Worst Case Execution Time Analysis, Case Study on Interrupt Latency, For the OSE Real-Time Operating System

By

Martin Carlsson

Royal Institute of Technology, Stockholm

Master's Thesis in Electrical Engineering
Stockholm, 2002-03-18

Abstract

In real-time systems the execution time of a program is crucial, missing a deadline can have catastrophically consequences. Today the estimation of the worst execution time is mostly done by measurements with the worst possible input to the program. These measurements are not totally reliable; there is a chance that the worst execution path of the program is not caught in the measurements. There have been a lot of research performed in the WCET (Worst Case Execution Time) analysis field in the last couple of years, and models of how to theoretically estimate the WCET have been thoroughly described. But there haven't been many attempts at applying the models to actual real-time operating system code.

The goal with this thesis project is to use today's research in the WCET analysis field, especially the work by the ASTEC WCET-group in Sweden, to develop a tool that can be used on object-code for an ARM microprocessor. The tool should be able to transform the binary executable file of a program into control flow graphs with basic blocks, so that a safe (no underestimates) and tight WCET analysis can be calculated. In this case study the WCET of the interrupt latencies in the OSE real-time operating system. A part of the work is to determine how much work that has to be done by hand, e.g. through program-specific input from the user, and how much that can be automated.

Table of Contents

1 Introduction

Estimating the worst-case execution time of a program is a very important task, especially when you are dealing with real-time operating systems and programs, which have deadlines that have to be kept. Missing a deadline can have catastrophically consequences, because real time operating systems and programs are used in all types of time sensitive embedded systems, e.g. in medical equipment, cars, mobile phones and airplanes.

To calculate a static estimate the worst case execution time of a program and get a both safe (no underestimates) and tight (as little overestimation as possible) WCET (Worst Case Execution Time) approximation is not an easy task. Several things have to be considered, such as how to model the caching behaviour to include it in the analysis and how to find the longest execution path in all the execution paths of the program. The most important task when performing a WCET analysis is to determine the number of loop iterations in the program, because it's here that the programs spend most of their execution time.

This thesis project was performed at OSE Systems, which is the developer and distributor of the OSE real-time operating system. They were interested in calculating the maximum Interrupt latency within the operating system kernel using static analysis of the compiled code.

OSE Systems' reason to get involved in this project is for one that today the estimations of the WCET are made by manual tests. Therefore a higher time limit on the deadlines for the operating system then necessary is set, because one can't be absolutely sure all execution paths have been tested. If this project could come up with a good way of representing the code for statically WCET Analysis, the theoretically WCET could be calculated and the deadline time limits could be reduced.

The main part of the thesis project was spent on the implementation of a tool prototype for preparing operating system code at object code level for static WCET calculation. The tool constructs a number of control flow graphs (CFG) from a compiled and linked binary file of the operating system kernel. These CFG's contain basic blocks of instructions, bounded by jump instructions and, in our case, instructions that change the Interrupt State. The approach was an up-and-down solution, which starts with the first binary and constructs basic blocks and control flow until all binaries are decoded and placed into basic blocks and the flow between the blocks is determined. There are others that have faced the same task, e.g. [16] where they had a bottom-up approach when constructing the CFG from the binary file.

This report will explain some of the background to WCET Analysis as well as some theory behind real-time operating systems in general and the OSE operating system in specific. The target architecture (ARM9) will be briefly discussed, but the main part of the report will be spent on the tool prototype implementation and the results from the experiments. In these chapters the problems we encountered and how we solved them will be discussed and the most important lessons of the project are presented in the conclusions and future work chapter

1. 1 Company Background

A brief summary of the company where the thesis was performed:

OSE Systems[1] is a subsidiary of Enea Data[2]. Enea Data was founded in 1968 by a couple of KTH (Kungliga Tekniska Högskolan, Royal Institute of Technology) and Stockholm University students. Enea is an abbreviation of Engström Elektronik AB (Engstrom Electronics Inc) where Engström is the last name of one of the founders. The first Unix system in Sweden stood in a room at Enea (1981), and the first E-mail in Sweden was sent to Enea (April 7, 1983). In the early Internet years, Enea was the .se NIC (where you register domain names), that of course means enea.se was the first Swedish domain registered. Enea also administrated the first Internet backbone in Sweden, which was later moved to KTH and is today known as SUNET[3]. Enea has since the

[1] www.ose.com
[2] www.enea.se
[3] www.sunet.se

start been in the embedded systems consulting business and the OSE operating system originated from a number of consulting projects for the telecom industry. OSE is today one of the largest operating systems in the world for embedded systems such as mobile phones and airplanes.

2 Real-Time Operating Systems

2.1 General

The definition of a real-time system is that the program execution has certain timing deadlines that have to be kept. Deadlines in real-time systems are often divided into two types: Hard and Soft. Where in hard deadline real-time systems, missing a deadline causes the system to crash in every case. For soft deadline real-time systems, deadlines can be missed without crashing the system. Another way to look at it is that the type of the deadline depends on the consequences for missing it, e.g. lowered efficiency in a car engine or an unstable measuring system.

Keeping real-time constraints can be difficult even on single processor systems. Therefore on distributed real-time systems it's even more difficult too keep the deadlines. A distributed system is a system with several processors, running separate from each other. The deadlines are kept through a carefully considered squeduling algorithm. An example of missing a deadline is shown in figure 2.1. Here a reply-message arrives too late.

In the figure process A needs some service form process B in order to continue the execution. The process sends a message to B requesting the information. B receives the message and sends a reply back to process A. If the reply message from B is not received by process A within the deadline, either an exception is raised (soft real-time systems) or the system requirements is not kept (hard real-time systems).

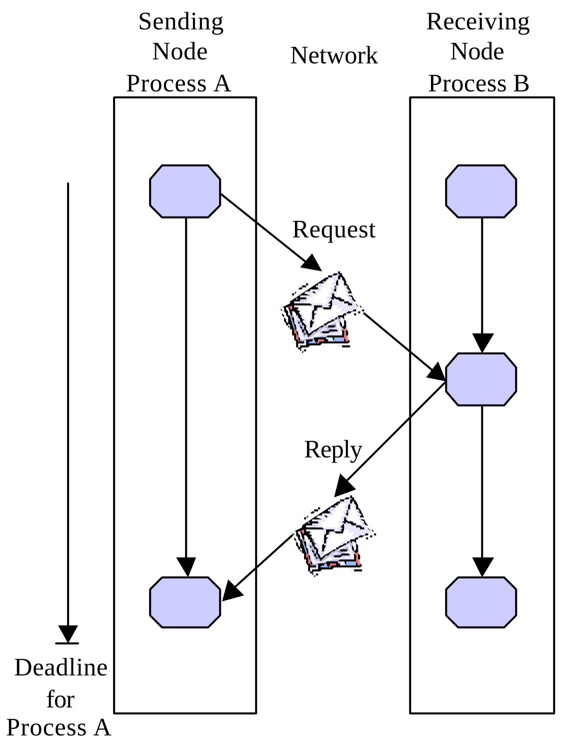


Figure 2.1 Deadline example in a distributed system, taken from [17]

2.2 The OSE Operating System

OSE is used in distributed systems with hard real-time deadlines but in reality most of these systems contain parts with hard real-time deadlines, and other parts with soft real-time deadlines.

2.2.1 Processes in OSE

There are five types of processes in OSE:

- Interrupt, trigged by either an interrupt, e.g. an Ethernet package has arrived and needs to be taken care of (before the queue of packages overflows), or some unit has been removed from the PCI bus.

- Timer Interrupt, used for periodic events, e.g. when it's desired to measure the temperature each 500 ms, or when a LED is set to blink every second.

- Prioritised, the most common kind of process. They are written as eternal loops (for(;;)) and runs as long as no interrupt occurs or a process with higher priority becomes ready.

- Background, run in a strictly time-sharing mode beneath the prioritised processes. Also written as eternal loops just like the prioritised processes. Background processes are pre -empted by prioritised or interrupt processes.

- Phantom, contains no executable code and are used only as a representation of another process. Used together with a redirection table to form a logical channel for communication between processes in distributed systems.

Every process has a priority from 0-31, where 0 is the highest. For interrupt processes, the priority is mapped against a hardware priority. At every point in time, the process with highest priority is run. Every process can be in one of three stages, waiting, ready and running, as shown in figure 2.1.

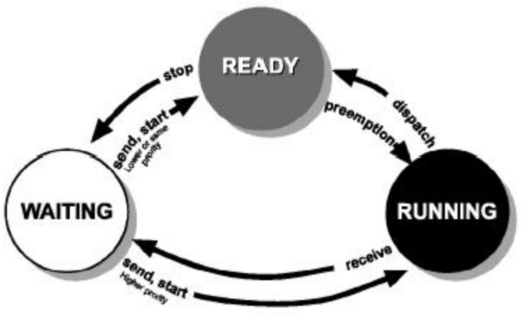


Figure 2.2 Possible process states in OSE from Enea OSE Documentation [19]

2.2.2 Memory Management

Memory management is important in every real-time system. In OSE the memory is organized into different pools and segments, as shown in figure 2.3. There is one system pool in OSE, where processes and blocks that execute in the system segment can allocate memory. The system pool is always located in the kernel memory. So if the system pool gets corrupted then the whole system will crash. The advantage of that each block can have its own memory pool is isolation. First isolation through memory protection and also isolation so that one process can't allocate all the memory in a segment and block other processes from allocating memory.

Processes can be put together into blocks that can have their own memory pool. One advantage of using blocks is that many system calls can operate on whole blocks instead on single processes. Another is that the blocks memory pool can be used for environment variables that are visible for all the processes within the block, but not for other processes.

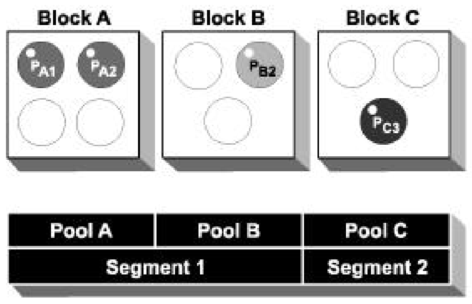


Figure 2.3 Memory organisation in OSE, taken from the OSE documentation [19]

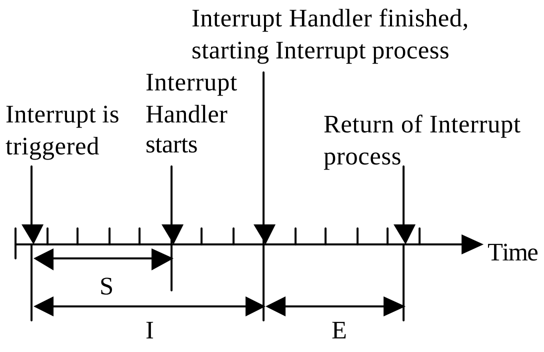
2.2.3 Interrupts

An interrupt occurs either when an outside device wants to notify the system that there is data available, e.g. an incoming signal, or when an error has occurred in the system, e.g. when a device has been removed or broken down.

When an interrupt occurs, the first thing that happens is that the program counter will be saved at the current execution address, and also if there are some registers that are needed for the active process, they are also saved. Then an interrupt handler is started which is interrupt type specific, that means that there are different interrupt handlers for different kinds of interrupts. As mentioned earlier interrupt processes also have priority. When an interrupt occurs and right after another interrupt with higher priority occurs, the current interrupt handler has to wait, i.e. gets pre-empted, until the higher priority interrupt is finished.

2.2.4 Interrupt Latency

The Interrupt Latency is the time from when an Interrupt is triggered until the Interrupt process starts. In figure 2.3 there is an overview of the different stages that passes when an interrupt occurs.



Interrupt Handler finished, starting Interrupt process
Interrupt Handler starts
Interrupt is triggered
Return of Interrupt process
Time
S
I
E

S = Time to start the Interrupt Handler
I = Interrupt Latency
E = Interrupt process execution time

Figure 2.3 Interrupt Latency in OSE, taken from [17]

The Interrupt Handler time (I – S) is, as mentioned before for saving the CPU register contents, find out cause of interrupt and set the interrupt mask. If the Interrupt mask is set so dis able Interrupts, then the system has to wait until Interrupts are enabled again before the Interrupt handling can start. This is were, in the worst case scenario, the most time of the Interrupt latency is spent and it is these parts of the operating system that this project will focus on and try to find the maximum bound for. In average, the time Interrupts are disabled is only about 1% of the total interrupt latency, but in the worst case, which we are interested in here, that time is approximately 50% of the Interrrupt latency.

2.2.5 Disable Interrupt regions

The operating system has been designed for real-time purposes. This means that the regions where Interrupts are disabled and the Interrupt latency is big, are many but very short. In a non real-time operating system, Interrupts can be disabled for longer periods of times and therefore have fewer but longer Interrupt latencies.

