloaded on the same way as on the WDS node. To analyse the CAN-interrupt either the start-address of the actual trigger or the address in the interrupt vector associated with the CAN-interrupts could be used as a trigger in the logic analyser. When a CAN-message had been received a trace of all addresses on the address bus was saved in the logic analyser. The execution time could easily be found by simply finding the last address of interrupt and see which time it had relative to the time of the trigger address occurring on the bus. The time is presented in ns and the smallest sampling rate possible was 8 ns. The trace could also be downloaded on a floppy disk so that it could be further analysed on a computer. Since the entire trace was saved the execution path could also be analysed with the help of a debugger or a control-flow graph.

There was a problem with testing the message types *STATUS* and *MESSAGE15*. The boot up process demanded the node to be able to send ordinary messages (*MESSAGE*) so the code couldn't be forced to always interpret a message as *MESSAGE15* or *STATUS* message. The *MESSAGE15* message isn't very interesting to analyse anyway since it is only used as a boot message when simulating. Since there is a constant flow of messages from the WDS node to the PSA node the CanTool was never needed to trigger this CAN-interrupt.

7.2 Regulator-interrupt

The regulator-interrupt is responsible for regulating the actual welding process by regulating the current and turning gas valves on and off etc. The interrupt is constructed as a state-machine with the state representing different welding-states such as overtemp and stop and start welding.

7.2.1 Regulator-interrupt on PSA

This interrupt was analysed with a logic analyser connected to the address bus in the same way as for the CAN-interrupt on this node. The code characteristics and structure of this interrupt can be seen in Section 5.2.2. The system had exactly the same setup as when the

CAN-interrupt was measured on this node. The WDS node that was used during the testing supported the welding methods Mma and Tig so these were the only welding methods that could be analysed using dynamic methods. Since the correct peripherals weren't connected a correct welding process couldn't be simulated. Some of the different states could be measured but only states that are used when the system isn't in welding mode. The state that was analysed without changing the code when the Mma welding method was used was overtemp. When Tig was used the only state that could be analysed without changing the code was idle.

8 Results

This chapter will go into the results of the thesis. In Section 8.1 the different methods are compared in order to see how time consuming they are and how hard it is to obtain a good WCET estimation. Section 8.2 will compare the results from dynamic and static analysis to see how big the overestimation of the static analysis is likely to be. Potential uses for these methods beside WCET analysis are discussed in Section 8.3.

8.1 Comparing Static and Dynamic WCET Analysis methods

CC Systems wanted to find out which analysis method that is best suited for their needs. It should be relatively easy to obtain WCET values and the values should be proven to be reasonable accurate. The general timing behavior of the program was also of interest.

It takes some time to get familiar with the aiT WCET tool, where to find all the settings, which annotations there are, how do they work and so on. It also took some time to be able to get the executable in a format that could be read by aiT. Once all the settings were correct and the correct executable has been loaded into the toolit is quite easy to do WCET calculation. It was often similar problems with loop bounds etc that occured in every analysis. The result produced by aiT can't just be accepted directly since it may include impossible paths or error-handling routines that shouldn't be included etc. This means that the user must have a very good understanding of the code in order to get a correct WCET value. A big advantage of this method is that it is very easy to redo the calculations, since all that is needed are different files. This is very useful if an error in the analysis is detected or the program code is changed. Thereby, the user doesn't have to start from scratch and redo all work for every new analysis. The graphs presented by aiT are very useful and can be used to see the structure of the code, understanding the code, finding out the path that lead to the WCET and also getting the execution time of each individual basic block.

