UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME JAMES DE ANGELIS FACHINI

Modular and Generic WCET Static Analysis with LLVM Framework

Trabalho de Graduação.

Dr. Thomas Kuhn
Orientador

Prof^a^. Dr^a^. Erika Fernandes Cota
Co-orientadora

Porto Alegre, Junho, 2011

ACKNOWLEDGES

First of all I would thanks God for his merciful grace and infinite wisdom. To my beloved parents, James Fachini and Ana Lúcia De Angelis Fachini, for all their support, care and, best of all, love, during the hard and challenging academic times. Thanks to my little big heart sister “irmã”, Clarissa Raquel De Angelis Fachini, for making my days full of joy, by the way, sorry about the dailies scares that I give you.

I would like to thanks my friends Daniel Petersen, André Damasceno, Bernardo Zoppas, Tiago Kommers and Tiago Landenberger, the friends since the begging of everything, for not being supportive about classes, exams and studies, making my days better and full of strange, funny and remarkable moments. To all the folks that I met in Kaiserslautern, in special, the Fraunhofer’s, for the good coffee breaks, talk moments and knowledge exchange, in special Matheus Vogel and Carmela Noro Grando.

To Prof^a. Dr^a. Erika Fernandes Cota a big thanks for all her support and opportunities given during the time I was working as undergraduate researcher at Embedded Systems Laboratory. For introduce and encourage me in the academic area and help me reviewing this work.

I would like to thank my Fraunhofer advisor Dr. Thomas Kuhn for the opportunity for develop my bachelor thesis during my time working at Fraunhofer Institute – IESE and all the research support and knowledge.

My thanks go also to the Informatics Institute at UFRGS, to Professors and community, for the very good work being done to improve each day our knowledge and the development of our university and country, through research and technological development, despite all politics obstacles. In special for the Professors who provide time and resources in exchange programs, particularly to Prof^a. Dr^a. Taisy Silva Weber and the exchange program with Technische Universität Kaiserslautern. To Prof. Dr. Dr. h.c. H. Dieter Rombach, for invests in this program and provide full support for the students at TU-Kaiserslautern.

Finally, I thank those who were and are part of my life, all my uncles, aunts, grandfathers, grandmothers, brothers and sisters, relatives or not.

SUMMARY

Análise estática, genérica e modular de WCET utilizando a infra-estrutura de compilação do LLVM

RESUMO

O cálculo do tempo do pior-caso de execução, do inglês Worst Case Execution Time (WCET) é um desafio na área de verificação de software para sistemas de tempo real. Essa análise faz parte do trabalho de escalonamento de tarefas de processos em sistemas multi-cores. A complexidade de prever esse tempo aumenta de acordo com a complexidade do hardware do sistema a ser analisado e seus componentes, já que muitas partes de uma plataforma, como pipelines e memória cache inserem variantes no tempo de execução difíceis de prever e analisar. Existem vários métodos com diferentes abordagens para se calcular o tempo de execução de um programa. Eles são principalmente baseados em análises estáticas e dinâmicas, de forma que a estática utiliza um modelo de hardware e analisa o código, enquanto a dinâmica necessita de algum simulador ou de uma plataforma real para realizar as medidas de tempo.

Esse trabalho apresenta um modelo de análise estática para prever o tempo do pior-caso de execução de códigos para sistemas embarcados de tempo real. Além disso, executa, para fins de comparação, uma análise dinâmica baseado na execução dos códigos de teste em um simulador. O modelo de análise estática é desenvolvido baseado em um assembly gerado pela infra-estrutura de compilação do LLVM, que gera uma representação intermediária de código que é independente de arquitetura.

O método tem o objetivo de ser escalonável e modular, isto é, quão mais precisa a análise deve ser, melhores modelos de análise devem ser implementados e usados no processo. O modelo de análise utilizado nesse trabalho possui uma abordagem clara e utiliza uma descrição de arquitetura simples exemplificando o processo. A análise dinâmica é baseada no método Monte Carlo de simulação e é executada em um simulador da arquitetura de um AVR, o Atmega128, chamado AVRORA.

ABSTRACT

The worst-case execution time (WCET) prediction is a challenge in the software verification area for real-time embedded systems. This analysis is part of the scheduling of parallelizable processes job on multi-core systems. The complexity of predict this time increases accordingly to the complexity of the target system hardware and its components, since much of this components, as pipelines and cache memory, attach hard to predict and analyze temporal variants. There are many methods with different approaches to calculate the execution time of a program and they are based mainly on static and dynamics analysis. The first takes account a hardware model and the code analysis itself, while the second needs some accurate simulator or the target platform to perform its time measurements.

This work presents a static method for predicting worst-case execution time (WCET) for embedded real-time systems. Furthermore, performs, as a comparison of accuracy, a dynamic analysis, running the test codes in a simulator. The static analysis is performed based on an assembly generated by the LLVM compiler. This compilation framework generates a code intermediate representation (IR) independent of architecture.

The model aims to be a modular and scalable, that means, it was build to accept different accuracy levels, depending on the accuracy of the developed hardware model and the analysis used in the process. In this work, this work analysis follows a clear approach and uses a simple architecture description for its execution. The dynamic analysis is done based on the Monte Carlo simulation method, performed over the AVR ATmega128 simulator, AVRORA.

1 INTRODUCTION AND MOTIVATION

In the past, processing speed of embedded platforms used to be increased by increasing their clock frequency, as multipurpose systems. Due to electromagnetic interferences, increase processing power by increasing the clock frequency is not possible anymore. To overcome this limitation, parallel processing units are replacing single processors running at high clock speed. However, to fully utilize the processing resources from parallel platforms, optimized algorithms and sophisticated parallelization process are required.

The parallelization of an algorithm implies the distribution of code blocks among the processing units at the multi core platforms and the management of communication and synchronization. Parallelization is only feasible if time savings due to parallel execution offsets the effort for synchronization and communication. Therefore, successful automatic algorithm parallelization depends on the knowledge of execution times.

For real time systems, Worst Case Execution Times need to be considered. Since execution times for code blocks are normally not known by developers, this information needs to be acquired during the deployment and parallelization process. This work was done during an internship at Fraunhofer Institute IESE where a project about scheduling process for multi cores architectures was being developed over Simulink. In this context, the WCET calculation was necessary for the scheduling algorithm. Here will be describe an approach and method to measure worst case execution time of code segments generated from Simulink models on different target platforms.

The problem of calculating WCET is well described and several approaches and methods have been developed [1]. However it keeps challenging research because of the difficulty of having more accurate timing prediction models. The issue comes mainly due to the variation of the code runtime depend on hardly or not predictable factors, such as:

- Input data: variables that affect the code flow.
- Loop and recursion bounds
- Initial state of the execution block: e.g. caches, pipelines, branch prediction.
- System interference: interruptions, preemptions.

Mainly there are two methods to predict the worst case execution time, namely static and dynamic: Static methods are based on the control-flow and call-graph analysis combined with some abstract hardware model, and does not depend on the execution of the code. Dynamic methods involve the execution of the code on some simulator or on the target hardware itself with some defined inputs and the WCET is calculated based on the observation of the execution times.

This work follows a static model approach based on a static analysis over the low level virtual machine (LLVM), which provides a layer between C and C++ code and concrete architecture machine code, in comparison with a dynamic analysis based on Monte Carlo simulation over AVRORA, an AVR architecture simulator. The dynamic analysis approach aims to validate the static one, which is based on code annotations and instructions mapping. Aiming an easy to understand and simple method, this work develops an architecture independent, modular and reusable static analysis model aiming an easy aggregation of new feature, analysis and more accurate hardware models.

