MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering (IDT)

Evaluation of a method for identifying timing models

MASTERS IN SOFTWARE ENGINEERING
30 CREDITS, ADVANCED LEVEL

Author:
Lidia Tesfazghi Kahsu

14th June 2012

Supervisor/Examiner:
Björn Lisper
bjorn.lisper@mdh.se

ABSTRACT

In today's world, embedded systems which have very large and highly configurable software systems, consisting of hundreds of tasks with huge lines of code and mostly with real-time constraints, has replaced the traditional systems. Generally in real-time systems, the WCET of a program is a crucial component, which is the longest execution time of a specified task. WCET is determined by WCET analysis techniques and the values produced should be tight and safe to ensure the proper timing behavior of a real-time system. Static WCET is one of the techniques to compute the upper bounds of the execution time of programs, without actually executing the programs but relying on mathematical models of the software and the hardware involved.

Mathematical models can be used to generate timing estimations on source code level when the hardware is not yet fully accessible or the code is not yet ready to compile. In this thesis, the methods used to build timing models developed by WCET group in MDH have been assessed by evaluating the accuracy of the resulting timing models for a number of combinations of hardware architecture. Furthermore, the timing model identification is extended for various hardware platforms, like advanced architecture with cache and pipeline and also included floating-point instructions by selecting benchmarks that uses floating-points as well.

Keywords: Real-time systems, WCET analysis, simulation, Early timing analysis, SimpleScalar, SWEET, Linear timing models

ACRONYMS

(BCET): Best-Case Execution Time

(HW): Hardware

(IDT): School of Innovation, Design and Engineering

(LOC): Lines Of source Code

(LSQ): Least SQuares method

(MDH): Mälardalen Högskola

(NCNP): No-Cache No-Pipeline

(NCSP): No-Cache Simple-Pipeline

(SA): Simulated Annealing

(SW): Software

(SWEET): SWEdish Execution Tool

(WCET): Worst-Case Execution Time

ACKNOWLEDGEMENT

This thesis is part of my Master degree in Software Engineering program and it is done at the school of IDT at Mälardalen University. First of all I want to thank the Almighty God for giving me the strength and power to do all this work (James 1:17). I would like to express my sincere gratitude to my supervisors Björn Lisper and Peter Altenbernd for your continuous support and flexibility throughout the whole process of this thesis.

Andreas Ermedahl and Jan Gustafsson for giving me timely help in time of need and trying to solve my problems despite your busy schedule. I would like to thank Andreas Gustavsson for being available whenever I knock at your door and Linus Källberg for replying to my mails and forwarding them to the responsible experts. Tim Bienias from Darmstadt University of Applied Sciences, Germany who helped me in supplying configuration of SimpleScalar and continuously strived to solve problem related to it. I want also to express my heartfelt gratitude to Tina Lempeä from KLOK, who has edited my report thoroughly word by word and gave it the final shape in which it is now.

My heartiest thanks go to my parents who provided me with proper education and made me dream of a bigger future despite discouraging circumstances. My family and friends in US, Sweden and Eritrea; I thank you for your contribution in my life.

Västerås, June 2012

Lidia Kahsu

AIM and CONTRIBUTION

The Programming Languages research group at MDH has developed a method to identify timing models for source code. Such a model is valid for a certain combination of compiler and target hardware. The method uses a test suite of programs that are compiled and run on the target hardware. The execution time is measured for the different runs, and a linear cost model for the source code is then fitted as to minimize the deviation of execution times predicted by the model from the real execution times. The cost model can then be used to predict the execution times for programs that are not yet compiled. In particular, the model can be used to make rough estimation of the Best- and Worst-Case Execution Times (BCET/WCET) of programs. The model fitting can be done in different ways, including linear regression (the "Least-Squares" method), and Simulated Annealing.

The purpose of this thesis is to evaluate this method to build timing models, by evaluating the accuracy of the resulting timing models for a number of combinations of hardware architecture. The models were built using predefined suites of test programs, using both the Least-Squares method and some variations of Simulated Annealing. The timing models were used in the WCET analysis tool SWEET to make an estimation of the BCET and WCET for a number of benchmark C programs, assuming predefined ranges of possible input values. The accuracy of the resulting BCET/WCET estimates, and thus of the timing models, were assessed by actually running the compiled benchmark problems on the SimpleScalar simulator for all possible combinations of input values in the prescribed ranges.

The work of this thesis has started by evaluating some of the results achieved in RNTS2011 paper [2] for standard hardware architecture. After the evaluation has conformed to the result of the paper then the evaluation of resulting timing models has been extended for systems with NCNP, NCSP and advanced architecture using a number of integer operation benchmarks. Finally, an evaluation scheme is purposed using floating-point benchmarks with the NCNP, NCSP and advanced architecture using a number of integer operation benchmarks.

Contents

INDEX OF TABLES

INDEX OF FIGURES

1. INTRODUCTION

1.1 Real-time systems

A system is called real-time system if:

i) The correctness is not only dependent on the logical order of events but also on their timing.

ii) It reacts upon outside events and performs function based on those and gives response within a certain time.

Real-time systems are classified into two by their consequence of missing deadlines: - Hard Real-time and Soft Real-time. *Hard real-time* systems have sharp and specified timing constraints from any system they control; otherwise failure to meet these timing constraints can have catastrophic consequences. For example, if a real-time system in an automobile fails to inflate an airbag rapidly during a collision, occupants can become severely injured due to striking interior objects like windows or the steering wheel. In order to avoid such hazardous outcome, the designer of a system has to be able to predict the peak-load performance and ensure that the system does not miss the predefined deadlines. *Soft real-time* systems are real-time systems where if the predefined deadlines are missed, the system quality degrades. For example, software that maintains and updates the trip plans for trains must be kept reasonably current but can operate to a latency of seconds.

1.2 WCET Analysis

In order to provide a safe operation of real-time systems WCET estimation is done for real-time tasks as shown in Figure 1. Worst-Case Execution Time (WCET), the upper bounds of a system or longest execution time of a program, is a very important aspect when verifying real-time properties. The input data space of a program, the logic of the program code and the timing properties of the target hardware determine in bounding the WCET. A reliable worst-case execution time can be generated if worst-case input for the task is known.

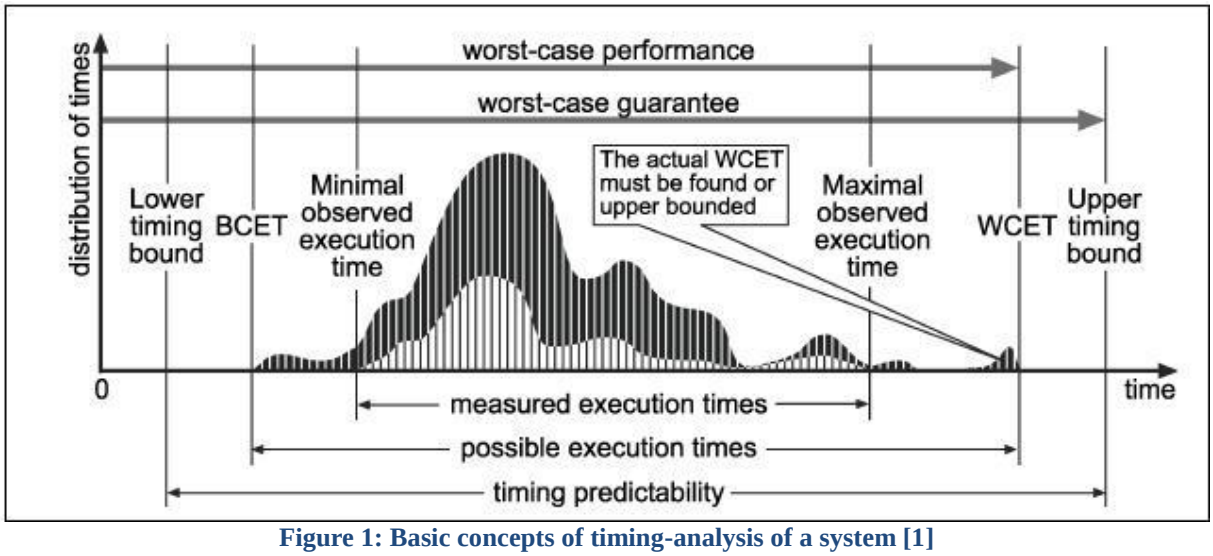

Figure 1: Basic concepts of timing-analysis of a system [1]

Timing analysis is the process of deriving execution-time bounds or estimates and tools that produce them are called *timing-analysis tools*. Timing analysis attempts to determine the

bounds of the execution time of a task when executed in a particular hardware. The time needed for a particular execution mainly depends on the path taken by the control flow and the time spent in the statement on this path or hardware. The determination of execution-time bounds has to consider the potential control-flow paths and the execution times for this set of paths. When the modular approach is used to solve timing-analysis problems, it may be divided into sub-tasks in which some deal with properties of control flow and others with the execution time of instructions or sequence of instructions of the given hardware. The methods to find the upper bound are divided into three classes:

1.2.1 Static timing analysis

Static timing-analysis is a method which attempts to analyze the code to obtain upper bounds having the set of possible control-flow paths in combination with abstract models of the hardware architecture without executing the code. Static methods can be achieved through value analysis, control-flow analysis and Processor-Behavior Analysis, estimation calculation and symbolic simulation.

Value analysis – is able to determine effective memory addresses of data which enables it to determine memory usage control. This is implemented in various tools like aiT and SWEET [1, 34].

Control-flow analysis – is used to collect the finite possible execution paths of a task taking task representation as input data. It can analyze source codes, intermediate codes and machine codes. Control-flow analysis is easier on a source-code level as the control-flow structure is not change by code optimization and linking as it is machine codes.

Processor-behavior analysis (a.k.a hardware-subsystem behavior analysis) – is finding precise execution-time bounds for a given task using linked executable based on an abstract model of the processor, the memory subsystem, the buses and the peripherals.

Estimation Calculation (a.k.a *bound calculation*) – finds the upper bound of all execution times of the whole task based on the flow and timing information derived, using control-flow analysis.

Static WCET analysis finds an upper bound to the WCET of a program using mathematical models of the hardware and software involved without actually executing the program. Mostly it is performed on some version of source code and in other cases in some form of binary code. In general, safe and tight results are expected from static WCET analysis methods. In order to control the consideration of infeasible execution paths, several path descriptions and analysis methods have been developed. The level of automation and the tightness of the results determine the usability of the static WCET analysis methods. MDH WCET researchers have been working to identify the best mathematical model for the past two decades. A model is evaluated to be correct if the analysis made derives a timing estimate which is greater or equal to the measured WCET [1].

1.2.2 Dynamic timing analysis

Dynamic timing analysis, also known as Measurement-based methods – executes the code in a given hardware architecture or simulators for set of inputs and each test run measured

execution time is given accordingly. The maximal and minimal observed execution is derived from the measured time. Generally it is difficult to explore all possible executions to derive the exact worst and best-case execution times. In most industries, the commonly used method to estimate execution time bounds is to measure the end-to-end execution time of the task for a subset of the possible executions. This timing analysis approach does not guarantee to give the exact WCET as each measurement exercises only one path. In the worst scenario, the set of inputs may not include the worst case path which leads to underestimation of WCET or overestimation of BCET.

