Faculdade de Engenharia da Universidade do Porto

FEUP

Methods for Dynamic Identification of Program Control-Flow Structures for FPGA-based Systems

Tiago José Rocha Alves da Costa

Thesis submitted for the degree of
Master in Electrical and Computers Engineering
Major in Telecommunications

Supervisor: Prof. Dr. João Manuel Paiva Cardoso
Co-supervisor: Prof. Dr. João Paulo de Castro Canas Ferreira

June, 2009

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2008/2009

A Dissertação intitulada

"METHODS FOR DYNAMIC IDENTIFICATION OF PROGRAM CONTROL-FLOW STRUCTURES FOR FPGA-BASED SYSTEMS"

foi aprovada em provas realizadas em 23/Julho/2009

o júri

Presidente Professor Doutor José Alfredo Ribeiro da Silva Matos
Professor Catedrático do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Mário Pereira Véstias
Professor Coordenador do Departamento de Engenharia Electrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa

Professor Doutor João Manuel Paiva Cardoso
Professor Associado do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

Professor Doutor João Paulo de Castro Canas Ferreira
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

Autor - TIAGO JOSÉ ROCHA ALVES DA COSTA

Faculdade de Engenharia da Universidade do Porto

Resumo

Actualmente, devido aos avanços contínuos no campo de sistemas embutidos um pequeno chip tem de ter uma alta velocidade de processamento e um consumo de potência tão baixo quanto possível.

Uma das maneiras de atingir estes objectivos é através do uso de aceleradores de hardware reconfiguráveis como uma extensão de um microprocessador. Dada a necessidade de assegurar portabilidade a compilação que estas arquitecturas permitem têm de ser efectuadas de modo dinâmico (quando o programa está a correr). O primeiro passo nesta aproximação é a detecção de quais partes do código do programa em execução é que serão os melhores candidatos para reprogramar o acelerador de hardware reconfigurável (zona crítica do código). Esta tese apresenta um método de fazer *profiling* do programa em execução capaz de identificar ciclos, já que tipicamente são as estruturas mais comuns nas zonas críticas da maior parte dos programas. Este *profiler* para além de identificar os ciclos, também recolhe variadas informações sobre eles, como o número de execuções, iterações e quão recentemente o ciclo foi executado, de modo a que seja possível classificar a importância de cada um dos ciclos detectados.

Resultados experimentais, obtidos para quatro *benchmarks* indicam que um esquema de *profiling* com um bom grau de simplicidade permite uma precisão elevada na detecção de ciclos.

Abstract

Continuous advances made in the field of embedded systems have lead to a time where a small chip has to be capable of high-speed processing and saving as much energy as possible.

A way of achieving these objectives is by extending microprocessors with reconfigurable hardware accelerators. However, to maintain portability and code legacy, the compilation for such architecture needs to be performed in runtime (dynamically). The first step of this runtime approach is the detection of which parts of code ("critical regions") to map to the reconfigurable hardware accelerator. This thesis introduces a runtime profiling scheme able to identify loops, the main computing construct in most critical regions of programs. The profiler not only identifies the loops, but also gathers information, such as number of executions, number of iterations, and when the loop was executed for the last time, allowing for a possible rating of the importance of the loop in the program.

Experimental results using four benchmarks indicate that a simple profiling scheme allows high accuracy as far as loop identification is concerned.

Acknowledgments

I would like to thank my supervisor, Prof Dr. João Manuel Paiva Cardoso for the assistance and support throughout the development of this thesis. Thank you for the documentation provided, ideas on how to implement in, and suggestions on the best way to perform some tasks.

I would also like to express my special gratitude to my parents and girlfriend for their constant concern, support, motivation, attempts to help and patience.

Finally, I would like to thank my friends for all the great moments we spent together, which also contributed to the achievement of my goals. I would particularly like to thank Vergilio Loureiro for the assistance with the software, Miguel Campos for helping with the webpage and Eduardo Sousa, Tiago Costa, Sergio Sà for the brainstorming that played a heavy part in the development of this thesis.

The Author

X

Index

Figure list

Table list

Acronyms

SOC	System on Chip
RC	Resistor Capacitor
FPGA	Field Programmable Gate Array
XOR	Exclusive OR Gate
NAND	Not And Gate
HDL	Hardware Description Language
ASIC	Application Specific Integrated Circuit
DSP	Digital Signal Processing
SRAM	Static Random Access Memory
CLB	Configurable Logic Block
IOB	Input Output block
BRAM	Block Random Access Memory
I/O	Input /Output
LUT	Look-up Table
GRM	General Routing Matrix
RISC	Reduced Instruction Set Architecture
PLB	Processor Local Bus
GPIO	Local Memory Bus
IP	Intellectual Property
P2P	Peer to Peer
LED	Light Emitting Diode
DIP	Dual In-line Packaging
UART	Universal Asynchronous Receiver/Transmitter
FSL	Fast Simplex Link
EDK	Embedded Development Kit
OPB	On-chip Peripheral Bus
sbb	small backward branch

FIFO	First in First Out
LSB	Least Significant Bits
MSB	Most Significant Bits
TP	Target Pattern
TPM	Target Pattern in Memory
CM	Count for each Target Pattern
$ADDR_n$	Address value in Memory position n
ADDR+1	Address value in Memory position n+1
-O2	Level 2 Optimization
mb-gcc	Microblaze Gnu Compiler Collection
ID	Identification

Chapter 1

Introduction

Miniaturization, low energy consumption and high performance are no longer mere features. These are now basic requirements in nearly any embedded system. The main problem now is how to combine these three requirements. There are a multitude of possibilities that explore the best options using the trade off involved. For instance a higher processing power will mean higher power consumption and a larger packaging.