Interrupts are disabled during critical operations, e.g. during memory access when a sensitive variable needs to be changed and mutual exclusion is necessary, or when new memory is allocated. Another time when Interrupts are disabled is when the scheduler is locked for temporarily preventing a context switch or when a process is removed or inserted from the process table containing all the processes.

In this case study we examine the OSE delta kernel for disable - to enable Interrupt regions.

3 Target Hardware

OSE is available for a number of different hardware targets. The delta kernel of OSE that has been used in this case study is available for ARM, StrongARM, PowerPC, Motorola 68k and MIPS R3000. These are all RISC (Reduced Instruction Set Computer) processors except for the Motorola 68k, which is a CISC (Complex Instruction Set Computer) processor.

We have chosen an ARM processor as target because it's one of the most common processors on the market, and it has a simple instruction set architecture. And we chose ARM9 because it's relatively new with an extended 5-stage pipeline and still has the same instruction set as the predecessor ARM7 family.

3.1 ARM 9

The ARM9 is a 32-bit RISC microprocessor that differs a bit from previous ARM releases. Instead of a three-stage pipeline, this new processor family uses a five-stage pipeline. It supports both Big- and Little -endian modes and can be switched to a 16-bit THUMB mode for sections where compact code size is required. THUMB is a separate instruction set derived from the 32 -bit ARM instruction set.

3.1.1 ARM Instruction Set

One property of the ARM instructions that makes it different from other processors instructions is that each instruction has a condition code attached to it. This condition code, which is the first four bits in the instruction binary, says if the instruction is to be executed or not. The most common condition is naturally the always condition. This solution reduces the number of branches in the code. The 32-bit ARM Instruction Set has ten standard formats:

- Data Processing
- Multiply
- Single Data Swap
- Single Data Transfer
- Block Data Transfer
- Branch
- Branch and Exchange
- Halfword Data Transfer
- Coprosessor Data Transfer
- Coprocessor Data Operation
- Coprocessor Register Transfer
- Software Interrupt
- Undefined

The processor has a total of 37 registers made up of 31 general 32 bit registers and 6 status registers. At any time, 16 general registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor mode and the other registers. The banked registers are switched in to support rapid interrupt response and context switching [20].

Interrupts are disabled and enabled by setting one or two of the Interrupt bits in the status register (CPSR) FIQ and IRQ in figure 3.1.

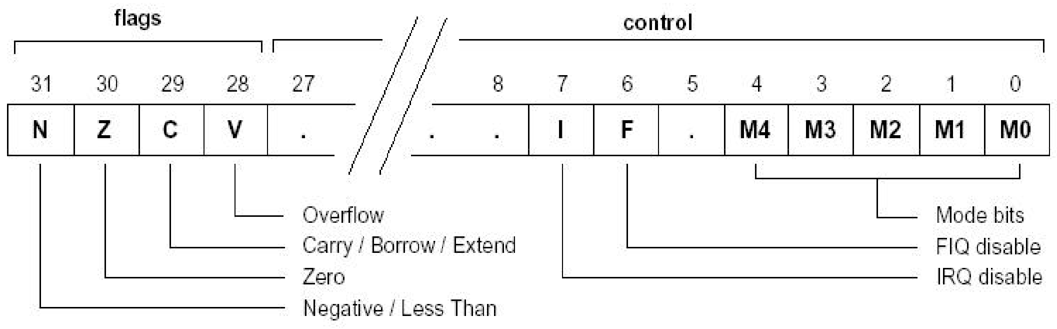


Figure 3.1 Format of the ARM Program Status Registers (PSR)

4 WCET Analysis – Theory

4.1 Introduction to WCET Analysis

Several attempts have been made to come up with a useful method for the estimation of the worst-case execution time of a program. It's hard to determine the best method because the research in the area only goes back about 10 years from now, and they haven't been applied to enough number of real systems to come to any conclusion. The main methods used in this report are based on the work of the ASTEC WCET-group in Sweden [1] [2] [3] [4] [5] [14]. They have come up with a powerful tool to express the different parts of the analysis in a language that can be used for the development of timing analysing tools, such as the one developed within this thesis project. Their method to calculate the WCET is divided into three steps.
First the program flow analysis step, where the control flow of the program is analysed, the code is grouped into basic blocks and a basic blocks graph and scope graph are constructed.
The second step is the low-level analysis, which determines the timing effects of external and internal parts. The external part is called global

low-level analysis and includes the timing effects of caches and pipeline timing effects. Local low-level analysis deals with machine timing effects that depend on a single instruction and its immediate neighbours.
The last step of the WCET analysis is the calculation step. Here the previous parts of the analysis are combined so that a total result can be reached.

4.2 Related Work

In [1], the previous work that has been conducted in Uppsala is presented. For example the introduction of a language representation for modelling complex program flows, and the timing behaviour of pipelines. It also gives an overview of the modular architecture of the WCET tool developed by the ASTEC WCET-group in Uppsala (see Figure 1).

In [2] Jakob Engblom has looked at the properties of embedded programs, such as how many condition statements there are and the depth of loops in a real-time program. The study was performed at object-code level, because program-code for embedded systems is often automatically generated, and therefore rather "ugly".

The researchers behind [3] [4] and [5] go deeper into the different parts of the WCET model from the ASTEC WCET-group. [3] concentrates on pipeline analysis, based on a trace-driven simulation, whereas [4] and [5] looks at the modelling of complex program-flows, i.e. how to model a complex flow in order to get the tightest possible WCET-estimate.

Colin and Puat have in [6] applied WCET methods on a real R-T operating system, namely the RTEMS operating system. They came to the conclusion that WCET analysis for a R-T operating system is feasible and located some of the difficulties one might have in performing the analysis.

A group of Korean researchers presents in [7] a technique for estimating the WCET for programs run on RISC processors. They have included cache analysis as well as pipeline timing considerations in their model.

In [8] the IPET (Implicit Path Enumeration Technique) is used for calculation of the worst possible execution path. A WCET analysis tool for hard real-time programs is presented: Cinderella (who had the hard real-time constraint that she had to be home by midnight…) that calculates the WCET for programs run on the Intel i960 KB processors.

The writers of [9] present models for instruction cache analysis using abstract interpretation and shows how to statically categorize the caching behaviour of each instruction.

The important task of how to bound the number of loop iterations inside R-T programs to be able to perform WCET analysis on it is closely examined in [10].

The WCET analysis in [12] also includes caching- and pipeline-analysis where the caching behaviour is attached to each basic block and the pipeline-analysis takes care of pipeline behaviour between the basic blocks.

[14] Gives an overview of the work performed by the ASTEC WCET-group and explains the pipeline timing-effects more in detail.

Theiling and Ferdinand try to combine Abstract Interpretation (AI) and Integer Linear Programming (ILP) in [13] and [15]. They use AI for the cache modelling and ILP for the program flow analysis. The advantage in using ILP for flow analysis is that the tool becomes more portable, as the user can specify constraints of the specific program before analysing it

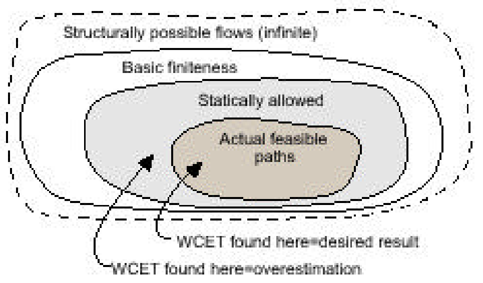
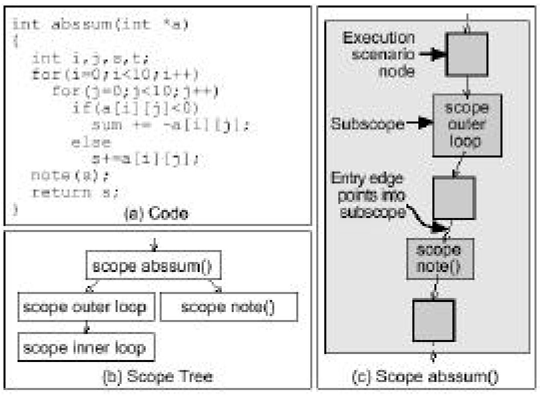

Structurally possible flows (infinite)
Basic finiteness
Statically allowed
Actual feasible paths
WCET found here=desired result
WCET found here=overestimation

Figure 4.1 Relation between possible executions and flow information [4]

4.3 Program Flow Analysis

The goal with the program-flow analysis is to model the possible execution paths the program can take when it runs, i.e. what functions get called, how many times loops iterate and dependencies between if-statements (figure 1). This is to ensure that we have all the feasible paths when we later, in the calculation step, will perform a search for the longest feasible path. In order to be able to do that, the first step is to divide the code into *basic blocks* and find the dependencies between them when constructing the *control flow graph*. Each block has the property that it doesn't contain any function calls or jumps to other procedures, i.e. we are ensured of this piece of code's execution-path. There are a number of different approaches to creating the control flow graph with basic blocks as nodes available. Either the analysis can be done automatically or it can be done by hand. With the automatic approach, one can use control flow information from the compiler, integer linear programming (ILP) constraints, or abstract interpretation. In abstract interpretation the idea is to extract properties of the run-time behaviour of a program by making an "interpretation" of the program using abstractions of values instead of concrete values [1]. The program behaviour can easily be modelled by ILP, but the analysis is likely to become inefficient for larger applications, since solving an ILP problem in general takes exponential time [12]. Abstract interpretation for flow analysis has been widely explained [4][5][7][9][12]. In [4], in order to represent the dynamic behaviour of the program, the concept of a scope is introduced. Here, each *scope* corresponds to a repeating or differentiating execution environment in the program, e.g. a function call or a loop, and can contain one or more basic blocks (figure 4.2).

All scopes are supposed to be looping, even if they iterate less then one time. Therefore all scopes can be assigned something that is called flow-fact-information, where a number of properties of the scope are given, such as the number of iterations of the scope and intervals of iterations where the expressions are valid.



```
int abssum(int *a)
{
  int i,j,s,t;
  for(i=0;i<10;i++)
    for(j=0;j<10;j++)
      if(a[i][j]<0)
        sum += -a[i][j];
      else
        s+=a[i][j];
  note(s);
  return s;
}
```

Figure 4.2 Example of code with associated scopes [4]

In figure 4.3 is an example of flow information facts attached to the scopes. The symbol [1…4] means that the facts are the total result from iterations one to four of the scope. <*Range…*> says that the facts are valid *for each* of the iterations in the range. When no ranges are given, the facts are supposed to be valid in all iterations of the scope. The right part of the information facts is a constraint specification, e.g. the first fact of scope foo in figure 4.3 means that the total of X_A in the iterations one to four is less than or equal to two.

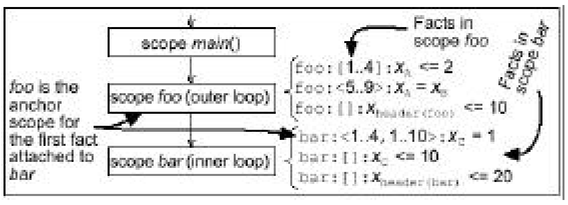

scope *main*()
scope *foo* (outer loop)
scope *bar* (inner loop)
foo is the anchor scope for the first fact attached to *bar*
Facts in scope *foo*
Facts in scope *bar*
foo: [1..4] : x_A <= 2
foo: <5..9> : x_A = x_B
foo: [] : $x_{header(foo)}$ <= 10
bar: <1..4, 1..10> : x_C = 1
bar: [] : x_C <= 10
bar: [] : $x_{header(bar)}$ <= 20

Figure 4.3
Example of facts attached to scopes [4]

The technique with expressing flow information using constraints is known from the implicit path-enumeration technique (IPET) that will be discussed later in this chapter.

4.4 Low-level Analysis

The second part of the WCET analysis is to consider the machine timing-effects. This includes both global low-level analysis, where the external timing effects such as caching are considered, and local low-level analysis, where the focus is on the timing effects from single instructions and pipeline effects. To get an overview of the timing effects from the low-level analysis, a *timing graph* is constructed. Also in this graph the nodes consist of basic blocks and we attach the timing facts to each node including both the global and the local low-level analysis. Relevant parts of the timing graph are later used in the calculation of the WCET.

4.4.1 Global Low-level Analysis

Global low-level analysis means analysing the effects on execution time from all parts of the machine. The main global contributor to lower execution times is the caching of instructions and data, and therefore it needs to be considered in the WCET analysis in order to get a tight estimation of the actual execution time. The other global timing effects are small in comparison and therefore I only focus on caching in the global analysis.

4.4.1.1 Including Cache Performance in the Analysis

Methods on how to model the caching behaviour have been presented in several research articles [7][9][12][13][15]. In [7] the authors divide the difficulty in predicting the caching behaviour into two problems: *intertask interference* and *intratask interference.* Intertask interference is caused by preemption of a task, and when the task gets to run after being pre-empted, it will refer to memory blocks in the cache that is no longer there. Intratask interference occurs when more then one memory block from the same task competes for the same cache memory block. This results in two kinds of cache misses: capacity, due to limited cache size, and conflict, due to limited amount of cache set associativity. The instructions or data in the cache references are divided into four categories. *Always-hit,* the referenced data or instruction is always in the cache. *Always-miss,* instruction or data is not in the cache in any cache reference. *First-miss,* means that the reference is not in the cache the first iteration, but can be supposed to be in the cache the rest of the iterations, e.g. a for() loop. Finally *conflict,* a reference that can't be determined whether it's in the cache or not.

