Porting the Bound-T WCET tool to Lego Mindstorms and the Asterix RTOS

Samuel Petersson (spn99007@student.mdh.se)

Department of Computer Science and Electronics
Autumn 2004 - Spring 2005
Supervisors: Andreas Ermedahl and Niklas Holsti
Examiner: Björn Lisper

ASTEC
Advanced Software Technology

Tid rum

MRTC

Abstract

Static Worst Case Execution Time (WCET) analysis is a technique that computes an upper bound of the execution time of a program before runtime. Static WCET analysis has a potential to cut costs and save time in the development of real-time systems. These are systems that do not just demands a correct result, but also demands correct timing.

However, static WCET analysis has not yet reached the industry in large scale. Measuring the WCET is the most common method to use, although it can not guarantee to find the WCET. The main goal of this thesis work was to bring static WCET analysis into the education of new real-time systems developers, thereby making them aware of the benefits of WCET analysis tools.

Bound-T is a commercially available static WCET-analysis tool. Asterix is a laboratory framework made for education purposes, including a real-time operating system (RTOS) and a development environment. The target system used to visualize the work for the students is the Lego Mindstorms development kit, which includes a Renesas H8/300 processor.

This thesis describes how we ported the Bound-T WCET tool to the Lego Mindstorms and the H8/300 processor. It also describes how we used the resulting Bound-T version to analyze and derive timing bounds for selected parts of the Asterix OS.

The resulting Bound-T tool version will be used in real-time courses given at the Department of Computer Science and Electronics at Mälardalen University.

Acknowledgments

After many years of study, it is now time for me to reflect on what I have learned. Advanced mathematics, various programming languages, real-time systems, project management, networking, WCET-analysis and writing scientific reports.

Knowledge is very valuable, but what is even more important is friends. When I first moved to Västerås from the little town of Öregrund, I did not know anyone here. Now, I have more friends here than I ever had in Öregrund. These friends have given med help, support and joy over the years that have helped me a lot to accomplish my goals. In the beginning, I was terrible home-sick. That is not a problem anymore. I am very grateful for their help and friendship.

This thesis project have been quite long and have involve much work. My work involved learning a completely new programming language and to figure out how a complex tool worked, Bound-T. Niklas Holsti, the creator of the Bound-T tool, did an extraordinary job with supporting me in this task. Niklas was also very helpful when this report was written. I have learned a lot from him, it is really nice to have worked with a professional.

My supervisor, Andreas Ermedahl have been my rock and support in this thesis. He have done a great job with encouraging and helping me to solve different problems. Just like Niklas, answering my questions were never a problem. I highly recommend him as a supervisor to any other student.

The Asterix crew, Anders Pettersson and Daniel Sundmark have also been great help. Anders helped me a lot configuring Asterix and Cygwin.

My parents (Arne and Ingrid) and siblings (Annakaisa, Sakarias and Linus) have also been great support to me, waiting for their son and brother to finally finish his education.

A beautiful spring day in April 2005 - Samuel Petersson

Contents

Chapter 1

Introduction

The goal of this thesis work was to bring static WCET-analysis into the education of new real-time software developers. This should make them aware of the benefits of WCET-analysis, and hopefully, make it more common in the industry.

The thesis work were performed with close cooperation with the owner of the Bound-T WCET-analysis tool, Tidorum Ltd, Finland. The thesis was on D-level of 20 credits size, and was a part of the authors Master of Science education. The site of this project was the Department of Computer Science and Electronics at Mälardalens University, Sweden.

The rest of this thesis is organized as follows: short introduction to important fields in chapter 1, describing the porting process of Bound-T in chapter 2, analyzing Asterix in chapter 3, describe related work in chapter 4 and finally present conclusions and future work in chapter 5.

1.1 Real-time systems

A real-time system is a computer system where not only the *result* of the computation is crucial, but also the *timing*. Most real-time systems are located in *embedded systems* i.e., computers embedded in products. Examples are cars, aeroplanes and televisions. To illustrate a real-time system, consider the airbag in a crashing car. We do not want the airbag to expand to early, because then the airbag will be almost empty when the driver or passenger hits the airbag. Of course, we neither want it to expand to late, because then it will not have expanded enough when the driver or passenger hits it. We want it to expand in such an interval that can be considered correct.

Real-time systems are often divided into *hard* and *soft*. In hard real-time systems, no task or activity are allowed to miss its deadlines. If this happens, it can result in heavy financial loss, damage to equipment, loss of revenues, or injury or death for operators or users. The airbag system described above is an example of a hard real-time system. In soft real-time systems, meeting deadlines for tasks or activities are desirable, but occasionally missing a deadline will not cause the system to fail permanently. Mpeg decoding is an example of a soft real-time system. No one gets killed if a few frames are lost, but the picture might get flickering.

Real-time systems often interact with the environment using sensors and actuators[1]. Sensors sample some value of interest, these value will be considered as input to the real-time system, which calculates an output value, and action will be taken by the actuators.

The need of WCET analysis in real-time systems is easy to see, especially in hard real-time systems. Here, the system must be able to handle all possible scenarios, including peak load. The WCET is a vital parameter in tools and formulas that are used to calculate and verify the performance of such real-time systems.

[1]Motor, brake or other device that could change the behavior of a real-time system

1.2 Why WCET-analysis?

The *worst-case execution time* (WCET) is defined as the longest execution time of a program that could ever be observed when the program is run on its target hardware [22]. The WCET is very important to know in real-time systems, where tasks needs to have a bound of their execution time to provide safe operation of the system. This is especially important in hard real-time systems, where no tasks are allowed to miss their deadlines, due to overrunning their execution-time or by being preempted. These systems are most often present in embedded systems.

For a WCET bound to be valid in hard real-time systems, it must be guaranteed to be *safe*, ie. the value should be greater than or equal to the actual WCET. To be useful, they must also be *tight*, which means that any overestimations must be acceptable [22].

WCET values can be obtained by two main analysis methods: *dynamic* and *static* analysis.

1.2.1 Dynamic WCET-analysis

Traditionally, software developers have used testing to get information about WCET of their applications. This testing is done by measuring the execution time gained for different inputs. The WCET-bounds of this method are thus quite unreliable, because it is often impossible to test all program behaviors of a normal size application. Other drawbacks are that dynamic analysis equipment is often very expensive, and that it takes a lot of effort to make good measurements. Users of dynamic WCET-analysis should also have in mind that measurements never gives WCET-values with an safety margin, which needs to be added explicitly.

Methods for dynamic WCET-analysis includes inserting code for measuring or measuring points, and special measurements equipment like oscilloscopes and logic analyzers [36]. Unfortunately inserting code snippets for starting and stopping measurements, might inflict the execution time of the programs real behavior, making measured WCET bounds rather unsure. Furthermore, in some environments, you are only allowed to use exactly the same code that you have tested, thus disqualifying insertions of measurement code [38].

Oscilloscopes and logic analyzers are methods that do not require measurement codes to be inserted. These devices allow us to connect to the relevant signals at the hardware level. It often takes a lot of effort to find relevant signals and use such a device, and there is still the problem of finding the worst case input.

Benefits of dynamic WCET-analysis includes know-how of the engineers i.e., it is a method that have been used in practice for a long time. Another benefit is that many tools and devices are available.

1.2.2 Static WCET-analysis

Static WCET-analysis does not run the program, instead it tries to derive WCET bounds for a program by analyzing it. This method builds upon mathematical models of program behavior and hardware timing that not should underestimate, rather overestimate, the real WCET. The result is a WCET bound. Given that the models are correct the analysis will derive a WCET bound which is guaranteed to be safe.

In [22], static WCET-analysis is divided in to three parts:

- *Flow analysis* - finds out possible flows through a program by analyzing the source- intermediate- and/or the object-code.
- *Low-level analysis* - determines the timing behavior by analyzing the object code and target hardware.
- *Calculation* - sum up the results of the low-level analysis and the flow analysis to obtain a WCET for the analyzed program.

The task of the flow analysis is to determine bounds on the dynamical behavior of a program. This includes safe execution bounds on which functions get called, how many times loops iterate, if there are dependencies between condition-statements and more. This information produced

must be safe, which means that all feasible execution paths must be included. To be useful, the information should also be tight, which means that it should include as few infeasible execution paths as possible [22]

The low-level analysis analyzes the final executable program and the hardware upon which the program runs. The execution time for each atomic unit of flow, such as an instruction or basic block, are determined. Note that the low-level analysis must analyze the executable binary program to get the actual hardware program timing. Overestimation are sometimes made in the low-level analysis to ensure safe timing. Other things to take into consideration in the low-level analysis is hardware features such as caches and pipelines [38].

The final calculation of the WCET program estimate, is done with the program flow information and the low-level analysis as an input. There are three main categories presented in literature: tree-based, path-based and IPET (Implicit Path Enumeration Technique) [16].

Tree-based calculation generates the WCET estimate by traversing a tree that corresponds to the syntactical parse tree of the analyzed program.

Path-based calculation obtains the WCET estimate by deriving execution times for different paths in a program.