The first step in optimizing a system is finding out where and what to optimize. Using software as an example, there is little advantage in optimizing a part of a program that is seldom used. Most software applications tend to follow the 90 – 10 rule of thumb. This rule states that 90% of an application running time is spent executing the same 10% of code [2]. This 10% of code which is being constantly used is referred as critical region.

These regions can be identified by profiling. This can be done by the designer during debugging (static profiling), or can be done during runtime by an external application (dynamic profiling). For many applications, static profiling is sufficient, however it has the disadvantage of being developed for a specific application, whereas a dynamic profiler can be used for a multitude of applications.

As previously stated, the first step in optimizing a system is finding out what has to be optimized. The purpose of this project is not to develop a method for optimizing a system, but to develop a method that can identify the areas of code with potential for optimization. Typically a critical region contains several loops, other structures like if-then or if-then-else, function calls. In many cases the critical region is a recursive loop.

There are several approaches to consider when analysing a system. A detection method can be dynamic or static:

- Static: A static optimization is performed at design time by the designer. It focuses a system for a specific application (or suite of applications). It can be

used, to monitor a data bus or a cache miss rate, among others. Despite its obvious lack of adaptability to different applications, it is effective in those cases, where only a pre-determined section is required to be optimal.

- Dynamic: The software is monitored in real time, in order to check which parts of the code run more often. When using this approach, there is no need of intervention from the application designer or to use simulations (which are usually costly). It is also transparent, causing no disruption in the standard software flow. A good application for this method can be for a programmable microprocessor. This kind of device is re-programmed constantly, which means that the critical regions are constantly changing, making this a good candidate for dynamic profiling;

The previous methods (dynamic and static) refer to how a profiler can probe a program in order to locate a specific pattern (such as the critical kernel). That probing can be made by two processes:

- Intrusive: as its name suggests, this method interferes with the normal operation of the system. It means the application will have to stop its regular operating procedure to communicate information (such as how many times a given loop has been executed). This method can be used in desktops, or other hardware that does not have tight timing constraints. As with the static approach, this method can be helpful at the design stage, allowing the design team to tweak the critical sections of the code.

- Non-intrusive: a non intrusive approach means that the actual running of the program suffers no interference. It can be compared to using a multimeter to measure the voltage on a RC circuit. The detection is adequate to real-time systems, and it has a low effect on current tool chains. This approach allows monitoring of a SoC with no runtime overhead.

Static methods are quite efficient and easier to implement than the dynamic ones, but they come with a price. As far as desktops or Virtual Machines go, there is not much of a problem, however when attempting to apply the same method to an embedded system, critical failures are likely to occur due to runtime overhead. Most embedded systems are designed to occupy the least possible physical area, and very close tolerances as far as timing is concerned. Implementing extra software in order optimize or detect a specific critical

section leads not only to a larger area (and by consequence, a bigger cost), but it might also affect how the system operates.

In order to be possible to detect critical regions in sensitive systems, or systems that undergo constant reprogramming (such as most SoC) a non-intrusive approach is needed.

As for choosing between dynamic or static approach, it depends a lot on the target system. A profiler can be designed specifically for a system (static) however if the goal is to develop a profiler that possesses portability (can profile multiple systems) the dynamic approach is required.

Once a critical area has been outlined and deemed suitable for optimization, binary translation techniques [15] can be used to enhance the system performance. One well documented method is referred to as Warp Processing [16]. This method uses the information gathered by the profiler to identify the best areas of code to optimize, de-compiles the code and programs an FPGA to assist in the processing.

1.1. Objectives

This dissertation's objective is to specify and develop a profiler that can analyse a program's instruction flow and determine its critical region, so that it can be optimized (if possible). This profiler is to be connected directly to a micro-processor instruction bus which will require study of the micro-processor's inner working to allow a better integration. A study was conducted on existing profilers, in order to gain new insight on current methods. Microblaze, a soft-core processor was selected as the target micro-processor because of its integration with FPGA's and because it allows the profiler to be directly connected to its instruction bus which would be more difficult if using a hard processor as target platform.

The context of the proposed work has already been outlined in the previous section but suffice it to say that efficiency is of paramount importance in embedded systems, particularly when an existing system can be optimized instead of updated, allowing for a better performance without added cost.

To achieve the proposed objectives, the first step is to be able to identify the critical region. Typically this region is made of loops. They might be recursive, or not, however, the best candidates for optimization are long loops (loops that contain many instructions), loops with many iterations and executions (a loop iterates n times per execution). A loop, per definition is something that its end meets its starting point. Using this as a starting point, it becomes clear that when a loop occurs the program flow has jumped backwards. Once a backward jump is detected, information must be gathered. There are two parameters that can identify a loop: the instruction address, or the instruction itself. Once a method to properly identify loops has been established, it becomes a matter of how to store and catalogue data, as well as what kind of data should be stored to best define a loop. The final

parameter should be how many loops should be stored. It would obviously be best to store all loops; however this would probably require too many resources. A Replacement policy is then in order, so that for a limited amount of storing space, only the best candidates for optimization are stored.