Whalley and others have a similar approach in [9], with the same categorizations of cache references except that they add the *first-hit* category, which means that the referred instruction (data caches are not considered here) is in the cache the first iteration and all remaining references will be misses. They also add some information to each basic block in the control flow graph that gives a set that abstractly represent what's in the cache at the entry and exit of each basic block. Here two new definitions are given to determine the sets:

1. A program line can potentially be in the cache if there exists a sequence of transitions in the combined control flow graph and call graph (graph with the external function calls and function instances) such that the program line is cached when the basic block is entered.
2. An abstract cache state of a basic block in a function instance is the subset of all program lines that can potentially be cached (1) prior to the execution of the basic blocks.

The other articles use combinations of the above.

The cache information is then included in the timing graph for later use as can be seen in the example in figure 4.4.

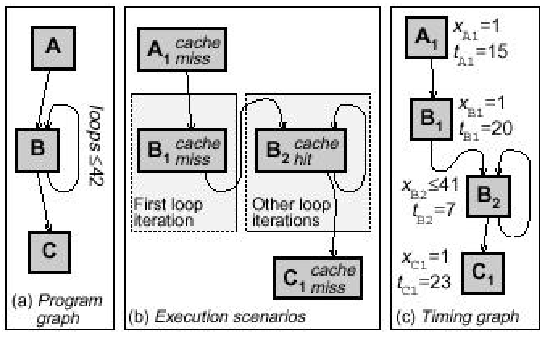


Figure 4.4 Example of cache execution scenarios included in the timing graph. [3]

4.4.2 Local Low-level Analysis

Single instructions and their pipeline timing effects are considered in the local low-level analysis. The timing contributions from the single instructions memory access time are first

added to the timing graph by summing up the execution time of every instruction in the correspondent basic block.
The pipeline effects between sequential basic blocks in a program can be achieved in two ways, either a simulation with first the basic blocks isolated and then together to get the pipeline timing effect, or through a statical analysis. In [7] the pipeline timing effects are considered statically, using so called reservation tables for each basic block. These tables are built up by examining the pipeline steps of each instruction in the block. Then the head and tail of the reservation table of depending blocks are analysed together to get the pipeline effect. An example of the reservation tables can be seen in figure 4.5, where the vertical axis corresponds to the pipeline steps and the horizontal is time.

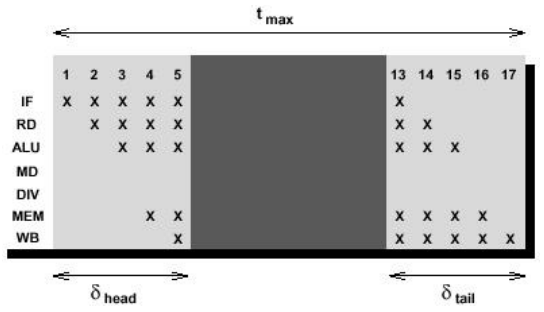


Figure 4.5 Example of a reservation table [7]

With simulation the basic blocks are first run isolated and then in sequence to get the pipelining difference. This difference is in [3][5] and [14] denoted with δ as can be seen in the example in figure 4.6

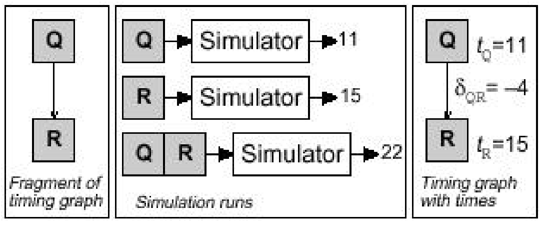


Figure 4.6 Example of pipeline overlap between consecutive blocks. [3]

4.5 Calculation

The final step of the WCET analysis is the calculation step. Here the results from the flow analysis and low-level analysis are put together to form a final estimate of the execution time. In research literature there are three main methods of calculation: *path-*, *tree-* or *IPET- (Implicit Path Enumeration Technique)* based [1].

4.5.1 Path based Calculation

When calculating the WCET with path-based calculation the goal is to get the longest feasible execution time of the program. This is achieved by first calculate the, e.g. five, longest paths of the program and then seeing which one of them is feasible [12].

4.5.2 Tree based Calculation

In tree-based calculation the WCET is generated by a traversal of a tree representing the program, starting from the bottom. Results from analysing smaller parts of the program are used to make the timing estimates for larger parts [7].

4.5.3 Implicit Path Enumeration Technique (IPET)

The way to implicitly find the longest executable path of a program is by setting algebraic and/or logical constraints to the basic blocks in the graphs and thereafter maximizing an objective function (1) and holding the constraints using integer linear programming (ILP) [8].

$$\Sigma_{\forall} (x_{\text{basic block}} * t_{\text{basic block}} - t_{\text{timing effect}}) \quad (1)$$

An explanation of the object function: $\Sigma_{\forall}$ means the sum of all elements in the following expression. $x_{\text{basic block}}$ is the number of iterations for a basic block. $t_{\text{basic block}}$ is the timing effect of each block and $t_{\text{timing effect}}$ is the gain of pipeline effects between blocks.

Most WCET tools use a combination of the techniques mentioned above for different parts of the calculation.

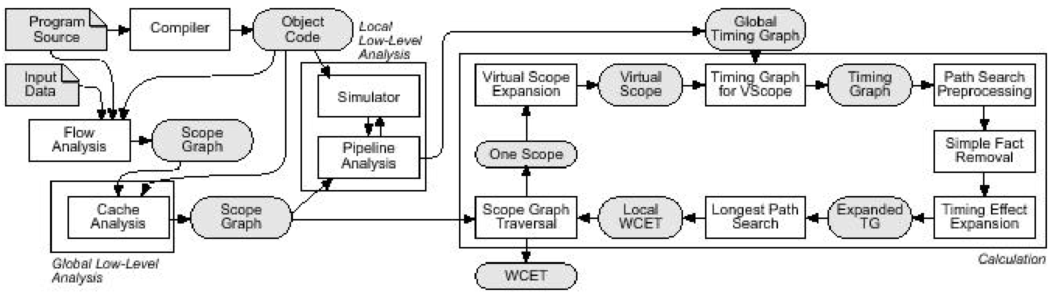


Figure 4.7 Overview of the WCET tool using path-based calculation [5]

5 WCET-Prepare tool prototype

5.1 Goal

The goal for the tool developed within this project is that it should be able to produce flow information graphs from a number of source code files and an ELF (Executable Linkable File) containing object code binaries. It should be able to detect interrupt pairs (disable interrupt -> enable interrupt) and construct a basic blocks graph for each such chain. This is needed to calculate the maximum interrupt latency. The result should be presented as a scope graph (in a .sg file) and as a special .tcd (Textual Code Description) file containing the basic blocks graph. The idea is that the graphs should be on a format they can be used for direct WCET calculation by using the tool developed by ASTEC.

I have chosen to call the tool WCET prepare because the actual calculation is performed by an automatic calculation too developed by ASTEC in Uppsala. So the tool developed in this project makes all the preparations to make a calculation possible.

As programming language, C++ is used for the most part, but some text comparing is done with AWK scripts.

The tool is divided in two separate parts, which internally are named “Find Functions and DI” and “ARM decoder”.

5.2 Find Functions and DI

The goal of this part of the tool is first to filter out each function in the source code files and then find out how many disable and enable interrupts they contain. This is all done with the help of AWK-scripts. Here I had to study the source files in order to find out how interrupts were disabled and enabled to be able to filter correctly. In the assembler source code, the disabling of interrupts is done by setting one of two (or both) interrupt bits in the ARM processor status register. This means that determining whether interrupts are disabled or enabled from just looking at the source code is very difficult, because a separate register is often used when changing the status register. And to determine the change correctly, the content of that register needs to be known. The solution to this problem was to categorize some changes of the status register as unknown when the register content could not be determined by looking at the instructions just preceding the interrupt change instruction (Move Register to Status Register, MSR [22]). Whether the change is a disable or enable interrupt is left open until later when the tool is put together with the basic blocks graph, where flow information is available. There is a possibility that the interrupt change is still undeterminable but this will be further discussed in the next chapter.

After the first filtering the function names are compared with a parsed ELF (Executable Linkable File) to get the physical starting address for each function, which then are linked together with their respective function.

5.3 ARM decoder

The second part of the tool is the part that handles the translation of binary instructions to an internal instruction format, and the construction of basic code blocks plus a basic blocks graph.

As a first step the ELF file obtained from the linking of the compilation is parsed in order to get the binary representation of each instruction. These binaries are put into a list with their respective address.

5.3.1 Instruction considerations

The ARM instructions have some different properties that have to be considered during decoding. First there are constant data regions embedded in the executable regions so that functions can access data very fast. There are also regions with THUMB instructions. THUMB is a 16-bit instruction set derived from the 32-bit ARM instruction set. These instructions are used when there is a need to save code size, which is an important cost factor in small embedded systems, e.g. a mobile phone.

6 Prototype Implementation

The largest part of the work in this thesis project was the implementation of the WCET prepare tool and the most workload within the implementation was put in the creation of basic blocks and managing correct program flows. I will discuss this later in this chapter.

As mentioned above, the tool is consists of two parts, the first is for finding functions and their properties, and the second for translating object code into a suitable instruction format, dividing the code into blocks and managing correct program flow.

6.1 Finding functions and interrupt changes

Figure 6.1 shows an overview of the first part of the tool, for finding interrupt changes, function names and their start-addresses. To get the function names from the source code files I used AWK scripts from the Cygwin tools. AWK is originally from UNIX but can be transformed to Windows through the use of GNU or Cygwin tools. That makes the tool more platform-independent. Cygwin is Open Source, so the tool is not depending on any commercial product. The GNU tools are also free to use, and they could just as well have been used here.

AWK scripts are used when you are parsing text files and need to do more complex searches within the text. It's possible to implement quite complex code in AWK that lets you do a lot with the text you're working with. Here the problems were to get the function names from each source code file and within each function detect the number of Disable and Enable Interrupts. The function names were no problem, but to detect the interrupt changes was a bit more problematic then I first thought.
In the c source files the interrupt disabling is done with a LOCK_PUSH() call or just LOCK(), and these were fairly easy to detect.

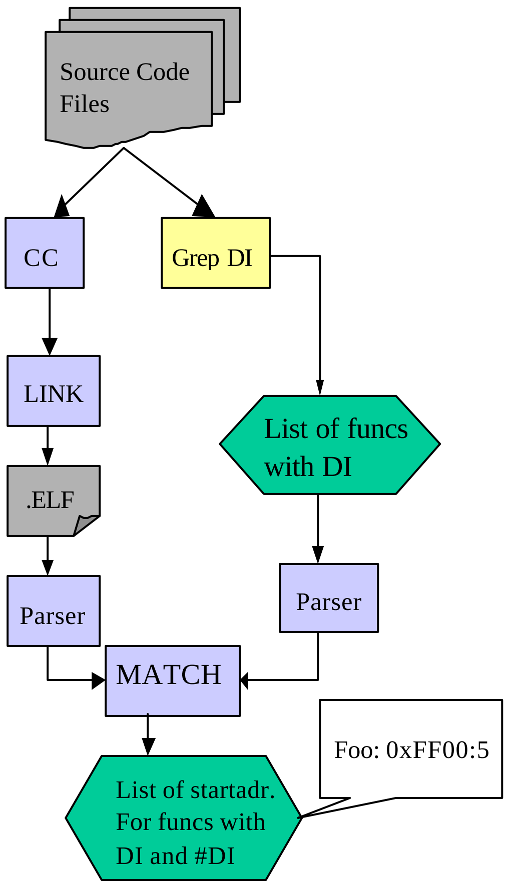


Figure 6.1 Finding Interrupt changes and function names from source code files

The problem is in the assembler source files. As mentioned in the section about ARM the interrupts are disabled and enabled by setting or clearing one or both of the interrupt bits in the status register. This is done with the MSR-(Move Register to Status-register) instruction where the second operand is one of the registers r0-r12. This means we have to know what's in this register to decide whether interrupts are disabled or enabled. In some cases checking the instructions just before the MSR-instruction can do this, e.g. when the interrupt bits of the local register are cleared using a BIC (Bit Clear) instruction. But in other cases, there is no certainty for what this register contains before the status register is changed, e.g. when the register is an argument into a function that changes the interrupt mode. There are also a great deal of lock-push and lock-pop in the assembly source files. For the code that means a LDR instruction, where a previously stored interrupt state is stored, and the status register is then changed accordingly to this previous state. In this case we also need to know what's the register contains in order to determine the type of interrupt change correctly. This could possibly be achieved with a more complex data analysis of the program flow.