Some of the important aspects to consider when using the measurement-based method are:-

i. code to be analyzed needs to be compiled and linked to binary form
ii. input data set, to which all possible paths must be provided
iii. a hardware configuration (could be simulator) should be set up to allow correct measurement

Even measurement-based approaches have tried to make more detailed measurements of the execution time to give better estimates of BCET and WCET but still it does not fully guarantee to give bounds of execution time since it uses abstraction of the task to make timing analysis of the task feasible. But abstraction loses information which leads to overestimation of the exact WCET and underestimation of BCET. The main crucial criterion to evaluate a method for timing analysis is safety and precision. Safety – does it produce bounds or estimates? And precision – are the bounds or estimates close to the exact values?

1.2.3 Hybrid timing analysis

This technique comprises of both dynamic and static timing-analysis. Hybrid timing-analysis tools use static analysis to deduce the final WCET estimate of a program without having to explore all paths, whereas measurement is used to extract timing estimates for small parts (basic block) of the program to be analyzed. It requires

- An analyzed program to be compiled and linked to executable binary,
- input data set which covers possible program paths and
- Hardware (or simulator) available in a setup to allow correct measurement.

1.3 Structure of Thesis

Chapter 2 introduces the background that is necessary for understanding the problem as well as for building solution. This chapter explores some basic concepts including SimpleScalar, SWEET, ALF, WCET benchmarks and some important mathematical equations for deriving the timing model analysis. Chapter 3 gives a detailed explanation of how the timing models are identified. Chapter 4 presents the problem formulation of this thesis followed by Chapter 5 discussing the result achieved. Chapter 6 presents related work. Chapter 7 is Future work and finally Chapter 8 Summary and Conclusion.

2. BACKGROUND

Why early-timing analysis is important?

As the demand of real-time embedded systems is growing in the market, one of the key aspects to consider in achieving the desired product is identifying suitable processor configuration (like CPU, memory and peripherals...etc). Usually the hardware and software parts of an embedded system are developed in parallel, which is quite often a potential problem for choosing an inappropriate hardware configuration. In order to avoid such costly change of hardware, configuration a WCET analysis of a system is inevitable.

However most existing WCET analysis methods of a system are carried out after the source-code is compiled and linked to an executable binary code; or the actual hardware configuration or input data is identified. This may rise a problem of redesigning the system if the timing properties are not met. This issue leads to a new perspective of early WCET estimation which enhances the possibility of selecting the right system configuration. An early WCET estimate is a very crucial aspect in the early stage of real-time embedded systems development for many different reasons. Most of these systems are comprised of a large variety of software engineering tools, like schedulability analysis or component frameworks and modeling etc. Moreover, these tools are used to for example, what hardware to use on the different nodes, what priorities to assign to tasks, etc. These tools need to have some type of execution time bounds in order to validate and verify early real-time properties of the system.

This thesis is carried out to evaluate a method for identifying timing models which estimates the WCET or timing-analysis of a system from a source-code. Mostly, early-timing analyses are done when the code is not ready to be compiled and linked to binary or the hardware is not accessible. Also in this thesis, SWEET [24] is used to predict the execution time of a program and SimpleScalar [23] is used to generate the corresponding measured execution time of a program [23, 24].

In this chapter, the background that is necessary for understanding the problem as well as for building solution is introduced. It explores some basic concepts regarding this thesis work. This chapter is organized as follows: Section 2.1 discusses SimpleScalar; Section 2.2 describes SWEET. Section 2.3 explores ALF; followed by Section 2.4 containing WCET benchmarks and finally Section 2.5 describes some mathematical equations which are useful to formulate the linear timing analysis models.

2.1 SimpleScalar

SimpleScalar is a tool widely used in research areas which is an instruction to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. Its development was started in 1994 by Todd Austin during his Ph.D. dissertation at University of Wisconsin in Madison, but today it is developed and supported by SimpleScalar LLC and distributed through SimpleSacalar's website at http://www.simplescalar.com. The first version was released in July 1996 and it is in a continuous process of producing new versions. It is a modeling applications that simulate real programs running on any range of processor architectures, and systems can be built using

SimpleScalar tools. The SimpleScalar tool includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution and state-of-the-art branch prediction [9]. There exists a version of `gcc` that compiles C code to the SimpleScalar instruction set. This version of `gcc` allows using a number of different optimization levels.

SimpleScalar simulators can emulate the Alpha, PISA, ARM and x86 instruction sets. In this thesis, Version 2.0 has been used. It builds on most 32-bit and 64-bit with UNIX and NT-based operating systems. Most SimpleScalar users, including this thesis, use SimpleScalar on Linux/x86-64 processor. SimpleSaclar is freely available for academic and non-commercial purposes and can be downloaded from SimpleScalar site [3].

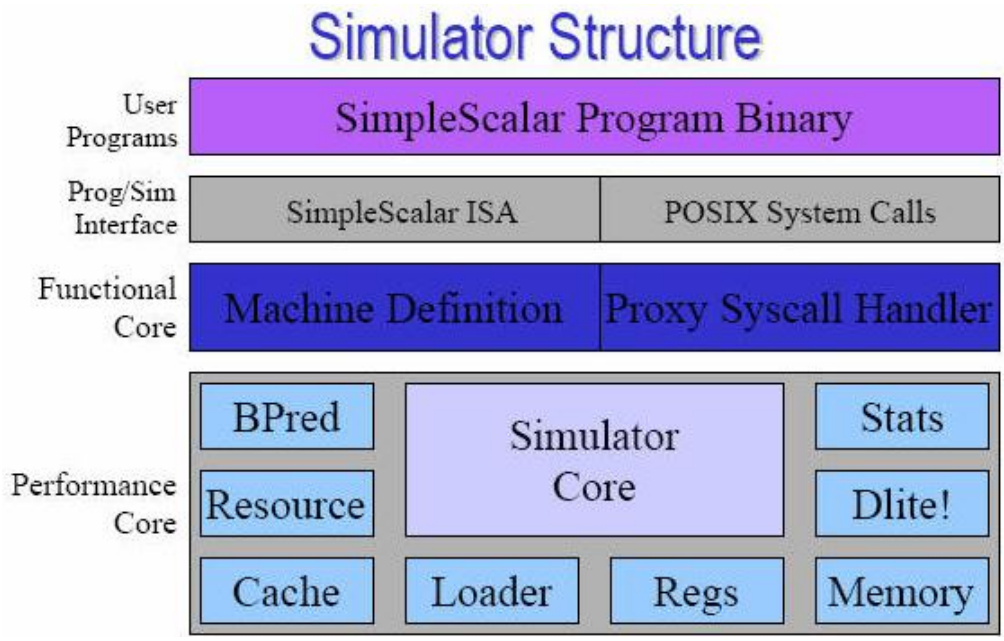

Figure 2: SimpleScalar Architecture [9]

The tool-sets that are available in SimpleScalar, which consists of a collection of microarchitecture simulators; which emulates the microprocessor at a different level of details, are:-

Sim-fast: fast instruction interpreter, optimized for speed. It does not take into account the behavior of pipelines, caches or any other part of the microarchitecture. Using the in-order execution of instruction, it performs only functional simulation.

Sim-safe: checks for memory alignment and memory access permission on all memory operations. It can also be used when the simulated program causes sim-fast to crash without explanation.

Sim-profile: is an instruction interpreter and profiler and keeps track of and reports dynamic instruction counts, instruction class counts, usage of address modes, and profiles of text and data segments.

Sim-cache: is a system simulator and can emulate a system with multiple levels of instructions and data caches, each of which can be configured for different sizes and organizations. When the cache performance on execution time is not important, this simulator

is ideal to simulate fast cache simulation.

Sim-bpred: is a branch predictor simulator. This tool can simulate various branch prediction schemes and report results such as prediction hit-and-miss rates. The effect of branch prediction on execution time is not simulated accurately.

Sim-outorder: is a detailed micro-architectural simulator. This tool models the details and out-of-order microprocessor with all of the bells and whistles including branch prediction, caches and external memory. It can emulate machines of varying numbers of executions units because it is highly parameterized.

In this thesis, sim-outorder is used to generate the estimated WCET and BCET of a training program in various architectures (i.e. no cache no pipeline, no cache simple pipeline, advanced and standard). It has used four different settings to represent standard architecture (standard), no cache no pipeline architecture (NCNP), no cache simple pipeline architecture (NCSP) and advanced architecture (advanced) as configuration executable files. The `<file.exe>` given in every command is a file produced after compiling c file using `sslittle-na-sstrix-gcc` which is a SimpleScalar compiler.

Architecture	Simulator Command
Standard	`sim-outorder <file.exe>`
NCNP	`sim-outorder -config outorder_no-cache_no-pipeline.config <file.exe>`
NCSP	`sim-outorder -config outorder_no-cache_simple-pipeline.config <file.exe>`
Advanced	`sim-outorder -config outorder_advanced.config <file.exe>`

Table 1: Hardware architecture vs. simulator command

2.2 SWEET

SWEET (**SWE**dish **E**xecution **T**ool) shown in Figure 3, is a tool which is developed at Mälardalen University, C-Lab in Paderborn, and Uppsala University [1, 11].

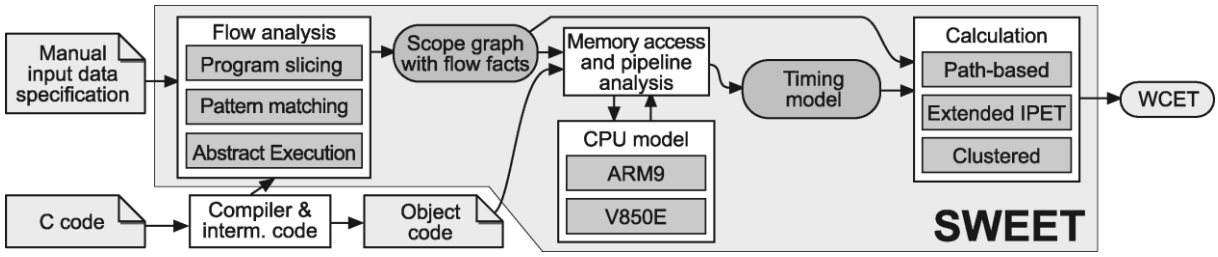

Figure 3: Architecture of the SWEET timing-analysis tool [1]

SWEET is developed in a modular fashion in order to allow different analysis and tool parts to work quite independently. It is designed to conform to the scheme of WCET analysis consisting of flow analysis, a processor-behavior and an estimate calculation. SWEET has some functionality that offers as a timing analysis tool:

1. Automatic flow analysis at the intermediate code level
2. Integration of flow analysis and a research compiler
3. The connection between flow analysis and processor-behavior analysis
4. Instruction cache analysis for level-one caches
5. Pipeline analysis for medium-complexity RISC processors
6. A variety of methods to determine upper bounds based on the results of the flow and pipeline analysis

In SWEET, the flow analysis is integrated with a research compiler in previous version. But the current version of SWEET instead uses the ALF language for its analysis, rather than the internal format of the compiler. The use of ALF makes SWEET compiler-independent; if there is a translator into ALF, then SWEET can analyze the code. The flow analysis is performed on the intermediate code (IC) of the compiler, after the structural optimizations is done. To find timing effects across sequences of two or more blocks in a code consecutive simulation runs is done starting with the same basic block of the code. The analysis in SWEET assumes that there is a known upper bound on the length of the block sequences that can exhibit timing effects [1].