These are the objectives for this thesis.

1.2. Document structure

This document is organized according with the following structure. The current chapter is an introduction to this dissertation. Here the context in which this work is inserted is described along with the proposed objectives. In chapter 2, all technologies that are relevant for the development and contextualization of this thesis are explained and analyzed.

Chapter 3 completes the information in Chapter 2 by providing a description of the State-of-the-Art of the most relevant technologies. While Chapter 2 describes technologies from a hardware point of view, Chapter 3 introduces some practical applications of the concepts behind this thesis. Chapter 4 describes the work that was done, the choices made and the used methodology. In chapter 5 an exposé and analysis of the obtained results is provided, leading to the next practical implementation, presented in chapter 6. Chapter 7 contains the conclusions and the achieved objectives along with references to future work that can be done.

Chapter 2

Relevant Technologies

This chapter introduces some of the relevant technologies for this thesis, the Xilinx Virtex II Pro FPGA [4] and the Microblaze [7].

2.1. Field Programmable Gate Array [4]

The Field Programmable Gate Array, also known as FPGA is a semiconductor based device.

It is made out of a number of "logic blocks" that, through a hierarchy of re-configurable interconnects, can be linked together. They can be configured to be simple logic gates (such as AND, XOR, NOT, NAND, etc) or into extremely complex combinational functions. Most also include memory blocks, that can be simple flip-flops, or even complete blocks of memory.
Its main operative advantage is that it can be reconfigured by a user at any time.

An FPGA can be programmed using a logic circuit diagram, or through application specific HDL code. FPGA's have the advantage of being capable of an application specific implementation, with costs lower than ASIC solutions, and of being updated should the need arise (even after shipping by the manufacturer)

FPGA's can be used in a multitude of environments and for completely different purposes. DSP applications, ASIC prototyping, medical imaging systems, computer vision, are only a few examples.

As an example we describe here the Xilinx Virtex II Pro [4].

2.1.1. FPGA Architecture

The user defined logic is configured by values stored in SRAM cells. Figure 2-1 represents the user-programmable gate array. In this gate array two major configurable elements have to be taken into account: The CLB's and the input/output blocks (IOB's). The CLB's provide the functional elements for constructing the logic (which is in SRAM-stored truth tables) and the IOB's interface between the package pins and the CLB's (see Figure 2-3).

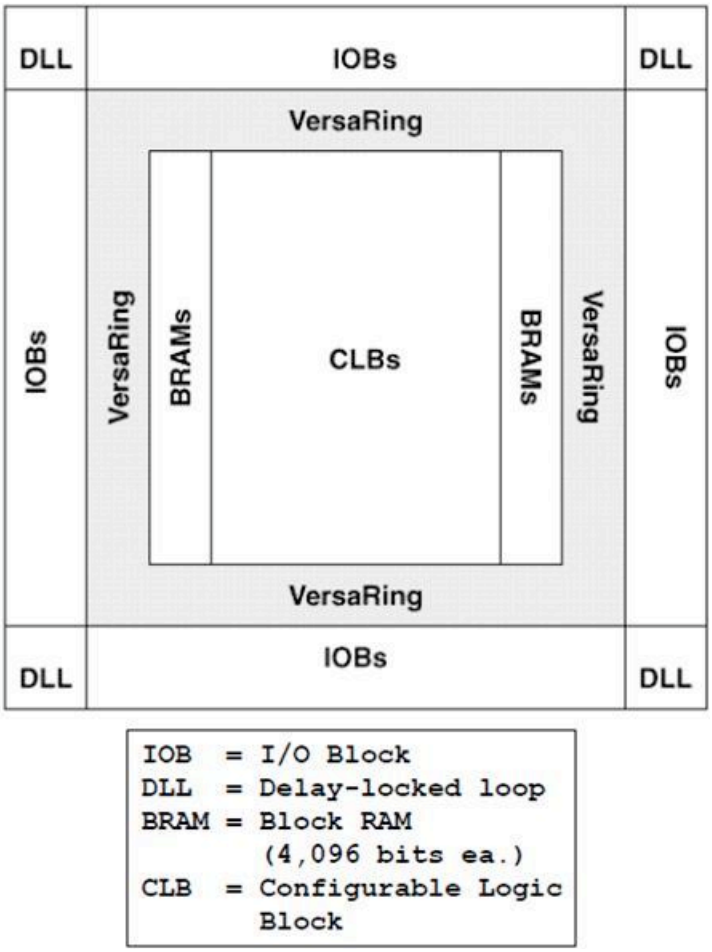

Figure 2-1 - Virtex Architecture Overview [4]

The FPGA architecture consists of several arrays of (CLBs), I/O pads, and routing channels. The routing channels have the same width. Multiple I/O pads may fit into the height of one row or the width of one column in the array.