The solution, or compromise one might say, was to label these hard-to-determine interrupt changes as 'unsure' and let the user look deeper into the specified function if the actual interrupt change is desired.

The thought behind detecting the interrupt changes at this stage was that the information was going to be used in the later stages, when constructing the basic blocks. But as it is not possible to determine an exact physical address for the change just by looking at the source code, the information can be used to compare with the basic blocks graph and see that each function contains the right number of interrupt changes.

Now we needed the physical starting addresses for each function. What we had to work with was the input ELF (Executable Linkable File) that contains information about each part of the execution region, for example where each function is located. First, an object dump was made of the ELF into a temporary file by using the Cygwin binutils command 'objdump –x <elf> > temp_file' (again, the GNU binutils are just as good as Cygwin in this case, but we chose to use Cygwin).
The temporary file was then parsed with an AWK script to dispose unnecessary information, before reading the function names and their start-addresses. The result was put in a function database so that it could be used when constructing basic blocks. Each function was then compared with the result from the source code analysing and when a function was found, its properties (start- and stop-line in the code file, di, ei, unsure interrupt changes and object code file) were added to the function object in the database.

All functions could not be found in this search because of two reasons:

First, the ELF contains all the functions in the entire executable program, and if not all the source code files of the program are given as input to the tool, all functions will not be found. Only the functions in these source code files will be found in the ELF.

Second, to keep OSE portable, OSE Systems has changed the names of some of the functions to a format that can be handled by all linkers, even older versions that can't take care of e.g. long function names.
To get the real function names, one has to look in a translation table that is kept safe from outsiders. Hence, some function names will be compared to the translated name and therefore not be applied it's properties.

6.2 Decoding binaries

This concluded the first part of the tool, the function handling. Next step was to translate the ELF binaries of each instruction to a suitable instruction format, which could be used in the creation of the basic blocks. Figure 6.2 shows an overview of this stage.

As a start, the binaries were read into a list with pairs of an instruction binary and its physical address. There were some problems that had to be solved here. The executable code image for ARM consists of different regions, namely data, THUMB- and ARM-regions (see chapter 4). Where these regions start and where another ends can be found by looking at the symbol table in the map-file created from the ELF. A

separate list with the region information was created from this table and used when reading from the binaries. First, the idea was to filter all the data regions at this stage, because they didn't contain any code that should be included in the basic blocks graph. But later in the project we discovered that at certain places in the code, conditional jumps were taken into some of the data regions, and that the code here looked as a normal subroutine with a normal return to the calling function. There are two kinds of data regions in the symbol table: d and f. When looking at the disassembly code in the ARM debugger I couldn't make out any difference between them, except that a small number of the f-regions looked as it actually was 'real' code. Most of the data regions looked, as expected, as total nonsense code.
The solution to this was, as I mentioned, to include the data binaries in the list and to take the different regions into account when constructing the basic blocks. This will be further discussed later in this chapter.

Now we have a list with all the instruction binaries, so the next step is to decode the binaries into a suitable format. As help, I received an ARM binary decoder from IAR Systems, which translated the binaries into a c-struct with a number of properties for each instruction (see figure 6.2). The decoder had to be slightly modified to suit my special wishes, but was very helpful in the translation. This c-struct was then used to create a new ARM instruction object from a format that I specified, which can be seen in Table 6.1.

This object has a number of properties to help the building of basic blocks. When an instruction is decoded, it's compared with the function database to see if it is the first instruction in a new function. If so, a variable 'functionstart' is set to true and later detected in the basic block creation. Each ARM instruction subclass (one for each instruction format [20]) has one important function, the print function. This is called when the TCD file for the basic blocks graph is constructed, and prints the instruction with its operators on the format specified for the ASTEC WCET calculation tool.

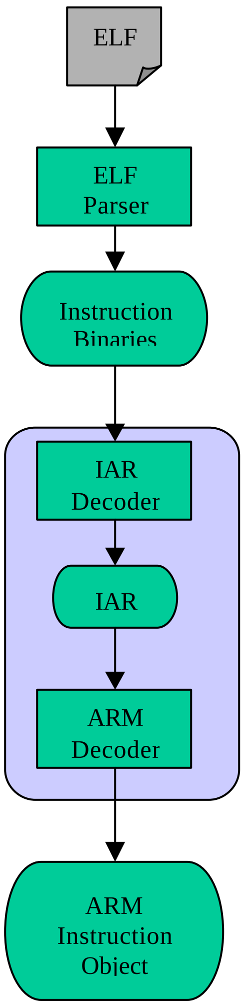


Figure 6.2 Parsing the ELF and decode ARM instruction

In the decoding stage, every change of the Interrupt State is detected again. It's necessary to have the exact physical address location of each change when the basic blocks are created and therefore the detection has to be done here. The interrupt properties for each function collected earlier can, when the program is finished, are used to check if the analysis was correct or not.

There are three types of interrupt state changes in the OSE source code. The first and simplest one is when the interrupt bits in a register is cleared or set right before it's applied to the status register. Then a Disable or Enable Interrupts can be determined directly.

A second variant of interrupt change is by storing the Interrupt State on a stack before disabling interrupts. When a lock pop is performed, it's done by restoring the previous Interrupt State from the stack, which therefore can lead to that interrupts still are disabled if the interrupts were disabled before the lock push. The reason to have this kind of lock push and lock pop instructions is that some parts of the code has to execute with interrupts disabled regardless if they were enabled or disabled before. This means that regardless if the calling function or subroutine has interrupts enabled or disabled, we are always certain that our little piece of code is executed safely and that the caller continues to execute unaffected by the interrupt changes in our code.

The third type of interrupt change is when a register with unknown content is used to change the status register. This occurs e.g. when the register contains an argument to a function that uses this argument to change the Interrupt State. Then it depends on what the calling function or subroutine had stored in this argument register before the call.

This tool only handles the first type of interrupt change completely. As in the first part of the tool, when interrupt changes were detected by looking at the source code, the changes are labelled according to its type. Direct changes are labelled Disable and Enable, lock push is detected as a Disable also at this stage and lock restore is labelled Lock Restore. The changes with unknown register content are labelled Unsure. This part of the tool has to be extended to fully take care of all types of interrupt changes and get a correct estimation of all regions of code were interrupts are disabled. Again, this can probably be done with a more complex data analysis of the program flow.

All interrupt changes are saved in a list with its type and the physical address of the change instruction (MSR Move Register to Status register). This list is then used in the creation of basic blocks.

Table 6.1 ARM Instruction Class

Member variable	**Type**	**Description**
cond	short	Condition code, e.g. 0xE for CondAL [20]
format	int	One of 15 instruction formats
binary	uns. int	The instructions 32-bit binary
instr_adr	uns. long	Physical address for the instruction
instr_name	char *	Name of instruction, e.g. "Bx"
cond_name	char *	Name of condition code, e.g. "CondAL"
change_pc	bool	True if the instruction changes the PC
updated_regs	short	Registers changed by the instruction
update	bool	True if a register updates itself, e.g. mov r0 <- r0+1
isthumb	bool	True for THUMB instructions
fstart	bool	True if instruction is first in new function
offset	int	Used to calculate jump destination addresses
targetadr	int	Destination address for jump instruction
bits	short	Condition bits, e.g. S for set conditions in status register
rn	short	Operand register
rd	short	Destination register
op2	Operand	Special class for second operand
Member functions	**Return type**	**Description**
Print()	string	Prints the instruction on the ASTEC ARM instruction format
FindCondName()	char *	Returns the name of the instruction condition, e.g. "CondAL"
IsJump()	bool	Checks if an instruction is a jump instruction

When we started the project we thought that all instructions were 32-bit ARM instructions (see chapter 4). But when we started decoding the instruction binaries we discovered that was not the case. Some parts of the operating system not critical to performance, executed in 16-bit THUMB mode, this was in the same time as we discovered the constant data regions. As THUMB instructions are just a subset of the ARM instruction set [20], the easiest way to adopt the tool for THUMB was to make the new THUMB root class a subclass of the ARM Instruction root class (see Table 6.1). That took about one extra day to implement.
In the decoding stage the target address for each jump is calculated. This is very straightforward with normal ARM instructions and most of THUMB instructions were the jumps are calculated directly with an offset. There is one special case however, THUMB Branch with link instructions (BL) are divided into two parts, to allow longer jumps. That just meant we had to save the first offset temporarily and calculate the target address when decoding the second branch with link instruction.

6.3 Construction of basic blocks

So, now we have a list of ARM Instruction objects, a list of regions, a database of functions and a list of interrupt changes. That is enough to start dividing the instructions into basic blocks. Figure 6.3 shows the basic steps in the process of creating basic blocks.

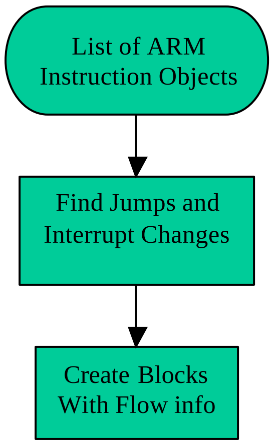

List of ARM Instruction Objects
Find Jumps and Interrupt Changes
Create Blocks With Flow info

Figure 6.3 Creation of Basic Blocks

The definition of a basic block is a group of instructions that always execute all of the instructions or none at all, and always in the same order.

Even if we only want to calculate the WCET of a small region in the code, all basic blocks of the entire program have to be created. This is necessary because the little region we want to examine might contain a subroutine call to a function in a totally different region, and if we haven't created the basic blocks for that region, we cannot calculate a WCET for it.

In ARM, all instructions are provided with a conditional (see chap. 4), telling whether the instruction will be executed and if so under what condition. The most common is obviously the always condition (CondAL), i.e. that the instruction will always be executed if it's in the execution path.
Strictly, every instruction that is not a CondAL instruction should form a separate basic block as shown in figure 6.4.

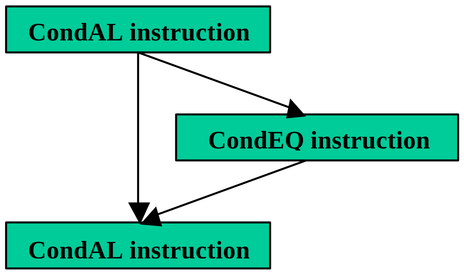


Figure 6.4 Separate blocks for each conditional instruction.

But that would lead to a tremendous amount of blocks, and the WCET calculation would be very complex. We decided to let conditional instructions to be considered as CondAL instructions and form basic blocks together with other instructions. This makes our WCET estimation not as tight as it would be if we let every conditional instruction form its own block, but the gain in simplicity is greater and our estimation will still be correct.

The bounds for a basic block are:

Start of new block:

- The instruction before was the end of another block.
- There is a jump somewhere in the code to the current instruction address (this will be further discussed later in this chapter).
- The current instruction is an enable interrupt or unknown interrupt change.
- The current instruction is the start of a new function.
- A new region starts at the current instruction address.

End of block criteria's:

- The current instruction is a jump instruction.
- The instruction following the current one is the start of a new block (see criteria's above).
- The current instruction is a disable interrupt, lock restore or an unknown interrupt change.

Because unknown interrupt changes are in both criteria's, these instructions will form separate blocks.

If an instruction is a jump to an address that is higher then the current (i.e. no block has yet been created at that address), an empty block is created at the target address and the target address is put in a queue of empty blocks. This queue is checked for each new instruction and when the start instruction for the empty block is found, the preceding block is finished and the empty block is filled with instructions.

If the target address for the jump instruction is within an already created block, that block is split in two blocks at that address, as shown in figure 6.5.
The new block names (BB1p and BB1c) stands for parent and child and will is applied each split so that each block have a unique name in the database. This can lead to block names with several p's and c's in them, e.g. "BB1ppcp".

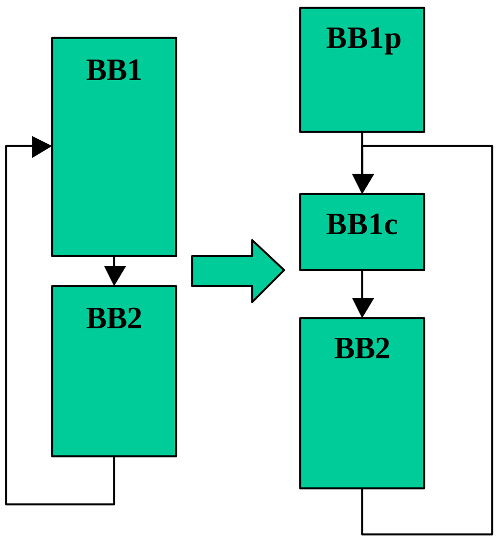


Figure 6.5 Splitting a basic block

When a new block is created, it is inserted into a database containing all the basic blocks that have been created. This database consists of a std::map in c++, where every basic block is indexed by the physical start address of the first instruction in the block.