An advantage that all dynamic analysis methods share is that they only give execution times on possible execution paths. With aiT the user can't be absolutely sure about the execution path leading to the WCET is possible if the program isn't very simple or dynamic measurement can confirm it. Dynamic methods can also give a general view of the timing behavior of a system with best case execution time and average case execution time. The execution times obtained can even be used to create a graph of the execution time distribution of the analysed code snippet. This sort of information cannot be given by aiT. A big problem with dynamic measurement is that the system must be connected and all needed hardware parts have to be present and working properly. Sometimes it can be very hard to do this and perhaps the hardware or other codes needed aren't developed yet. When aiT is used the only thing that is needed is the code to be analysed. The biggest problem when using dynamic measurement is that is hard or even impossible to guarantee that the execution time measured is the WCET value. Often it is very hard to generate the worst case since it may only occur rarely. To make sure that a value can be used as a WCET value a safety margin is often added to the measured value to make sure that it is greater than the WCET. But there is no way of knowing how big the safety margin should be to ensure a safe WCET value.

The oscilloscope is one of the most frequently used ways of obtaining WCET values. Once the system is set up it is pretty easy to measure different code snippets. The problem is that the code usually has to be changed between each measurement. The fact that the code has to

be changed in order to measure can mean that the system behavior is affected. Another problem is that it is very hard to see which execution path that was measured or if preemptions have occured.

The logic analyzer can be used in the same way as an oscilloscope and has then the same advantages and disadvantages. In this thesis the logic analyzer was however connected to the address bus in order to obtain WCET estimations. If the address bus is monitored there is no need to change the code in order to do measurements and this is a big advantage compared to using an oscilloscope. Since all traffic on the address bus can be recorded it is also possible, but not very easy, to see which path that was executed. Since all memory accesses are time stamped the execution times for different code pieces can be calculated separately. It is also easy to do many different measurements on the same code since the only thing that has to be changed is the trigger address.

8.2 Comparing Static and Dynamic WCET Analysis results

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

The first values that were compared were the values for the CAN-interrupt on the WDS node. Since the dynamic measurement method used for this code was the oscilloscope the execution path was hard to find out but there weren't many different paths to execute so the values should be comparable with the ones obtained by the static method. The three different message types were compared and the two types *MESSAGE* and *MESSAGE15* were compared with 1 byte of data and with 8 bytes of data. The STATUS message had to be faked so it might be hard to validate the times for this type of message. The different values and the difference between the values can be seen in Table 8.1. The difference was between 4 and 8 percent for the two types of ordinary CAN-messages (*MESSAGE* and *MESSAGE15*) and that could be considered as acceptable. The execution path for the message type *MESSAGE* and *MESSAGE15* was forced at one place in the code in aiT to better correspond to the path taken when the dynamic measurements were made. The differences of the *STATUS* messages are between 19 and 117 percent and that should be considered as being too much. It wasn't possible to get a good WCET value using aiT when the *STATUS* message type was analysed. There are three possible scenarios when a *STATUS* message is received. The message can be informing the system that a *warning* has occurred, the bus is off or both. The code to handle these errors was hard to analyse with aiT. The analyses of these errors could include errors or impossible paths and that could be the reason to the big differences. The path taken by the oscilloscope for these errors could also be different since we didn't have all the information about which path was taken when the dynamic measurements was made. The values where both *warning* and *bus off* was false in the table should correspond to the same execution path and the difference here was much lower than for the other cases. However this scenario couldn't actually happen on the real system. So even if the execution path leading to the

values from the oscilloscope wasn't completely known the values from the static method seem pretty accurate for the ordinary message types.

Message Type	Conditions	Dynamic measurement	Static measurement	Difference
MESSAGE	1 byte of data	111 µs	116,15 µs	4,6 %
	8 bytes of data	135 µs	143,80 µs	6,5 %
MESSAGE15	1 byte of data	102 µs	109,25 µs	7,1 %
	8 bytes of data	130 µs	136,90 µs	5,3 %
STATUS	warning = bus off = 0	60 µs	71,75 µs	19,6 %
	warning = bus off = 1	9,75 ms	21,089 ms	116,3 %
	warning =1, bus off = 0	4,80 ms	10,223 ms	113,0 %
	warning = 0, bus off = 1	4,80 ms	10,223 ms	113,0 %