SWEET uses the language ALF as input for its flow analysis. ALF is a language mainly intended to be used in conjunction with WCET analysis [5]. ALF will be discussed in detail in the next section.

2.3 ALF (ARTIST2 Language for WCET Flow Analysis)

ALF is a language used for flow analysis for WCET calculation. It is an intermediate language which was mainly developed for flow analysis instead of code generation. It is also designed to represent a code on source-level, intermediate-level and binary-level through direct translation. It maintains the information in the original code while performing a precise flow analysis. ALF is a sequential imperative language which has a fully textual representation which makes it seems as an ordinary programming language though it is generated using a tool rather than written by hand [4].

2.3.1 Syntax

The syntax of ALF is similar to the LISP programming language which makes it easy to parse and read. ALF uses the same prefix notation as in LISP, but with curly brackets “{”, “}” as parentheses as in Erlang. An example is

```
{dec_unsigned 64 0}
```

which denotes the unsigned 64-bit constant 0 [4].

ALF is an imperative language with standard semantics based on state transitions. The state consists of the contents in data memory, a program counter (PC) holding the label of the current statement to be executed, and some representation of the stacked environments for function calls [4].

Example of ALF program

The following C code:

```
if (x>y) z = 42;
```

Equivalently it can be translated into ALF as follows:

```
{ switch { s_le 32 { load 32 { addr 32 { fref 32 x } { dec_unsigned
32 0 } } }
{ load 32 { addr 32 { fref 32 y } { dec_unsigned 32 0 } } } }
{ target { dec_unsigned 1 1 }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } } } }
{ store { addr 32 { fref 32 z } { dec_unsigned 32 0 } }
with { dec_signed 32 42 } }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } }
```

The if statement of the C program is translated to a switch statement, jumping to the exit label if the (negated) test becomes true (return one). The test uses the s_le operator (signed less-than or equal), taking 32 bit arguments and returning a single bit (unsigned, size one). Each variable is represented by size 32-bit frame.
ALF AST is an abstract Syntax tree, is a tree representation of the abstract syntactic structure of source code after translated to ALF statement, built by parsers and additional information is added to the AST by semantic analysis. The syntax is known as abstract as it does not give all details as in a real syntax.

2.3.2 ALF and SWEET

ALF is used as input to the WCET analysis tool SWEET. The figure shown below describes the uses of ALF in conjunction with SWEET.

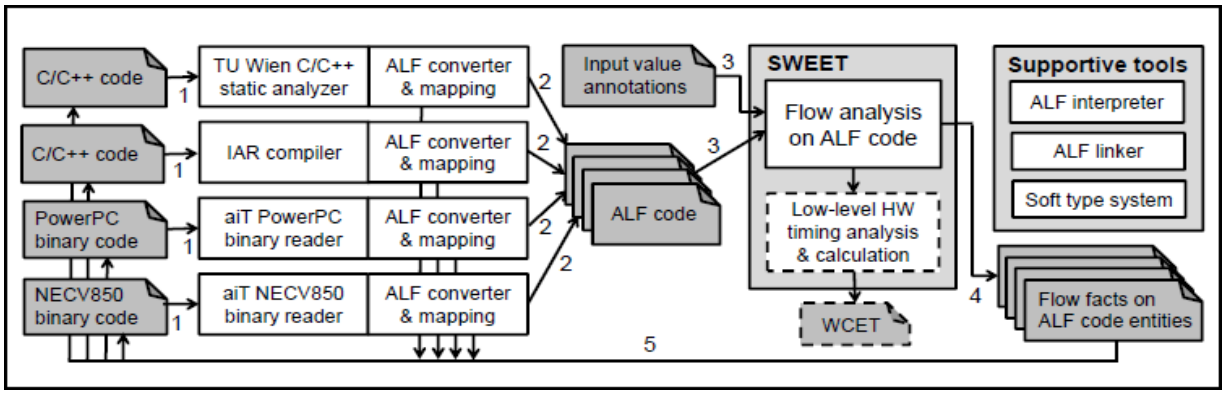

Figure 4: The use of ALF with the SWEET tool [4]

First the input program code is read, which is represented in different formats and levels. Then the output is given in the generic ALF format. Then the ALF code is used as an input to the flow analysis in SWEET which outputs the results as flow constraints on the ALF code entities. Using the mapping information created earlier, these flow facts can be mapped back to the input formats [4].

2.3.3 C to ALF Conversion

SWEET cannot perform its analysis directly on C source or executable files. In order to achieve its analysis SWEET uses the ALF format. As ALF is the only input to SWEET, there are a number of translators to ALF being developed. All three types of sources (i.e. source code, intermediate code and binary code) are covered in those translators. The C to ALF translator which has been used in this thesis can be found on the MELMAC website [33]. MELMAC is a tool that translates C source code to the ALF format. A shell script[1] has been used in this thesis to make easy the conversion of C to ALF when it is run in a Linux environment (i.e. OPENSUSE). The shell script takes a C source file and generates the corresponding converted ALF file as output [4, 6].

2.4 WCET Benchmarks

In recent years a number of WCET analysis tools have emerged: both fully-fledged commercial tools, and research tools. In order to compare these tools the associated methods and algorithms, require common set of benchmarks. The crucial evaluation metric is accuracy of the WCET estimates but there are other evaluation metrics which are equally important such as performance (i.e. scalability of the approach) and general applicability (i.e. ability to handle all code constructs found in real-time). In order to enable comparative evaluation of different algorithms, methods, and tools, it is very important to have easily available, thoroughly tested, and well documented common sets of benchmarks [7].

2.4.1 Mälardalen WCET Benchmarks

The Mälardalen benchmarks are small (all except two are less than 900 LOC) and assembled with the same goal as mentioned above in mind. All these benchmarks are written in C and were collected in 2005 from several researchers within the WCET field. The benchmarks contain a broad set of program constructs to support testing and evaluation of WCET tools. The complete set of Mälardalen benchmarks can be found in Mälardalen WCET Benchmarks web page [8]. The main categories of benchmarks are well-structured code, unstructured code, Array and matrix calculations, Nested loops, input dependent loops, inner loops depending on outer loops, switch cases, nested if-statements and floating-point calculations, bit manipulation, recursive code and automatically generated code. Some Mälardalen benchmarks used in this thesis are given below in Table 2.

Program	Description	Comments
`bs`	Binary search for the array of 15 integer elements.	Completely structured
`cover`	Program for testing many paths.	A loop containing many switch cases.
`edn`	Finite Impulse Response (FIR) filter calculations	A lot of vector multiplications and array handling
`fdct`	Fast Discrete Cosine Transform	A lot of calculations based on integer array elements
`fibcall`	Iterative Fibonacci, used to calculate fib(30)	Parameter-dependent function, single-nested loop
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample	Inner loop with varying number of iterations, loop-iteration

[1] "c_to_alf_using_christers_machine"

		dependent decisions
janne_complex	Nested loop program	The inner loops number of iterations depends on the outer loops current iteration number
ns	Search in a multi-dimensional array	Return from the middle of a loop nest, deep loop nesting (4 levels)
nsichneu	Simulate an extended Petri net	Automatically generated code with more than 250 if-statements

Table 2: Some Mälardalen benchmark programs [7]

Single-path/multi-path benchmarks and inputs to the benchmarks: If a program runs the same path regardless of the inputs then it is a single-path program while multi-path program is a program where the execution path can differ for different inputs. But in reality most programs are run with different inputs in different invocations. During analysis of WCET, it is important to know the possible values of the input variables. In general, in order to obtain tight program flow constraints from the flow analysis, the input value needs to be constrained as much as possible. The possible input variables for an embedded program or task (possibly written in C) can be:

- Values read from the environment using primitives such as ports or memory mapped I/O,
- Parameters to **main()** or the particular function that invokes the task, and
- Data used for keeping the state of tasks between invocations or used for task communication, such as external variables, global variables or message queues.

In order to be able to test and evaluate such input dependency, multiple input values have been defined for some of the benchmarks. The inputs are provided as intervals i.e. limits to the inputs. The inputs are stored on the Mälardalen WCET Benchmarks web page [8] as "input annotations" (.ann files) in SWEET format [7].

2.5 Mathematical Equations

In this section the most important mathematical formulas, used to develop the timing model that will predict the BCET/WCET of a program, will be briefly discussed. Section 2.5.1 presents the linear equation followed by least-square methods in Section 2.5.2 and finally simulated annealing in Section 2.5.3 is discussed.

2.5.1 Linear Equation

A Linear equation is "*an algebraic equation in which each term is either a constant or the product of a constant and the first power of a single variable*" [13]. It is an equation in the form of

$$m * X + c = 0$$

where X is unknown [16]. The name "*linear*" comes from the set of solutions of such an equation which forms a straight line in the plane [13].
A general formula of m linear equations with n unknown can be written as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \cdot \quad\quad \cdot \quad\quad \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \quad\quad \cdot \quad\quad \cdot \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

and can be also written as a weight for a column vector in a linear combination:

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \cdot \\ \cdot \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \cdot \\ \cdot \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \cdot \\ \cdot \\ a_{mn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ b_m \end{bmatrix}$$

The vector equation is equivalent to a matrix equation of the form

$$Ax = b$$

where A is an $m \times n$ matrix, X is a column vector with n entries, and b is a column vector with m entries [12].

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m1} & a_{m2} & \cdot & \cdot & a_{mn} \end{bmatrix}, X = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ \cdot \\ b_m \end{bmatrix}$$

2.5.2 Least-squares method

The Least-squares method is one of the standard mathematical approaches to approximate solution of sets of equations in which there are more equations than unknowns. It is the simplest and most commonly used form of linear regression. The most significant application is in data fitting where the best fit is found when the sum of squared residuals (i.e. the difference between an observed value and the fitted value provided by the model) is minimized [14, 17].

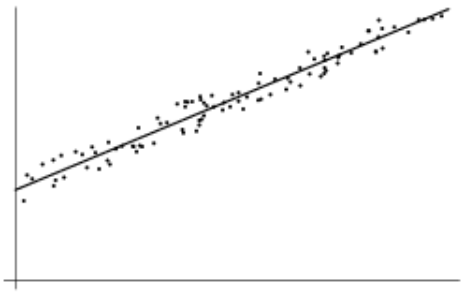

Figure 5: Data fitting using least-square [17]

The Least square method is categorized into linear and non-linear least squares, depending on whether or not the residuals are linear in all unknowns. There is a close-form solution which

can be evaluated in a finite number of standard operations for linear least-squares problem [14].