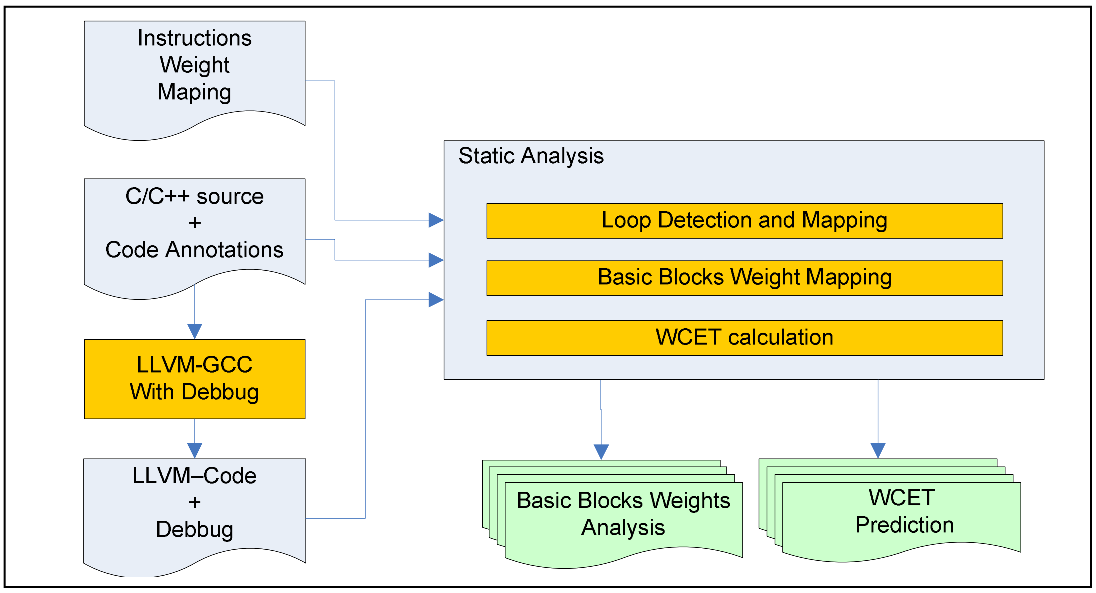

Instructions Weight Maping
C/C++ source + Code Annotations
LLVM-GCC With Debbug
LLVM–Code + Debbug
Static Analysis
Loop Detection and Mapping
Basic Blocks Weight Mapping
WCET calculation
Basic Blocks Weights Analysis
WCET Prediction

Figure 1.1 – Static Analysis Workflow

Figure 1.1 gives an overview about the static analysis process made in this work. Each step is described in more details in Chapter 3.

The work has the following structure: Chapter 2 describes related works and gives an overview about the tools used in this work. Chapter 3 explains the static analysis method and the mapping between real architecture and LLVM instructions, besides showing some results. Chapter 4 gives an overview about Monte Carlo analysis and the data generation for the simulation. The evaluation and comparison of results are presented in chapter 5, while a discussion about them is given in chapter 6. The last chapter presents the conclusion and describes future implementation and improvements for the static method.

2 RELATED WORKS AND TOOLS

Related works involving approaches based on dynamic and static analysis are discussed, and some description about the used tools will be presented.

2.1 Related Works

[R. Wilhelm, et al.] in [1] gives an overview of the whole problem of calculating WCET describing its sub problems and the existing methods and tools to solve it. The approaches are divided in static and measurement-based and they are compared considering their aims, abilities, technical problems and research directions. One of the main problems of the static analysis is about having precise processors models such that results are not overestimated. Some of the static methods described in [1] are shown by Figure 2.1, where *a* is a CFG example with timing on the nodes, *b* is a path calculation method, *c* shows the Implicit Path Enumeration Technique and *d* shows how a structure-based method proceed according to the syntax tree. This is also discussed by [Y.-T.S. Li, S. Malik, A. Wolfe] in [12] where an integer linear programming formulation is used to solve the problem, targeting an Intel i960KB, their solution address pipeline instruction execution units and cached memory modeling.

[S. Thesing] in [3] implements a technique to describe pipeline models. There they use the AbsInt's WCET analysis tool [13] and a full analysis is made in eight phases: reconstruction of the control-flow graph from the binary executable, loop transformation, loop analysis, value analysis, pipeline analysis, path analysis (where the tools generates a integer linear program), ILP solver and the last phase computes as display results, this approach follows the scheme in Figure 2.1. Using the same analysis tool [13], [M. Schlickling, M. Pister] semi-automatically derives timing models from formal VHDL specifications to compute a cycle abstract semantic for further use as hardware model on AbsInt's tool. Also working over pipeline modeling [M. Langenbach, S. Thesing, R. Heckmann] in [9] developed a tool called *ColdFire WCET Tool*. Their analysis is performed on the control flow graph representation and is divided in two phases: execution modeling and program path analysis. [F. Mueller] implements a technique called *static cache simulation* which statically simulates a large portion of cache behavior of programs. Based on the call graph of the program and the control-flow graph of each function the instructions references are statically determined as *always cache hit* or *always cache miss*.