Table 8.1: Comparison of execution times of the CAN-interrupt on the WDS node

The dynamic measurements on the PSA node were made by the logic analyser. The execution path of the dynamic measurements was extracted from the traces and then aiT was forced to execute the same path so that the execution time could be compared correctly. The execution path in aiT was changed with different types of flow-annotations. The different values and the difference between the values can be seen in Table 8.2. The analysis is made both with and without memory annotations. The memory annotations are based on information from the traces and they tell aiT where some of the instructions read and write in memory. These annotations have a big influence on the execution times and there are still instructions without memory annotations that aiT doesn't know which parts of memory they access. This means that it is possible to get tighter values but it is a very time consuming job and there is no simple way of knowing that the memory accesses always access the same parts of memory. As seen in the table the difference is between 3 and 8 percent and that is acceptable. The difference is between 1 and 5 percent lower when memory annotations are used. The reason that the CAN-interrupt hasn't been analysed with memory annotations is that the difference was so low without memory annotations and that most of the destinations of memory accesses were found automatically by aiT.

Which interrupt	Dynamic measurement[13]	Without memory annotations		With memory annotations	
		Static measurement	Difference	Static measurement	Difference
Can	56,3 µs	58,60 µs	4,1 %	—	—
	75,1 µs	79,00 µs	5,2 %	—	—
Regulator Mma	99,5 µs	104,00 µs	4,5 %	102,90 µs	3,4 %
	107,2 µs	113,45 µs	5,8 %	110,60 µs	3,2 %
Regulator Tig	112,8 µs	119,30 µs	5,8 %	117,10 µs	3,8 %
	128,0 µs	138,20 µs	8,0 %	132,50 µs	3,5 %

Table 8.2: Comparison of execution times on the PSA node

[13] The values from the logic analyzer varies with about 0,5 µs so the values in the table aren't exact values

It was very hard to find the WCET path with dynamic measurements. Yina tried to force the execution to take the path aiT chose as the WCET path, but this required a lot of modification on the code. This could affect the system behaviour and it was also not certain that the execution times are correct. The overestimation of that analysis was 12,5 percent without memory annotations and 10,2 percent with memory annotations and that is much more than the differences in Table 8.2. The reason could be that the executed code was much larger than the code snippets analysed in Table 8.2. This could lead to bigger overestimations in aiT.

The values in Table 8.2 was the result of much work to give annotations to aiT since there is no way for aiT to know that the values are correct. The aiT tool was also improved during the thesis and this helped to lower the differences.

8.3 Usages of WCET analysis methods for CC Systems

CC Systems (CCS) wanted to know if the methods tested in this and Yina's thesis could be used for other things than just WCET analysis.

The times from the logic analyser and aiT can be used to find places in the code that is extra time-consuming and therefore a good place to optimise the code if optimisation is needed. The times can also be used to make simulations more time accurate. CCS has developed their own simulation technology that they uses to be able to simulate the functional behaviour of an implemented system before they download it to the target system. Time accurate behaviour has been added to this technique. This work has been done in another thesis [26] described in Section 2.6. The thesis focused on how to add the time accurate behaviour to the system but not on how to obtain the execution times needed to make the simulation time accurate. Times for entire interrupts can be obtained by all the methods used in this and Yina's thesis. However to get a good timing behaviour it is not enough to give a WCET value for the entire interrupt. The best thing is probably to give execution times for every basic block in the code to ensure a good timing behaviour, but this can introduce too much overhead in the system. There is a balance on how small pieces of code that should be measured. Both aiT and the logic analyser attached to the address bus can be used to obtain execution times for such kinds of small code snippets. The problem of suing aiT for obtaining timing for smaller code snippets is that all the times have potential overestimations and that can be a problem if the average rather than the worst behaviour is going to be simulated. The overestimations are however not very big as can be seen in Section 8.2. The logic analyser gives correct times but it can be hard to analyse all needed code pieces, without manipulating the system i.e. changing the code to force it to take different execution paths. With aiT it is easy to force the execution to take a specific path without affecting the system.