IPET is in short a method that is based on a representation of program flow and execution times using algebraic or logical constraints. Each node or edge in the graph are given a timing (from the low-level analysis) and a count variable. The latter will hold the number of times the node or edge is executed in the worst case. The result of the flow analysis is expressed as additional constraints on how many times certain nodes or edges the control-flow graph can be taken.

Benefits of static WCET-analysis includes safer and more precise results of WCET, less amount of work for calculating WCET once we have a good and verified tool, and that there is no need to set up the real system when performing the calculations.

Disadvantages with static analysis includes that a this is a new technology, there is only a small number of tools developed, and the number of supported platforms is rather small.

Static WCET-analysis is performed with different tools that have been ported to different hardware platforms to match the processor and hardware timing characteristics. One such tool is Bound-T which is described more in section 1.3. A large part in static WCET-analysis is to create a sufficiently detailed model of the target processor and chip, that can be used in the WCET analysis tool. This thesis will describe such work.

1.3 The Bound-T tool from a user perspective

Bound-T is a static WCET-tool, supplied by Tidorum Ltd, which provides support for the following features [5]:

- Bounds on *execution time*
- Bounds on *stack usage*
- Generation and visualization of *control flow graphs* and *call graphs*

The typical users of this tool are developers of real-time, embedded software. Bound-T is suitable for both hard real-time applications, as well as soft real-time applications.

Bound-T is independent of the programming language the source code is written in, as it performs its analysis directly upon the binary files [5].

Bound-T can be used on Linux and Microsoft Windows. Bound-T supports a couple of target platforms, including Intel 8051 and ADSP-21020 (Digital Signal Processor). Ports to ATMEL AVR and ARM7 is under development.

Bound-T performs flow-analysis using a Presburger-model of the arithmetic, a low-level analysis that can handle deterministic pipelines but not caches, and calculates the final WCET-value using the IPET-method. The Bound-T tool is shell-based, and have no graphical interface, but it can provide flow-graphs for visualization of the analyzed program. Configuration of Bound-T is performed by different user options. A wide range of these options are provided, for example for drawing graphs, giving assertions and tracing of the program flow.

Assertions are used to support Bound-T's flow-analysis, for example, by giving loop-bounds on loops that are not automatically solved using the Presburger analysis.

Assertions also lets the user give additional information to provide better analysis results, for example bounds on variables, loops or calls. Assertions are defined by the user and are given in a separate text file.

```
subprogram "binary_search"
    loop repeats <= 7 times; end loop;
end "binary_search";
```

The above example, taken from [26] shows an assertion in the subprogram `binary_search`, where the user have set a loop to iterate at most 7 times in the analysis.

An example of Bound-T usage is shown in Appendix A.

1.4 The Asterix and Obelix laboratory environment

Asterix [37] and Obelix [28] form a laboratory environment designed for education in real-time systems. It is developed and used at the Department of Computer Science and Electronics at Mälardalen University. The Lego Mindstorms [11] package is used in the real-time courses to visualize the students work. The Lego Mindstorms package consist of a main unit (called RCX) with a Renesas H8/3292 chip, featuring a H8/300 CPU, sensors, motors and many different Lego building parts (see section 1.5 for further details).

1.4.1 Asterix

The aim for the Asterix framework is to provide a real-time kernel that supports state-of-the-art real-time methods, such as Fixed Priority Scheduling for task scheduling, Immediate Inheritance Protocol for semaphore usage and interprocess communication with Wait and Lock-free channels. Asterix is a multitasking RTOS and supports preemption of tasks. Asterix replaces the original software on the RCX [37].

All these features make the system developer able to insert their own tasks to run on the Lego Mindstorms platform. Asterix has been designed so that the kernel overhead should be predictable. This is very important, because it is a principle of an RTOS. Asterix is a very small RTOS, and the source code is available for everyone to use in education or studies.

1.4.2 Obelix

Obelix is the development environment in the laboratory framework. It consists of a configuration tool, a down-loader, an up-loader and a GCC cross-compiler. The configuration tool allows the user to set up the Asterix system, and generates C-code for compilation with the kernel code. Properties that the user can set for their tasks are priority, period, offset, deadline and more. The user can also name their tasks, and set them to be periodic or aperiodic. The down-loader downloads the complete Asterix system on the target machine, while the up-loader allows the kernel to upload data to the development environment. The cross-compiler is a compiler that runs on your development computer and produces object code for the target system.

1.5 The target platform

1.5.1 Lego Mindstorms

Lego Mindstorms is a kit containing sensors, Lego bricks, motors and a control unit (RCX). With this kit we can build and program our own robots. The kit is widely used in real-time and embedded system courses in universities, often with modified software, which means that the original software will be replaced with custom-made software. It is suitable for educational

purposes, because it is an embedded system with simple design and the kit contains sensors and actuators that makes it possible to simulate real-time systems. Figure 1.1 depicts the RCX-unit and some sensors and actuators.

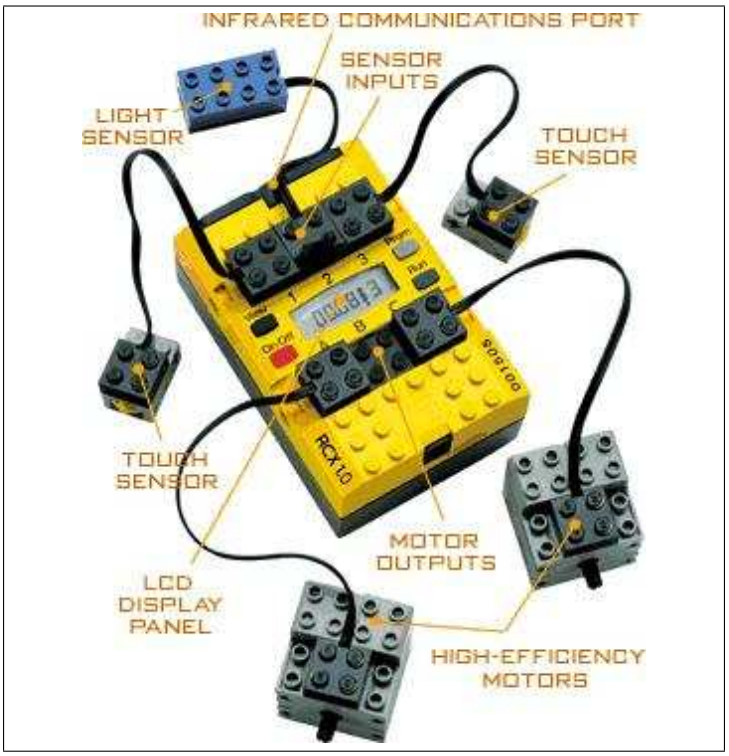


Figure 1.1: The Lego Mindstorms RCX-unit and some sensors and actuators

1.5.2 The Renesas H8/300 processor

The chip of the RCX-unit of Lego Mindstorms is Renesas H8/3292. It includes a H8/300 CPU which is the common core of all chips in the H8/300 series, such as H8/3292 and H8/3294 [12]. The different chipsets differ by size of RAM and ROM, operating voltage and memory types. An overview of the H8/3292-chip is given in [13].

The H8/300 core is a single-chip RISC (Reduced Instruction Set Computer) architecture running at 16 MHz and has eight 16-bit registers or sixteen 8-bit registers[2]. The H8/3292 has 16 kilobytes Read-Only-Memory (ROM) and 512 bytes of on-chip Random-Access-Memory (RAM) and an additional 16 kilobytes of external RAM in a separate circuit. The ROM includes functions for reading sensors, controlling motors and control the LCD-display. It has a serial communication interface and an 8-channel 10-bit analog-digital converter. The H8/3292 also features two 8-bit timers, one 16-bit timers and a watch-dog timer. It also has an IR-transceiver, used to downloading programs or communicating with other units.

Two important missing features are pipelining and caches. This makes the processor predictable, and simplifies the modeling of the processor timing.

[2]Accessed as RnL or RnH, where L is Low byte and H is High byte.

1.6 The Ada programming language

Bound-T is written in the Ada programming language. The porting work described in this thesis was therefore made in Ada. Furthermore, Ada includes a lot of interesting features, it is therefore motivated with an short introduction of the language.

Ada was developed by a request of US Department of Defense in the late seventies [15]. Their different military applications where developed in many different programming languages that not where common elsewhere in industry or academia. The cost for developing software under such circumstances was high; staff had to be trained and different development environments had to be developed and maintained. Another aspect is that it was hard to share experiences between different developers.

To cut costs, a standard programming language would be a good solution. None of the existing languages was considered good enough to be a standard, so a new language was decided to be developed. A set of requirements where set up to work after, and after a couple of years of testing in the science world, Ada83 was launched in 1983. Today, a version called Ada95 is used. The new version was developed during the early nineties, and was released in 1995. The new version supports object-oriented programming techniques.

Adas strong sides includes its package construction and strong typing by data structures, which makes it a very reliable language and makes programming in the large easier. It is a versatile language that is well suited both for big and small systems.

Ada is most often used in the military and spacecraft industry. These systems are mostly embedded systems, where efficiency and reliability are the most important aspects. Well known Ada programmed systems are Boeing and Airbus aeroplanes, and projects in the French train industry. These systems are very safe-critical, and thats a field that Ada is very well suited for.