The 2 Slice Virtex CLB depicted in Figure 2-2 contains two logic cells, and each logic cell contains:

- 2 Look-up tables (LUTs)
- 2 D flip-flops/latches
- Fast carry logic
- Three-state drivers
- SRAM control logic

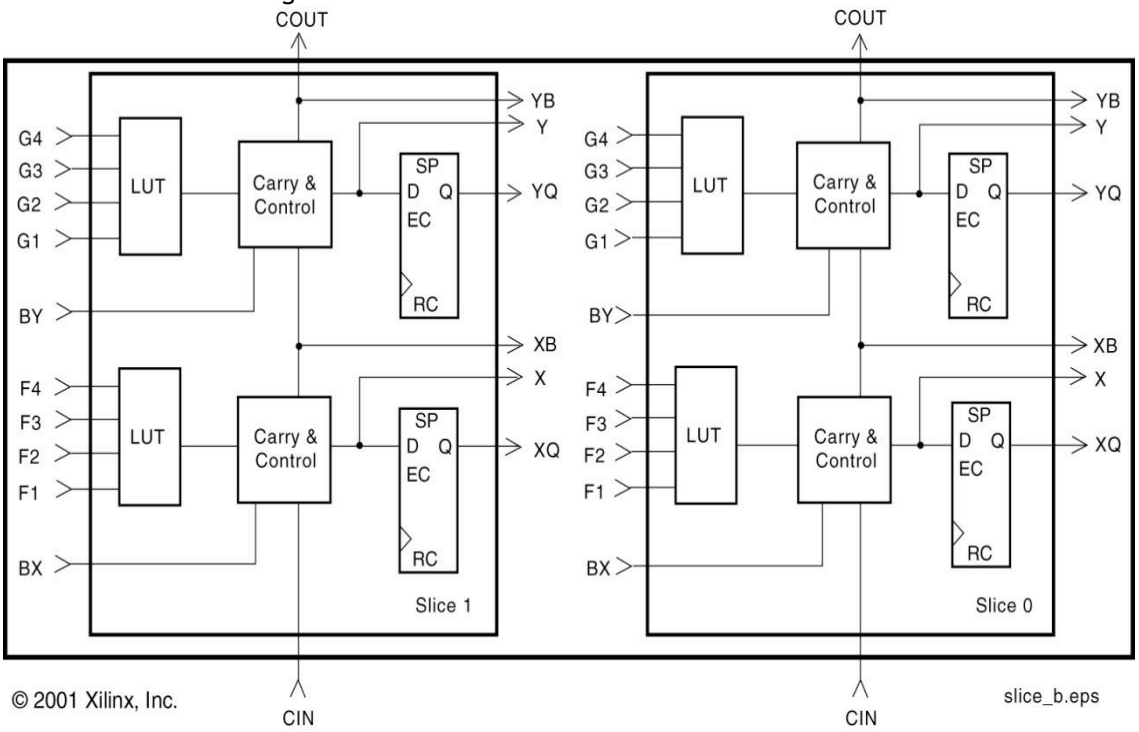

Figure 2-2 - 2-Slice Virtex CLB [4]

These slices can be programmed to perform any number of logic functions. For example, the LUTS can implement logic gates and the Flip-Flops can hold the resulting values if needed.

- Output Features
  - Optional data output D flip-flop with clock enable and shared asynchronous Set/Reset
  - Optional 3-state control D flip-flop with clock enable and shared asynchronous Set/Reset
  - Three-state output buffer

- Input Features
  - Input Buffer

- Programmable delay for forcing pad-to-pad hold time to zero
- Optional data D flip-flop with clock enable and shared Set/Reset
- Optional pull-up and pull-down resistors (Same ones as for output use)

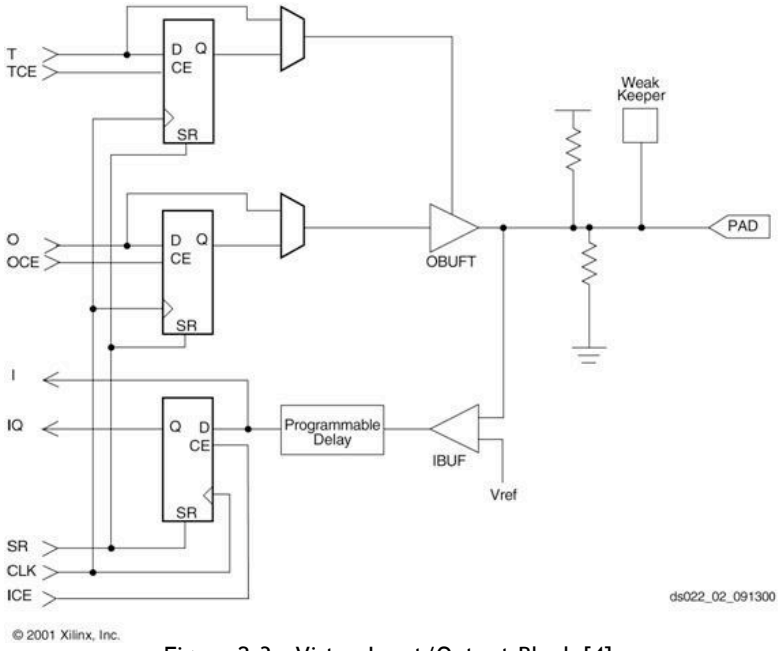

Figure 2-3 - Virtex Input/Output Block [4]

- Local Routing (Figure 2-4)
  - Interconnections among LUTs, flip-flops, and General Routing Matrix (GRM)
  - Internal CLB feedback paths that can chain LUTs together
  - Direct paths between horizontally-adjacent CLBs
  - Short connections with few “pass” transistors => low delay => high-speed connections