As mentioned above, the objective of the least-square method is to adjust the parameters of a model function to best fit a data set, or a technique applied in the form of linear regression through a set of points that provide a solution to the problem of finding the best fitting straight line, $y = ax + b$. A simple data set can have n points of data pairs (x_i, y_i), $i = 1,...,n$ where x_i an independent variable, while is y_i is a dependent variable whose value is found by observation. The fitting curve $f(x)$ has a deviation (error) d from each data point, i.e., $d_1 = y_1 - f(x_1)$, $d_2 = y_2 - f(x_2)$, ..., $d_n = y_n - f(x_n)$. Then the best fitting curve has the property that minimizes the sum of squares as shown:

$$\prod = d_1^2 + d_2^2 + \cdots + d_n^2 = \sum_{i=1}^{n} d_i^2 = \sum_{i=1}^{n} [y_i - f(x_i))]^2 = aminimum$$

Detailed information about least-squares method can be found in [14, 17, 18 and 21].

2.5.3 Simulated Annealing

Simulated annealing (SA) is a probabilistic technique which can find an optimal solution of a cost function that may possess several local minima [15]. In SA, when the physical process is emulated, the solid part gradually cools down and reaches “frozen” stage which happens at a minimum energy configuration.
Each iteration of SA algorithm randomly generates a new point. The distance between the new point and the current point is determined by the probability distribution with a scale proportional to the temperature. The SA algorithms collects all the new points that lower the objective including certain probability that raise the objective, but SA tries to avoid a local minima in early iteration by exploring a better solution globally [20].
The probability of taking a step is determined by the Boltzmann distribution as,

$$p = e^{-(E_{i+1} - E_i)/(kT)}$$

if $E_{i+1} - E_i$, and $p = 1$ when $E_{i+1} <= E_i$

Temperature T is inversely proportional to the energy difference $E_{i+1} - E_i$. The temperature T is initially set to a high value and a random walk is carried out at that temperature. According to the cooling schedule, for example: $T -> T\mu_T$ where μ_T is slightly greater than 1 [25].
This method is an easy-to-implement, probabilistic approximation algorithm, (even though the structure of the problem might not be fully understood) which is able to produce good solutions for an optimization problem [19].

3. TIMING MODELS

In this section, the identification of linear timing model is done is going to be discussed in detail in order to grasp the idea behind the work of this thesis from the RNTS2011 paper [2]. Moreover, this thesis is the evaluation and extension of the result of this paper and some results of this thesis has been used in the RNTS2011 paper.

3.1 Identification of Linear Timing Models

The linear timing model identification is done assuming that the source language is emulated by an abstract machine which leads to tracing the virtual instruction in the source language of a program.
For each virtual instruction i_k, the trace then contains c_k occurrences of i_k (the execution count of i_k). The *linear timing model* for the abstract machine computes the execution time T for a trace as

$$T = w_0 + \sum_{k=1}^{n} w_k c_k \qquad (1)$$

w_0 is a constant startup time, and w_k, $k = 1,\ldots,n$ are constant execution times for the respective virtual instructions. If we assume that i_1 is a virtual "startup" instruction, which occurs once in each trace, then (1) can be simplified as

$$T = \sum_{k=1}^{n} w_k c_k \qquad (2)$$

The linear timing model for a code compiled with non-optimizing compilers executed on a simple hardware without features like pipelining or cache is expected to be more accurate than a code executed with more complex hardware architectures containing varying instruction execution times. Moreover, a code that has been heavily changed by using an optimizing compiler has a less accurate timing model. The goal is to find the best model which produces minimized deviation of predicted execution times from the real ones having the same number of observations. There is a measured execution time t_j for each observation j for the compiled binary and an array $(c_{j1},\ldots,c_{jn})$ of execution counts for the emulated source code executed with the same inputs as the compiled binary. If we assume that we have made m observations, the model predicts an array of $C\vec{w}$ execution times, where C is an mXn-matrix whose rows are the observed arrays of the execution count, and $\vec{w} = (w_1,\ldots,w_n)$ is an array of virtual instruction execution times. Let $\vec{t} = (t_1,\cdots,t_m)$ be the array of measured execution times for the different observations. The best model then amounts to finding a $\vec{w}$, that minimizes the overall deviation of $C\vec{w}$ from $\vec{t}$ [2].
There are various ways to define the overall deviation and the Euclidean distance is used for this purpose:

$$\|\vec{x} - \vec{y}\|_2 = \sqrt{\sum_{j=1}^{m} (x_j - y_j)^2}$$

The least-square method (LSQ) will find a $\vec{w}$ that minimizes $\|C\vec{w} - \vec{t}\|_2$. The minimization of the overall deviation can also be heuristically done using SA which is applied in the experiments.

3.2 Early Timing Analysis Approach

After training programs were designed, they were compiled and executed on target hardware or a simulator for it. As mentioned earlier, the SimpleScalar simulator is used in this thesis in order to ease the task. SimpleScalar was configured to simulate required architecture with a variety of different features, and it records the number of cycles needed to execute the program on the virtual hardware. The number of cycle values forms a vector $\vec{t}$ [2].

Following that the training programs were translated to an intermediate format ALF language using the C-2-ALF translator as mentioned in Section 2.3.3, which provides the virtual instruction set. The execution counts for the ALF instructions were produced after SWEET interprets the ALF code which forms the matrix C [2]. Then the model has been identified, i.e. the vector $\vec{w}$ was determined using the LSQ method and SA (see Section 2.5). The last step is to use the model for timing analysis which could be done either through simulation or a static timing analysis. SWEET has been used to do both simulation and approximate static WCET analysis on source level using the timing models. ALF timing models are used to extend the interpretation mode of SWEET to provide timing estimates during simulation as shown in Figure 6.

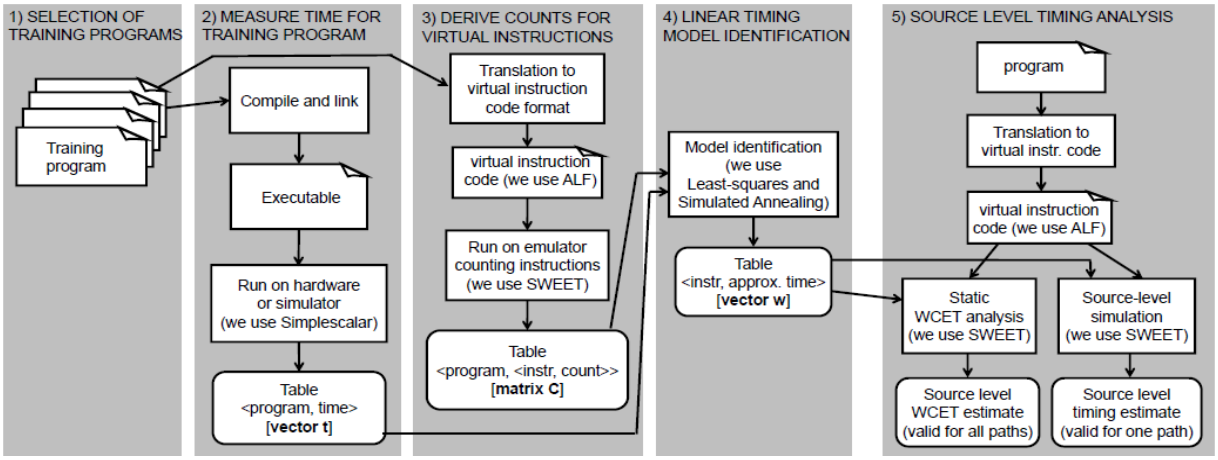

Figure 6: Early-Timing analysis approach [2]

3.3 How are training programs constructed?

One important aspect of timing model identification is selecting good and an adequate number of training programs. The training programs used are synthetic program suites which allow more control over the virtual instruction traces by avoiding problems either with linear dependency or highly correlated execution counts. In this thesis, training program suites for three scenarios are designed; simple architecture, advanced architecture and floating-points.

3.3.1 Training programs for Simple Architecture

The training program suites for simple architecture are constructed as follows

- The first program is the "empty" program. Execution yields the startup time for a program (i.e. the time for a virtual RUN_PROG statement):

```
int main() {}
```

  First this program is put in the suite, as every emulation of virtual instructions for a source program will execute RUN_PROG.

- Any nonempty C program must contain an assignment. Such a program must execute at least one virtual STORE instruction. Thus, the next program executes exactly one STORE:

```
int main() {int j=17;}
```

- Correspondingly, a third program executes along with STORE and RUN_PROG a LOAD instruction. Until the full set of instructions is executed the program suite continues with a series of simple programs executing each remaining instruction. For example, INT_MULT instruction:

```
int main() {int j=42; j=j*3}
```

The number of function calls was equal to the number of executed RETURN instructions. In order to avoid this dependency, which can occur due to RETURN instruction, it is replaced by a superinstruction carrying their added execution times which will help to yield a lower-triangular execution count matrix C with nonzero diagonal entries. There were no linear dependencies between the column vectors of such matrices. As there was exactly one program execution per instruction, C becomes a quadratic matrix. Furthermore, the absence of linear dependency causes C to be invertible and the linear system $C\vec{w} = \vec{t}$ can be solved directly to yield $\vec{w}$ such that $\|C\vec{w} - \vec{t}\|_2 = 0$.

3.3.2 Training programs for Advanced Architecture

The advanced architecture, having caches and pipeline features, will cause instructions to have highly context-dependent execution times. In order to have the model capturing the influence of the context on the execution time, the "real" instructions must be executed in a variety of contexts when identifying the timing model. Moreover, in order to capture the cache and pipelining influence, longer instruction sequences must be executed. One way of accomplishing this is to introduce loops in the code. The advanced architecture test suite is built up on simple architectures and extends the programs with loops executing the instruction under test a number of times. This extension gives reasonably good results even though it does not capture more complex timing effects involving several different instructions [2].

Some instructions were invariably needed during the loop introduction in the code; a *STORE* to initialize the loop variable, some arithmetic operation to increment/decrement

the loop counter, some test instruction to decide the exit condition, and a *JUMP* to return to the entry point of the loop. This resulted in a matrix without lower triangular execution count. But the execution counts made linearly independent by introducing several loops executing different arithmetic operations to increment/decrement the loop counter, and different test instructions to break the loop. In order to break the possible linear dependencies to instructions (like *JUMP*) that appear in all loops, a third part of the code executes the loop body outside any loop. The linear dependencies between any instructions has been broken and also the correlation minimized if the training program has been executed a number of times, with different loop bounds set in a linearly independent fashion.