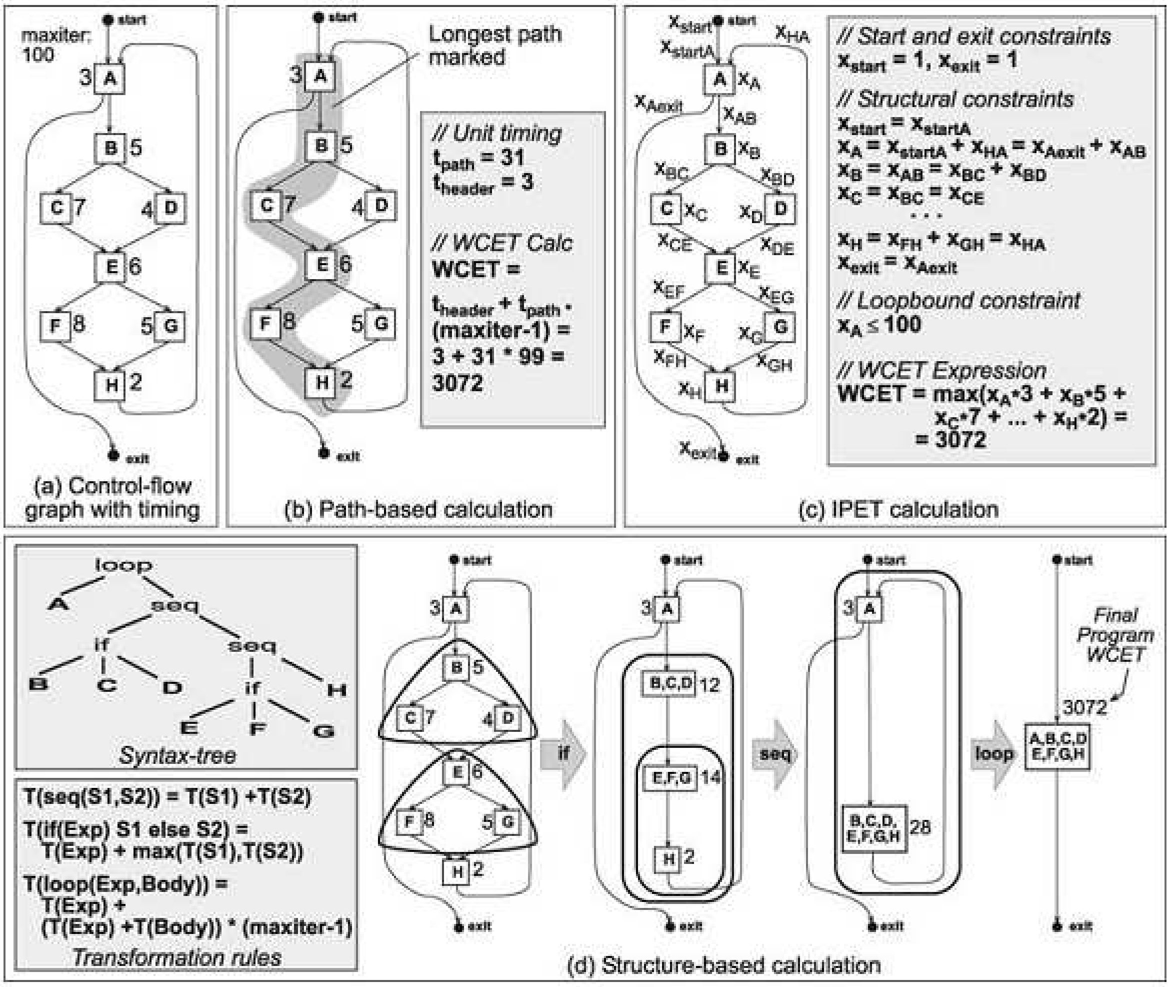

maxiter: 100
start
exit
(a) Control-flow graph with timing
Longest path marked
// Unit timing
t_path = 31
t_header = 3
// WCET Calc
WCET =
t_header + t_path * (maxiter-1) =
3 + 31 * 99 =
3072
(b) Path-based calculation
// Start and exit constraints
x_start = 1, x_exit = 1
// Structural constraints
x_start = x_startA
x_A = x_startA + x_HA = x_Aexit + x_AB
x_B = x_AB = x_BC + x_BD
x_C = x_BC = x_CE
...
x_H = x_FH + x_GH = x_HA
x_exit = x_Aexit
// Loopbound constraint
x_A ≤ 100
// WCET Expression
WCET = max(x_A*3 + x_B*5 + x_C*7 + ... + x_H*2) =
= 3072
(c) IPET calculation
loop
seq
if
Syntax-tree
T(seq(S1,S2)) = T(S1) +T(S2)
T(if(Exp) S1 else S2) = T(Exp) + max(T(S1),T(S2))
T(loop(Exp,Body)) = T(Exp) + (T(Exp) +T(Body)) * (maxiter-1)
Transformation rules
Final Program WCET
3072
(d) Structure-based calculation

Figure 2.1 – WCET calculations

A measurement-based analysis is presented by [I. Wenzel, R. Kirner, B. Rieder, P. Puschner] in [6] where test data are generated automatically. Their approaches are based on decomposing program paths into sub paths of program segments and measure the runtime from each segment. After measuring instruction timing of sub paths, they use a static calculation method for calculate the final WCET. That work describes a hybrid approach using static and dynamic analysis to predict WCET. Following a measurement-based approach J. Hansen, S. Hissam, G. A. Moreno] estimates WCET values based on a statistical method, the extreme value theory. This method not only estimates the WCET, it also gives a probability that a possible execution time will exceed the estimations.

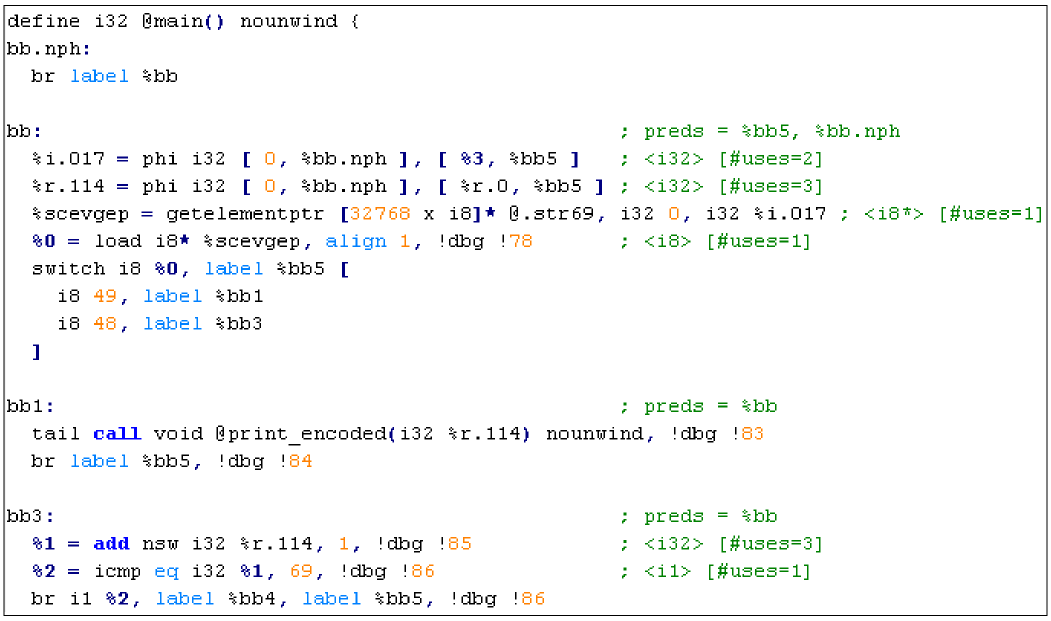
2.2 Tools

In this work two open source tools were used as base for WCET analysis. The main tool was the LLVM compiler infrastructure [14], version 2.7, which provides an architecture independent compiler (based on GCC 4.2), optimizer and an analyzable intermediate code representation. Besides the C and C++ front-end, the LLVM project provides several others tools for code analysis, optimization, code generation, etc. For this work the llvm-gcc front-end with debug information was the base for the static analysis, while for the dynamic analysis is also used the native machine code generator (llc).

Specifically for the dynamic analysis, a cycle accurate simulator was chosen, Avrora, an AVR framework for simulation and analysis [15].

2.2.1 LLVM

The LLVM project aims for versatility, flexibility and reusability, these characteristics fit into our approach for a WCET analysis method, since LLVM provides also a well documented and architecture independent assembly. It specifies three different representations of assembly, an in-memory compiler, an on-disk bitcode and a human readable assembly, all generate from C/C++ codes. For this work, the static analysis was made over the human readable assembly aggregated with the debug information, Figure 2.1.



```
define i32 @main() nounwind {
bb.nph:
  br label %bb

bb:                                                  ; preds = %bb5, %bb.nph
  %i.017 = phi i32 [ 0, %bb.nph ], [ %3, %bb5 ]    ; <i32> [#uses=2]
  %r.114 = phi i32 [ 0, %bb.nph ], [ %r.0, %bb5 ] ; <i32> [#uses=3]
  %scevgep = getelementptr [32768 x i8]* @.str69, i32 0, i32 %i.017 ; <i8*> [#uses=1]
  %0 = load i8* %scevgep, align 1, !dbg !78        ; <i8> [#uses=1]
  switch i8 %0, label %bb5 [
    i8 49, label %bb1
    i8 48, label %bb3
  ]

bb1:                                                 ; preds = %bb
  tail call void @print_encoded(i32 %r.114) nounwind, !dbg !83
  br label %bb5, !dbg !84

bb3:                                                 ; preds = %bb
  %1 = add nsw i32 %r.114, 1, !dbg !85              ; <i32> [#uses=3]
  %2 = icmp eq i32 %1, 69, !dbg !86                 ; <i1> [#uses=1]
  br i1 %2, label %bb4, label %bb5, !dbg !86
```

Figure 2.2 - LLVM code Example

The human readable assembly is a designed set of instructions with low-level representation but with support for high-level analyses. For this the instruction set does not define any machine specific constraints or features such pipelines, physical registers or call conventions. The registers are in Static Single Assignment (SSA) form used for compiler optimization and can only hold scalar values as Boolean, integer, floating point and pointer. All memory is explicitly allocated and it is partitioned into stack, heap and global memory, and its data are accessed only with *load* and *store* instructions. About the LLVM instruction set types, the system is very simple and can easily represent high-level classes combining the low-level types, it is also strongly-typed (every SSA value and memory locations has an type associated as seen in Figure 2.1) allowing easy mismatch type detection and optimizations.