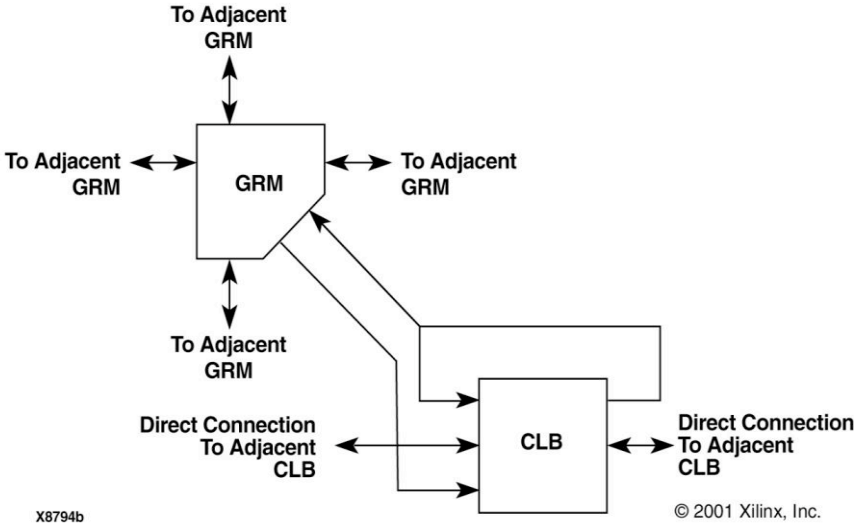

Figure 2-4 - Virtex Local Routing [4]

- Global Routing
    - Distribute Clocks and other signals with high fanout
    - Primary Global Routing
    - Four dedicated global nets with dedicated input pins for clocks
    - Driven by global buffers
- Secondary Global Routing
    - 24 backbone lines, 12 across top of chip and 12 across bottom of chip. From these, can distribute 12 unique signals/column via 12 longlines in column
    - Not restricted to routing only clock pins

Summarizing, the use of FPGA has multiple advantages from a hardware standpoint. One of the most significant ones is the fact that FPGA's are an ideal platform for soft-core processors, like the Microblaze.

The Microblaze will be introduced in the next section along with the reason of it being advantageous. This issue will be address more thoroughly ahead.

2.2. Microblaze™ based embedded systems

The MicroBlaze™ core is a 32-bit RISC Harvard architecture soft processor core with a rich instruction set optimized for embedded applications.

Typically the MicroBlaze instruction Set allows it to achieve single-cycle execution throughput. The Microblaze key advantage is its versatility. It can be configured by the user to best suit the system requirements as much as possible. Cache size, pipeline depth, memory management unit, peripherals and bus-interfaces can all be configured (or disabled). Hardware support for floating point operations is also included (it can be enabled or disabled according to the user will). This freedom allows for performance/cost relation that makes the MicroBlaze useful in industrial controllers, consumer applications, data processing and communications.

The Microblaze has several I/O buses. The Processor Local Bus (or the PLB) represented by the orange line (Figure 2-5) links the microprocessor to external memory, GPIO's, and to any other module (or IP core).

The Local Memory Bus (also known as LMB) is the blue line, and connects the microprocessor to the internal block ram, where Data and instructions are (usually) located.

Should the program be too large to be accommodated on this internal memory, then it can be stored on the external SRAM. The magenta line represents a dedicated bus, called Xilinx P2P. Its use is to connect the cache to the microprocessor.