Currently a new version is being developed, called Ada 2005. The aim for this update is to remove limitations and increase the usefulness. Other goals is to make the object-oriented programming in Ada more powerful, and to better adapt the language to real-time programming [4].

The Ada language is not so frequently used in small systems. This is because Ada gives much overhead, and this is not good at all for smaller embedded systems with sparse resources. One way to work around this is to use different subsets of Ada, like The Ravenscar Profile [17].

The Ada language is defined in [1] and an open-source compiler can be found at [2].

Chapter 2

Porting Bound-T

2.1 Bound-T general structure

Bound-T builds on a general model that consist of two parts. The first part is specialized for the host platform, while the second is more general and used for all the different host platforms. This is very useful when porting Bound-T to different host platforms, and encourages the adaption for many different processors.

Figure 2.1 illustrates the general Bound-T modules (packages Flow, Arithmetic and others), the target-specific Bound-T modules (packages Processor, Decoder and others), the target-specific support modules and the library modules such as the packages for reading COFF[1] files.

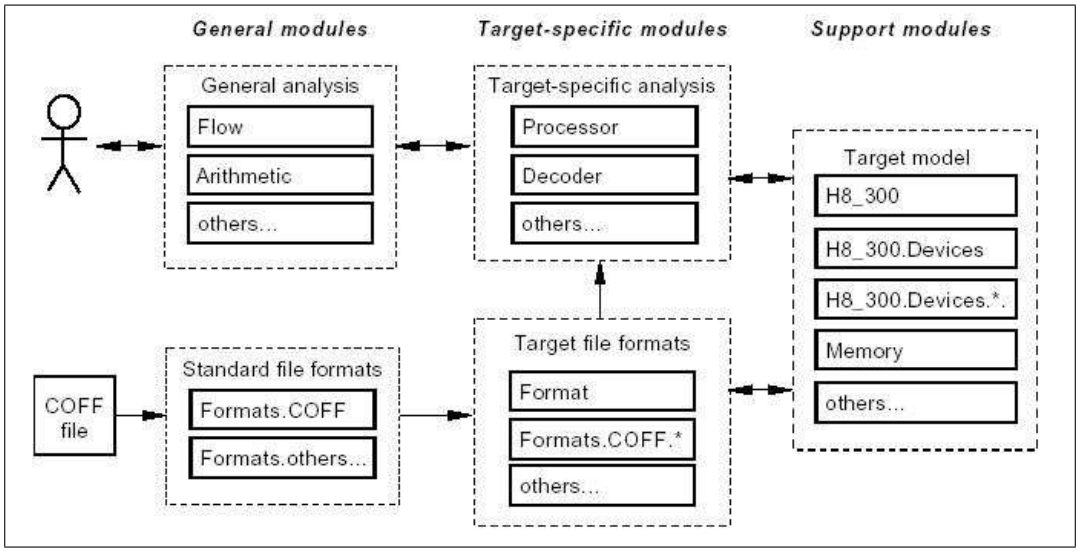


Figure 2.1: General structure of Bound-T.

The two main target-specific Ada packages have the following roles:

- The `Processor` package defines the target-specific aspects of the program model as a set of Ada types and related operations. These types include for example the type `Code_Address_T` that represents a code address and the type `Cell_Spec_T` that identifies a storage cell (a register, flag or memory location). The general modules use these target-specific types in the program model.

[1] COFF is short for Common Object File Format and is the binary file format used in the Asterix/Obelix tool-kit.

- The `Decoder` package provides the operations that read binary executable files, decode machine instructions and build the program model: the control-flow graphs and the call graph with all their attributes and decorations.
- The structure of the target processor is usually defined in a set of *support packages* that do not depend on the general Bound-T modules and are not directly used by Bound-T's general modules. In this work, the main support package is called `H8_300`. It provides types and constants that represent the H8/300 architectural concepts such as addresses, octets, words and instructions. The child package `H8_300.Devices` defines types that represent the properties of specific H8/300 chips, chiefly the memory map. Child packages of `H8_300.Devices` define derived types for chip families such as the H8/3297, defined in the package `H8_300.Devices.H8_3297`.

Target programs to be analyzed must be read from the binary executable file that the linker creates. Tidorum Ltd has a library of packages for reading standard executable formats such as COFF and ELF and some proprietary formats defined by compiler vendors. It is usually necessary to create also some target-specific extension packages that interpret the data from the executable file according to rules specific for this target, for example register numbers and symbol-table structures.

To port Bound-T to a new target processor, one must implement the main target-specific packages `Processor` and `Decoder` and as many support packages and file-format packages as necessary. Together, these packages define how the target processor is modeled and provide operations to translate a program from the executable file into a Bound-T program model.

Bound-T uses state-of-the-art methods to analyze a program. It uses Presburger arithmetics implemented with the Omega Calculator [30] to automatically calculate any possible loop bounds and IPET implemented with lp_solve [16] to sum up the WCET [27].

2.2 Processor timing in Renesas H8/300

The instruction times for H8/300 instructions is counted for in "execution states" in the H8/300 manual [31]. One "execution state" corresponds to one clock cycle. One clock cycle corresponds to 62,5 nanoseconds, which is 1/16 of a microsecond.

The instructions in the H8/300 processor generally have fixed execution times, the exceptions are EEPMOV, MOVTPE and MOVFPE, all of them are described in detail below [25]. The execution time for an instruction depends on its length, the addressing mode, the data width and the memory areas that are accessed. The H8/300 processor has no caches or pipelines, which makes the porting of Bound-T relatively simple.

2.3 Porting Bound-T to H8/300

To port Bound-T to the Renesas H8/300 CPU core, some changes in the earlier mentioned host platform part of Bound-T, was required: We decided to divide the adoption of Bound-T to H8/300 host platform into two steps:

1 *Binary instruction to Abstract instruction*
2 *Abstract instruction to Element in Bound-T flow-graph*

The binary instruction are read from the executable binary program, which consists of a set of binary instructions. An abstract instruction is an instruction that is decoded from the binary instruction and inserted in an abstract data structure. This abstract data structure is defined in package `H8_300`, which is closer described in section 2.1. An element in a Bound-T flow-graph is a representation of a decoded instruction in the flow-analysis model of Bound-T.

There is a number of reasons for making the adaption into two separate steps:

- ***Abstraction:*** For a developer that is new to the Bound-T model and the Ada programming language, this model will make it easier to understand the developing process and easier to learn Ada and the Bound-T structure.

- ***Testing:*** The two-step model simplifies the testing process. When the Binary instruction to Abstract instruction was implemented, we could test each part in isolation.
- ***Processor-families:*** Many processors belong to a "family" of processors that only differ by small details, such as the set of instructions. This implies that when porting Bound-T to another member in the same processor family, the changes required will mostly be located in the binary to abstract instruction step.
- ***Practical reasons:*** Dividing the program into different modules allow us to have a logically correct structure. The two-step model encourages this.

Tidorum Ltd provided an input interface to the first decoding step which allowed us to read one or several binary words from the executable. These words where then to be interpreted as instructions and inserted into the abstract data type (step 1). These abstract instruction data types where then interpreted, arithmetical effects and timing where added, and used to build a control-flow graph (step 2).

The main work of this thesis was concentrated at these two steps, the decoding of binary instruction to abstract instruction and the conversion of abstract instruction to element in the Bound-T flow-graph.

2.3.1 Development environment

The porting process of this thesis work was performed in Microsoft Windows XP on a normal PC. The main tool that was used for programming was the SciTE text editor from SCIntilla [14]. The compiler used was the Gnat version 3.15p. A Cygwin shell [7] was used to run Bound-T and Gnat.

2.3.2 Binary instruction to Abstract instruction

A more precise description of the work in step one is that we decode instructions. The input is one or two 16-bit binary words, in which certain bits need to be interpreted. Each instruction has a unique "instruction code" that need to be identified, along with address mode and parameters. Consequently, much work was spent reading and learning to understand the processor manual [31].

The function `H8_300.Decode` is responsible for decoding a binary instruction, represented as one or two 16-bit instruction words, to an abstract instruction, represented as an object of type `H8_300.Instruction_T`.

When the instruction code and its address mode and parameters are identified for an instruction, they are stored in the abstract data type. As an illustrating example, consider Figure 2.2, taken from the H8/300 Programming Manual [31]:

Addressing mode	Mnem.	Operands	Instruction code								No. of states
			1st byte		2nd byte				3rd byte	4th byte	
Immediate	MOV.W	#xx:16, Rd	7	9	0	0	rd		IMM		4
Register indirect	MOV.W	@RS, Rd	6	9	0	rs	0	rd			4
Register indirect with displacement	MOV.W	@(d:16,Rs),Rd	6	F	0	rs	0	rd	disp.		6
Register indirect with post-increment	MOV.W	@Rs+, Rd	6	D	0	rs	0	rd			6
Absolute address	MOV.W	@aa:16, Rd	6	B	0	0	rd		abs.		6

Figure 2.2: Table for the H8/300 instruction `move.w`.