For the WCET analysis one very important characteristic from the LLVM assembly code is the explicit Control Flow Graph (CFG) organization. The code is structured in many Basic Blocks (BB), which are defined as a linear sequence of code having one entry point (labeled normally with “bb” and explicitly listing the successors basic blocks) and one exit point (a Terminator Instruction) also described by [Frances E. Allen] in [20]. This construction allows the full flow mapping with a near to machine, but independent code. These mappings are widely use for loops identification, described in chapter 3.2, and the full path calculation for the static analysis. This feature is used by the LLVM infrastructure for analysis and optimizations of control flow, for example.

All these characteristics makes the LLVM project a base for several other projects that have to deal with low level, platform independent code, which gives, since the beginning a very modular foundation.

2.2.2 Avrora

Avrora [15] is a research project of the UCLA Compilers Group and is a set of simulation and analysis tools implemented in Java [16] for AVR microcontrollers produced by Atmel. The framework provides also a Java API and infrastructure for experimentation, profiling and analysis. The core from the whole Avrora project is their cycle-accurate simulator for AVR microcontroller that allows a precise timing analysis for real programs.

For the purpose of simulation and debugging breakpoints are inserted into the code to terminate or pause the simulation. After the complete simulation a report with the results is automatically generated, Figure 2.2 shows an Avrora report example. In addition to the timing analysis, this framework can also emulate the behavior of on-chip devices, like led blinking.

```
Avrora [Beta 1.6.0] - (c) 2003-2005 UCLA Compilers Group

This simulator and analysis tool is provided with absolutely no warranty,
either expressed or implied. It is provided to you with the hope that it be
useful for evaluation of and experimentation with microcontroller and sensor
network programs. For more information about the license that this software is
provided to you under, specify the "license" option.

Loading RLencode.od...[OK: 0.484 seconds]
=={ Simulation events }=================================================
Node        Time   Event
========================================================================
Simulated time: 2436497 cycles
Time for simulation: 3.937 seconds
Total throughput: 0.61887145 mhz
```

Figure 2.3 – Avrora Report Example

For this work the single simulation mode in the dynamic analysis is used. This mode offers several configuration options, like clock speed, interrupt schedule, microcontroller model, monitors, platform, etc.. The microcontroller model used here was the ATmega128, briefly explained in chapter 4.1.

The Avrora framework proved to be easy to use and deploy. Their reports and analysis fits on what was necessary for this project and provided a feasible result on measurements.

3 PROPOSED STATIC BASED WCET ANALYSIS

Static analysis is a well know technique for evaluation and verification of code. Specifically for WCET prediction, its allows a fast and modular analysis. For this work, a technique of basic blocks mapping and instructions counting is used.

Based on the human readable LLVM assembly with debug information and the source code in C or C++, two mappings are performed, one is the loops mapping from de source code to the LLVM, and another is the target architecture instructions cycles to LLVM instructions mapping. These informations are the inputs for the whole execution time analysis and from them, is possible to predict the worst execution time from a program code.

3.1 Source Code Programming

As previously mentioned, there are some non-predictable factors in calculating de worst case execution time from a program. This implies some programming restrictions and code annotations in order to make any execution time prediction possible.

Some programming problems mentioned by [P. Puschner, Ch. Koza] in [22] are that loops and recursions end‘s conditions, or maximum number of iterations or recursion depth, cannot be easily calculated or determined statically, because of the possibility of very complex conditions that cannot be automatic determined. Furthermore pointers to functions can reference functions that the timing is not known and implicitly implement recursion, besides the case of GOTO usage, which can disturb with the whole program structure. These conditions can be avoided by some programming restrictions and code descriptions.

3.1.1 Code Restrictions

In order to avoid the determination of a recursive and loops end’s condition and non structured control flow, some programming restrictions were defined:

- Program cannot contain any kind of recursion (direct or indirect) [21] [22].
- GOTOs and unconditional changes of flow are not allowed, which do not really performs a restriction according to [22].
- Loops should have a specified iteration or time bound, and it cannot overcome this limit.
- Functions should be called explicitly and each function must have a calculated WCET.

3.1.2 Annotations

In this work, some additional information is defined to be inserted in the analyzed code. This information aims at specifying which functions should be analyzed and the loops iteration bounds.

The annotations follow a very simple syntax and should be added as normal comments in the source C code, as shown in Figure 3.1. The beginning and the end of a function to be analyzed should follow the syntax "<F><Function_Name>" and "</F><Function_Name>" respectively. This syntax was defined to facilitate the identification of the functions to be analyzed.

The loop bounds annotation defines how many times each loop would iterate. This should be provided by the programmer, so this must be known or pre defined during developing phase. Following the same idea at the function annotation, "<I><#Iterations>", defines the loop opening, and "</I><#Iterations>", the loop closure. Each loop statement (for, while, do) should have its own bound, allowing nesting. These bounds are used later on the loops identification and mapping (after optimization some loop can be unrolled) and for the WCET counting, to determine how many times a basic block, inside a loop, will be executed.

```
//<F><example>
void example()
{
    //<I><12>
    while( i++ < T )
    {
        //<I><127>
        for(j=0; j<W; j++)
        {
        ...
        }//</I><127>
    }//</I><12>
}
//</F><example>
```

Figure 3.1 – Function and Loop Bounds Syntax Example

3.2 Loops Mapping

As shown in Figure 1.1, two codes are used for the static time analysis, the source code in C and the human-readable LLVM assembly. The LLVM assembly is the analyzed base code, because it has been already processed in compilations. That means, it does not contain architecture dependent directives and already has been optimized and transformed, becoming quite near the actual target code. However many loops annotated on the C code can be optimized and unrolled during compilation. Those loops would not exist anymore in LLVM code, being necessary thus a mapping between these two codes for a correct loop counting.

To determine which loops have not been optimized a control flow analysis was used to identify the loops and then, with the assistance of debug informations, map the loop bounds described on C code to the LLVM basic blocks. To find the loops, a well described algorithm to find Natural Loops as described by [A. V. Aho, R. Sethi, J. D. Ullman] in [23] was used.

3.2.1 Natural Loops

Before presenting the natural loop identification algorithm, the loop itself must be defined. The loop intuitive properties are that, they should have a single entry point and the edges must form at least a cycle. From this it is inferred that not every cycle is a natural loop.

The algorithm presented in [23] uses a technique that consists of three components to find and identify natural loops. The first component is to build a dominator tree out of the Control Flow Graph (CFG). The formal definition of dominators is:

- Node *d* dominates *n* (*d* **dom** *n*) in a graph if every path from the start node to *n* goes through *d*.

A dominator can be found if all paths to a given node have to go though another node. Starting from the entry node in a CFG, the algorithm needs to check if there is a path to the slave node from the entry. This path must avoid the master node, so, if it is possible to reach the slave node without touching the master node, it can be determined that the master node does not dominate the slave node.

The second component is identifying the back edges, this step use the dominator list done in the previous component. The algorithm performs a depth first search in the CFG and for each retreating edge *tail->head*, where *head* dominates *tail* (checking if *head* is *tail's* dominator list), defines a back edge.

Finally the natural loop of a back edge is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessor outside the set, except for the predecessors of the header. The algorithm in short is, delete the *header* from the flow graph and find all nodes that can reach the *tail* (those nodes plus *header* form the natural loop of *tail->header*).

3.3 Instructions Mapping

Besides the two input files, C code and LLVM assembly, another file is used in the analysis as instructions cycles reference. The Instructions Weight Map describes the number of cycles from the target architecture for each LLVM assembly instruction. For the static analysis performed in this work it is the only information used from the target architecture.