An example of a training program for INT_MULT, consisting of two independent loops and a section with straight-line code:

```
int main () {
        int max1 = ...;
        int max2 = ...;
        int i,j;
        for (i=1; i<=max1; i++) {
            j = I; j = j*3;
        }
        for (i=max2; i>0; i--)
             J = I; j = j*3;
        }
        J = i;  j = j *3;
        }
```

INT_MULT would always been executed as many times as the ADD that increments i, and the test operation that compares i with max1 if only the first loop was present. As a result, linear dependency would have been created between these execution counts. A different test operation is created using a second loop which uses SUB to decrement the loop counter.

Using max1 and max2 a number of executions could be made in such a way that the resulting execution counts for the involved virtual instructions are linearly independent. But both loops had a single JUMP each and INT_MULT will still be the same no matter what max1 and max2 are and in order to break linear dependency, the third appearance of the loop body has been added. This is automated to generate training programs automatically and the constant max1 and max2 can be varied [2].

3.4 Model Identification

There have been different approaches tried to the problem of choosing $\vec{w}$ such that predicts the execution time well for the compiled binary, running on the chosen target platform. Even though LSQ (see Section 3.1) gave best fit for the set of programs by selecting real-valued weights to minimize distance, it can yield models that did not predict the execution time well for programs outside the training set. For example, it could yield negative cycle counts for instructions which are unrealistic. Moreover, it may yield very poor predictions for programs that execute instructions frequently as compared to other

instructions.

They have used also a more general search method that allows more freedom in specifying constraints and objective function. The SA approach has been used, in which each step of the SA algorithm replaces the current solution by a random solution from the neighborhood. If the result of SA is better, it is accepted or else it may still be accepted with a probability that depends both on the difference between the subsequent objective function values, and on a global parameter T (the "temperature"). As the temperature is decreased during the process, jumps leaving the local solution space will become less and less likely, so eventually the result will stabilize.

Adapting SA to minimize $\|C\vec{w} - \vec{t}\|_2$ is easy, and it is done according to the following:

- All elements in $\vec{w}$ are initialized to zero
- For producing a solution in the neighborhood of $\vec{w}$, its elements will be randomly incremented or decremented by one, or kept as is (while upholding any imposed constraints on the solution).

SA has been executed several times with varying parameters to get the best result since SA is very sensitive to its steering parameters like temperature.

3.5 Experiment done by Mälardalen WCET group

The MDH WCET group has already made an evaluation on the precision of the identified models, as well as the influence of the training program suite. They have used two sets of training programs for advanced architecture and simple architecture and they have tried both LS and SA as shown in Table 4. Using a distinct set of programs consisting of fifteen programs from the Mälardalen WCET Benchmark Suite, the models were evaluated as shown in Table 3. Table 3 gives some basic data about the programs, including lines of C code **(#LC**), the number of functions (**#F**), loops (**#L**), and conditional statements (**#C**).

Program	#LC	#F	#L	#C
bs	114	2	1	3
cover	640	4	3	6
edn	285	9	12	12
Esab_mod	3064	11	1	292
fdct	239	2	2	2
fibcall	72	2	1	2
fir	276	2	2	4
inssort10	92	1	2	2
inssort15	92	1	2	2
inssort20	92	1	2	2
inssort30	92	1	2	2
jcomplex	64	2	2	4
loop3	76	1	150	150
ns	535	2	4	5
nsichneu	4253	1	1	253

Table 3: Benchmark Programs [2]

First a comparison was done between predicted and real running time result generated by running each benchmark with its specified input (all these benchmarks have their hard-

coded inputs). From the results, estimations are taken how well the derived timing models predict real running times. Then, by removing the hard-coded inputs for some selected benchmarks, they changed the hard-coded inputs into programs having different paths through the code for different inputs. Later possible input values were defined for each selected benchmark. Finally, a static BCET/WCET analysis for these benchmarks was performed, so as to evaluate the precision of static timing analysis based on the timing models. The real best/worst case obtained using an exhaustive search over the possible inputs and compared with the static BCET/WCET estimates [2].

SWEET has been used for both single runs and the static analysis and it has a "*single-path mode*" that can be used to emulate ALF code. SWEET is turned into a source-level simulator estimating execution times using ALF timing models in extension of the single-path mode. Moreover, SWEET's static analysis has been extended so as to perform BCET analysis on top of WCET analysis [2].

Both the training programs and benchmarks have been compiled using sslittle-na-sstrix-gcc with no optimization and for sim-outorder executes with its standard configuration in SimpleScalar. Sim-outorder simulates a processor with out-of-order issues of instructions, main memory latency 10 cycles for the first access and 2 cycles for the next accesses, memory access bus width 64 bytes, 1KB L1 instruction cache (1 cycle, LRU), no data cache, no L2 cache, no TLB's, 1 integer ALU, 1 floating point ALU, and fetch width 4 instructions. The branch prediction is 2-level with 1 entry in the L1-table, 4 entries in the L2-table and history of size of 2.

The experiment done in [2] had selected benchmark programs that did not use floating-point instructions which make ALF use 31 different ALF instructions that has formed the virtual instruction set for the experiment. These instructions included program flow control instructions, LOAD/STORE, and arithmetic/logical instructions excluding floating-point arithmetic.

3.5.1 Training Programs

The "*simple*" training program suite has been used from Section 3.3.1. The average deviation obtained for the set of benchmark programs in Table 4 is 29%. This result showed that this suite is not well suited to identify models for architecture like sim-outorder.

In order to see the influence on precision of the predicted execution, the "*advanced architecture*" suite tried by executing loops with varying number of iterations. As it has been noted, architectural features like cache and branch predictors tend to yield shorter instruction execution times within loops which seemingly influence the identified model and its resulting precision. To estimate the influence a program suite instantiation "*small*" (loop iterating 7-17 times), "*medium*" extending "*small*" with instances of the programs iterating up to 29 times, and "*big*" extending "*medium*" with instances iterating up to 61 times is used. Furthermore, to see the influence on the precision they tried adding the "*simple*" training program suite. As a result a total of six test cases with different variations of the training program suite produced [2].

3.5.2 Model Identification Method

The LSQ and SA have been tried with different variations. For LSQ, both direct solution and with instruction execution times rounded to the closest integer, has been tried. The

outcomes showed that in both cases, there was no restriction on the instruction execution times: therefore, e.g., negative execution times were possible and would indeed appear sometimes.
The Euclidean deviation $\|C\vec{w} - \vec{t}\|_2$ is used as an objective function for SA with three variations. In the first variation, SA was run without any constraints on the final solution and in the second run, with constraints that all virtual instruction execution times enforced to be non-negative. This was done to observe whether it would yield a better prediction when the constraints were added. In the third variation, the search space size was restricted for SA by enforcing an upper limit on virtual instruction execution times. This was created to shorten search times for SA; it was interesting and important to see whether the identified model precision was affected by it.

All these variations of LSQ and SA were tested with all six different training program suite combinations as shown in Table 4. The result shows the relative average deviation of predicted running times from measured running times.

Training suite	LSQ	LSQ rounded	SA	SA≥0	0≤SA ≤2×
small	39%	34%	10%	10%	10%
medium	50%	45%	14%	12%	12%
big	64%	63%	16%	13%	13%
small + simple	18%	17%	15%	10%	20%
medium + simple	19%	15%	16%	10%	10%
Big + simple	17%	14%	16%	10%	10%

LSQ: standard least squares method, LSQ rounded: LSQ rounded to closest integer
SA: Simulated Annealing with no constraints on the solution
SA≥0: SA restricted to nonnegative instruction times, 0≤SA≤2×: SA additionally restricted from above

Table 4: Average deviation of predicted vs. real execution times for benchmarks with different model identification methods [2]

As shown in the above table, SA has much better result for all examples compared to LSQ but LSQ gave improved results when a simple training suite was added. Even rounding the LSQ result did not bring a significant change, whereas SA consistently gave the best results when restricted to nonnegative values. The restriction has a significant effect for the small + simple training suite which resulted in faster convergence in SA solutions in all cases.

The result obtained in [2] shows deviation between the measured and the predicted running time for the individual benchmark programs as shown in Table 5. All the programs have a deviation from close to zero up to about 20% except for cover which has an extreme outlier with more than 50% underestimation.
A subset of benchmark codes has been used to evaluate the approximate source-level timing analysis. Since the programs are multipath, in which execution times vary with inputs and their worst- and best-case input are known. Therefore, by running SimpleScalar with proper input, the real BCET and WCET could be approximated.

The result of the analysis is shown in Table 6 of [2], along with BCET and WCET recorded for SimpleSacalar. The result obtained is reasonable BCET and WCET estimate

as compared to the precision measured by the single execution times.

In [2], the model identification method was tried for advanced architecture, in which sim-outorder is configured to simulate a processor with some characteristics like out-of-order issues instruction, main memory latency 18 cycles for first access and 2 cycles for next accesses, 8KB L1 data and instruction caches, respectively (1 cycle), 256KB L2 data and instruction cache (6 cycles), all caches LRU, no TLB, 2 integer ALU's, 2 floating-point ALU's, fetch decode, issue, and commit width all 4 instructions, perfect branch prediction. Then the model identification was rerun for this hardware configuration and the best model obtained was by SA>=0 and evaluated the precision of the timing model using a single benchmark program run. The deviation obtained was 0-30% and the average deviation is 15%.

Program	Model	Measured	Diff	Rel. diff
bs	274	317	43	13.6%
cover	3515	8388	4873	58.1%
edn	244189	232561	11628	5.0%
esab_mod	698848	699934	1086	0.2%
fdct	9250	11294	2044	18.1%
fibcall	788	901	113	12.5%
fir	6973	8468	1495	17.7%
inssort10	3674	3529	145	4.1%
inssort15	549	579	30	5.2%
inssort20	729	759	30	4.0%
inssort30	1089	1119	30	2.7%
jcomplex	671	673	2	0.3%
loop3	11999	13371	1372	10.3%
ns	31897	33718	1821	5.4%
nsichneu	19744	18545	1199	6.5%

Table 5: Predicted vs. measured times for single benchmark program runs [2]

Program	Model	Measured	Diff	Rel. diff
bs	130	184	54	29.3%
cover	1837	3605	1768	49.0%
edn	140136	119291	20845	17.5%
esab_mod	368743	408076	39333	9.6%
fdct	4998	3940	1058	26.9%
fibcall	283	377	94	24.9%
fir	3923	4035	112	2.8%
inssort10	2094	1678	416	24.8%
inssort15	284	303	19	6.3%
inssort20	379	395	16	4.0%
inssort30	569	568	1	0.2%
jcomplex	282	307	25	8.1%
loop3	5017	6290	1273	20.2%
ns	18758	18725	33	0.2%
nsichneu	10969	10129	840	8.3%

Table 6: Predicted vs. measured times for single benchmark program runs, advanced architecture [2]

4. PROBLEM DESCRIPTION, PROJECT SETUP and METHODS

The objective of this thesis is to evaluate the method used to build timing models, by evaluating the accuracy of the resulting timing models for a number of combinations of hardware architecture. The models have been built using predefined suites of test programs, using both the LSQ method and some variations of SA. To ease the task of measuring execution times, the hardware simulator SimpleScalar has been used as target hardware rather than real hardware. The thesis project started with analyzing the result that has been produced and published by the WCET group in RTNS11 paper [2].