The figure shows all the different kinds of addressing modes of the `move.w` instruction. The particular instruction of `move.w` that we should look into is the one located at row one. Under the column with the caption "Operands" in the Figure 2.2, we can see that this particular `move.w` instruction has a source that is a word of data and the destination that is a general register. For this particular instruction, there are 5 different addressing modes. When interpreting a number of bits, both the instruction codes that matches, and their parameters need to be identified. For example for the `move.w` instruction:

```
7903 0006 => mov.w #6,r3
```

The two 16-bit words operation code (`7903 0006`), which is read from the binary, gets decoded to the assembler instruction `mov.w #6,r3`. The value in the first octet is static[2]. In the lowest three bits in the second octet, the destination register is set (`r3`). In the second word, the immediate part is set (`6`). These entities will be placed in an abstract data type, which in this case means a data type for two-operands instructions:

```
type Two_Op_T is record
   Source      : Operand_T; -- The source operand that is read.
   Destination : Operand_T; -- Destination operand that is
                            -- usually written (except for
                            -- CMP) and is usually also
                            -- read (except for MOV).
   Width       : Width_T;   -- The data width.
end record;
```

The code for decoding the real entities and inserting the values in the abstract data structures looks as follows for the `move.w` instruction:

```
case High_Octet is
.
.
.
when 16#79# => -- 16#79# for value 79 of the
               -- high octet in the first word
               -- in hexadecimal.

   Check (Second_Quartet = 0 and not Third_Bit);
   Get_Word_Two; -- Gets the second word.

   Make_Move (
      Source      => (Kind => Immediate, Value => Word_Two),
      Destination => Second_Register (Word),
      Width       => Word);
```

What we see here is a `case-when` statement that checks the high octet of the operation code. All instructions that have a high octet of 79 is move-instructions, so in this case an special procedure (`Make_Move`) has been created to handle the insertions of entities into the abstract data type.

For an example of a bit-instruction with a little more complex instruction code decoding consider Figure 2.3, row two.

It shows a `band` instruction with an 3-bit immediate data and an address that is in a general register. For example, an instruction code like `7C30 7660` will be decoded to:

```
7C30 7660 => band #6,@r3
```

Investigating the table we see that the fourth to sixth bits in the second byte gives us our destination register, and that fourth to sixth bits in the fourth byte gives us our immediate data. These entities are inserted into the abstract data type for bit operations:

[2]Bits that are static will not be changed by changing general register or any other parameter.

Addressing mode	Mnem.	Operands	Instruction code								No. of states
			1st byte		2nd byte		3rd byte		4th byte		
Register direct	BAND	#xx:3, Rd	7	6	0 IMM	rd					2
Register indirect	BAND	#xx:3,@Rd	7	C	0 rd	0	7	6	0 IMM	0	6
Absolute address	BAND	#xx:3,@aa:8	7	E	abs		7	6	0 IMM	0	6

Figure 2.3: Table for the H8/300 instruction `band`. Taken from [31]

```
type Bit_Op_T is record
   Bit     : Operand_T;-- This operand defines the bit number.
   Host    : Operand_T;-- This operand is the octet in which the bit lies.
   Inverse : Boolean;  -- Whether the bit is used/stored with inversion.
                       -- Significant only for Bit_Invertible_Kind_T
                       -- instructions.
end record;
```

The code for inserting the entities, earlier found in the operation code 7C30 7660 looks as follows:

```
case High_Octet is
.
.
.
when 16#7C# => -- 16#7C# for value 7C of the
               -- high octet in the first word
               -- in hexadecimal.

   Check (First_Quartet = 0 and not Seventh_Bit);
   Get_Word_Two; -- Gets the second word.
   Check (First_Quartet_Word_Two = 0);

   case High_Octet_Word_Two is
   .
   .
   .
   when 16#76# =>

      Instruction := (
         Kind => Bit_And,
         Bit  => Bit_Immediate_Indirect (
            Inverse => Seventh_Bit_Word_Two));
```

All instructions with the `High_Octet` of `7C` are bit-manipulation instructions, but an extra case-when statements is used to divide them. Our operation code `7C30 7660` has an high octet of word two of `76`, and that is where we set our entities. A function is used to set them. This function have an argument to set the instruction to inverse, which depends on the seventh bit in word two.

The instruction set of the H8/300 architecture consists of 57 different instructions and it has eight different addressing modes. This implies that it is a lot of work to separate the different instructions and their different addressing modes. The fact that they only differ by a value of a bit sometimes forced us to look at one instruction, one addressing mode at a time. To complicate the matter further, no instruction has instances of all the addressing modes. For practical reasons

it was necessary to divide the instruction set into different subsets for later use in the architecture.

2.3.3 Abstract instruction to Element in Bound-T flow-graph

In step two, the control-flow graphs was built. In the normal case, edges and steps[3] are added, based on the earlier mentioned abstract data type. Cycles per instruction are also added, and the arithmetic effects of the different instructions is set. The arithmetic effect is the effect that an instruction has on internal parts in the CPU. The effect on registers, condition code register and other parts of the memory needs to be modeled. For example, an instruction like `move.w r3, r4` will have an arithmetic effect on the general register four, where the value gets updated by the value in register three, and on the condition code register, where the bits N, Z and V are updated[4].

The edges and steps are a vital part in the timing model of Bound-T, thus Bound-T uses IPET (Implicit Path Enumeration Technique), explained in section 1.2.2. The construction with nodes and edges will also make Bound-T able to build flow-graphs, using Dot GraphViz [9] to visualize the flow-graphs.

The procedure `Decoder.Decode` is responsible for translating the abstract instruction, as received from the function `H8_300.Decode`, to its model in the control-flow graph. Edges and steps are added for each instruction using procedures `Add_Edge` and `Add_Step`. There are different instances of these procedures, here are two of the most commonly used procedures:

```
procedure Add_Step (
   To       : in out Graph_T;
   Address  : in     Processor.Step_Address_T;
   Effect   : in     Arithmetic.Effect_T;
   Info     : in     Processor.Step_Info_T;
   Giving   :    out Step_T);
```

The `Add_Step` procedure adds a step with a given address and other attributes to the graph. The step do not have any edges yet, they will be added with the `Add_Edge` procedure.

```
procedure Add_Edge (
   To      : in     Graph_T;
   Source  : in     Step_T;
   Cond    : in     Arithmetic.Condition_T := Arithmetic.Always;
   Time    : in     Processor.Time_T;
   Target  : in     Step_T);
```

As we can see in this procedure, a general edge has a source and a target. The `Add_Edge` procedures binds already defined steps together. In this case of a `Add_Edge` procedure, a condition attribute, specifying under what particular condition a certain edge will be taken, can be given. Default value is an unconditional edge.

How is the time to the WCET added more in detail? It consist of two parts: A static time and a dynamic time. The static portion of the WCET is fetched from [31]. The dynamic portion is depending on the memory configuration, which can vary depending on which chipset that the user has chosen. This is described more in detail in section 2.4.2.

2.4 Special cases of the H8/300

2.4.1 Troublesome instructions of the H8/300

Most instructions where easy to handle, like the ones described in Sections 2.3.2 and 2.3.3. However, some instructions were more troublesome, like the `EEPMOV`-instruction. `EEPMOV` is an acronym

[3]Steps are the connections between nodes in flow graphs and in the Bound-T model.

[4]N is set to 1 if the data value is negative; otherwise cleared to 0. Z is set to 1 if the data value is zero; otherwise cleared to 0. V is cleared to 0.

Mode	MD1	MD0	Address space	On-chip ROM	On-chip RAM
Mode0	Low	Low	-	-	-
Mode1	Low	High	Expanded	Disabled	Enabled
Mode2	High	Low	Expanded	Enabled	Enabled
Mode3	High	High	Single-chip	Enabled	Enabled

Table 2.1: Modes and their settings.

for "Move data to EEPROM" and moves a block of data from a memory location specified in register R5 to a memory location specified in register R6. Register R4L gives the byte length of the data block. The instruction transfers one byte of data at a time, decrementing R4L and incrementing R5 and R6 during every data transfer. When R4L reaches 0, the transfer ends and next instruction is executed. The following pseudo code illustrates the function of `EEPMOV`:

```
if R4L != 0 then
  repeat
     store @R5 in @R6
     store R5+1 in R5
     store R6+1 in R6
     store R4L - 1 in R4L
  until R4L = 0
else next instruction;
```

`EEPMOV` is not believed to be used often in systems where Asterix is running, but the instruction is although implemented in this version of Bound-T. The implementation consist of three nodes instead the normal case of one node per instruction. This is done to simulate the loop and to let Bound-T use its loop-analysis to find the an upper iteration bound of the loop.

Two other special instructions in H8/300 are `MOVFPE` and `MOVTPE`. They are special in the way they synchronize with the peripheral clock. The access time of the peripheral clock is variable, thus these instructions are set to have an execution time equal to the worst access time to the peripheral clock. This is a safe time, but will for most cases be an overestimation.

Another interesting case was instructions that affect an octet register. They needed to be decoded into two flow graph nodes. The first node models the execution time and the effect on the octet register. The second node models the implied effect on the word register that contains the octet register. Only the first node were given an execution time.