This mapping is extracted compiling each LLVM instruction to the ATmega128 and which is possible because the AVR Microcontroller has a very simple architecture. The processor achieves a throughput of 1 MIPS per MHz, meaning that almost all its instructions execute in a single clock cycle. However LLVM deals with high level instructions that cannot be directly mapped as 1-to-1. For example, an 32 bits *AND* operation between two registers in LLVM is written as follows:

```
%0 = and i32 %i, %j
```

While in AVR assembly it becomes, besides some context instructions:

```
ldd r24, Y+1
ldd r25, Y+2
andi r24, 0x00
andi r25, 0x28
```

For the *and* instruction it is easy to detect the reason for the difference, which is the fact that the microcontroller ATmega128 works with 8 bits registers, while the LLVM instruction is explicitly dealing with 32 bits. However when dealing with more complex LLVM instructions, that do not have any near representation in AVR, the difference becomes really big. For instance, in the case of floating point instructions, an *fadd* instruction in LLVM is mapped to a 768 lines AVR assembly code.

LLVM Instruction	AVR cycles
add	14
alloca	55
and	10
ashr	8
bitcast .. to	8
br	23
call	6
extractelement	47
extractvalue	39
fadd	1085
fcmp	16
fdiv	819
fmul	917
fpext .. to	18
fptosi .. to	417
fptoui .. to	1512
fptrunc .. to	18
frem	823
fsub	1123
getelementptr	8
icmp	24
indirectbr	0
insertelement	118
insertvalue	93
inttoptr .. to	8
invoke	0
load	20
lshr	8
mul	25
or	9
phi	0
ptrtoint .. to	8
ret	8
sdiv	75
select	8
sext .. to	8
shl	8
shufflevector	0
sitofp .. to	616
srem	78
store	18
sub	14
switch	75
trunc .. to	8
udiv	46
uitofp .. to	652
unreachable	0
unwind	0
urem	41
va_arg	0
xor	12
zext .. to	8

Table 1 – LLVM instructions cycles mapping

The Table 1 shows the number of cycles mapped from ATmega128 [17] to each LLVM instruction. This table was generated with a reverse engineering process, since LLVM is not portable to AVR yet. Each LLVM instruction was compiled back to a low level C (by the LLVM tool chain) and the C code compiled to AVR assembly, from which the cycle counting was taken.

3.4 WCET Mapping and Calculation

The full static analysis is based on the basic blocks control flow and in the instructions counting from each BB. The process begins on the algorithm planning phase, where developers should define which functions or methods will be measured and the others functions related to the main one. During development programmers should define also the loops bounds for each loop defined in the algorithm, as explained in section 3.1.2. Another pre-analysis task is to map the target architecture instructions cycles to LLVM instructions, as shown in section 3.3.

Having the code ready, it is necessary to compile it to the readable LLVM assembly version with the debug information and the intended optimization level. This is done by the following command line:

llvm-gcc (LLVM optimizer and code generator) **–emit-llvm** (emit LLVM IR) **–S** (readable assembly) **–g** (debug information) **–O0..4** (desired optimization level) ***.c –o *.ll**

Generated the three inputs (Instruction Weight Mapping, Source Code and LLVM Code), the analysis begins with the Natural Loops detection on the LLVM code, as described in section 3.2. The detection generates a Natural Loop list that is mapped with the loops annotations included in the source code with the aim of extracting the bounds information. The mapping involves the use of the debug information as guide for the lines positions from the loops. The debug information in LLVM use a special type called *metadata*; this directive can be attached to instructions in the program to provide extra information about the code to the optimizer, code generator, debugging or any other analysis.

All *metadata* has the *metadata* type and is identified syntactically by a preceding exclamation point '!'. There are also two primitives defining this type: strings and nodes, the debug information uses metadata nodes which are represented with notation similar to structure constants, for example:

```
"!20 = metadata !{i32 51, i32 0, metadata !15, null}"
```

Here "!20" represent location information metadata. The four fields represent respectively: line number, column number, scope and original scope from the source code. This information was used to map the loop bound from the C code to LLVM IR.

```
//<F><main>
int main(void) {
   int i, j, k,l;
   //<I><100>
   for (i = 0; i < 100; i++) {
          //<I><100>
          for (j = 0; j < 100; j++) {
                 //<I><100>
                 for (k = 0; k < 100; k++) {
                        l = i+j+k;
                 } //</I><100>
          } //</I><100>
} //</I><100>
return l;
}
//</F><main>
```

Figure 3.2 – C source Example

As observed in Figure 3.2, the first annotated loop begins at line 6. Searching for the metadata node that represents line 6 in the LLVM code, in Figure 3.3, we find the metadata “!11”. So on, all loops are mapped this way, and from this it is possible to decide which basic blocks are inside each loop, and how many times they will iterate and their nesting level. Of course when a loop is unrolled by the compiler, it will not have a related metadata with its line.

```
define i32 @main() nounwind {
entry:
 %i = alloca i32
…
store i32 0, i32* %i, align 4, !dbg !11
br label %bb7, !dbg !11
...
bb7:
%16 = load i32* %i, align 4, !dbg !11
%17 = icmp sle i32 %16, 99, !dbg !11
br i1 %17, label %bb, label %bb8, !dbg !11
…
return:
%retval9 = load i32* %retval, !dbg !15
ret i32 %retval9, !dbg !15
}
...
!11 = metadata !{i32 6, i32 0, metadata !1, null}
```

Figure 3.3 – LLVM IR with metadata

Continuing the analysis, each basic block is then mapped into a list with the following information: *Name, Predecessor, Metadatas, Branches, Calls and Weight*, where:

- *Name* is the tag given by the compiler for the basic block, each BB has a unique name
- *Predecessor* is a list with all BB that reaches the mapped BB in the CFG
- *Metadatas* are the metadatas associated to this BB
- *Branches* is the list of reachable BBs in the CFG

- *Calls* is a list with all other functions called in this particular BB, used for later link, when all functions are analyzed
- *Weight* is the sum of the weight, mapped in Instruction Weight Mapping file, from all instructions in this BB

After the loop and the weight mapping, these two information are merged into one, where the matches between them generates the full BB mapping, and then their weight are multiplied by the iteration number when applicable (if the BB received a loop bound). In the loop mapping description, the fields mean respectively: *metadata*, *BB name*, *nesting level* and *iterations*.

```
(!13,bb3,3,1000000)
(!11,bb6,1,100)
```

Figure 3.4 shows the simple BB report on the left with only the BB informations and the weight from each BB, while on the right is the full analysis report, with the BB weights already multiplied by the iteration number resulting in the expected cycle's number for each BB. As final result, the sum from all BB cycles is given.

```
Simple Report:
Name: entry
Preds:
Metadatas: !7 !11
Branchs: bb7
Calls: llvm.dbg.declare
Weight: 450
-----------------------
...
-----------------------
Name: return
Preds: bb8
Metadatas: !15
Branches: END
Calls:
Weight: 28
-----------------------
```

```
Full Report:
Function - main

******BASIC BLOCKS EXECUTED******
Name: entry
Calls: llvm.dbg.declare
Loop Iteraction: 0
BB cycles: 450
------------------------------
...
------------------------------
Name: return
Loop Iteration: 0
BB cycles: 28
------------------------------
******************************
Total Executed Cycles: 290844777
******************************
```

Figure 3.4 – Simple and Full WCET Repo

3.5 Analysis

In this work, two classical algorithms [24] were used to evaluate the accuracy of the researched static analysis. The chosen algorithms were a Dijkstra shortest path algorithm, and an encode algorithm that uses the Huffman Compress code. They were selected because of their structure. Dijkstra presents interesting nesting levels between loops, while the encoder works with a single loop in the main function but with large case structure. The C codes of each algorithm can be found in annexes.

3.5.1 Dijkstra's Algorithm

The Dijkstra's Algorithm [24] created by Edsger Dijkstra and published in 1959 is a search algorithm for graph that solves the single-source path problem producing the shortest path between the source and all other reachable nodes. In a graph, given a source vertex, the algorithm finds the shortest path leaving the source to all other vertex. This algorithm is widely used to find the shortest route between cities and in network routing protocols.