To help the finding of interrupts and building of a scopegraph, symbols are applied to the block names according to what kind of block it is.
First, each block is given a name in increasing number order ("BB1", "BB2"...), except if the start address for a block is found in the function database. Then that block will have the same name as the found function, e.g. "foo".
Second, all blocks names are applied their region, "_a" for a block that executes in ARM mode, "_t" for a block that executes in THUMB mode and "_d" or "_f" for a block that contains constant data. The region is attached to the end of the name, e.g. "BB1_a" or "foo_t". This is the most basic form of a block name.
If the block is the start of a function or a subroutine (The target block for a branch with link jump is said to be the start of a subroutine) a "f_" is applied to the beginning of the name, resulting in "f_BB1" etc.
If the last instruction of the block is a branch with link instruction (also discussed later in this chapter), "b_" is applied to the name, in the beginning, but after "f_" if present.
Last, the interrupt blocks are applied the type of interrupt change to their name, "di_" for disable interrupt, "ei_" for enable interrupt, "lr_" for lock restore and "ic_" for unknown interrupt change.

All of these naming conventions makes the block names a bit complex, but it makes it very easy to parse the basic blocks graphs for e.g. interrupt chains. The longest possible name would be: “f_di_ei_function-name_a”, saying that the block starts a new function or subroutine, the first instruction is a disable interrupt, the last instruction is an enable interrupt and the block executes in ARM mode.

Every block has a list of predecessor blocks and a list of successor blocks. These can be empty if the block is a single block function that never is called, but for most blocks they are not both empty. The predecessors and successors form the program flow information, i.e. which paths a certain block can take and what paths that could have lead to this block.

A basic block is given its flow information facts during the construction of basic blocks. If the preceding block of a newly created block did not end with an unconditional jump, that preceding block is put in the list of predecessors of the new block. And respectively, the new block is put in the list of successors for the preceding block.

There is also another case for adding a successor or predecessor, and that is when a basic block ends with a jump instruction.

There are three kinds of jump instructions in ARM: Branch (B), Branch with link (Bl) and Branch and exchange (Bx).

Branch instructions are used when there is no need to save the return address, that is for internal jumps and PC-relative jumps.

Branch-with-link instructions are used for subroutine calls when the return address has to be saved. This address is then put in the link register (Reg 14 in ARM) before the jump is taken.

When a subroutine ends, its final instruction is a Branch and exchange. Then the PC-value is restored, either by what’s in the link register, or in which other register that is given as argument for the branch and exchange instruction. Because the branch instruction is word aligned, the final bit can be used to change mode between THUMB and ARM. If the least significant bit in the branch instructions offset is set, then the mode of the code starting at the restored PC-value is THUMB mode.

When a Branch and exchange has a register that is not the link register as argument, the value of that register has to be known in order to give a correct basic blocks graph and scope graph.

In our WCETprepare tool and in the scope graph creator, we consider all branch with exchange jumps as a return from subroutine. This problem is very similar to that of finding the lock push and lock pop discussed earlier, and can probably be solved by using a more complex data flow analysis.

So, to get back to the flow information, there are some considerations that need to be done.

For a normal Branch instruction there is no problem, if it’s an unconditional jump (B CondAL), only the target block is put in the successor list. If it’s a conditional Branch instruction, both the target block and the block at the next address (in ascending order) are put in the list of successors.

Branch and exchange instructions are always unconditional jumps, so no blocks are put in the successor list.

But for unconditional Branch with link instructions the block in ascending address order after the jump block will be taken as soon as a return from subroutine is reached. Should then the jump block be in the predecessor list for the block that is taken after return from subroutine? And should that block be in the successor list for the Branch with link block? The answer is of course no in both cases, because there is no direct execution path between the two blocks, but we had to adjust the WCETprepare tool a bit to make the automatic creation of scope graphs work. This is where the “b_” in the jump block comes in. Every time the scopegraph converter finds a block with ”b_” in the name, it knows that the block at the next address should be included in the scope, even though it is not in the successor list.

Also, every time a subroutine is called with a Branch with link instruction, the called block gets a “f_” in its name so that the scope graph converter and the WCET calculate tool recognize it as a function.

When a basic block is split into two blocks, due to a jump where the target is an instruction within an already created block (see earlier in this chapter), the flow information is updated for that block. The successors of the parent block are transferred to the child block, and the only successor the parent block has left in the list is its child block. The child block of course puts its parent block in its predecessor list.

6.4 Creation of Basic Blocks Graph

When all blocks are created, and all flow information have been put into the blocks, its time to create a basic blocks graphs and TCD-files. A TCD-file is a file that contains the basic blocks graph, with all the blocks and their instructions, on a special format that the WCET calculation tool can recognize. The format is basically:

```
Begin block name
        List of predecessors
        List of successors
                Instructions
End block

Next block…
```

The instructions are printed on a format that looks like this:
<Address> : Instruction length : Instruction-name : Condition : Operands ;

The tool is able to build three kinds of basic blocks graphs: one big for the entire program, one for each function in the program and finally one for each Enable - to Disable Interrupt chain.
The graphs for each function are constructed by stepping through all the blocks, and when a new function is found, a tree is built starting from that block's successors, until there are no successors left. Then a print function is called that only prints the blocks in the tree. If a block has a predecessor that is not in the tree, that block is not included in the predecessor list.

The same strategy is used when the basic blocks graphs for the Disable- to Enable Interrupt chains are created. A brief overview is given in figure 6.5. Here a tree for each Disable Interrupt is built at first. When the tree is finished, it is searched for any Enable Interrupt blocks. For each Enable Interrupt found, a backward search is performed, until the starting Disable Interrupt block is found. This is done because we want to avoid all unnecessary blocks that are not in the execution path for that particular chain. The backward search also has to consider unconditional jumps to subroutines, e.g. if a block ends with an unconditional Branch-with-link instruction. Then that block will only have the starting block of the called subroutine in its predecessor list, but the execution continues with the block at the next address when the subroutine is finished. If such a block is found during the backward search then the whole subroutine is added to the chain, unless an Enable Interrupt block is found in the subroutine. Then that path is excluded from the chain.
Also, if an Interrupt change that is of unknown type, e.g. a LOCK_POP (see earlier section in this chapter), then the backward search is stopped and the chain is discarded.
When everything goes well, a file with the name DI<address>-EI<address>.tcd is created for each chain.

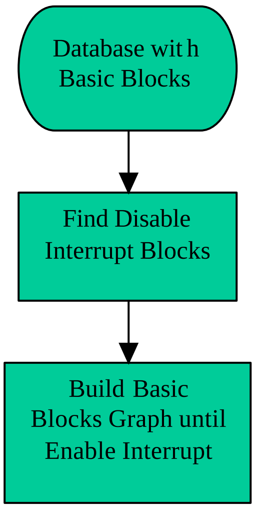

Database with Basic Blocks
Find Disable Interrupt Blocks
Build Basic Blocks Graph until Enable Interrupt

Figure 6.6 Building a Basic Blocks graph for each Disable to Enable Interrupt chain

Because the calculation tool developed by ASTEC that is used in the last step of the process checks the names of the basic blocks for certain properties, e.g. if a basic block is the start of a new function (block name starts with "f_") or if a basic block calls a subroutine (block name contains "b_"), some modifications had to be made to the block names. In this case study we make basic blocks graphs for Disable to Enable Interrupt chains. That means that we stop constructing the graph when we find an Enable Interrupt. Sometimes a subroutine is called in the middle of the chain and Interrupts are enabled before a return from subroutine comes. But the calculation tool expects to find an end for each function it detects, i.e. for each block that begins with "f_", otherwise an exception is raised and the calculation stops.

This means we have to temporarily change the name of some blocks inside the chain if no return from subroutine is found. It's a kind of "flattening" for unfinished functions so that for the calculator it looks like the half function is a part of the main function and not a subroutine. The same goes for the basic bock that calls the subroutine. That block will at the beginning have "b_" inside its name, and when the calculator detects that kind of block it expects a return from subroutine somewhere in the following execution path otherwise an exception is raised. So the basic blocks with "b_" are also changed so that it looks as if they call a normal basic block inside the current function.

All the basic blocks names are then restored in the last stage when the whole graph is constructed and a .tcd file has been created.

7 Experiments

In this chapter I will present some information about the results from the test runs of the tool together with the scopegraph converter and WCET calculation tool from the ASTEC group.

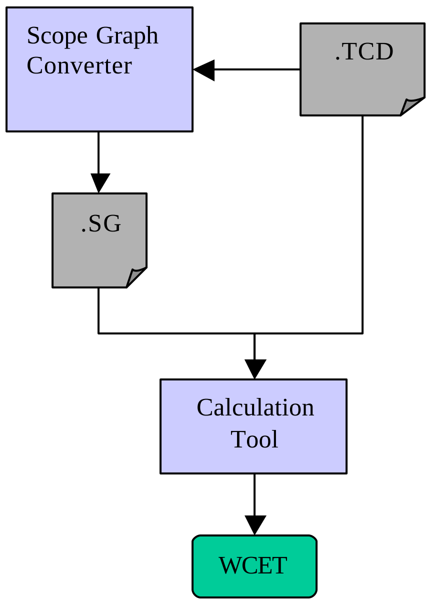


Figure 7.1 Last step in the work chain, calculating the WCET

The .tcd files containing the basic blocks graphs of the found Disable to Enable Interrupts chains are run through an automatic scopegraph converter that produces a .sg file with the scopegraph for the chain as shown in figure 7.1.

The definition of a scope is, as explained in the chapter about WCET theory, that each scope corresponds to a repeating or differentiating execution environment in the program, e.g. a function call or a loop, and can contain one or more basic blocks (see figure 3.2, page 4). All scopes are also supposed to be looping, even if they iterate equal to or less then one time. The calculator needs both one basic blocks graph and one scope graph of the same chain to calculate the WCET. The scopegraph if needed to get the loop bound for the loops within the chain.

So after the scope graph is constructed, one have to find the loop bounds manually, by first looking at the code in the basic blocks graph if the number of iterations can be found directly. If its not possible to determine the bound directly one have to find the corresponding source code to see if the loop bound can be found there. Often the loop bound is set by the size of a vector or variable and therefore hard to determine. One possibility is to look if there exists a maximum size for that vector or variable in a header file that is included in the source file.

I had some problems to find the corresponding source code for the basic blocks graphs. This could probably be done pretty easily with the ARM Debugger, but I couldn't find the right settings. So what I did was I looked at the physical starting address for each chain and then compared it with the starting addresses of the functions that I calculated in the first step of the tool. This method is not the best since if the chains start in a subroutine relocated between two functions, then the source code for that chain doesn't have to be in the same source file as that function. And if it is, the code could be placed in another section in the source code file. Of the chains I looked at I could only find the correct loop bound for one, because of that I couldn't trace the code back to the actual source code.

The total number of Disable to Enable Interrupts chains found in the compiled operating system kernel: 612

This is approximately half of the total number of interrupt chains in the operating system kernel. The other half is from those where storing the state on a stack or restoring the state from a stack, i.e. lock-push and lock-pop instructions, makes the interrupt state change.

Chains that contain 3 or fewer basic blocks: 554

That means that more then 90 percent of the found chains are very short. This was expected, because interrupts are not supposed to be disabled for a long period of time, except during boot.

I've chosen 10 of the chains that were produced by the WCETprepare tool, to look a little closer on the properties. These chains either contain a number of loops or are longer then most chains. Table 7.1 shows the properties of the selected chains. Of these chains I could only determine the loop bound for one of them directly because I could not trace the chains back to the source code as I mentioned above.

Notable is also that because not all types of interrupt changes are handled (see previous chapter), there are a lot of chains that will not be detected by this tool. One of the most common ways to disable and enable interrupts in the OSE operating system kernel, is by saving the locks (disable) and unlocks (enable) on a stack, i.e. through lock push and lock pop instructions. This is to make a section of code execute with interrupts disabled no matter what state the calling function was in when the call was made. This means that the calling function doesn't need to worry about if the interrupt state has changed after the subroutine has finished. The interrupt state that was before the call will be restored and the execution can go on as normal.

The types of Disable to Enable Interrupts chains that are found by this tool is typically when interrupts need to be disabled for a very short period of time, e.g. in memory handling when a sensitive system variable needs to be updated and mutual exclusion is needed to secure a safe execution. And as said in the beginning of this chapter, 90 percent of the found chains consists of 3 or fewer basic blocks, i.e. very small. An explanation to the chains containing loops in Table 7.1.

The WCET is given by the number of cycles for the basic blocks outside the loop(s) plus the number of cycles for the basic blocks inside the loop multiplied by the number of iterations. When the number of iterations couldn't be determined directly, e.g. when the loop bound is set by a variable, the limit has been set to 100, 50 and 20. This is to show the big time contribution loop is. When the loop bound gets high, then the WCET gets worse fast. If the unsure loop bounds are set to 100 it's sure that no underestimations are made. The probability that there are chains where interrupts are disabled containing loops that iterates more then 100 times is very small. When there is a nested loop, as there is in two cases, the total time for the nested loop is calculated and added to the outer loops basic blocks total time before multiplied with the outer loop bound.