2.4.2 Limitations of the WCET-analysis

The Renesas H8/300 processor can change the memory-configuration during runtime. This complicates the calculation of the WCET, because addresses and values located at different memories take different amount of time to fetch in different memory configurations [31].

The H8/300 supports four different memory modes [24] shown in Figure 2.1. Which mode that are set depends on the settings on the mode pins, high or low. They are depicted as MD1 and MD0 in the figure.

Mode 0 is inoperative in H8/3297 and should be avoided. Mode 1 and 2 are called expanded modes, which permits access to off-chip memory and peripheral devices. Mode 3 allows access only to on-chip RAM and ROM and the on-chip register field.

To simplify WCET calculation and to avoid overestimation, Bound-T assumes that the mode is not changed during runtime i.e., instructions for changing the mode are not to be used.

The actual memory map depends on which chip-model that is in use, for example H83292 (the one used in Lego Mindstorms) or H83297. Depending on the chip-model, the access times for different addresses will vary. The user must therefore specify what chip model that is in use.

Ada Modules	Ada statements and declarations	Ada subprograms
Processor and its children	504	53
Decoder and its children	1213	87
Format and Formats and their children	339	30
H8_300 and its children	1266	77
Others	13	2
Total	3335	249

Table 2.2: Software metrics in Bound-T.

2.5 Testing of decoding process

To verify the binary to abstract data type decoding, systematic testing was used. The testing consisted of an input-file with systematically chosen tests of each instruction with all their possible operands, and Ada-module that checked the incoming value and the output value from Bound-T. The file consisted of 154 different calls to an instruction-testing function in the `H8_300.Verify` testing-module. This testing-module helped to remove a lot of typing errors that might have been harder to find later in the porting process.

Here is an example of testing the earlier mentioned `move.w` instruction:

```
Try ("mov.w #6,r3",     16#7903#, 6,
    (Kind => Move,
     Two  => (
        Source      => (Immediate, 6),
        Destination => (Register, 3, Word),
        Width       => Word)));
```

What we see here is that the properties of this particular instruction has been hardcoded, and that `H8_300.Verifiy` invokes the `Try`-function to see if the output produced by `H8_300.Decode` is correct.

2.6 Software metrics

The size of the 23 target-specific Ada packages for the H8/300 port (version 2a2) are summarized in Table 2.2.

The first column gives the name of the Ada packages, the second gives the number of Ada statements and declarations, and the third column gives the number of Ada subprograms[5].

The numbers for the `H8_300` include the test package `H8_300.Verify` which contains 456 Ada statements and declarations divided into 14 subprograms. This package is not really part of the port. The actual decoding from binary instructions to abstract instructions (in the body of package `H8_300`) consisted of 315 Ada statements and declarations divided into 28 subprograms.

The total number of source-code lines in these 23 packages is 17057 including blank lines and comments. This can be compared to the total number of source-code lines in the 92 general Bound-T packages which is about 90000.

[5]Typical subprograms in Ada are functions and procedures.

Chapter 3

Adaption of the Bound-T tool to the Asterix laboratory environment

3.1 What can Bound-T offer?

Asterix is the laboratory environment for real-time courses used at the Department of Computer Science and Electronics. The most important features that Bound-T have to offer for Asterix are:

- *WCET for the student codes.*
- *WCET for the overhead of the operating system.*
- *Illustrative control-flow-graphs and call-graphs of the analyzed code.*

WCET for student codes (tasks in the system) and WCET for OS system-calls are important for giving overall system timing guarantees. These WCETs will be used to create schedules and to verify response times using scheduling analysis.

The Asterix framework have a mode for measuring execution times (a dynamic WCET-analysis method), However, the students have experienced difficulties to determine reliable WCET using measurements making static WCET analysis an interesting alternative [29].

3.2 System-calls in Asterix

To calculate WCET for the tasks in the system, it is essential to know how long time a system-call of the operating system will take. The Asterix real-time operating system is a small OS with a limited number of supported system-calls. In Table 3.1 the system-calls in Asterix that the user can invoke and their sizes are shown:

System-call	Lines of C-code
self	5
getSemaphore	66
releaseSemaphore	84
getReadPointerWF	26
getWritePointerWF	28
writeChannel	21
raiseSignal	39

Table 3.1: Table for system-call sizes.

There are of course more system-calls in Asterix, but they should not be invoked by the user, so we assume that they belong to the operating system overhead. Any other functions that gets invoked from the system-calls that the user can call will be included in the WCET-analysis.

Here follows a short description of all the system-calls that the user in Asterix can invoke:

- **self:** Returns the tasks own identification number.
- **getSemaphore:** When this call is invoked by the user, it will try to take hold of the specified semaphore for the calling task. The tasks priority will be set to the priority ceiling of the semaphore.
- **releaseSemaphore:** Simply drops the access to the claimed semaphore. Resets the tasks priority to normal.
- **raiseSignal:** Wakes up all tasks waiting for a signal.
- **getReadPointerWF:** Returns a pointer to the buffer where the value to read is.
- **getWritePointerWF:** Returns a pointer to the buffer where the value to write is.
- **writeChannel:** Writes to the waitfree-channel specified by getReadPointerWF return value.

3.3 Analysis environment

For each system-call a number of selected test cases was constructed, containing a number of semaphores, signals, tasks or anything else that could inflict their WCET. After compiling Asterix with these prerequisites, analyzes with Bound-T were performed.

The system-calls were analyzed using the following Bound-T options:

`-device=Lego:` to set the memory map and chip-set to H8/3292.

`-trace decode:` to have a trace of the instructions being decoded.

`-assert name.bta:` to use an assertion file.

The different system calls took about 1 to 2 minutes to analyze using Bound-T.

The minimum number of tasks in the system was two (Idle and test) when testing self, getSemaphore, releaseSemaphore and raiseSignal. When writeChannel, getReadPointerWF and getWritePointerWF was tested, at least three tasks were needed, Idle, test (Writer), test2 (Reader).

3.4 WCET and code-properties

The user-code in Asterix consists of one file for implementing tasks, and one Obelix configuration file for giving the task attributes like deadline, period, priority, etc. We can also set our tasks to be periodic or aperiodic. The tasks are implemented in C, and the system-calls are ordinary functions that the user can invoke. Parameters that will make the WCET for a system-call to vary are the number of signals, semaphores and tasks in our system.

Another factor that can inflict the WCET-analysis is function-pointers. These will be compiled to a JSR @Rn H8/300 instruction, which is a dynamic jump. Such instructions are hard to analyze, because the possible target addresses are hard to deduce statically. However, function-pointers are not commonly used by the Asterix users, but are used in the Asterix kernel to call error handler routines.

3.5 Analysis of Asterix system-calls

The result of the system-call analysis are presented in tables 3.2 to 3.8.

In Table 3.2 the results for the `self` system-call is shown. The system-call has a constant WCET. The result is not an surprise, since `self` is a simple function which only reads a single value from a variable.

The results for `getSemaphore`, `releaseSemaphore` and `raiseSignal` are given in Table 3.3 to 3.8. The WCETs of the `getSemaphore` are dependent on the number of semaphores which the calling task are using. The WCET of `releaseSemaphore` are dependent of the number of

System-call	WCET(cycles)	WCET(ms)
self	34	2,125

Table 3.2: System-call self.

Semaphores	WCET(cycles)	WCET(microseconds)
1	906	56,625
2	1264	79
3	1622	101,375
4	1980	123,75
5	2338	146,125

Table 3.3: WCET for system-call getSemaphore.

semaphores and tasks that are included in the Asterix system. The WCET of `raiseSignal` depends on the number of signals the calling task is using and the number of tasks that can get invoked by the particular signal.

Two system-calls are dependent on the number of tasks that are running in the system, namely `releaseSemaphore` and `raiseSignal`. Both calls the system-call `hal_yield` which is a function used for task-switching. The results of having two, three, four or five tasks in your system is interesting. The difference between the results is inconsistent. This depends on that if you change the number of tasks in your system, the Asterix kernel is recompiled and therefore some small changes can affect the WCET.

For three of the system-calls we were able to construct parametrical WCET formulas:

```
getSemaphore:
    WCET = 358 * (#Semaphores - 1) + 906

releaseSemaphore:
    WCET = 362 * (#Semaphores - 1) + Starting value for specific #tasks

raiseSignal:
    WCET = 112 * (#Signals - 1) + Starting value for specific #tasks
```

The first value in the formulas is a constant corresponding to the increase in WCET-value when the number of semaphores (or signals) in the system are increased by one.

The last constant, e.g. 906 in the getSemaphore formula, gives the "start-value" for taking one semaphore. A condition for these formulas to be correct is that the number of semaphores or signals in the system are at least one. The "Starting value for specific #tasks" is the starting value for the system-calls that is dependent on how many tasks that are running in the Asterix system. Because of the invariant difference between number of tasks, you need to set these starting values according to the tables 3.4 and 3.7.

The results on the waitfree-communication was expected. All three system-calls look quite the same, and therefore there is no surprise that their estimated WCETs are close to each other. The results of the waitfree-communication analysis are shown in Table 3.6.