The algorithm is implemented in this way:

1. Beginning with the source node, current node:
    a. Set its value to 0.
    b. Set value of all other nodes to "infinity".
    c. Mark all nodes as unvisited
2. For each unvisited node, adjacent to the current node:
    a. If the value of the current node plus the value of the edge is less than the value of the adjacent node, change the value of the adjacent node to this sum value.
    b. Otherwise leave the value as it is.
3. Set the current node to visited.
4. Finish if every node have been visited. Otherwise, set the unvisited node with the smallest value as the new current node a go to step 2.

The value recorded in each node is the final and minimal distance between this node and the source.

3.5.2 Huffman Compression Algorithm

Huffman compression [24] is an encoding algorithm used for lossless compression. It belongs to a family of algorithms called entropy encoding algorithms which refers to the use of a variable-length code table. That means the characters symbols are replaced by bit sequences with different lengths. Based on the estimated probability of occurrence of each symbol in the source, symbols that tend to occur more, are encoded into fewer bits, while rarely used symbols gets longer bit sequences.

The code used in this work implements a simple Huffman compression that expects a binary file with a sparse distribution of bits with the value 1. The compression process is performed by counting the number of zeroes between each successive pair of ones and then, encoding these outcomes using a symbol code table.

3.6 Results

Following the approach described previously, Table 2 shows some results acquired from the algorithm with static analysis, with the instructions weights from Table 1, in two optimization level, O0 and O3, meaning no optimization at all and the common optimization level used in general, all performed by LLVM tool chain. These results will be compared with the Simulations results acquired by Monte Carlo Analysis on Avrora AVR simulator.

	-O0	-O3
Dijkstra	4.814.878.170 Cycles	2.181.701.037 Cycles
Huffman Compress	66.603.932.946.841 Cycles	3.138.178.705 Cycles

Table 2 – Static Analysis Results

4 SIMULATION AND EVALUATION

This chapter discusses the results from the static and dynamic analysis. Both evaluations are compared in order to ensure some validity and quality of the static analysis. First of all, the simulations process is explained.

4.1 Simulations

The simulations were performed to be compared with the static analysis results and give an approximation of the real WCET. Following a Monte Carlo approach, the inputs were randomly generated and simulated on the cycle accurate simulator AVRORA. By the end of the execution the simulator provides a report informing the number of cycles for the program execution.

4.1.1 Simulation Architecture

Performing a simulation, AVRORA was used as the main engine for the dynamic analysis approach.

4.1.1.1 AVRORA

The AVRORA framework is implemented in Java and focus on a clean design and program representation. Each type of instruction has it own class and, instances of these classes represent the instructions from a program. The core of the framework is the Java package *avrora.sim* which contains a set of classes that implements the simulator.

The simulator is the execution engine of a simulation. It contains an interpreter for all AVR instructions in the set and store states of the program including SRAM, IO registers and general purpose registers. Inserted in the layer of a microcontroller (devices built on a chip) it emulates the behavior of on-chip devices and provides an interface with off-chip devices (platform level). The simulations for this work were performed over the ATMega128 microcontroller class.

4.1.1.2 ATmega128

ATmega128 is a simple microcontroller with RISC architecture. With most of its 133 instructions executing in one clock cycle, so it can reach up to 16 MIPS at 16 MHz. It has also 128 KB of program memory, 4 KB EEPROM and 4 KB SRAM. This architecture was chosen by its simplicity, since it doesn’t implements any kind of pipeline, cache protocol and branch prediction, features expected for future works improvements.

4.1.2 Monte Carlo Method

Monte Carlo methods consist in repeatedly run some algorithm with random inputs and collect their results. These methods are frequently used for simulating physical and mathematical systems, when is not feasible or is impossible to generate and run a deterministic algorithm to calculate an exact result, like simulations of systems with a large degree of freedom, multiples variables and complicated boundary conditions.

Algorithms itself are process that fits on “complicated to predict and simulate” concept, so, for this work the Monte Carlo method was used. As Monte Carlo doesn’t define a single method indeed, but a set of algorithms that follows the described concept, the approach used for generating the inputs and simulating the algorithms in this work was:

- Define the inputs bounds, based on their respective types and memory size
- Generate random inputs between the defined limits
- Run the algorithm on simulator
- Save execution time from simulation

The inputs for the evaluation algorithms were generated by a script written in the language Perl [25]. The scripts generate random inputs values in the model expected by the evaluations algorithms. The model used for the inputs were C header codes included during compilation time, since the simulator do not accept dynamic inputs references. Huffman compress and Dijkstra’s inputs scripts are in the appendix.

4.1.3 Analysis

Using the same algorithms described in Chapter 3.5, the dynamic analysis was performed by a Shell Script following these steps executed inside a “while loop” (iterating as much as possible):

- Generate random inputs; input scripts, in appendix.
- Compile code using “*avr-gcc*” and generate a .elf binary file.
- Dumb object from binary file using “*avr-objdump*” (the simulator expect this format).
- Run simulator and keep the simulated time result.

The results generated on the dynamic analysis are shown in Table 3 and also each algorithm was tested with two different optimization levels. The minimum and the maximum number of cycles reached during the simulations shows the difference between the best and the worst case of executed cycles during simulation.

4.2 Results Evaluation

The Table 3 shows the absolute values from the dynamic and static analysis. The difference between the maximum number of cycles simulated and the WCET prediction is also calculated. This first comparison shows only the big difference between both analyses, being necessary a more meaningful examination.

		Simulation		Static	max - static
		min	max		
Huffman Compress	-O0	16670	10715636	6,66039E+13	6,66039E+13
	-O3	950	5816066	3138178705	3132362639
Dijkstra	-O0	209	190375882	4814878170	4624502288
	-O3	145	31839495	2181701037	2149861542

Table 3 – Absolute values from analysis

Over the simulated data, displayed graphically in Figures 4-1 and 4-2, was taken more accurate information to be compared with the static analysis. The graphic in Figure 4.1 shows the distribution in the simulation results, where the X axis is represented by the number of vertex in the input graph and the Y axis the source node of the Dijkstra's shortest path. Following the same idea of showing the number of simulated cycles by the algorithms inputs, the axes X and Y in Figure 4.2 represents N (number of 0's) and M (number of 1's) respectively.