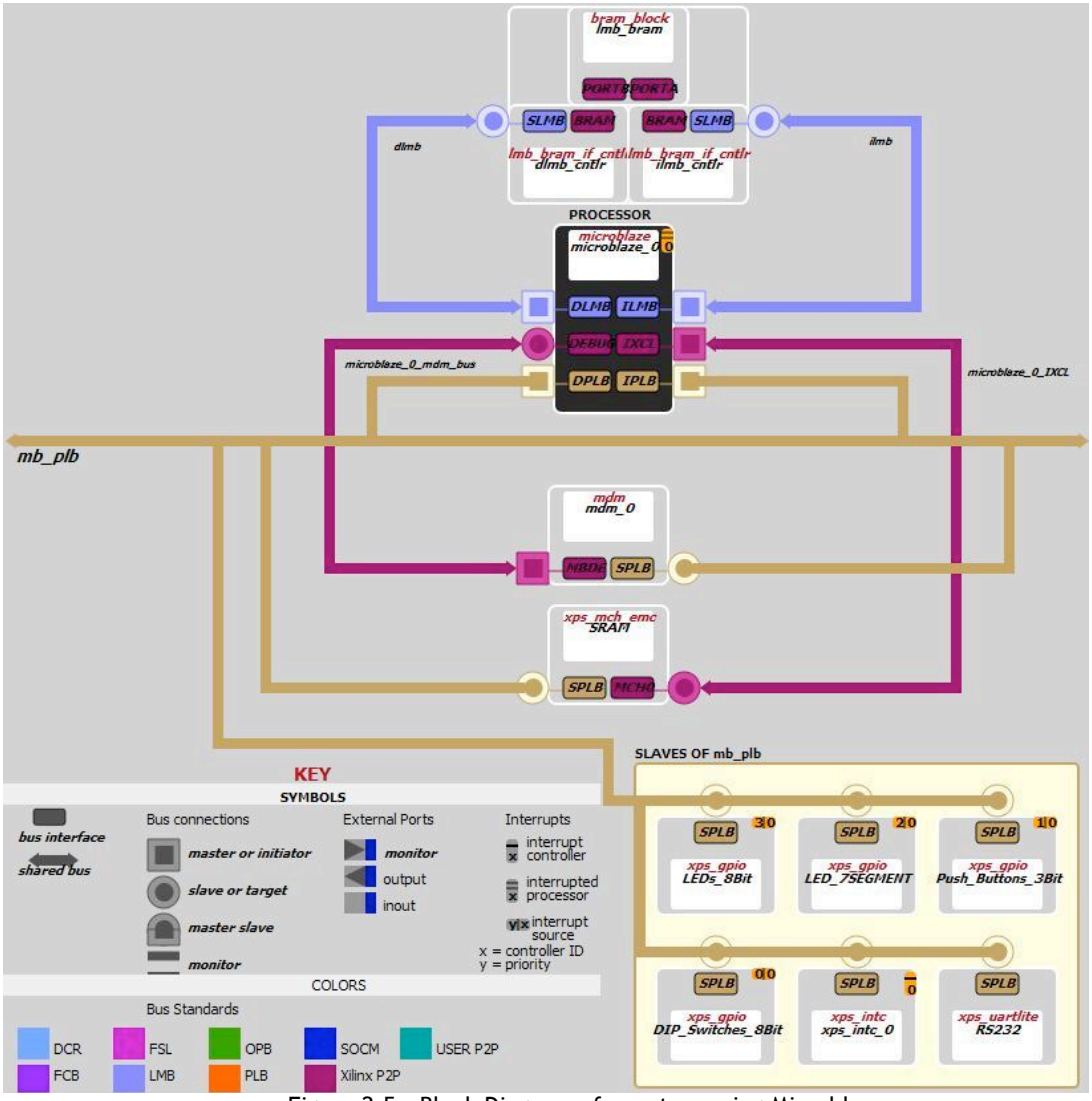

Figure 2-5 - Block Diagram of a system using Microblaze

On this generic system, several general purpose input output (GPIO) peripherals are present: a 7 segment LED matrix, 8bit LED's, DIP switches, Push buttons, and a UART interface. Several more peripherals can be hooked up to the PLB or to the FSL and interact with the microprocessor.

Xilinx's Embedded Design Environment, EDK [14] allows a designer to specify the system specifications and to program the Microblaze using C language. The tools made available are able to generate the configuration that program the FPGA to implement the system. Figure 2-6 shows a small extract of C code containing the sum of all elements within an array, and its equivalent in Microblaze assembly code.

C code

```
int j =0, sum=0;

for(j = 0; j<11;j++)
{

sum = sum + vector[j];
}

return soma;
}
```

Assembly code

```
$L3:
$LM14:
    lwi r4,r19,8
    addik   r3,r19,4
    addk    r3,r3,r4
    lbui    r3,r3,8
    sext8   r3,r3
    addk    r4,r3,r0
    lwi r3,r19,4
    addk    r3,r3,r4
    swi r3,r19,4
$LM15:
    lwi r3,r19,8
    addik   r3,r3,1
    swi r3,r19,8
$L2:
    lwi r3,r19,8
    addik   r18,r0,10
    cmp r18,r3,r18
    bgei    r18,$L3
```

Figure 2-6 Microblaze assembly code

By analysing the code it becomes apparent how the loops are implemented in Microblaze. After the instructions contained inside the loop are executed (in this case, the sum), a comparison is made between two registers (r18 and r3) where one of the registers contains the number of current iterations and the other contains the value of the maximum number of iterations. Depending on the result of the comparison (which is stored in r18) the program will either branch to the begging of the loop again or moves to the next instruction in the next line.

2.2.1. Possible monitor locations

EDK [14] allows the insertion of custom IP's. A custom IP is a user developed peripheral for the Microblaze that can range from a new set of LED's to a sensor interface, communications, etc. By creating a custom IP, a user has the possibility to connect it directly to the PLB or the OPB. If it is necessary to connect said peripheral to another bus (like the LMB) it cannot be done using the EDK [14] tools. However, it can still be done by directly editing the Microblaze Hardware Specification file.

Chapter 3

State of the art

The previous chapter presented and described technologies relevant for this. This chapter presents the State of the Art in the area of hardware profiling, so that the reader may get familiar with some of the solutions that these technologies allow developers to achieve and also to justify some of the methodologies and paths used in the development of the present work.

There are many techniques to discover the critical region of a program, though not all fit the characteristics relevant for this thesis. The Dynamo [8] greatly increases performance while reducing power however this is achieved by software optimization (intrusive method) and is out of this work's scope. Other optimization methods resort to instruction compression [9] and [10], or to storing some instructions in a low-power cache [11] and [12]. All these methods have their advantages and disadvantages, but the objective of this thesis is a non-intrusive dynamic application, and the previously mentioned work does not fit the profile.

3.1. DYNAMIC APPLICATION PROFILER (DAProf) [1]

The Dynamic Application Profiler (DAProf) [1] is a hardware module to allow non-intrusive dynamic profiling. This module is connected to the microprocessor base system as illustrated in Figure 3-1.

The idea behind DAProf is loop identification. It allows the dynamic identification of different loops and their parameters, like its size (or number of instructions), number of iterations and even how recently has the loop been executed (when compared with the other loops being monitored). This information is used to determine the critical section(s) of the program currently being executed by the microprocessor.