A part of the thesis has focused on giving the CAN-interrupts on both the PSA and WDS nodes, and the regulator-interrupt (when Mma is used) on the PSA node, correct timing behaviour. Breakpoints[14] were added in the code to give it the correct timing behaviour. Breakpoints were for example placed in large if-statements so that the timing behaviour would be different if the if-statement was entered or not. Breakpoints with different times were also added in the different case-statements so that the execution times would be different depending on which case-statement that was executed. All this was done to be able to get as

[14] See Section 2.6 for more information about breakpoints and Time Accurate Simulation.

correct timing behaviour as possible. One problem that was encountered was to get the interrupts associated with the main loop of the system i.e. share the same CPU-time. Different threads must be created and associated with the correct parent in order to make the simulation behave correctly. Once all the breakpoints were added and the interrupts were using the same CPU time as the rest of the node, the Time Accurate Simulation made it possible to step through the execution one breakpoint at a time. It was also possible to pause, speed up or slow down the execution. The times for the breakpoints were taken from aiT but times obtained by the logic analyser could also have been used.

WCET analyses methods could also be used to obtain other times than just WCET. Best Case Execution Time (BCET) or Average Case Execution Time (ACET) could also be interesting to know. AiT can't be used in a good way to obtain BCET since it sometimes overestimated the WCET and therefore can't provide a safe BCET value. The way of finding BCET is however not very different from finding WCET, but aiT is currently no supporting BCET computation. It is also impossible for aiT to calculate things like ACET since it haven't got a clue on the likelihood of different execution paths. To be able to analyse the general timing behaviour of the system and obtain values of BCET and ACET dynamic measurements are better suited than aiT. There can be a problem to obtain the BCET with dynamic measurements since it hard to find the path leading to it. However, in comparison to aiT there are no overestimations made, so the value will at least be closer to the real BCET value.

The traces produced by the logic analyser can be used to do functional testing or to correct functional errors. If a trace has captured an error it is easy to see the execution path and find the place in the code where the error occurs. These kinds of errors can often also be found with the help of a debugger but if the errors depend on some hardware failure the only way of finding the error might be to execute the code on the target system, and then a logic analyzer attached to the address bus can provide assistance in finding the error.

One advantage of aiT is that it is relatively easy to force the execution to take a certain path without affecting the system. This is useful if all possible execution paths in the system or all basic blocks should be analysed. Since the execution path leading up to the execution time in aiT is visible it is easy to see that all different paths or at least all basic blocks have been analysed. If aiT is used to get execution times for Time Accurate Simulation this approach is a good way of making sure that all the execution times needed actually have been calculated.

9 Conclusions

A part of the purpose of the thesis was to find out if it is possible to integrate the static WCET tool in CC Systems (CCS) development tool chain (see Section 2.1). The three methods tested in this and Yina's thesis [2] i.e. aiT, logic analyser and oscilloscope could all be used as a part of the development process at CCS. AiT can be used during the implementation part of the development process since it doesn't require a target system while the oscilloscope and logic analyser cannot be used until the target system is available. AiT is also the best method if a safe WCET value is desired. The values produced by aiT are also relatively tight so overestimations aren't a big problem. The overestimations are also getting smaller and smaller since aiT is constantly improved. There can be some problem to set all the correct settings in aiT so it can therefore be a good idea to compare the times with traces from a logic analyser or an emulator so that the results can be verified. The aiT tool could be integrated in the Construction phase of the development[15] and used in every iteration to evaluate the WCET of different parts of the system. A final timing analysis could also be made in the Transition phase and for that analysis a logic analyser or aiT could be used. The best result should however be obtained if both a logic analyser and aiT are used in the Transition phase.

To get times for CCS time accurate simulation aiT can be used during the development phase so that the worst possible timing behaviour is simulated. Once the system is downloaded on the target system a logic analyser can be used to get an even more accurate and tight timing behaviour. If the result from aiT can be considered tight enough the logic analyser wouldn't be needed to give these times but that depends on how time accurate the simulation have to be.

The logic analyser could be used when the general timing behaviour with ACET etc. is examined. It can also be used to give the execution time distribution of the different parts of the system. This would however demand that the system is running as it would in its real environment.