Table 7.1 Selected Disable to Enable Interrupts chains

Chain	Size	Blocks	Loops	Nested	Scopes	WCET (cycles)
DI156588-EI156828	244	11	2	-	4	64 + 25*(20) = 564 64 + 25*(50) = 1314 64 + 25*(100) = 2564
DI159444-EI159608	168	7	-	-	2	45
DI181296-EI147608	200	9	-	-	2	85
DI182248-EI147608	308	16	-	-	2	89
DI183924-EI183796	160	11	1	-	3	51 + 23*(20) = 511 51 + 23*(50) = 1201 51 + 23*(100) = 2351
DI183924-EI184392	160	11	1	-	3	63 + 23*(20) = 523 63 + 23*(50) = 1213 63 + 23*(100) = 2363
DI185276-EI185524	248	11	-	-	2	86
DI194280-EI194376	100	6	-	-	2	31
DI230288-EI230508	240	12	3	1	5	29+(7*32+29)*(20) + 14*(20) = 5369 29+(7*32+29)*(50) + 14*(50) = 13379 29+(7*32+29)*(100) + 14*(100) = 26729
DI261180-EI261432	212	9	3	1	5	28 + 14*20 + (16+7*32+6)*20 = 5228 28 + 14*50 + (16+7*32+6)*50 = 13028 28 + 14*100 + (16+7*32+6)*100 = 26228

The loop bounds in the table that could not be decided directly are set to 20, 50 and 100 to have something to compare with. In the real case however, this is probably an huge overestimation, since most loop bounds are kept low in the speed critical operating system kernel.

Diagram 7.1 Distribution of chai n sizes in bytes

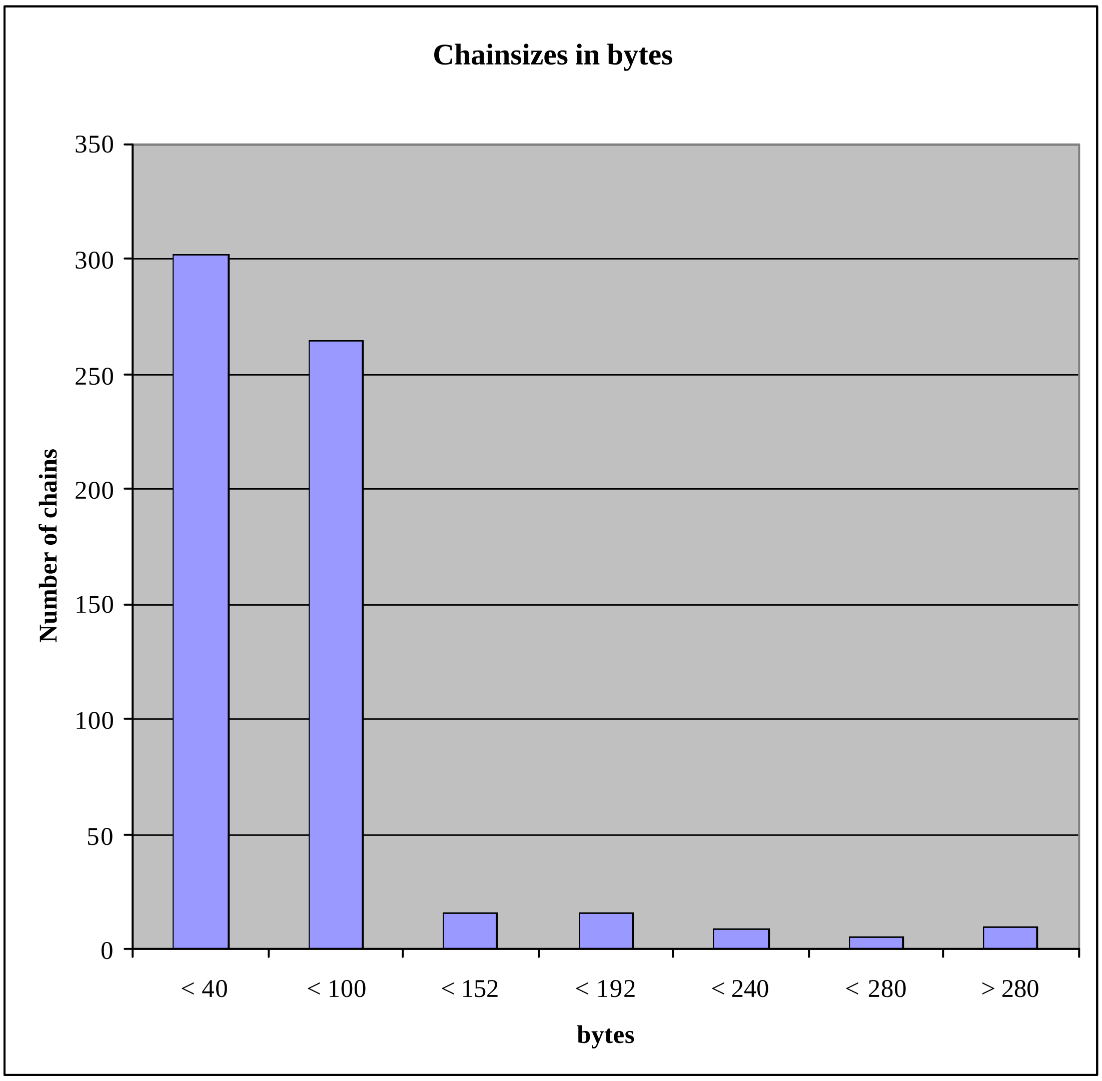


Diagram 7.1 shows the size distribution of the 612 found chains. This also confirms the fact that almost all Disable to Enable Interrupt chains are very short.

7.1 DI-EI chain properties

All the times given in the blocks are given in clock cycles. The negative values between some blocks are the pipeline effects.

The first chain is shown in figure 7.1 and contains 11 basic blocks with 2 loops. In Appendix B there is the corresponding scope graph file. In this chain I have attached the actual name of each basic block to make it possible to follow the scope graph file in Appendix B.
The loop bounds could not be found directly by looking at the basic blocks graph with the translated instructions. Therefore the chain needed to be checked against the source code file to get the correct bounds.

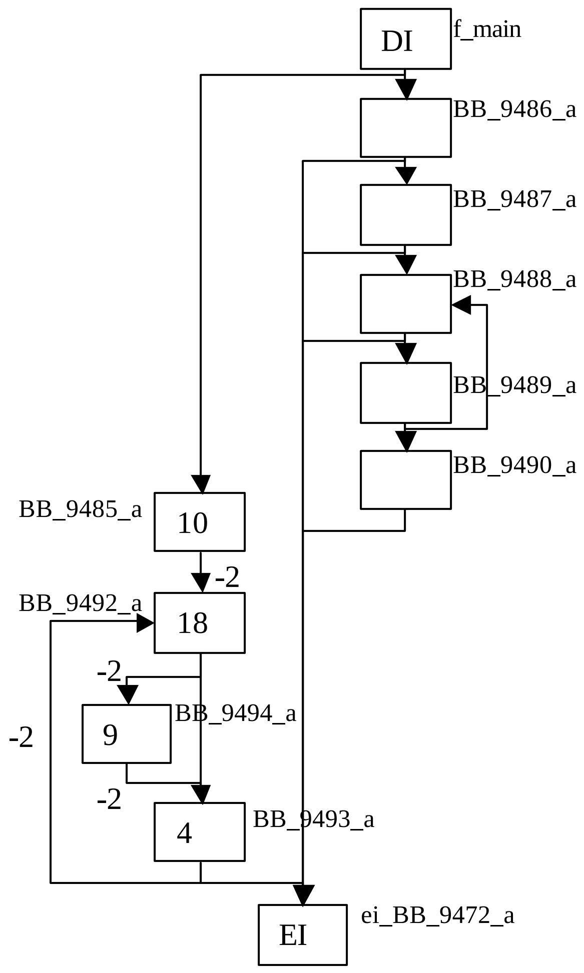


Figure 7.1 DI156588-EI156828

Here we see that only one of the loops will be taken, since they are in different execution paths. So, the path with the highest loop bound will most likely give the WCET.

Largest block (last block): 18 instructions with WCET: 25 cycles

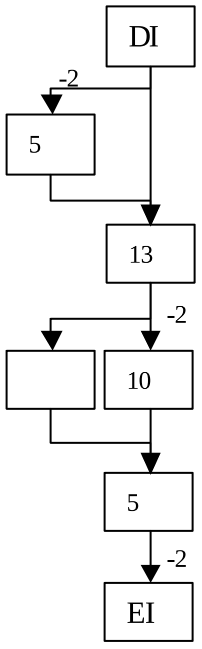


Figure 7.2 Chain DI159444-EI159608

This is the second shortest of the selected chains, with a WCET of 45 cycles.

First block: 13 cycles, last block: 5 cycles

Largest block: 9 instructions with WCET 13 cycles

Here we can see that the longest path search works, as the execution path is given by the blocks with number of cycles information inside them, and the block not taken has a WCET of 8 cycles.

Otherwise this is a pretty normal DI to EI chain, with not so many blocks in it, no loops and no function calls.

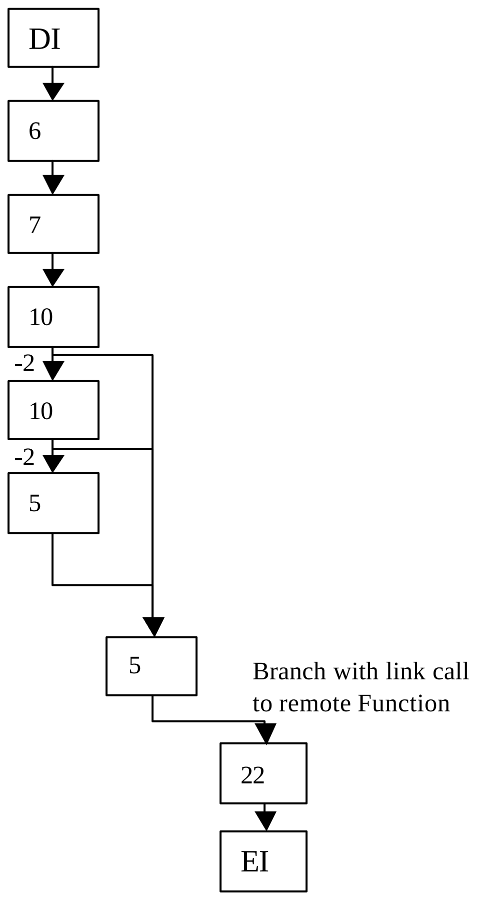


Figure 7.3 Chain DI181296-EI147608

In figure 7.3 we see a chain that contains a call to a subroutine, which enables the interrupts. This is also very common, often there is a function called something like "clear_interrupts" that is a function calls instead of turning interrupts on inside it.

First block: 13 cycles and last block: 12 cycles.

Largest block: 10 instructions with WCET 22 cycles.

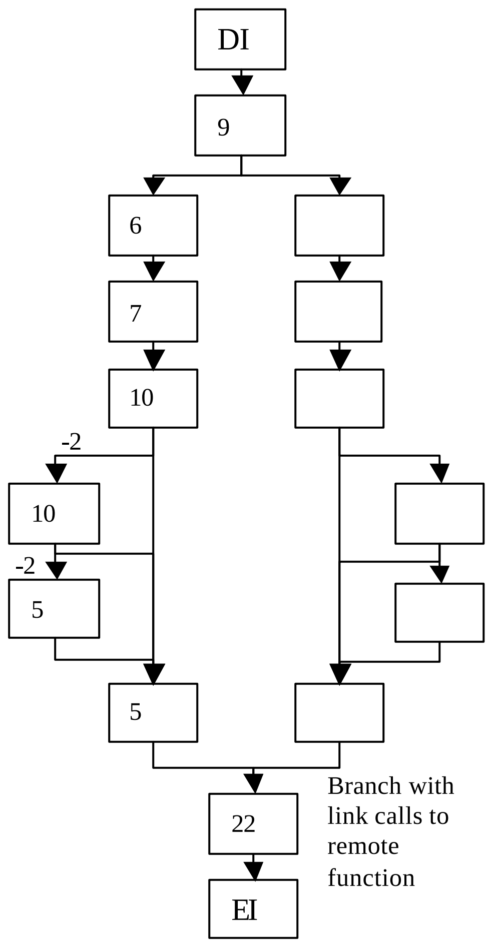


Figure 7.4 Chain DI182248-EI147608

Another call to the same subroutine, but from another function, is shown in figure 7.4. Here there are two almost identical paths, but one contains more instructions then the other, and therefore has the highest WCET.

First block: 7 cycles and last block: 12 cycles.

Largest block: 10 instructions with WCET 22 cycles.