The main hardware used in this thesis was a PC with Opensuse 11.4 version. To ease the task of measuring execution times is carried out by using SimpleScalar version 3.0e simulator. SimpleScalar can be configured to simulate a variety of processor architectures, and there exist a version of gcc that compiles C code to the SimpleScalar instruction set. This version of gcc allows using a number of different optimization levels. There have been four different configurations of SimpleScalar that simulated the NCNP architecture, NCSP architecture, Standard architecture and advanced architecture mentioned in Table 1. Moreover, alongside with SimpleScalar, SWEET has been used as one of the main software or tools in this thesis. So far, no version has been set on SWEET because it is a research prototype. SWEET was modified several times throughout the thesis by fixing bugs found in it. Some shell scripts has been used during the project setup and analysis because SWEET is a command based tool without any GUI. Firstly, the platform was set up after installing SimpleScalar and tried out with some Mälardalen benchmarks that have been used in Table 5. The next step was to install SWEET and attempt to execute some Mälardalen benchmarks in order to make sure that SWEET was installed properly. In both these startups of the project, only single-path mode was taken into account.

4.1 Virtual Instructions

The first step to identify a source-level timing model for a given combination of hardware configuration is to select a set of virtual instructions (such as arithmetic/logic operations, branching, function calls/returns etc). An abstract machine that can execute source code has been defined using the set of virtual instructions.

4.2 Analysis timing models using SimpleScalar

The following steps were involved for analyzing timing models with SimpleScalar.

i) The training program generator was an executable file and it has been executed to generate specific c-file training programs. The training program generator should be set according to the matter at hand to produce the specific training suite (for example, if floating-point benchmarks were used, we have to use the proper combination in order to identify the corresponding model).

ii) Then the training programs c files are compiled using this command:

```
sslittle-na-sstrix-gcc $file.c –o $file.exe
```

in which `$file` is replaced with respective c file and outputs an `$file.exe` file.

iii) Then it is executed using `sim-outorder` with different combinations of

SimpleScalar processor configurations and it produced the cycle counts for each exe file:

```
sim-outorder -config outorder_<processor configuration>.config
   $file.exe
```

where `<processor configuration>` is replaced by a specific SimpleScalar processor configuration of the hardware architecture and produces cycle counts for each training program suite.

4.3 Analysis timing models using SWEET

The following steps were involved for analyzing timing models with SWEET.

i) For SWEET, the first step was done to transform all the training program suite c-files to an intermediate format, ALF, which provides virtual instruction using a shell script. The `c_to_alf_using_christers_machine.sh` was a shell script that would run the C-2-ALF translator melmac on a machine where it was installed during the thesis project and written as follows:

```
c_to_alf_using_christers_machine.sh $file.c
```

where `$file.c` is replaced with training program suite c-file

ii) Then the ALF file is executed to produce the count of statements occurrences (statements are like store, call… etc.)

```
sweet -i=file.alf -ae pu tc=st
```

where `-i` option represent input-files, which `file.alf` are the particular ALF files and `-ae` is used to give abstract execution to produce flow facts. The option `pu` is used when `-ae` should be run on code containing imports (i.e. undefined).

Both steps done in 4.1 and 4.2 are written together in a shell script[2]. The shell script is executed as follows:-

```
./produce-input-data.sh
```

which gives a matrix to produce a result line for an equation file in the form of $Ax = b$, where A is the result generated in 4.3 using SWEET and b is generated in 4.2 using SimpleScalar. These results are written to `Axb.csv` file as an output of the shell script[2].

4.4 Identification of Linear model

The output produced in Section 5.3, named `Axb.csv`, is fed to the file which is an equation solver, as input, in order to identify a linear timing model with an execution time for each virtual instruction from the measured execution times and recorded instruction counts.

```
./EquationSolver.sh -i58 Axb.csv
```

[2] produce-input-data.sh

As mentioned in Section 3.4, the model identification is summarized into a shell script[3] which takes `Axb.csv` and solves equations of the form `A*x=b` using three different algorithms, namely LU decomposition, linear regression and SA. The `-i58` option is used to remove column 58 in `Axb.csv` which is a "RETURN" statement in order to avoid dependency as explained in Section 3.3.1. This will also create several CSV files in which all output files are summarized in `overview.csv` where the best vector is to the left of a corresponding CSV file with the same name as a headline of the left-most column in the directory [26].

4.5 Source Level Timing Analysis

A source level timing analysis is done based on the best vector produced in Section 4.4 using SWEET.

4.5.1 Single path timing estimates

i) An alf file is executed using SWEET

```
sweet -i=$file.alf -ae pu css vola=i tc=st,op
```

where `-i`, `-ae` and `pu` options has already been explained in Section 4.3. The `css` option is used to check if a single state is generated, i.e., it throws run-time error if more than one state is generated during the abstraction execution. `$file.alf` is replaced by a particular Mälardalen WCET benchmark. Moreover, `tc=st,op` is a type counting which counts the number of occurrences for each type during an execution. In this particular situation, it counts the occurrence of statements (i.e. store, call... etc) and operators. This produces predicted times for single benchmark programs, similar to Table 5 and 6.

ii) The same procedures as Section 4.2 is done using SimpleScalar except that `$file` is replaced by a benchmark file and produces measured times for single benchmark programs.

Both i) and ii) are summarized in a shell script[4]. The shell script is executed as follows: -

```
./produce-output-data.sh <bestvector.csv> <benchmarks>
```

where `bestvector.csv` is produced in Section 4.4. `<benchmarks>` is replaced by a directory containing all benchmarks selected to be used for single path estimates. A for-loop is iterated throughout the whole benchmarks. The output of this shell script is written to `compare.csv` file containing measured and predicted timing estimates into two different columns.

4.5.2 Multi-path timing estimates

The source level timing analysis is also evaluated using a subset of benchmark codes which are multipath programs. The BCET and WCET can been known for such programs since BCET and WCET of the inputs are known. This thesis has only worked to estimate BCET and WCET of proper benchmarks using SWEET. The measured BCET/WCET using SimpleScalar is not included since it is not a priority in this thesis.
During this thesis, some benchmarks, which has been evaluated using multi-path source level

[3] EquationSolver.sh
[4] produce-output-data.sh

timing, are identified by the WCET Mälardalen research group. These benchmarks are `bsort100`, `esab_mod`, `insertsort`, `minmax`, `ndes`, and `nsicheu`.
In order to produce estimated BCET/WCET of these benchmarks, the following command is executed:-

```
sweet -i=file.alf annot=file.ann -ae aac=file.clt tc=st,op
merge=all
```

The abstract execution is done by using an annotation using the option `annot=file.ann` which contains the input range. The option `aac=file.clt` generates BCET and WCET estimate using ALF AST (i.e. ALF abstract syntax tree) construct costs. As mentioned in Section 4.5.1, `tc=st,op` is type counting the occurrence of statements (i.e. store, call... etc) and operators.

4.6 Floating-Point Instruction

Single path runs with floating-point operation benchmarks have been extended to untried task which has never been done before, either as a thesis work or by the WCET research group. It is accomplished by identifying a set of considered benchmarks with floating-point instructions. The first step was to check if SimpleScalar can compile floating point instructions. In order to achieve this, simple c-file with a floating-point number is created and compiled using the SimpleScalar compiler as follows:-

```
sslittle-na-sstrix-gcc float.c
```

where `float.c` is a simple code which prints a floating number. After SimpleScalar successfully compiled the c-file, 15 benchmarks that use floating-point instructions were identified from the set of considered Mälardalen benchmarks. These benchmarks are `bsort100`, `cnt`, `expint`, `lcdnum`, `ludcmp`, `matmult`, `minver`, `mm`, `qsort-exam`, `qurt`, `select`, `sqrt`, `st`, `ud`, and `whet`.
Then the training programs generator code is assembled in a way that enables it to produce training programs that contain floats. The next step was to follow the steps starting from Sections 4.2 to 4.5, which are analyses of the timing models using the SWEET and SimpleScalar. During the process of analysis of timing models using SWEET, `sqrt`, `st` and `whet` benchmarks could not produce result, so they are removed from further procedures.
The only difference from Section 4.5.1 is that SWEET has a new way of handling floating-point values and new flags have been implemented. The command used to execute the floating point is:-

```
sweet -i=$file.alf -do floats=top -ae pu tc=st,op
```

where `-do` is used to configure the domain (i.e. integer or float) to be used by the abstract execution. The `floats=top` is the default setting is used all calculations of floating-point values result in TOP (safe). The other flags used have already been mentioned in Section 4.5.1.

5. RESULTS and DISCUSSIONS

The purpose of this thesis is to evaluate the method used for identifying timing models. The result achieved after the experiment done in Section 4 for identifying timing models is presented in this section for different hardware architectures (standard architecture, NCNP architecture, NCSP architecture and advanced architecture) using integer and floating-point benchmarks.

`sim-outorder` is configured to simulated a processor (i.e. standard architecture) with the following characteristics: out-of-order issues of instructions, main memory latency 10 cycles for the first access and 2 cycles for the next accesses, memory access bus width 64 bytes, 1KB L1 instruction cache (1 cycle, LRU), no data cache, no L2 cache, no TLB's, 1 integer ALU, 1 floating point ALU, and fetch width 4 instructions. The branch prediction is 2-level with 1 entry in the L1-table, 4 entries in the L2-table and history of size of 2 and 2 memory system ports. After that, `sim-outorder` is configured to simulate a processor (i.e. NCNP) with the following characteristics, out-of-order issues of instructions, main memory latency 10 cycles for the first access and 2 cycles for the next accesses, memory access bus width 64 bytes, 1KB L1 instruction cache (1 cycle, LRU), no L1 data cache, no L2 cache, no TLB's, 1 integer ALU, 1 floating point ALU, and fetch width 1 instructions. The branch prediction is 2-level with 1 entry in the L1-table, 4 entries in the L2-table and history of size of 2, 1 memory system port, 1 instruction fetch queue size.

Next `sim-outorder` is configured to simulate a processor (i.e. NCSP) with the following characteristics, out-of-order issues of instructions, main memory latency 10 cycles for the first access and 2 cycles for the next accesses, memory access bus width 64 bytes, 1KB L1 instruction cache (1 cycle, LRU), no L1 data cache, no L2 cache, no TLB's, 1 integer ALU, 1 floating point ALU, and fetch width 1 instructions. The branch prediction is 2-level with 1 entry in the L1-table, 4 entries in the L2-table and history of size of 2, 1 memory system port, 4 instruction fetch queue size. Finally, `sim-outorder` is also configured to simulate a processor (i.e. advanced architecture) with the following characteristics: out-of-order issues of instructions, main memory latency 18 cycles for the first access and 2 cycles for the next accesses, 8KB L1 data and instruction cache, respectively (1 cycle), 256KB L2 data and instruction cache (6 cycles), all caches LRU, no TLB, 2 integer ALU's, 2 floating point ALU's, and fetch decode, issue, and commit width all 4 instructions, perfect branch prediction. The integer benchmarks were experimented for both single and multi-path benchmark program runs whereas floating-point benchmarks for single benchmark program runs. Furthermore, the findings and questions that could be a good ground for future works are also discussed.