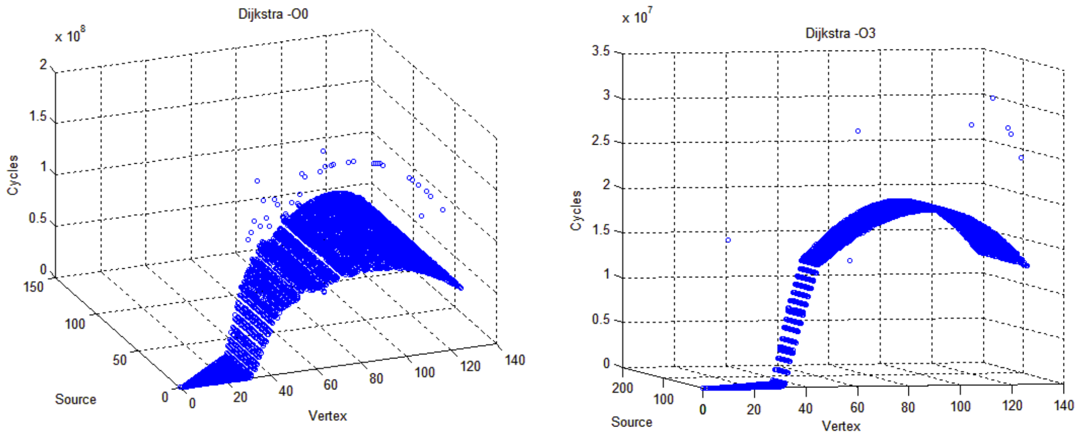


Figure 4.1 – Dijkstra's Simulation

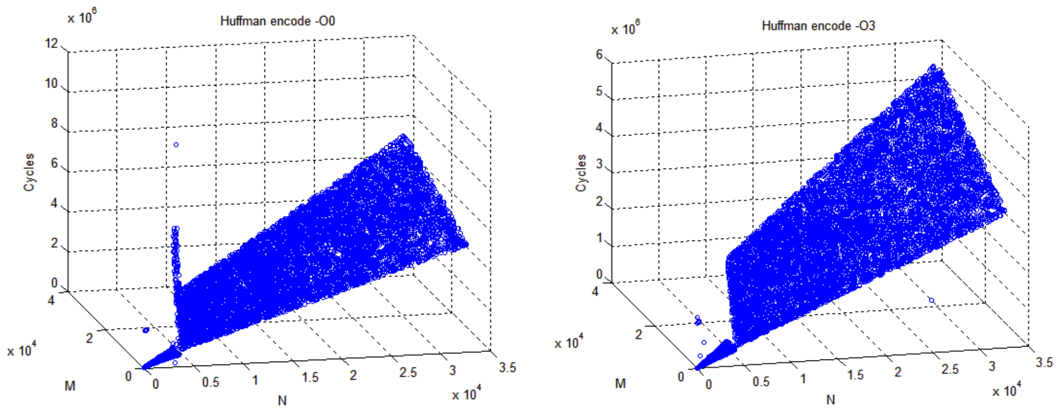


Figure 4.2 – Huffman Compress Simulation

From the dynamic analysis results was calculated the standard deviation of each algorithm and their optimization level in order to measure how many deviations the static analysis is from the longest simulated time. This difference, shown in Table 4, is also compared with how many times the static analysis is bigger than the dynamic. These evaluations allow some conclusions to be taken.

The measures shows that the static method clearly produced higher values than the dynamic, meaning that, it predicted a number of cycles for these algorithms bigger than the observed in simulations by Monte Carlo method. This observation shows that for a real execution of these algorithms over the Atmega128 microcontroller the execution time won't almost certainly exceed the worst execution time statically predicted.

		Dynamic	Static	static/dynamic(max)	Deviations
		max			from max
Huffman Compress	-O0	10715636	6,66039E+13	6215583,746	36008296,31
	-O3	5816066	3138178705	539,5706832	2229,93
Dijkstra	-O0	190375882	4814878170	25,29142935	125,21
	-O3	31839495	2181701037	68,52184801	272,15

Table 4 – Relation between results

5 CONCLUSION AND FUTURE WORK

This chapter explains the whole development of this work and the conclusions taken during the research. Also describes the challenges faced to active the goal and the next steps to improve it.

5.1 Conclusion

Timing verification in real-time systems algorithms is essential for a feasible parallelization between processes. This kind of verification aims calculation of the upper bound time from processes, called worst case execution time. This task can be performed mainly with two methods, dynamic and static. While the dynamic analysis involves timing accurate simulators or a platform, and a lot of effort running algorithms a bunch of times trying to get a good input and scenario to generate “worst cases”, a static analysis look for a accurate architecture/platform model to perform the measures over the code.

Following the static approach, the main problem in build the hardware models for the code evaluation is their features, like pipelines and cache, used to improve the hardware performance. They difficult the timing prediction, once there is a lot of variation on their execution. This way, the goals of the WCET static predictions are perform a safe and precise analysis, generating a result that could not be exceeded by a real execution and won’t be so long that the effort of the parallelization process would not have benefits.

This work follows a simple static approach applied over an architecture independent infrastructure, the LLVM project. Over this core, was possible build a modular WCET static analysis. The analysis performed showed that the results taken from the static analysis have a good margin of safety, however, based on the dynamic analysis results, not very precise. Once the LLVM compiler don’t create code directly to the used AVR architecture another compiler [26] was used, so the assembly executed in the simulation and the one analyzed are similar but different in structure and size.

5.2 Future Work

As a continuation of this work, solving the problem of the difference between the codes analyzed statically and the one executed in the simulation, would be develop a backend for the LLVM code generator. This way, the code generated for static analysis would be closer to the real executed code. However, if the target platform is one already supported by LLVM, this wouldn’t be a problem.

Another solution to reduce and to accurate the WCET prediction is applied more complex platform modeling to the analysis. Since the example platform was very simple, this wasn’t necessary, but for more complex and modern platforms, models with pipelines and cache simulation/analysis are also needed. According to the references works much of this kind of modern processors modeling is already being done. However, is missing integration between all the models and modeling process to develop more reliable and accurate WCET predictions. Also important to note that the improvements in this area should make WCET prediction each time more applicable and automatic in real-time systems projects context.

LIST OF ABBREVIATIONS AND ACRONYMS

WCET – Worst Case Execution Time

LLVM – Low Level Virtual Machine

IR – Intermediate Representation

CFG – Control Flow Graph

BB – Basic Block

FIGURE LIST

TABLE LIST

REFERENCES

[1] R. Wilhelm, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embedd Comput. Syst. 7,* 3, Article 36 April 2008.

[2] F. Mueller. Static Cache Simulation and its Applications. Doctor Dissertation, Florida State University of Tallassee, July 2004.

[3] S. Thesing. Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models. Doctor Dissertation, Universität des Sarrlandes, July 2004.

[4] M. Schlickling, M. Pister. Semi-Automatic Derivation of Timing Models for WCET Analysis, LCTES'10, April 2010.

[5] G. Bernat, A. Colin, S. Petters. pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. Technical Report YCS-2003-353, University of York, January 2003.

[6] I. Wenzel, R. Kirner, B. Rieder, P. Puschner, "Measurement-Based Worst-Case Execution Time Analysis," seus, pp.7-10, *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems* (SEUS'05), 2005.

[7] Y. Zhou, L. R. Welch, E. Huh, C. Alexander, D. Lawrence, S. Mehta, C. Cavanaugh. Important considerations for execution time analysis of dynamic, periodic processes. *Parallel and Distributed Processing Symposium., Proceedings 15th International*, vol., no., pp.1024-1031, April 2001.

[8] J. Hansen, S. Hissam, G. A. Moreno. Statistical-Based WCET Estimation and Validation. *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, November 2009.

[9] M. Langenbach, S. Thesing, R. Heckmann. Pipeline Modeling for Timing Analysis. *Lecture Notes In Computer Science; Vol. 2477, Proceedings of the 9th International Symposium on Static Analysis*, pp. 294 – 309, 2002.

[10] Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu. Performance Comparison of Techniques on Static Path Analysis of WCET. *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing.* EUC '08, pp. 104 – 111, 2008.

[11] P. Pushner, Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, Volume 1, Number 2, pp. 159-176, September 1989.

[12] Y.-T.S. Li, S. Malik, A. Wolfe. "Efficient microarchitecture modeling and path analysis for real-time software," *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE* , vol., no., pp.298-307, 5-7 Dec 1995.

[13] http://www.absint.com/ait/, *AbsInt Home Page.* Last accessed 14.05.2011

[14] http://llvm.org/, *LLVM Project Home Page.* Last accessed 14.05.2011

[15] http://compilers.cs.ucla.edu/avrora/, *Avrora Home Page.* Last accessed 17.05.2011

[16] http://www.java.com, *Java Home Page.* Last accessed 14.05.2011

[17] ATMEL, *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash Datasheet*

[18] C. Lattner, V. Adve. The LLVM Instruction Set and Compilation Strategy. Technical Report #UIUCDCS-R-2002-2292, Computer Science Dept., Univ. of Illinois, Aug. 2002.

[19] V. Adve, C. Lattner, M. Brukman, A. Shukla, B. Gaeke. "LLVA: A Low-level Virtual Instruction Set Architecture". *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36),* San Diego, California, Dec. 2003.

[20] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*. ACM, New York, NY, USA, 1-19. DOI=10.1145/800028.808479, 1970.