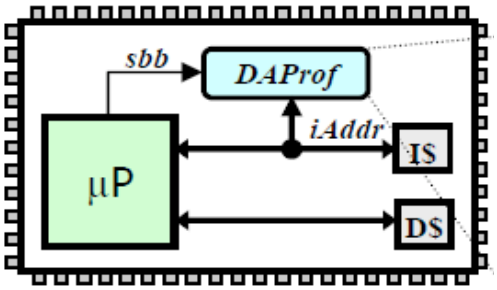

Figure 3-1 – The DAProf profiler, integrated within a microprocessor [1]

It monitors the instruction bus of the microprocessor in order to determine the address of the instruction that is currently being executed every time a short backward branch is detected.

This profiler considers a small backward branch as a branch instruction whose target address is a negative offset of 1024 or less.

Given the size of each instruction, this offset means that only loops containing less than 256 instructions will be considered.

Even though DAProf could decode the contents of the instruction bus directly, it is assumed that the microprocessor has an output signal named **sbb**, which indicates that a Small Backward Branch has been executed. This would require only minor modifications on a microprocessor decoding logic, or the use of a microprocessor that already has this output.

The DAProf is composed by a profiler FIFO, a profile cache and profiler controller (Figure 3-2).These components have the following functions:

The FIFO is responsible for synchronizing the microprocessor with the profiler.

The Cache is used for storing profile statistics (for the loops currently under analysis)

The Controller analyzes the short backward branches and updates the cache when needed.

More detailed explanations on the purpose and functionality of each of the previously mentioned components are available in the next sections.

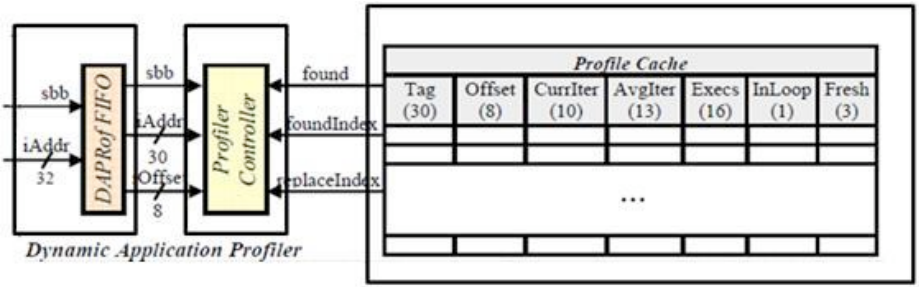

Figure 3-2 - DAProf, general overview [1]

3.1.1. The Profiler FIFO

As can be observed in Figure 3-3, the FIFO is composed by 2 inputs and 3 outputs. Its function is to monitor both the instruction bus and the **sbb** signal of the microprocessor. Every time the **sbb** signal is activated (which means a small backward branch has occurred), the branch instruction address and its offset are stored in the FIFO.

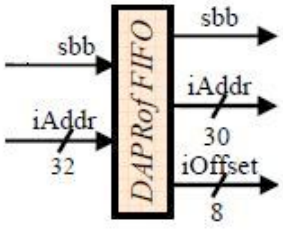

Figure 3-3 - DAProf, Profiler FIFO [1]

The offset is the number of instructions inside the loop currently being profiled, and together with the branch instruction address allow the profiler to determine the beginning and ending of the loop.

The FIFO is also responsible for synchronizing the microprocessor with the internal components of the profiler. The relevance of this action is that the microprocessor may have a higher clock frequency than the DAProf. This does not affect the effectiveness of the profiler because backward branches do not occur on every clock cycle, allowing the DAProf to have a smaller clock frequency and still have the ability to correctly identify loops. Since any relevant loop will have at least two instructions plus the short backward branch, this allows for a possible clock frequency of one third of the microprocessor frequency. This does not take into account that the profiler FIFO needs a large enough size in order to allow for bursts of short backward branches, which may eventually occur (even periodically). For the applications used in the experiments, it was determined that a four entry FIFO is sufficient to maintain a clock frequency rate of one third.

3.1.2. The profiler Cache

The profile Cache (see Figure 3-4) is a small cache that stores the profiling results, as well as extra information required by the controller to identify loops, profile statistics, and loop execution monitoring. It also identifies which entry is to be replaced in case a new loop is executed. The current cache has a 32 entry profile cache with a width of 81 bits.

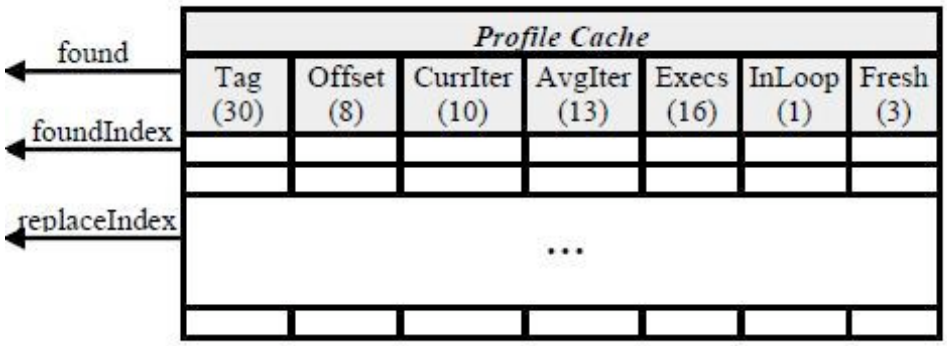

Figure 3-4 - DAProf - Profiler Cache [1]

A description of each of the cache's fields follows.