5.1 Single path runs with Integer operation benchmarks

The results shown below were achieved after running Mälardalen benchmarks which only uses integer operations. In Table 7, the result produced using the standard architecture using `sim-outorder` executed with its standard configuration is presented. All programs have deviations from close to zero up to about 20% except `cover`, which is an extreme outlier with more than 50% underestimation similar the result produced in Table 5 in the RNTS2011 paper.

Program	Model	Measured	Diff	Rel. diff
bs	309	317	8	2.5%
cover	4060	8388	4328	51.6%
edn	286775	232561	54214	23.3%
esab_mod	795679	699934	95745	13.7%
fdct	10265	11294	1029	9.1%
fibcall	762	901	139	15.4%
fir	10162	8468	1694	20.0%
inssort10	4377	3529	848	24.0%
inssort15	642	579	63	10.9%
inssort20	852	759	93	12.3%
inssort30	1272	1119	153	13.7%
jcomplex	719	673	46	6.8%
loop3	12608	13371	763	5.7%
ns	37840	33718	4122	12.2%
nsichneu	22235	18545	3690	19.9%

Table 7: Predicted vs. measured times for single benchmark program runs, standard configuration

In Table 8, the NCNP architecture is relatively similar to the result from the standard configuration, except that the `nsichneu` is an extreme outlier benchmark with more than 115% underestimation. On the contrary, `cover` has improved to 35% compared to Table 7. In Table 9, for NCSP architecture, the `nsichneu` benchmark again showed to be an extreme outlier with more than 132% underestimation. Moreover, the `fdct` benchmark is an outlier with more than 53% underestimation but `cover` is still better than the standard configuration with less than 43% underestimation. As the hardware configuration gets more advanced, the underestimation of `cover` gets closer to the standard architecture. The `nsichneu` benchmark result is quite strange that in NCNP and NCSP architecture is an extreme outlier but when it is estimated in advanced architecture the deviation is only 8.3% as shown below in Table 10. The `nsichneu` program is somewhat special in that it is automatically generated, so its control structure can be slightly different from a hand-written code, but why does it behave so differently for these architectures and not in the advanced one? An inquiry was made in order to realize what property of the `nsichneu` code makes it behave so differently. Is it rich in some particular kind of instruction? Or is it something in its control structure, or data access pattern, which makes it behave like this? As further checks were done with the output of `nsichneu` from SimpleScalar, the main cause was branch prediction hit ratio which creates poor predictions by timing models.

Program	Model	Measured	Diff	Rel. diff
bs	288	281	7	2.5%
cover	4780	7310	2530	34.6%
edn	268190	264317	3873	1.5%
esab_mod	796055	931622	135567	14.6%
fdct	10223	8299	1924	23.2%
fibcall	928	991	63	6.4%
fir	11672	9941	1731	17.4%
inssort10	3789	3639	150	4.1%
inssort15	579	614	35	5.7%
inssort20	774	824	50	6.1%

inssort30	1164	1244	80	6.4%
jcomplex	802	735	67	9.1%
loop3	14212	14867	655	4.4%
ns	35164	37670	2506	6.7%
nsichneu	22244	10302	11942	115.9%

Table 8: Predicted vs. measured times for single benchmark program runs, NCNP architecture

Program	Model	Measured	Diff	Rel. diff
bs	210	187	23	12.3%
Cover	2941	5094	2153	42.3%
edn	213518	168261	45257	26.9%
esab_mod	582251	679220	96969	14.3%
fdct	7620	4953	2667	53.8%
fibcall	526	588	62	10.5%
fir	7026	6292	734	11.7%
inssort10	3227	2691	536	19.9%
inssort15	457	421	36	8.6%
inssort20	612	566	46	8.1%
inssort30	922	856	66	7.7%
jcomplex	519	492	27	5.5%
loop3	9122	9008	114	1.3%
ns	27975	30478	2503	8.2%
nsichneu	17460	7539	9921	131.6%

Table 9: Predicted vs. measured times for single benchmark program runs, NCSP architecture

Program	Model	Measured	Diff	Rel. diff
bs	130	184	54	29.3%
cover	1837	3605	1768	49.0%
edn	140136	119291	20845	17.5%
esab_mod	368743	408076	39333	9.6%
fdct	4998	3940	1058	26.9%
fibcall	283	377	94	24.9%
fir	3923	4035	112	2.8%
inssort10	2094	1678	416	24.8%
inssort15	284	303	19	6.3%
inssort20	379	395	16	4.0%
inssort30	569	568	1	0.2%
jcomplex	282	307	25	8.1%
loop3	5017	6290	1273	20.2%
ns	18758	18725	33	0.2%
nsichneu	10969	10129	840	8.3%

Table 10: Predicted vs. measured times for single benchmark program runs, advanced architecture

5.2 Multi path runs with Integer operation benchmarks

The outcome of the multipath runs from the model running in SWEET using a subset of Mälardalen benchmarks is:

Program	BCETe	WCETe
bsort100	6770	446998
esab_mod	693	1963089
insertsort	487	3682
minmax	84	145
ndes	170844	172748
nsichneu	12444	15344

BCETe/WCETe: estimated BCET/WCET

Table 11: BCET/WCET using SWEET analysis result, standard configuration

5.3 Single path runs with Floating-point

The results achieved by using floating-point instructions with a standard configuration are:

Program	Model	Measured	Diff	Rel. diff
bsort100	392446	422446	30000	7.1%
cnt	19031	15776	3255	20.6%
expint	7788	7226	562	7.8%
lcdnum	381	473	92	19.5%
ludcmp	9288	3131	6157	196.6%
matmult	676874	506351	170523	33.7%
minver	6536	3235	3301	102.0%
mm	9772493	8242615	1529878	18.6%
qsort-exam	2888	4594	1706	37.1%
qurt	3537	272	3265	1200. 4%
select	3259	5933	2674	45.1%
ud	8838	11689	2851	24.4%

Table 12: Predicted vs. measured times for single floating-point benchmark program runs, standard configuration

Program	Model	Measured	Diff	Rel. diff
bsort100	506982	453129	53853	11.9%
cnt	22715	11875	10840	91.3%
expint	8888	8648	240	2.8%
lcdnum	438	501	63	12.6%
ludcmp	13148	3236	9912	306.3%
matmult	885923	571057	314866	55.1%
minver	8952	3393	5559	163.8%
mm	12878120	9172484	3705636	40.4%
qsort-exam	3900	3823	77	2.0%
qurt	4283	164	4119	2511.6%
select	4459	3961	498	12.6%
ud	11870	11553	317	2.7%

Table 13: Predicted vs. measured times for single floating-point benchmark program runs, NCNP configuration

Program	Model	Measured	Diff	Rel. diff
bsort100	371415	312013	59402	19.0%
cnt	14701	9323	5378	57.7%
expint	6495	5710	785	13.7%
lcdnum	276	319	43	13.5%
ludcmp	9757	2369	7388	311.9%
matmult	597420	442143	155277	35.1%
minver	6500	2550	3950	154.9%
mm	8678437	7339252	1339185	18.2%
qsort-exam	2667	2698	31	1.1%
qurt	2802	102	2700	2647. 1%
select	3039	2939	100	3.4%
ud	8600	8651	51	0.6%

Table 14: Predicted vs. measured times for single floating-point benchmark program runs, NCSP configuration

Program	Model	Measured	Diff	Rel. diff
bsort100	240614	191037	49577	26.0%
cnt	8660	6095	2565	42.1%
expint	4680	3619	1061	29.3%
lcdnum	161	229	68	29.7%
ludcmp	6746	1635	5111	312.6%
matmult	362720	286220	76500	26.7%
minver	4397	1887	2510	133.0%
mm	5264241	5079570	184671	3.6%
qsort-exam	1610	1964	354	18.0%
qurt	1866	123	1743	1417. 1%
select	1895	2039	144	7.1%
ud	5599	5947	348	5.9%

Table 15: Predicted vs. measured times for single floating-point benchmark program runs, Advanced configuration

As shown in Table 12, 13, 14 and 15 for all the four hardware architecture (standard, NCNP, NCSP and advanced) there exist same trend that three huge outliers that underestimate the timing analysis, but the deviation produced by the `qurt` benchmark in all the hardware architecture is too large and it is assumed that SWEET cannot produce such deviation from the real measured times. Both `ludcmp` and `minver` are matrix computations having relatively regular loop structures, admittedly with some conditionals in the bodies of some loops. The deviation produced by them is 100-200%. One way to find the source of that outlier of these two benchmarks was to try with advanced architecture like NCSP to see if the result is equally bad. Moreover, `qurt` is short code having nested if-statements which may have an effect on the outcome. In order to find the source of this outlier, some checks have been done which identified the potential cause:-

- `qurt` is converted to a pure-integer program (just replace every `float` to `integer`) to check if the problem persists. The hypothesis is that if the outlier did not persist then it is not related to the handling of the float. After all occurrence of float in `qurt` converted to integer. The result achieved is:-

	Qurt (float, original)	**Qurt** (integer, converted)
SimpleScalar	272	4531
SWEET	3537	4003
Relative difference	**1220.4%**	**11.65%**

Table 16: Comparison Integer vs. float qurt benchmarks program measured times

The result shown in Table 16 proved that the reason for the outlier is not related to handling float. Further checks, like:

- Checking the output from SimpleScalar for the outliers to see whether there are similar discrepancies for the branch predictor hit ratio as for `nsichneu`, was carried out. It proved the branch prediction produced for `qurt` by SimpleScalar could not show any difference that causes the problem. Another check was:

- A SimpleScalar simulator to see if it behaves the same as running the program on a PC, i.e. insert some "`printf ()`" test outputs - to make sure the problem is not the simulator.

Benchmark	**X64_output**	**Sim-outorder_output**
qurt	1.000000 -3.000000 1.000000 1.000000 0.000000 1.000000 -16.000000 -2.000000	0.000000 0.000000 0.000000

Table 17: Comparison simulator vs. real hardware 64 bit architecture Printf () result

The result shown in Table 17 demonstrates that the SimpleScalar simulator is not simulating the float-benchmarks properly in the machine (64-bit OS and hardware architecture) used for the thesis. This result made the output achieved from SimpleScalar in Table 12, 13, 14 and 15 are questionable, which may also be useless.

A further inquiry has been done to find out why the SimpleScalar simulator was not working to see properly with the floating-point instruction. More benchmarks were tried using `printf ()` if further benchmarks could be found that has the same result with the simulator running on the same real hardware architecture. The hypothesis was that if some benchmark could produce the same result, the problem must be in the benchmarks in question. This would enable the dropping of those that are not simulated properly from further experiment. But none of the benchmarks produced the same result and it has been found later that the reason was that simulator does not simulate properly with 64-bit OS. The solution proposed is to setup a VM of 32-bit and run experiment.