The traces produced by the logic analyser could also help to get tighter WCET estimates with aiT since it can for example show which parts of memory that certain instructions access.

The only thing where the oscilloscope is useful is when the system is functional and a fast WCET estimation is needed. The result from the oscilloscope is however undetailed and can't even be proven to just include the execution time of the code that should be measured, i.e. it can't show if a pre-emption has occurred. Using the oscilloscope also requires changing the code and that can affect the code and it also take some time to change the code and compile and link it between each new measurement.

The conclusion is that a combination of a static tool and a logic analyser is the best way to go since they can complement each other. AiT's strength is that it produces safe WCET estimates (as can be seen in Table 8.2) and the strength of the logic analyser is that it captures the overall timing behaviour of the system. The traces from the logic analyser can be used to get tighter WCET estimates by aiT and the graphs produced by aiT can be used to interpret the traces from the logic analyser.

[15] For more information about the development process at CC Systems, see Section 2.1

CCS was also interested in ways of making the WCET analysis as easy as possible. The following rules should be applied if aiT is going to be used for WCET estimations. When the code is developed the basic rules for programming embedded systems apply. The code-structure should be as easy as possible, goto-statements shouldn't be used and returns and breakpoints in the middle of loops should be used as infrequently as possible. Pointers should also be avoided as much as possible. Recursion and especially indirect recursion shouldn't be used if possible. Dynamic memory should be avoided as much as possible since the functions malloc and calloc are hard to analyse. The code should be well commented so that someone else than the programmer can understand the code and do WCET calculations. Loop bounds should be specified in the code if possible and if they aren't too obvious. All loops that are included in the WCET calculations have to have an upper bound so no non-terminating loops are allowed in the analysed code. The program flow should if possible be controlled from as few functions as possible to make the WCET analysis easier.

The object oriented programming approach used in the ESAB-code complicated the analyses somewhat. The dynamic memory allocations used at some places in the code made the analyses harder and inheritance also caused some problems. Object-oriented code is easier to learn so that is a plus. Overall you can say that there are both benefits and drawbacks with object-orientated code. The conclusion is that object-oriented code can be analysed with aiT if it is somewhat simple and doesn't contain too much inheritance and dynamic memory allocation.

The person that do these analyses in the future should preferable be the person that wrote the code or at least someone who have some idea how it is constructed. If this isn't possible the person that performs the analyses should be able to contact the person who wrote the code. It is very hard to get tight WCET values if the person performing the analysis doesn't have a good understanding of the code. The goal of doing static analyses by pressing a button and then the WCET value pops up is currently just a utopia and will probably never happen. However, in the future the static WCET analysis tools will probably be even better in finding loop bounds automatically and giving tighter WCET values

10 Future work

The regulator-interrupt hasn't been tested with all different welding methods so there are a lot of work that can be done in that area. All the different welding methods have different states and if all states and state-transition are to be tested there are a lot of analyses that have to be done.

If the entire ESAB welding system is going to be simulated time accurately there are a lot of calculations or measurements that have to be done. The code doesn't have to be studied as thoroughly as when WCET analysis are made but almost. If the entire ESAB welding system can be simulated time accurately it means that there are not many tests that have to be done on the target system since the timing behavior can be tested on a PC instead. It is very time consuming to test different execution paths so that all basic blocks are analyzed. It would therefore be great if aiT could give execution times for all basic blocks at ones.

AiT could of course also be used to calculate the WCET for other systems than just the ESAB welding system. The same goes for the logic analyzer.

Something that would decrease the workload would be if aiT could take a trace from a logic analyzer and display which path in the graph that it represents.

There is a problem when simulating periodic interrupts with the Time Accurate Simulation. The only way to make it work is to start a new simulation thread that pretends to execute for the time between the executions of the periodic task and it is a bit hard to make this work. It would be much better if a special type of sleep could be implemented that didn't affect the execution of the other treads and also could be paused, slowed down or speeded up as well as every thing else that is simulated with Time Accurate Simulation.