[21] Mingsong Lv, Zonghua gu, Nan Guan, Qingxu Deng, Ge Yu. “Performance Comparison of Techniques on Static Path Analysis of WCET*”, in Proc. of the 6th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (EUC 2008).

[22] P. Puschner, Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.* 1, 2, 159-176. DOI=10.1007/BF00571421, September 1989.

[23] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms (Second ed.), MIT Press and McGraw-Hill.*ISBN 0-262-03293-7, 2001.

[25] http://www.perl.org/, *Perl Programming Language Home Page.* Last accessed 14.05.2011

[26] http://winavr.sourceforge.net/, *Atmel AVR Compiler Home Page.* Last accessed 14.05.2011

ANNEXE A - DIJKSTRA’S ALGORITHM IN C

```
#include "input.h"
void findTheRotines(int source)
{   int i, j, k, my_vertex, ek;
    SHORTEST_PATH[source ] = 0;
    my_vertex = source;
    i=0;
    while( i++ < vertexNum )
    { //we will decide to which vertex is closer that is not in Y (not visited)
    for(j=0; j<vertexNum; j++)
    { //if we find a vertex that is not Y, lets work on it
    if( VISITED[j] == 0 )
    { //if there is a connection with the vertexes
    if( PATHS[my_vertex][j] != -1 )
    { //decide the shortest paths
    if( SHORTEST_PATH[j] > PATHS[my_vertex][j] + 
SHORTEST_PATH[my_vertex] )
    { //we found more shortest way.. change the old one
    SHORTEST_PATH[j] = PATHS[my_vertex][j] + 
SHORTEST_PATH[my_vertex];
    for(k=0; k<vertexNum; k++ )
    { ROUTE2[j][k] = ROUTE2[my_vertex][k]; }
    k=0;
    while(ROUTE2[j][k] != -1 )
    { k++;}
    ROUTE2[j][k] = my_vertex;
    }}}}
ek = MAX_INT;
for( j=1; j<vertexNum; j++)
{
if( VISITED[j] == 0 )
{
if( SHORTEST_PATH[j] < ek )
{
ek = SHORTEST_PATH[j];
my_vertex = j;
}}}
VISITED[ my_vertex ] = 1;
}
for( i=0; i<vertexNum; i++)
{
if( ROUTE2[i][0] != -1 || source == i)
{
k=0;
while(ROUTE2[i][k] != -1 )
{k++;}
ROUTE2[i][k] = i;
}}}
```

ANNEXE B - HUFFMAN COMPRESS ALGORITHM IN C

```
#include "input.h"
// maximum permitted run of zeroes.
#define RMAX 69
//maximum loop interaction
#define LOOPMAX 32766
// compressor (run length) for a sparse file.
// usage: RLencode <  filep.01.10000 > file.RLZ
// Uses Huffman codewords that were generated using huffman.p
//  http://www.inference.phy.cam.ac.uk/mackay/perl/huffman.p
// (c) Davi d J.C. MacKay
// License: GPL http://www.gnu.org/copyleft/gpl.html
// Originates from:
// http://www.inference.phy.cam.ac.uk/mackay/itprnn/code/c/compress/
void print_encoded(int);
void printfa(char*);
char* sconcat(char*,char*);
char* out;
int size = 0;
int main() {
    char *fp;
    fp = STRING_FILE;
    int r;
    //  unsigned char c ;
    int c, tot0 = 0, tot1 = 0;
    r = 0;
    //Iteraction limit
    unsigned int iteraction = 0;
    int i;
    for (i = 0; i < STRING_SIZE && iteraction < LOOPMAX; i++)
      {
            c = fp[i];
            if (c == '1') {
                  print_encoded(r);
                  r = 0;
                  tot1++;
            } else if (c == '0') {
                  r++;
                  tot0++;
                  if (r == RMAX) {
                        print_encoded(r);
                        r = 0;
                  }
            } else { // carriage returns}
            iteraction++;
    } //</I><32766>
    // clear the buffer of remaining stuff.
    if (r > 0) {
            r = RMAX;
            print_encoded(r);
    }
    return 1;
}
```

APPENDIX - INPUT SCRIPTS

Huffman Compress Input Script:

```
#!/usr/bin/perl -w
#
# returns a subset of M 1s among N-M zeroes
# usage:
#   randNchooseM.p N=10000 M=100
# MAX N = 32766

$max = 32766;
$N = int(rand($max));
$M= int(rand($N));

if($N<$M){
    $N = $N + $max/2;
}
$NTOTAL = $N;
$seed="123"; # new feature 99 07 30

print "N = $N, M = $M\n";

open TEMP,">>simulated_time.txt" or die("Cannot create file.");
print TEMP "N = $N, M = $M\n";
close (TEMP);

open FILE,">input.h" or die("Cannot create file.");
```

```
eval "\$$1=\$2" while @ARGV && $ARGV[0]=~ /^(\w+)=(.*)/ && shift;
srand($seed) ;

print FILE "#define STRING_FILE \"";

while($N ) {

   if ( ($M *1.0/$N) > rand() ) {
     print FILE "1" ;
     $M -- ;
   } else { print FILE "0" ; }
   $N -- ;
}

print FILE "\"\n#define STRING_SIZE $NTOTAL\n";

close (FILE);
```

Dijkstra Input Script:

```
#!/usr/bin/perl -w

$max = 127;
$vertnum = int(rand($max));
$source = int(rand($vertnum));
$seed="123"; # new feature 99 07 30

print "Vertex number: $vertnum, source: $source\n";
open FILE,">input.h" or die("Cannot create file.");
srand($seed) ;

#Define "#defines"
print FILE "#define MAX_INT 65535\n";
print FILE "#define MAX_ITERACTION $max\n";
print FILE "#define VERTNUM $vertnum\n";
```

```
#vertexnum and Source Node
print FILE "int vertexNum = VERTNUM;\n";
print FILE "int source = $source;\n";
print FILE "int SHORTEST_PATH[VERTNUM] = {";
for($i = 0; $i < $vertnum-1; ++$i)
{
    print FILE "MAX_INT,";
}
print FILE "MAX_INT};\n";
#PATH definition
$upperLimit = 90;
print FILE "int PATHS[VERTNUM][VERTNUM] = { ";
for($i = 0; $i < $vertnum-1; ++$i) {
    print FILE "{";
    for($j = 0; $j < $vertnum-1; ++$j) {
            if($i == $j){
                    print FILE "0" ;
            } else {
                    $rand = int(rand($upperLimit));
                    if ( $rand > 60 ) {
                            print FILE "-1" ;
                    } else {
                            print FILE "$rand";
                    }
            }
            print FILE ",";
    }
    $rand = int(rand($upperLimit));
    if ( $rand > 60 ) {
            print FILE "-1" ;
    } else {
            print FILE "$rand";
    }
    print FILE "},";
}
print FILE "{";
for($j = 0; $j < $vertnum-1; ++$j) {
    $rand = int(rand($upperLimit));
    if ( $rand > 60 ) {
            print FILE "-1" ;
    } else {
            print FILE "$rand";
    }
    print FILE ",";
}
print FILE "0}};\n";
#ROUTE2 definition
print FILE "int ROUTE2[VERTNUM][VERTNUM] = { ";
for($i = 0; $i < $vertnum-1; ++$i) {

    print FILE "{";
    for($j = 0; $j < $vertnum-1; ++$j) {
            print FILE "-1,";
    }
    print FILE "-1},";
}
print FILE "{";
for($j = 0; $j < $vertnum-1; ++$j) {
    print FILE "-1,";
}
print FILE "-1}};\n";
#VISITED definition
print FILE "int VISITED[VERTNUM] = {";
for($i = 0; $i < $vertnum-1; ++$i)
{
    print FILE "0,";
}
print FILE "0};\n";
close (FILE);
```