Semaphores/Tasks	2	3	4	5
1	2844	3036	3200	3372
2	3206	3398	3562	3734
3	3568	3760	3924	4096
4	3930	4122	4286	4458
5	4292	4484	4648	4820

Table 3.4: WCET in cycles for system-call releaseSemaphore.

Semaphores/Tasks	2	3	4	5
1	177,750	189,750	200,000	210,750
2	200,375	212,375	222,625	233,375
3	223,000	235,000	245,250	256,000
4	245,625	257,625	267,875	278,625
5	268,250	280,250	290,500	301,250

Table 3.5: WCET in microseconds for system-call releaseSemaphore.

System-call	WCET(cycles)	WCET(microseconds)
getReadPointerWF	515	32,1875
getWritePointerWF	509	31,8125
writeChannel	588	36,75

Table 3.6: WCET for system-calls for waitfree-communication.

Signals/Tasks	2	3	4	5
1	2502	2694	2858	3030
2	2614	2806	2970	3142
3	2726	2918	3082	3254
4	2838	3030	3194	3366
5	2950	3142	3306	3478

Table 3.7: WCET in cycles for system-call raiseSignal.

Signals/Tasks	2	3	4	5
1	156,375	168,375	178,625	189,375
2	163,375	175,375	185,625	196,375
3	170,375	182,375	192,625	203,375
4	177,375	189,375	199,625	210,375
5	184,375	196,375	206,625	217,375

Table 3.8: WCET in microseconds for system-call raiseSignal.

3.6 Bound-T and Asterix in education

Asterix, Obelix and Lego Mindstorms are used in Real-time courses at Mälardalen University, and the use of Bound-T with Asterix will put static WCET-analysis on the students minds. These students are the future developers of embedded systems, and they will hopefully consider WCET tools when developing new applications.

Since the real-time courses at Mälardalen University started to use Asterix and Lego Mindstorms, around 500 students have completed approximately 200 robot projects [29].

Chapter 4

Related work

4.1 Related thesis and articles

We will here give short descriptions of three master thesis that where performed in the field of WCET.

4.1.1 WCET Analysis, Case Study on Interrupt Latency, for the OSE Real-Time Operating System

In [19] and [20], Martin Carlsson uses an existing static WCET-tool and adds on functionality for performing analysis on object-code for an ARM microprocessor. Analysis was performed on operating system code from the OSE RTOS.

The final result of this thesis was a functional WCET-tool, with pipeline analysis and able to draw control flow-graphs. One downside with the tool was that no loop-bound analysis is included, every loop-bound had to be set by hand.

4.1.2 Evaluating Static Worst Case Execution Time Analysis for a Commercial Real-Time Operating System

In [34] and [33], Daniel Sandell look into static WCET-analysis of the commercial RTOS, ENEA [8], with the AbsInt aiT-tool [3]. The aim for this thesis work was to investigate if a commercial WCET-tool was suitable for analyzing a commercial RTOS. The analysis was performed on selected system-calls and on regions where interrupts were disabled.

The conclusions of this thesis was that the tool used in this investigation was able to obtain WCETs for this kind of code, with more or less user interaction to support the analysis. For all selected code, the tool was able to obtain WCETs.

4.1.3 Evaluation of Static Time Analysis for Volcano Communications Technologies AB

In [18] and [32], Susanna Byhlin evaluates static WCET-analysis for Volcano Communications Technologies (VCT). VCT has a tool chain for development of real-time communications solutions for embedded network systems, and their customers are mostly located in the car manufacturing industry. In this thesis, the author examined if a commercial WCET tool could be integrated into VCTs tool chain.

The conclusions of the thesis was that it is possible to integrate an commercial WCET tool into the tool chain, but that it would require a lot of workload and detailed knowledge about the analyzed system. Both static and dynamic analyzes (using an oscilloscope) where performed by the author. The dynamic analyzes where hard to perform and good results was difficult to achieve.

Most time consuming part of the static analyzes was to set iteration bounds for loops that aiT did not find automatically.

4.2 Related tools

In this section, a couple of static WCET-tools will be described for orientation.

4.2.1 aiT

aiT, from AbsInt GmbH is together with Bound-T the commercial WCET-tools that are available on the market. aiT statically analyzes binary executables and includes cache and pipeline behavior using formal cache and pipeline models [38]. Similar to Bound-T, the user set annotations to give information on flow and hardware characteristics.

aiT has a graphical user interface that visualizes control-flow-graphs, derived WCET values and shows pipeline states. The output from this tool is an upper bound of a tasks execution time and a flow graph that contains all vital information.

aiT has been used by Airbus France to validate the timing behavior of critical avionics software [3].

4.2.2 Heptane

Heptane is short for Hades Embedded Processor Timing ANalyzEr, and is a WCET tool available under GPL license [10]. It is developed by IRISA (Institut de recherche en informatique et systèmes aléatoires) in France. Heptane is designed to be retargetable, and is available for the Renesas H8/300. Heptane has generally a tree-based approach, see section 1.2.2. For the moment, Heptane can analyze C-code, but it will soon be able to analyze disassembled binary. In [21], Heptane is used to analyze system-calls in the RTEMS real-time OS.

4.2.3 SWEET

SWEET is an acronym for SWEdish Execution time Tool, and is a tool that is under development at Department of Computer Science and Electronics, Mälardalen University.

SWEET architecture is build on the principle of having two analysis phases (flow analysis and low-level analysis) and one calculation phase. The tool can perform WCET analysis for the NECV850E and ARM7 processors. The tool includes pipeline and cache analysis modules and three different calculation methods.

The current research aim for SWEET is to develop flow analysis methods, thus avoiding the need for manual annotations from the user [23]. SWEET analyzes intermediate code inside a compiler.

Chapter 5

Conclusions and future work

5.1 Conclusions

To develop an accurate WCET-tool requires much work. To port an accurate WCET-tool to another host platform also demands much work, because all processors have their own little quirks to consider and special features to learn. Even though the H8/300 is a simple processor, having no caches or pipelines, it has quite large instruction set and number of addressing modes, which made the implementation non-trivial.

Bound-T was used to analyze Asterix. WCETs were derived for different system-calls and for some we were able to obtain parametrical WCET formulas. These formulas can be used by the students in the real-time courses, but they will have to be recalculated if any changes are made in the Asterix framework. Assertions where necessary for analyzing most system-calls. To set these assertions correctly, the user must know how the system calls are supposed to work.

Other results of this MSc thesis work are two articles, one for the WCET2005 workshop [29] and one planned for an education conference.

5.2 Future work

Future work includes validating the timing results of this port to the H8/300 processor. The execution time of instructions were taken from the processor manual [31], which we do not know if it is fully correct. The manual is written by humans, and humans sometimes make mistakes. Without thorough testing, we do not know if the Bound-T timing model for this version is correct either. Systematic validation of the timing model could include oscilloscopes or logic analyzers. This will minimize the possibility of implementation faults and verify that the values presented in the H8/300 processor manual is correct.

All parts of Asterix have not yet been fully analyzed and validated. The users have not tested Bound-T together with Asterix yet. The analysis of Asterix system-calls is also interesting for the development of Asterix, many thesis and extensions have been made to it from the start, is it still a fully predictable RTOS? Bound-T might tell.

BrickOS [6] is another OS used in education (by Uppsala University for example) with Lego Mindstorms as the target platform. It would be interesting to analyze BrickOS with Bound-T and document the differences between Asterix and BrickOS. This could be an interesting future MSc thesis.

Appendix A

An example of Bound-T usage - Intel 8051

A test run with an existing version of Bound-T will here be shown for introduction to the user interface and outputs for different commands. The already supported CPU is in this case the Intel 8051 [35].

The sample program analyzed is a very simple C-program that we here called demo.c, consisting of three functions: `main`, `foo` and `bar`:

```
unsigned char bar(unsigned char i) {
    unsigned char j, k;
    k = 50;
    if (i < 100) {
        j = 0;
        while (j < k) {
            i = i + j + 2;
            i = i + k - 3;
            j ++;
        };
    };
    return(i);
}

unsigned char foo(unsigned char i) {
    unsigned char j, k, m, p;
    k = 20;
    m = 0;
    if (i < 100) {
        j = 0;
        while (j < k) {
            i = i + j + 2;
            i = i + k - 3;
            j ++;
        };
        i = 0;
        for (j = 0; j < k; j ++) {
            i ++;
            p = j + 1;
            m = m + bar(p);
        };
    };
    return(m);
}
```

```
main() {
    unsigned char i;
    i = 1;
    i = foo(i);
    return 0;
}
```

As we can see in the code, `main` calls the function `foo`, which in turn will call function `bar`. Both `foo` and `bar` do some calculations.

To prepare demo.c for analysis, we need to compile and link the program. An executable binary for Intel 8051 also needs to be generated. If we want to analyze the executable binary demo.exe with Bound-T, we can use the command:

```
boundt_8051 demo.exe main
```

This command specifies that we want to analyze the executable binary file demo.exe, and that we want Bound-T to calculate the upper bound WCET for the subprogram main.