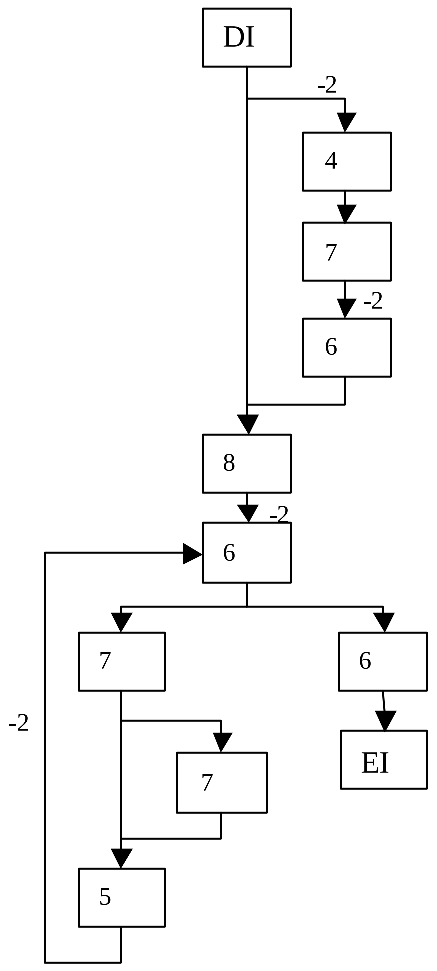


Figure 7.5 Chain DI183924-EI183796

This is the first chain containing a loop. It's also one of the larger of the selected chains.

The loop bound could not be determined directly, and is therefore set to a high number in the calculation to be on the safe side. Here we can see that the WCET rapidly gets high, when the number of loop iterations increases, as the WCET is given by:

51 + 23*(number of loop iterations)

First block: 13 cycles and last block: 5 cycles.

Largest block: 6 instructions with WCET 13 cycles.

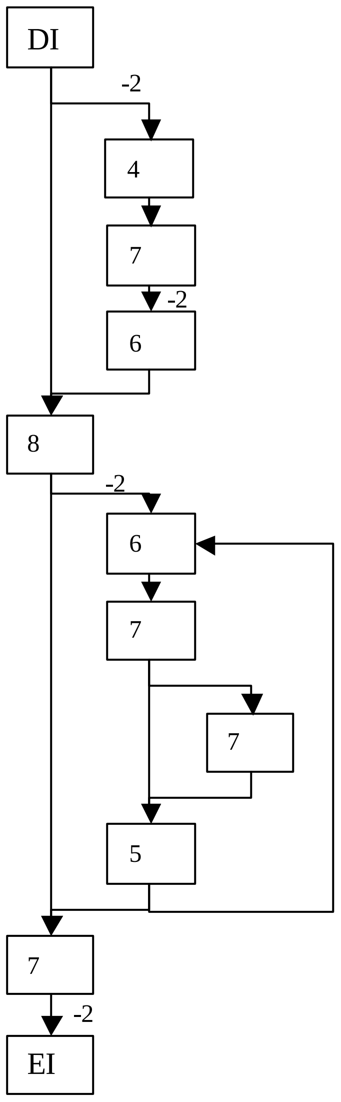


Figure 7.7 Chain DI183924-EI184392

This chain is similar to the previous example, but here the enabling of interrupts is done in a different place. Also the execution path is a little different in this example and therefore the WCET is a little different.

First block: 13 cycles and last block: 5 cycles

Largest block: 6 instructions with WCET 13 cycles.

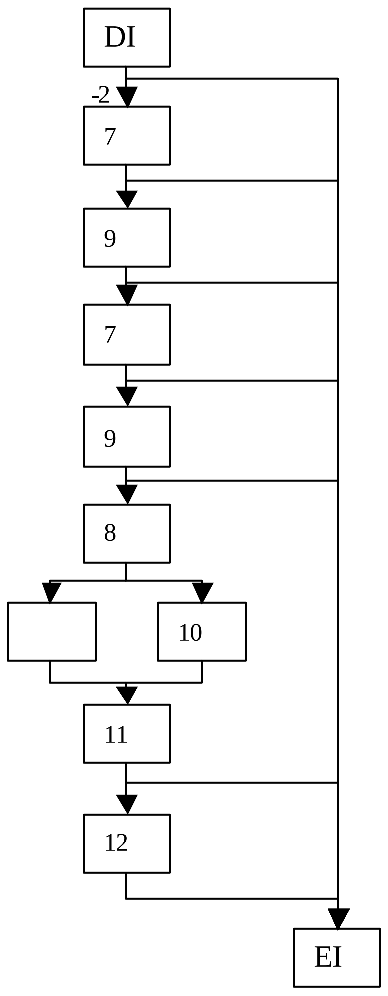


Figure 7.7 Chain DI185276-EI185524

Pretty straightforward chain where the longest path is found from a number of possible chains. But it's still very unusual to find paths this long with interrupts disabled in the results from my experiments.

First block: 22 cycles and last block: 5 cycles.

Largest block: 14 instruction with WCET 22 cycles.

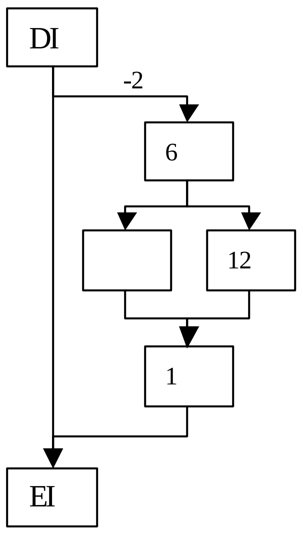


Figure 7.8 Chain DI194280-EI194376

This is the smallest chain that's included in the selected chains. That gives an idea about how the more usual chains look like, a bit smaller then this one. The most common chain contains only 3-4 blocks and rarely any conditional. This is logical, since often you want to change a variable or register that is sensitive and disables interrupts just long enough to make the change.

Here ones again, the longest path is found (the execution path is given by the blocks with cycles inside them) and the WCET can be calculated, in this case it's 31 cycles.

First block: 9 cycles and last block: 5 cycles.

Largest block: 8 instructions with WCET 12 cycles.

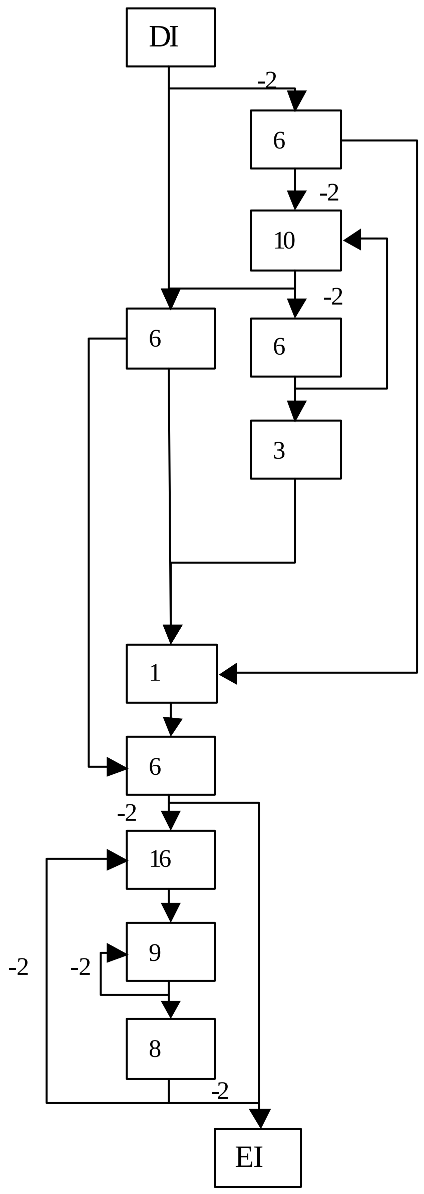


Figure 7.9 Chain DI230288-EI230508

This chain is one of the two most complex chains I found during my experiments. It contains three loops, where one of the loops is a nested loop. Again, the loop bounds where hard to determine, but the small nested loop has a loop bound of 32 iterations. And as you can see, it doesn't contain a lot of code, so to set the bound for the larger loops to 100 iterations are on the safe side.

First block: 18 cycles and last block: 5 cycles.

Largest block: 11 instructions with WCET 18 cycles.

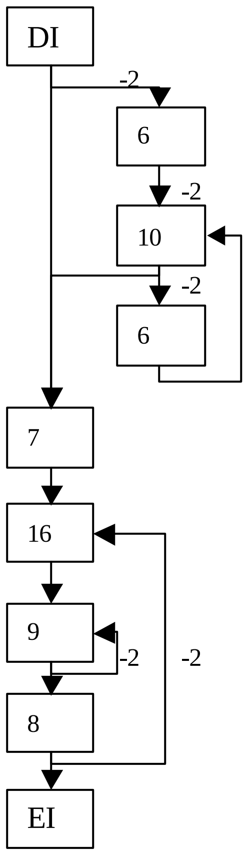


Figure 7.10 Chain DI261180-EI261432

Very similar to the previous chain, also with three loops, where one of the loops is nested. Also this nested loop has a bound of 32 iterations.

First block: 14 cycles and last block: 5 cycles.

Largest block: 11 instructions with WCET 16 cycles.

8 Conclusions

The first, and most important, conclusion of this project is that we have showed that it is possible to use WCET analysis on real-time operating system code. But there are still a number of issues that can be improved, these will be further discussed in the next chapter, future work.

The current tool is not compared to real WCET numbers from physical tests. This is because with this tool only half of the chains from Disable Interrupt to Enable Interrupt could be found and therefore one cannot be sure that the WCET is found. It would be very interesting to see how good the results from the tool are, but only if its certain that the WCET is found.

I think the tool can be very helpful to determine the WCET of a desired region of compiled operating system code if the improvements that are suggested in the next chapter are implemented. The biggest downside right now, as I see it, is that each loop bound needs to be determined by hand. If you have to analyse a large piece of code, such as an operating system kernel, this can be very time consuming.

Another downside, as we discovered after the project, is that the tool is compiler dependent. This is because all the instruction binaries and MAP files are generated from code that is compiled and linked. In this project the ARM C-compiler armcc is used, but when the tool was tested with object code compiled by an IAR compiler, it did not work.

8.1 Detailed Conclusions

In the beginning of the project, we set up a number of goals and wishes, which we wanted as results:

* See if it was possible to convert operating system code into a format that could be used to statically calculate the execution time of specified regions of code.

* How much of the process of converting operating system object code into basic blocks and basic blocks graphs could be automated and how much had to be done manually.

* Look at the typical properties of operating system code, such as nested loops, function pointers, recursion and so on.

The answer to the first goal is yes, but it was not as easy as we first thought when we started the project. A number of unpredicted problems occurred that we had to solve, such as how to handle the conditional branch with link jumps and adapt the tool for THUMB mode. But in the end we were able to convert the source code like we wanted in the beginning (see work overview, Appendix A). Very helpful in the later stages, during the tests, was the scope graph converter developed by ASTEC in Uppsala that converts a basic blocks graph into a scope graph. This would otherwise be done manually and that would have taken a lot longer time for such large pieces of code.

We had to make some assumptions along the way though, in order to get it to work. First we assumed that the code didn't contain any THUMB code, but that proved to be false so we had to adopt the tool. Another assumption was that a Branch-and-exchange jump (Bx instruction [20]) always means return from subroutine. This is true for almost all cases, where the operand of the instruction is the link register (register 14 in the ARM9 family [20]). But sometimes the operand is another register and then the result depends on what is saved in that register before the jump is taken. In most cases it is the link register that is temporarily saved in that register, but sometimes the content is undefined, e.g. when a switch table is used. We then assume that the meaning nonetheless is return from subroutine. The exact content can probably be determined by a larger data analysis of the program flow, just as for the determining of interrupt types mentioned in the prototype implementation chapter. Another assumption was that a subroutine called by a Branch-with-link instruction [20] always has to end with a Branch-and-exchange instruction. Because when the subroutine is finished, the execution starts at the address after where the Branch-with-link jump was taken, i.e. the content of the link register. This is not necessarily an assumption since it's logical and we couldn't find any case where this was not true.

Another of the things we discovered was, discussed in the implementation chapter, some peculiarities with the constant data regions.

There are two types of data regions in the ELF, marked as ‘d’ and ‘f’ in the symbol table. When looking at the code in the ARM debugger, all the ‘d’ regions are, as expected, just constant data, and all the ‘f’ regions looked to be the same. But when we worked on the control flow between the basic blocks, we found blocks that jumped from normal ARM execution, to ‘f’ blocks. And the ‘f’ blocks performed a normal execution with return from subroutine when finished. The conclusion was that not all ‘f’ regions consist of constant data, but are infact executable code. There was no information on what exact difference there are between the two data region types as far as we could see, but as we treat all blocks the same it doesn’t effect the WCET calculation.