The aiT tool is very user friendly but it could get even better. It would be great if aiT could automatically find the relative address of instructions so that the user doesn't have to calculate that himself since it can be very error-prone. It would also be good if context-sensitive loop bounds could be set. The graphical presentation could also be improved by showing flow constraints set by the user and show places where aiT can't find the destination of memory accesses. It would also be useful if the actual analysis in aiT could be paused since it sometimes may take a long time and the user may want to do something else that requires a lot of CPU-time for a short while. Another thing that would be very useful would be if the project file from Tasking could be loaded directly into aiT so that all settings were read from it. This would save a lot of trouble with finding and applying the correct settings.

Bibliography

[1] CC-Systems (CCS) company homepage (2005)
URL: http://www.cc-systems.com/

[2] Zhang, Y.: *Evaluation of Methods for Dynamic Time Analysis for CC Systems AB.* Master´s thesis, Mälardalen University, Västerås, Sweden (2005)

[3] ASTEC WWW homepage
URL: http://www.astec.uu.se

[4] Vinnova WWW homepage
URL: http://www.vinnova.se

[5] Engblom, J., Ermedahl, A., Stappert, F.: *Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor.* Uppsala University, Dept. of Information Technology. Technical Report 2001-030. (Dec 2001)

[6] Byhlin, S.: *Evaluation of Static Time Analysis for Volcano Communications Technologies AB.* Master´s thesis, Mälardalen University, Västerås, Sweden (Sept 2004)

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

[8] Sandell, D.: *Evaluating Static Worst-Case Execution-Time Analysis for a Commercial Real-Time Operating System.* Master´s thesis, Mälardalen University, Västerås, Sweden (2004)

[9] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, Björn 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)

[10] Peterson, S.: *Porting the Bound-T WCET tool to Lego Mindstorms and the Asterix RTOS.* Master´s thesis, Mälardalen University, Västerås, Sweden (2005)

[11] Bound-T WCET tool and Tidorum company WWW homepage (2005)
URL: http://www.tidorum.fi

[12] Colin, A., Puaut, I.: *Worst-Case Timing analysis of the RTEMS Real-Time Operating System.* Research report IRISA, NoPI1277, (Nov 1999)

[13] Holsti, N., Långbacka T., Saarinen S.: *Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE software.* In Proc. of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457), (Sep 2000)

[14] Thesing, S., Heckman, H., Souyris, J., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: *An Abstract Interpretation-based Timing Validation of Hard Real-Time Avionics Software.* International Performance and Dependability Symposium 2003 (IPDS03), (2003)

[15] Healy, C., Arnold, R., Müller, F., Whalley, D., Harmon, M.: *Bounding Pipeline and Instruction Cache Performance.* IEEE Transactions on Computers 48 (1999)

[16] Thesing, S.: *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Universität des Saarlandes, Saarbrücken Germany, 2004

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

[18] Engblom, J., Ermedahl, A., Stappert, F.: *A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems.* In: Workshop on Real-Time Tools (RTTOOLS'2001), affiliated to CONCUR'2001. (Aug 2001)

[19] AbsInt Angewandte Informatik GmbH company WWW homepage
URL: http://www.absint.com/

[20] Rapita Systems Ltd company homepage
URL: http://www.rapitasystems.com/

[21] Bernat, G., Colin, A. Petters, S.: pWCET: *A tool for probabilistic Worst-Case Execution Time Analysis of Real-Time Systems.* Technical Report YCS-2003-353, Department of Computer Science, University of York, UK (Feb 2003)

[22] Gustafsson, J., Lisper, B., Sandberg, C. Bermudo, N.: *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)

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

[24] Heptane WCET Tool WWW homepage
URL: http://www.irisa.fr/aces/work/heptane-demo/heptane.html

[25] Colin, A., Puaut, I.: *A modular & Retargetable Framework for Tree-based WCET Analys.* Research report IRISA, NoPI1386, (Mar 2001)

[26] Nilsson, M.: *Time Accurate Simulation*, Master´s thesis, Uppsala University, Sweden (2001)

[27] ARTIST2 WWW homepage
URL: http://homepageartist.cs.uni-sb.de