5.4 Problem Encountered

As the linux version of SWEET is hardly used, the setup process took more time than expected. Moreover, regular updates made in SWEET often during the whole process of the thesis and it created several times minor problems on several occasions. These could have easily been avoided by sending some information of what changes had been made every time it was updated.

After the SimpleScalar configuration files for various hardware architecture (i.e. NCNP, NCSP, advanced) was supplied, the result that was produced was similar to standard configuration. The problem was with the advanced configuration it has not been a real "advanced" configuration but it was just the standard configuration dump by sim-outorder without changes. Later, a real advanced configuration with pipelining, caching and multiple alus/multipliers/dividers was implemented.

Last but not least, even though I have worked with a hardworking, cooperative and supportive group, I would like to mention that there was communication latency as I had more than four experts actively involved during the whole process of the thesis. Moreover, one of my important supervisors is located in Germany, which meant that I was able to meet him in person only twice. Most communication was through telephone or emails, when face to face meetings would have made my understanding of some concepts easier and less time-consuming.

6. RELATED WORK

The concept of early timing analysis has been investigated using different approaches throughout the last two decades. In this thesis some of the related works, which are similar to the work of this thesis are present briefly.

In [27] Nenova, Kästner used the TimingExplorer tool based on aiT [28] which aids to predict the early timing analysis using a source-code and enables the identification of suitable hardware architecture for real-time systems. The available source code is compiled and linked to each core in question and analyzed using TimingExplorer. The result of analysis is a WCET estimation of each code with the given hardware configuration.

Ferdinand and Heckmann [29] presented the aiT combined with a model-based design and automatic code generation SCADE and ASCET to achieve more secure and better-performing systems while decreasing time-to-market. aiT tries to find the upper bounds of a given program in a reasonable time while taking all possible hardware architecture into consideration. The integration of aiT with ASCET and SCADE is designed to be accessible from within respective graphical user interface. This combination made the static analysis tool achieve a high precision of the estimation.

Engblom, Ermedahl, Sjödin and Gustafsson has presented the WCET analysis of an embedded system in [30] focusing on some aspects which affect the methods and tools being developed. Their main target was to module architecture for WCET tool in which various WCET analysis components were used. The architecture allows integration of components along with comparison of methods to implement different components. They have done two control-flow analysis methods and a pipeline analysis. The target hardware used in this paper is a micro-controller simulator and a prototype of that architecture is developed with several of the components. To calculate tight and safe WCET estimates, a method is integrated that allows flow analysis and hardware analysis including the effects of caches. Moreover, a methodology is used to validate the components, pipeline analysis and calculation methods.

Guisto, Martin, and Harcourt [31] discuss in their paper how to derive a method for identifying a linear timing model for Source-Level Simulation based on task samples from particular domains using regression analysis and statistical-based predictor equations for SW estimation. Lisper and Santos [32] present a new approach to measurement-based WCET based on model identification which is end-to-end measurements of programs. The result achieved from the model does not underestimate any observed execution times. This is done to identify execution times for basic blocks in specific programs using binary level code.

7. FUTURE WORK

The target of this thesis was to evaluate the method for identifying timing models. The result achieved for single paths using integer operation benchmarks has been satisfactory in the entire hardware configuration (i.e. standard, NCNP, NCSP and advanced) produced deviation of less 20% on average. But the result achieved from multipath runs needs to calculate the real BCET/WCET for those selected benchmarks in Table 11, and compare the outcome and find out the range of the deviation if it is similar to the result achieve in the RNTS 2011. Furthermore, it is also important to identify floating-point benchmarks which can be accessed using multi-path source level timing.

The evaluation of floating-points has not been done prior to this thesis, needs to be deepened by setting up a 32-bit VM (Virtual machine) and see if the SimpleScalar simulator will work properly. Above all, it would be interesting and, could mean advancement for the research of WCET in evaluating the timing models of floating-points benchmarks are similar to integer operation benchmarks. This will enable the early timing analysis to be vast and applied to a wide range of hardware architecture configurations and in different kinds of operations. As a result, the risk of not meeting timing property of safety critical systems will be reduced.

8. SUMMARY and CONCLUSION

As the traditional systems have been fully replaced by embedded systems, early timing analysis is vital during the early stages of software development for real-time systems even before hardware is supplied and any code is compiled. To achieve this, early analysis is done on source code to produce a source-level timing analysis. The work that has been accomplished in this thesis is to evaluate the methods used to identify timing models. SWEET is a research WCET analysis tool developed by Mälardalen University and is used to produce the source-level timing analysis and calculations.

The methods used to identify timing models for a given hardware architecture has been evaluated using SWEET, and is compared with a result that is generated from a tool (i.e. SimpleScalar) which simulates hardware architecture. The result of this thesis showed that the source-level timing analysis for single paths using integer operation benchmarks has been satisfactory and the entire hardware configuration (i.e. standard, NCNP, NCSP and advanced) used in this thesis produced a deviation of less than 20% on average.

Moreover, it has produced a multipath source-level analysis using subsets of benchmark codes which are multipath programs. Floating-point source level timing analysis has also been addressed in this thesis (which is in its early stage). This thesis gives a starting point and roadmap on how to continue with that experiment in related future work. Finally, some of the results produced in this thesis have been used in the RNTS2011 which was published in September 2011.

REFERENCES

[1] A. Ermedahl, R. Wilhelm, S. Thesing, *The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools*, ACM Transactions on Embedded Computing Systems, Vol 7, No. 3, Article 36, April 2008

[2] P. Altenbernd, A. Ermedahl, B. Lisper and J. Gustafsson, *Automatic Generation of Timing Models of Timing Analysis of High-Level Code*, 19th International Conference on Real-Time and Network Systems (RTNS 2011), Nantes, France, September 2011

[3] SimpleScalar LLC, http://www.simplescalar.com/ [Accessed 31/08/2011]

[4] A. Ermedahl, B. Lisper and J. Gustafsson, L. Källberg, C. Sandberg, *ALF – A Language for WCET Flow Analysis*. In Niklas Holsti, editor, Proc. 9Th International Workshop on Worst-Case Execution Time Analysis (WCET'2009), page 1-11, Dublin, Ireland, June 2009. OCG

[5] Saranya Ntarajan, *Developing an ALF interpreter for the SWEET WCET analysis tool*, EURECA Exchange Student, Master's thesis, Department of Computer Science and Engineering, Mälardalen University, Sweden

[6] Mohammad Nazrul Islam, *Extending WCET benchmark programs*, Master's thesis, Department of Computer Science, Mälardalen University, Sweden, November 2011

[7] J. Gustafsson, A. Betts, A. Ermedahl and B. Lisper, *the Mälardalen WCET Benchmarks: Past, Present and Future*, Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010), pages 137-147, Brussels, Belgium, July 2010.

[8] Mälardalen WCET Benchmarks, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html [Accessed 09/09/2011]

[9] Introduction to SimpleScalar, http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/SimpleScalar_introduction.htm [Accessed 13/10/2011]

[10] T. Austin, E. Larson and D. Ernst, *SimpleScalar: An Infrastructure for Computer System Modeling*, University of Michigan, USA, February 2002

[11] WCET project/SWEET http://www.mrtc.mdh.se/projects/wcet/sweet [Accessed 23/09/2011]

[12] System of linear equations http://en.wikipedia.org/wiki/System_of_linear_equations [Accessed 02/11/2011]

[13] Linear equation http://en.wikipedia.org/wiki/Linear_equation [Accessed 02/11/2011]

[14] Least squares http://en.wikipedia.org/wiki/Least_squares [Accessed 02/11/2011]

[15] Simulated Annealing http://en.wikipedia.org/wiki/Simulated_annealing [Accessed 02/11/2011]

[16] Linear Equations http://cs.gmu.edu/cne/modules/dau/algebra/equations/linear1_frm.html [Accessed 02/11/2011]

[17] Least Squares Fitting
http://mathworld.wolfram.com/LeastSquaresFitting.html [Accessed 11/11/2011]

[18] The Method of Least Squares
http://web.williams.edu/go/math/sjmiller/public_html/BrownClasses/54/handouts/MethodLeastSquares.pdf [Accessed 11/11/2011]

[19] D. Bertsimas and J. Tsitsiklis, *Simulated Annealing*, Statistical Science, Vol.8, No.1, 10-15, 1993

[20] Simulated Annealing: Find global minima for bounded nonlinear problems
http://www.mathworks.se/discovery/simulated-annealing.html [Accessed 02/11/2011]

[21] Least Square Method
http://www.efunda.com/math/leastsquares/leastsquares.cfm [Accessed 02/11/2011]

[22] R. Kirner, P. Puschner, *Classification of WCET Analysis Techniques*, Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), May 2005

[23] D. Burger, T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, ACM SIGARCH, Vol. 25 Issue 3, June 1997

[24] D. Devaki AR, *A Translator from CRL2 representation of PowerPC Assembly to ALF*, Master's thesis, Department of Computer Science, Mälardalen University, Sweden, July 2009

[25] Simulated Annealing algorithm
http://www.gnu.org/software/gsl/manual/html_node/Simulated-Annealing-algorithm.html [Accessed 11/11/2011]

[26] P. Altenbernd, Equation Solver documentation [Accessed 10/05/2011]

[27] S. Nenova, D. Kästner, *Source-level worst-case timing estimation and architecture exploration in early design phases*. In N. Holsti editor, Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009), pages 12-22, Dublin, Ireland, June 2009

[28] C. Ferdinand, R.Heckmann, and B.Franzen, *Static memory and timing analysis of embedded systems code*, 3rd European Symposium on Verification and Validation of Software Systems (VVV'07), Eindhoven, The Netherlands, number 07-04 in TUE computer Science Reports, page 153-163, 2007

[29] C. Ferdinand, R. Heckmann, *Worst-Case Execution Time – a Tool Provider's Perspective*, 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC2008), pages 340-345, Orlando, Fl, USA, May 2008

[30] J. Engblom, A. Ermedahl, M. Sjödin and J. Gustafsson and H. Hansson, *Worst-case execution-time analysis for embedded real-time systems*, International Journal on Software Tools for Technology Transfer, vol 4, nr 4, p437-455 , 2002

[31] P. Guisto, G. Martin, and E. Harcourt, *Reliable estimation of execution time of embedded software*. In Proc Conference on Design, Automation and Test in Europe (DAC 2001), Los Alamitos, CA, USA, 2001. IEEE Computer Society

[32] B. Lisper and M. Santos, *Model identification for WCET analysis*. In Proc 15Th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09), page 55-64, San Francisco, CA, Apr. 2009. IEEE Computer Society

[33] MELMAC, MELMAC homepage, 2009
http://www.complang.tuwien.ac.at/gergo/melmac [Accessed 12/05/2012]

[34] aiT Worst-Case Execution Time Analyzers,
http://www.absint.com/ait/ [Accessed 01/06/2012]