The second goal was to make as much of the process as possible automated. Because the tool developed in this project is used on large code sections, here the kernel of an operating system, a great deal of manual interference would make it very slow to calculate the WCET. All the finding of interrupt changes, function names and their starting addresses in the tool are done automatically. Also, all conversions from binaries to the new ARM instruction format are done automatically. And the creation of basic blocks of code and their graphs are also automated. The parts where manual interference is needed are during the creation of scope graphs and the finding of loop bounds. To create a scope graph, one only need to start the tool from ASTEC, which constructs a scope graph file from a basic blocks graph file. It should not be too difficult to integrate this automatic scope graph converter into the tool and make that part automated as well. But determining the loop bounds is very hard to automate, since it depends on the circumstances, e.g. when a constant set the loop bound or if a pointer or variable determines it.

But the rest of the process is automated. All the program needs to run is an ELF (Executable Linkable File) of the program and a text file containing the paths to the source code files that are to be examined.

Third goal was to look at typical properties of operating system code. There are a number of aspects that one can look at the properties of source code, such as function pointers, loops and nested loops.

The part of OSE that I’ve worked with, that is the parts where interrupts are disabled, have a number of properties.
We could not find any function pointers, which would have caused a big problem for calculating the WCET. If we would have found function pointers, then data flow analysis would have had to be used to try and find out what function the pointer points to.

Nested loops were only found in two of 612 chains, and then only one in each chain. This was also expected since nested loops highly contributes to the WCET of the chain.
Single loops are more common, there are loops in about 5% of the chains found, but rarely more then one loop per chain. These loops contain basic blocks with a total WCET between 10 and 60 clock cycles for each loop iteration.

One interesting property that we found was conditional subroutine calls. These are specific for ARM and cannot be found on any other processor. That made it a bit harder to calculate the WCET, but since the calculation tool had been modified for conditional subroutine calls, we had no problems.

The current tool does include pipeline analysis, which is a part of the calculation tool developed by ASTEC. The pipeline effects can be seen in the example chains in the previous chapter (Experiments) as negative contributions to the WCET.

The tool is today also able to create basic blocks graphs for each function in a compiled program. This also needs to be integrated with a data flow analysis to correctly determine all subroutine returns that are non-link-register content dependent. With that I mean return jumps with a register that is not the link register as jump operand.

9 Future Work

There are some issues that could be considered in the future work with this type of tool.

First an extended data flow analysis on the control flow graphs would be helpful when determining jump targets or the type of an interrupt change that depend on a register which content can’t be determined directly. Then all types of interrupt changes can be determined and the correct WCET for all the interrupt latencies can be calculated. This would important especially for the OSE operating system, where there are a lot of lock push and lock pop instructions when disabling and enabling interrupts.

Of course the correct loop bounds need to be set in order to get the correct WCET estimation. In the OSE operating system kernel, the ranges for the variables that determines the loop bounds are known in most cases. It should be possible to include some form of parameterisation for these variables so that when the tool detects a certain type of variable it can set a worst case loop bound automatically.

Another aspect that this tool doesn't consider is cache analysis. A full correct cache analysis included in the control flow graphs would make the WCET estimation tighter. It doesn't effect the safeness of the estimation to exclude cache analysis, we only assumes that each instruction reference will result in a miss in the cache. There has been a lot of previous work for including cache-analysis when calculating the WCET of a program [7][9][12][13][15]. When implementing an extended dataflow analysis for determining unsure register contents, the caching part should also be possible to include.

In the future development of this tool I think that the major work should be spent on integrating the tool with a more extended data flow analysis and parameterisation for loop bounds. I know there has been work done in this area before by researchers within the WCET area. If the integration is done then it would be possible to correctly calculate all the interrupt chains in the programs, because all different types of interrupt chains would be possible to detect.

Acknowledgements

I would like to thank my supervisor at Enea OSE systems, Jan Lindblad, for the help, support and suggestions on problem solving in this thesis project. Also I want to thank my supervisor at KTH, Björn Lisper for help within the project and correction suggestions on this report.

A special thanks to Jakob Engblom and Andreas Ermedahl at the Department of Computer Studies at Uppsala University for all their help and support in the every day work within the project and correction reading of this report.

10 References

[1] J Engblom et. al. "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems", *Software Tools for Technology Transfer,* February 2001.

[2] J Engblom, "Static Properties of Commercial Embedded Real-Time Programs, and Their Implication for Worst-Case Execution Time Analysis", *In Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99),* IEEE Computer Society Press, June 1999.

[3] J Engblom, A Ermedahl, "Pipeline Timing Analysis Using a Trace driven Simulator", *In Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99),* IEEE Computer Society Press, December 1999.

[4] J Engblom, A Ermedahl, "Modelling Complex Flows for Worst-Case Execution Time Analysis", *In Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00),* November 2000.

[5] F Stappert, J Engblom, A Ermedahl, "Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects", *in Proc. 22nd IEEE Real-Time Systems Symposium (RTSS'01),* 2000.

[6] A Colin and I Puat, "Worst Case Timing Analysis of the RTEMS Real-Time Operating System", *Technical Report No 1277*, IRISA, November 1999.

[7] S-S Lim et. al. , "An Accurate Worst Case Timing Analysis Technique for RISC Processors", *IEEE Transactions on Software Engineering*, 21(7): 593-604, July 1995.

[8] Y-T S Li and S Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", *In Proc. 32nd Design Automation Conference,* pages 456-461, 1995.

[9] D Whalley et. al., "Worst-Case Instruction Cache Performance", *In Proc. 15th IEEE Real-Time Systems Symposium,* pages 172-181, December 1994.

[10] C Healy et. al., "Bounding Loop Iterations for Timing Analysis", *In Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98,* June 1998.

[11] L Ko, D Whalley, M Harmon, "Supporting User-Friendly Analysis of Timing Constraints", *in Proc. ACM SIGPLAN Workshop on Language, Compilers, and Tools for Real-Time Systems,* June 1995, pages 107-115.

[12] F Stappert, P Altenbernd, "Complete Worst-Case Execution Time Analysis of Straight-line Hard Real Time Programs", *Technical Report 27-94,* C-LAB, Paderborn December 1997.

[13] C Ferdinand et. al., "Run-Time Guarantees for Real-Time Systems – The USES Approach", Universität des Saarlandes, Saarbrücken.

[14] J Engblom, "Modeling and Analysis of Pipeline Timing Behavior for WCET Analysis", Draft of *PhD Thesis, Department of Computer Systems Uppsala University,* Uppsala, June 2001.

[15] H Theiling and C Ferdinand, "Combining Abstract Interpretation and ILP for Microarchitecture Modeling and Program Path Analysis", *In Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98),* December 1998.

[16] H Theiling, "Extracting Safe and Precise Control Flow from Binaries", *Technical Report within the USES group,* Saarbrücken, Germany.

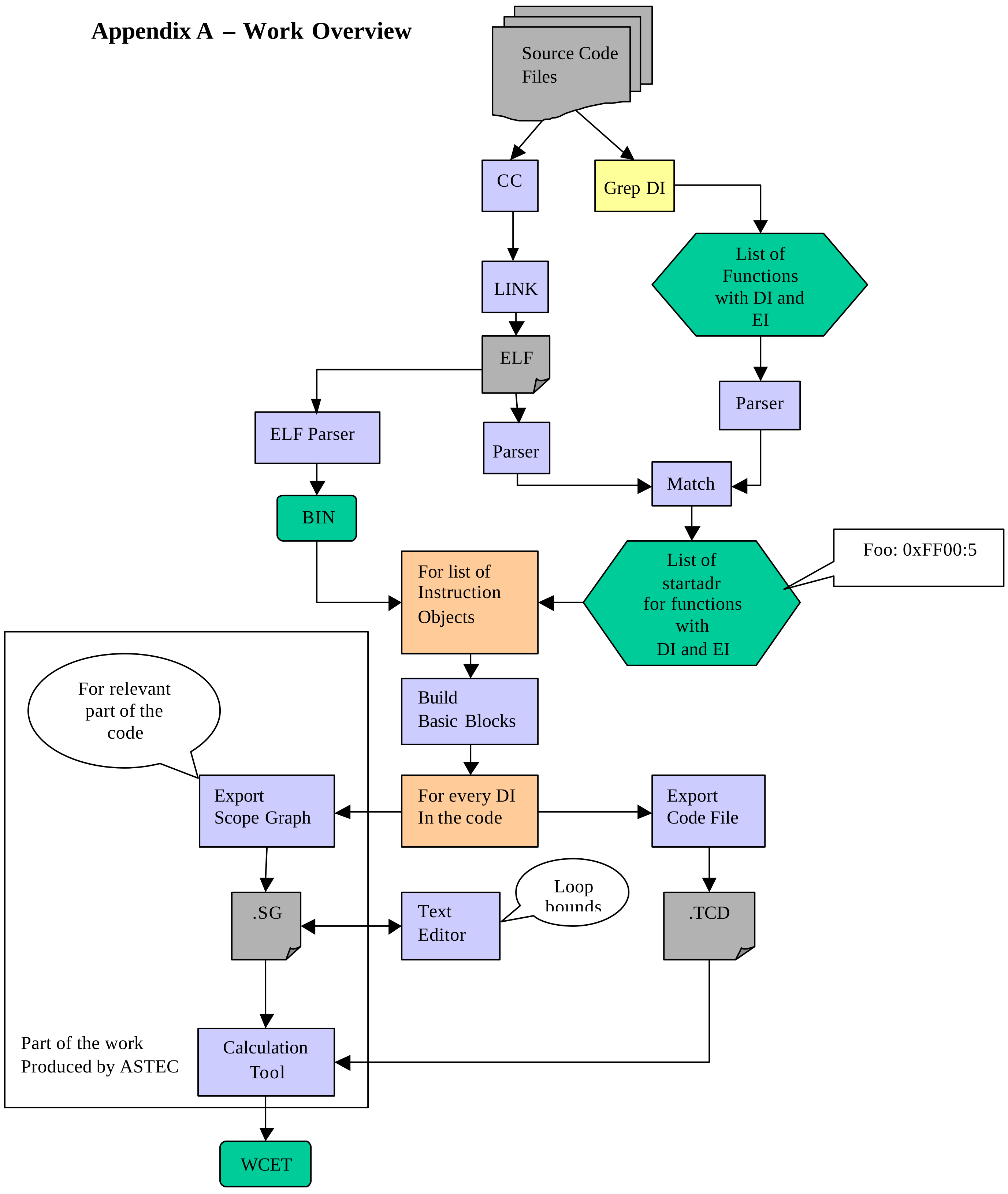
[17] D Bucar, "Reducing Interrupt Latency using the Cache", *Master's thesis in Electrical Engineering,* Stockholm January 2001.

[18] Tannenbaum, "Modern Operating Systems", chapters 2, 6, 9-12.

[19] Enea OSE Systems, "OSE 4.3 Documentation, Volume 1 – Kernel", Copyright © 2000.

[20] Advanced RISC Machines Ltd. "ARM7TDMI Data Sheet", ARM DDI 0029E.

Appendix A – Work Overview



Appendix B Scope Graph File for chain DI156588-156828

```
scopegraph

 scope f_main :
   maxiter 1 ;
   header f_main ;
   facts
   subordinates
     loop_BB9488_a ;
     loop_BB9492_a ;
   basicblocks
     f_main , [] ;
     BB9486_a , [] ;
     BB9487_a , [] ;
     BB9490_a , [] ;
     BB9485_a , [] ;
     ei_BB9472_a , [] ;
   internaledges
     f_main -> BB9486_a ;
     f_main -> BB9485_a ;
     BB9486_a -> BB9487_a ;
     BB9486_a -> ei_BB9472_a ;
     BB9487_a -> ei_BB9472_a ;
     BB9490_a -> ei_BB9472_a ;
     BB9485_a -> ei_BB9472_a ;
   exitedges
     BB9487_a -> ( loop_BB9488_a , BB9488_a ) ;
     BB9485_a -> ( loop_BB9492_a , BB9492_a ) ;
     ei_BB9472_a -> exit ;
 end scope

 scope loop_BB9488_a :
   maxiter 1 ;
   header BB9488_a ;
   facts
   subordinates
   basicblocks
     BB9489_a , [] ;
     BB9488_a , [] ;
   internaledges
     BB9489_a -> BB9488_a ;
     BB9488_a -> BB9489_a ;
   exitedges
     BB9489_a -> ( f_main , BB9490_a ) ;
     BB9488_a -> ( f_main , ei_BB9472_a ) ;
 end scope

 scope loop_BB9492_a :
   maxiter 1 ;
   header BB9492_a ;
   facts
   subordinates
   basicblocks
     BB9494_a , [] ;
     BB9492_a , [] ;
```

```
      BB9493_a , [] ;
    internaledges
      BB9494_a -> BB9493_a ;
      BB9492_a -> BB9494_a ;
      BB9492_a -> BB9493_a ;
      BB9493_a -> BB9492_a ;
    exitedges
      BB9493_a -> ( f_main , ei_BB9472_a ) ;
  end scope

end scopegraph
```