Here is the output from Bound-T:

```
Loop_Bound:demo.exe:DEMO.C:_B:8-12:50
Loop_Bound:demo.exe:DEMO.C:_A:25-29:20
Loop_Bound:demo.exe:DEMO.C:_A:32-35:20
Wcet:demo.exe:DEMO.C:_B:1-15:765
Wcet:demo.exe:DEMO.C:_A:18-38:15989
Wcet:demo.exe:DEMO.C:main:41-47:15999
```

The resulting WCET bound for `main` in demo.c is 15999 cycles. The output is also giving WCET bound for the subprograms called by `main`, and upper bounds for the loops in all analyzed subprograms.

Appendix B

An example of Bound-T usage - H8/300

A test run with our developed port to H8/300 will here be shown. Source-code of analyzed program, Bound-T command and options, assertions, disassembly of the object code and result of analysis with Bound-T will be included in this appendix.

The sample function that should be analyzed is a small system-call in the Asterix kernel, `writeChannel` from file os_waitfree.c:

```
void writeChannel( buffertype_t *ut, uint16 value , bufferid_t
channel_id ) {
  taskid_t task_id;
  bufctrl_t *bufctrl;
  bufferid_t nof_entries;
  uint16 i;

  task_id = self();
  bufctrl =TCBList[task_id].bufpointer;
  nof_entries = bufctrl->buf_id; /* The dummy entry contains the
                                    number of total entries in
                                    the array */
  bufctrl++;  /* the first index bufctrl[0] is just a dummy for
                 storing array information */


  for( i=0; i<nof_entries; i++,bufctrl++)
  {
     if(channel_id == bufctrl->buf_id && bufctrl->access==WF_WRITER )
     {
        bufctrl->written = WF_WRITTEN;
        *ut = value;
     }
  }
}
```

As we can see in the code, it contains a `for`-loop that needs to be asserted, because `nof_entries` are obtained via pointers, which are tricky for Bound-T to analyze.

If we want to analyze the function `writeChannel` with Bound-T, we set up the Asterix and Obelix environment with some tasks that reads and writes to a waitfree-buffer. Task `test` is the writer, `test2` is the reader. The environment is compiled and linked. A coff-h8300-file is generated using a GNU cross-compiler. Then we can run the following command:

```
boundt_h8_300 -device=lego -trace decode -assert wf-assert.bta
wf.coff-h8300 _test >wf.txt
```

Note that an underscore is added to function names when the cross-compiler converts the names into coff-h8300-format. We only analyze the function test, because that is where writeChannel is called at.

This command says that we want to analyze the file `wf.coff-h8300`, and that we want Bound-T to calculate a WCET bound for the user task `test`, which is where the call to `writeChannel` is located. An assertion file `wf-assert.bta` is used, chip-set is Lego or H8/3292 and the -trace decode option tells that we want an listing of decoded instructions in our output.

Here is the disassembler of `writeChannel`, note that writeChannels physical address is `00008ca0`:

```
00008ca0 <_writeChannel>:
    8ca0:   79 03 00 08     79 03 00 08       mov.w #0x8,r3
    8ca4:   19 37           19 37             sub.w r3,r7
    8ca6:   6d f4           6d f4             mov.w r4,@-e7
    8ca8:   6d f5           6d f5             mov.w r5,@-e7
    8caa:   6d f6           6d f6             mov.w r6,@-e7
    8cac:   6f f0 00 0c     6f f0 00 0c       mov.w r0,@(0xc:16,e7)
    8cb0:   6f f1 00 0a     6f f1 00 0a       mov.w r1,@(0xa:16,e7)
    8cb4:   0d 26           0d 26             mov.w r2,r6
    8cb6:   5e 00 86 56     5e 00 86 56       jsr   @0x8656:0
    8cba:   6b 03 92 2c     6b 03 92 2c       mov.w @0x922c:16,r3
    8cbe:   0d 02           0d 02             mov.w r0,r2
    8cc0:   09 22           09 22             add.w r2,r2
    8cc2:   09 22           09 22             add.w r2,r2
    8cc4:   09 22           09 22             add.w r2,r2
    8cc6:   09 02           09 02             add.w r0,r2
    8cc8:   09 22           09 22             add.w r2,r2
    8cca:   09 32           09 32             add.w r3,r2
    8ccc:   6f 21 00 0e     6f 21 00 0e       mov.w @(0xe:16,r2),r1
    8cd0:   69 14           69 14             mov.w @r1,r4
    8cd2:   6f f4 00 08     6f f4 00 08       mov.w r4,@(0x8:16,e7)
    8cd6:   89 06           89 06             add.b #0x6,r1l
    8cd8:   91 00           91 00             addx  #0x0,r1h
    8cda:   19 55           19 55             sub.w r5,r5
    8cdc:   6f f5 00 06     6f f5 00 06       mov.w r5,@(0x6:16,e7)
    8ce0:   1d 45           1d 45             cmp.w r4,r5
    8ce2:   44 40           44 40             bcc   .+64 (8d24)
    8ce4:   0d 10           0d 10             mov.w r1,r0
    8ce6:   0b 80           0b 80             adds  #0x2,er0
    8ce8:   0b 00           0b 00             adds  #0x1,er0
    8cea:   0d 13           0d 13             mov.w r1,r3
    8cec:   0b 83           0b 83             adds  #0x2,er3
```

We are also using a simple assertion to support Bound-T in the analysis, located in the assertion file, `wf-assert.bta`, which look like this:

```
subprogram "00008ca0"
   loop repeats <= 1 times;
   end loop;
end "00008ca0";
```

The loop will repeat at most once, because we only have one waitfree-buffer in our system, which the loop condition depends on.

Here is the output, which in the command are flushed to `wf.txt`, in this example, only the writeChannel-part are shown:

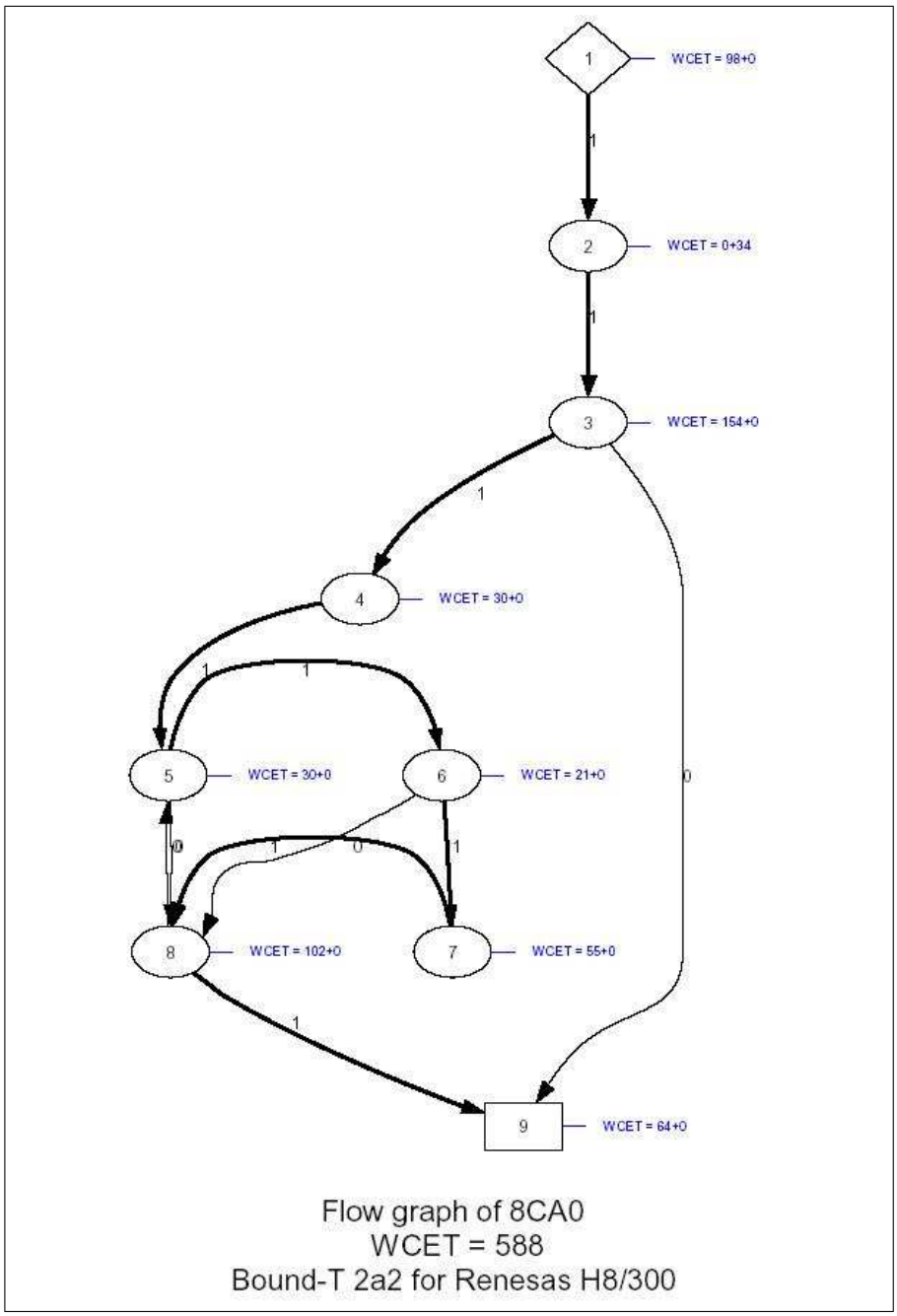
```
Address Code       Assembly          Time    Steps  Remarks
8CA0 7903 0008     mov.w #8,r3       12      1
8CA4 1937          sub.w r3,sp       6       2
8CA6 6DF4          push r4           10      3
8CA8 6DF5          push r5           10      4
8CAA 6DF6          push r6           10      5
8CAC 6FF0 000C     mov.w r0,@(12,sp) 14      6
8CB0 6FF1 000A     mov.w r1,@(10,sp) 14      7
8CB4 0D26          mov.w r2,r6       6       8
8CB6 5E00 8656     jsr @H'8656       16      9 call to 8656
8CBA 6B03 922C     mov.w @H'922C,r3  18      11
8CBE 0D02          mov.w r0,r2       6       12
8CC0 0922          add.w r2,r2       6       13
8CC2 0922          add.w r2,r2       6       14
8CC4 0922          add.w r2,r2       6       15
8CC6 0902          add.w r0,r2       6       16
8CC8 0922          add.w r2,r2       6       17
8CCA 0932          add.w r3,r2       6       18
8CCC 6F21 000E     mov.w @(14,r2),r1 18      19
8CD0 6914          mov.w @r1,r4      12      20
8CD2 6FF4 0008     mov.w r4,@(8,sp)  14      21
8CD6 8906          add.b #6,r1l      6       22 combined with next
8CD8x 9100         addx #0,r1h       6       23 combined with preceding add
8CDA 1955          sub.w r5,r5       6       24
8CDC 6FF5 0006     mov.w r5,@(6,sp)  14      25
8CE0 1D45          cmp.w r4,r5       6       26
8CE2 4440          bhs *+66          12      27 branch to 8D24
8CE4 0D10          mov.w r1,r0       6       28
8CE6 0B80          adds #2,r0        6       29
8CE8 0B00          adds #1,r0        6       30
8CEA 0D13          mov.w r1,r3       6       31
8CEC 0B83          adds #2,r3        6       32
8CEE 6912          mov.w @r1,r2      12      33
8CF0 1D26          cmp.w r2,r6       6       34
8CF2 4612          bne *+20          12      35 branch to 8D06
8CF4 683A          mov.b @r3,r2l     9       36, 37 compose-step
8CF6 460E          bne *+16          12      38 branch to 8D06
8CF8 FC01          mov.b #1,r4l      6       39, 40 compose-step
8CFA 688C          mov.b r4l,@r0     9       41
8CFC 6F74 000A     mov.w @(10,sp),r4 14      42
8D00 6F75 000C     mov.w @(12,sp),r5 14      43
8D04 69D4          mov.w r4,@r5      12      44
8D06 6F75 0006     mov.w @(6,sp),r5  14      45
8D0A 0B05          adds #1,r5        6       46
8D0C 6FF5 0006     mov.w r5,@(6,sp)  14      47
8D10 8806          add.b #6,r0l      6       48 combined with next
8D12x 9000         addx #0,r0h       6       49 combined with preceding add
8D14 8B06          add.b #6,r3l      6       50 combined with next
8D16x 9300         addx #0,r3h       6       51 combined with preceding add
8D18 8906          add.b #6,r1l      6       52 combined with next
8D1Ax 9100         addx #0,r1h       6       53 combined with preceding add
8D1C 6F74 0008     mov.w @(8,sp),r4  14      54
8D20 1D45          cmp.w r4,r5       6       55
8D22 45CA          blo *-52          12      56 branch to 8CEE
8D24 6D76          pop r6            10      57
8D26 6D75          pop r5            10      58
8D28 6D74          pop r4            10      59
8D2A 7903 0008     mov.w #8,r3       12      60
8D2E 0937          add.w r3,sp       6       61
8D30 5470          rts               16      62
Decoding finished for subprogram 8CA0: 62 steps, 65 edges.
```

The output shows the instruction address, the instruction words, the decoded instruction in assembler, the instruction's execution time in cycles, the numbers of the steps in the flow-graph, and any special remarks on the instruction. And last in wf.txt, the estimated WCET is shown:

```
Wcet:wf.coff-h8300:user.c:8CA0:-31:588
```

WCET is 588 clock cycles which corresponds to 36,75 microseconds with a CPU clock of 16 MHz.

In Figure 5.2 the flow-graph of `writeChannel` is shown, generated by Bound-T.



Bibliography

[1] Ada Language Reference Manual. http://www.adaic.com/standards/ada95.html.

[2] Download Site for Gnat. http://libre.act-europe.fr/.

[3] AbsInt AiT homepage. http://www.absint.com/ait, April 2005.

[4] AdaCore and Ada 2005. http://www.gnat.com/ada_2005.php, April 2005.

[5] Bound-T homepage. http://www.tidorum.fi/bound-t/, April 2005.

[6] BrickOS homepage. http://brickos.sourceforge.net, April 2005.

[7] Cygwin homepage. http://www.cygwin.com, April 2005.

[8] Enea Company Homepage. http://www.enea.com, June 2005.

[9] Graphviz - Graph Visualization Software. http://www.graphviz.org/, May 2005.

[10] Heptane homepage. http://www.irisa.fr/aces/work/heptane-demo/heptane.html, April 2005.

[11] Lego Mindstorms homepage. http://www.legomindstorms.com, April 2005.

[12] Renesas Technology - H8/300 Series. http://www.renesas.com/fmwk.jsp?cnt=h8300_series_landing.jsp&fp=/products/mpumcu/h8_family/h8300_series/, May 2005.

[13] Renesas Technology - H8/3297 Group. http://www.renesas.com/fmwk.jsp?cnt=h83297_root.jsp&fp=/products/mpumcu/h8_family/h8300_series/h83297_group/, May 2005.

[14] SciTE homepage. http://www.scintilla.org/scite.html, April 2005.

[15] M. Ben-Ari. *Ada Programming for Software Engineers*. John Wiley and Sons, Chichester, UK, 1998.

[16] Michel Berkelaar. lp_solve: (Mixed Integer) Linear Programming Problem Solver. ftp://ftp.es.ele.tue.nl/pub/lp_solve, 2004.

[17] Alan Burns. The Ravenscar Profile. *ACM SIGADA Ada Letters*, 19(4):49–52, 1999.

[18] Susanna Byhlin. Evaluation of Static Time Analysis for Volcano Communications Technologies AB. Master's thesis, Mälardalen University, Västerås, Sweden, Sept 2004.

[19] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. Worst-case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. 2002.

[20] Martin Carlsson. WCET Analysis, Case Study on Interrupt Latency, for the OSE. Master's thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, Mars 2002.

[21] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*.

[22] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Phd thesis, Uppsala University, Uppsala Sweden, 2003.

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

[24] Hitachi Ltd. *Hitachi Single-Chip Microcomputer H8/3297 Hardware Manual*, 3rd edition edition.

[25] Niklas Holsti. *Bound-T Application Note Renesas H8/300*. Tidorum Ltd, Helsinki, Finland, May 2005.

[26] Niklas Holsti. *Bound-T User Manual*. Tidorum Ltd, Helsinki, Finland, May 2005.

[27] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-Case Execution Time Analysis for Digital Signal Processors. In *Proc. of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, September 2000.

[28] Joakim Lindgren. Obelix Development Environment. Master's thesis, August 2000.

[29] Samuel Petersson, Andreas Ermedahl, Anders Pettersson, Daniel Sundmark, and Niklas Holsti. Using a WCET Analysis Tool in Real-Time Systems Education. Technical report, 4 2005.

[30] William Pugh. The Omega test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*, pages 4–13, 1991.

[31] Renesas Corporation. *H8/300 Programming Manual*.

[32] S. Byhlin, A. Ermedahl, J. Gustafsson, B. Lisper. Applying Static WCET Analysis to Automotive Communication Software. July 2005.

[33] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static Timing Analysis of Real-Time Operating System Code. Oct 2004.

[34] Daniel Sandell. Evaluating Static Worst Case Execution Time Analysis for a Commercial Real-Time Operating System. Master's thesis, Mälardalen University, Västerås, Sweden, June 2004.

[35] Ville Sipinen. *Bound-T Application Note MCS-51(8051) Family*. Tidorum Ltd, Helsinki, Finland, March 2001.

[36] David B. Stewart. Measuring Execution Time and Real-Time Performance. In *Proceedings of the Embedded Systems Conference (ESC SF)2002*, Columbia, USA. Embedded Research Solutions, LLC.

[37] Henrik Thane, Anders Pettersson, and Daniel Sundmark. The Asterix Real-Time Kernel. In *13th Euromicro International Conference on Real-Time Systems, Industrial Session*. IEEE Computer Society, June 2001.

[38] Stefan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. Phd thesis, Universität des Saarlandes, Saarbrücken Germany, 2004.