- Tag and Offset (Loop identification):

The Identification of the profiled loops is done within the profile cache by using the address of the loop's short backward branch (used as the Tag entry) and the loop's offset (determined by the FIFO).

Since in the case of a 32-bit ARM processor and byte addressable memory, the 2 LSB for all instructions are always the same, they can be ignored. So, The Tag entry is the 30 MSB of the loop's branch address.

The offset is an 8 bit entry, and as said previously, refers to the number of instructions contained in that loop These two fields are obtained via the FIFO and are used to determine the boundaries of the loop.

- Loop Executions

This field stores the number of times a loop has been executed during the execution of the application. Since the purpose of DAProf is to monitor a program for an extended period of time, regardless of the number of bits chosen for this entry, It will eventually become saturated. By using a 16-bit entry, DAProf can profile 65536 loop executions before it reaches its maximum capacity.

- Current Iterations

This field stores the number of times a loop has iterated for the current loop execution, as a 10-bit entry, allowing DAProf to profile loops up to 1024 iterations per execution.

- Average Iterations

This entry stores the average number of times a loop iterates per loop execution. Taking into account that loops usually don't iterate a fixed number of times per execution, this field cannot be an integer. The Average Iterations are stored as a 13-bit fixed point number, where 10 bits represent the integer part and 3 bits the fractional part.

- Loop Execution Monitoring

This entry is a flag. A 1-bit value that indicates that the loop is being executed and makes the distinction between a new loop execution and an iteration on the current execution.

- Freshness

To prevent a recently added loop from being almost immediately discarded, the cache includes a 3-bit freshness value to indicate how recently a loop has been executed (or iterated). A large freshness means that a loop that has just been executed. Only a non-fresh loop (zero freshness) can be replaced.

3.1.3. The Profiler Controller

The profiler controller provides interface between the FIFO and the cache. Its inputs (see Figure 3-5) are the short backwards branch address (*iAddr*) and offset (*iOffset*) values from the FIFO, as well as the control signals (*found*, *foundIndex*, *replaceIndex*) from the profiler cache.

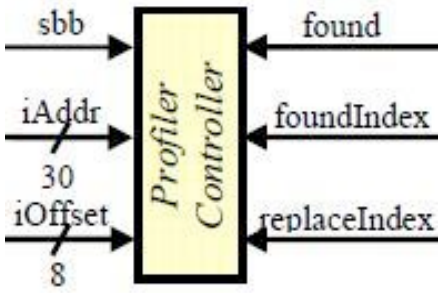

Figure 3-5 – DAProf Controller [1]

The signals from the cache are responsible for communicating to the controller if the current short backward branch exists in the cache (*found*), where it is (*foundIndex*), and in case the short backward branch does not exist, the location (index) for the loop entry that it will replace (*replaceIndex*).

Whenever the FIFO signals a short backwards branch the controller attempts to locate the loop within the cache. If a cache hit occurs then the controller determines if the loop is currently executing (through the *InLoop* flag). If this is true, the loops current iterations are incremented. If not, and the loop is not currently being executed, this implies that this is a new loop execution. For this, the controller increments the loops executions, sets the *InLoop* flag, current iterations to one, decrements the freshness for all loops (except for the current one), and sets the freshness for the current loop to maximum.

If and when the profiler controller detects that the loop's executions are saturated, all loop executions are divided by two (the handling of execution saturations are dealt in the

same way than total iteration saturations). Unlike what occurs with the frequent loop detection profiler, saturations in DAProf only affect the accuracy of the loop executions. The precision of the average loop iterations per execution remains the same.

Besides ensuring that the executions for any loop will not become saturated, this scheme guaranties that older loops, which might have been considered critical, may no longer be executed. This mechanism assures the dynamism required in DAProf. For example, a previously executed loop with high executions will not be replaced during the profiling operation, however, after a few saturations, the total iterations profiled might be less (possibly) than other loops. This assures that if a loop stops being executed, its data will eventually be removed from the profiler.

In case a loop's short backwards branch is not stored in the cache (cache miss), the profile controller will replace the entry pointed by *replaceIndex*. The initialization is done by setting the *TAG* and *Offset* to the values of the recently profiled loop, executions are set to one along with the *Inloop* flag and current iterations. Freshness is decremented for all other loops and the new one's freshness is set to maximum.

For all short backwards branches, the profiler controller checks all entries of the profile cache whose *InLoop* flag is set to determine if the application is still executing within those loops. If a loop is no longer being executed, the profile controller resets the *InLoop* flag and updates the loop's average iterations.

3.1.4. Results

After testing, it was proved that the dynamic application profiler (DAProf) [1] is capable of providing a highly efficient, non-intrusive and accurate profile of an application's execution.

With a 11% area increase, the profiling of the running application of a 495MHz processor yields a 90% accuracy for average iterations and 97% for loop executions.

However, given the multitask and multithread component in most of today's applications, a future research effort will have to include a minimally intrusive, task-aware profiling method to allow DAProf to achieve the same level of accuracy with multitasked and multi-threaded applications, as it has with single-task and single-thread programs.

3.2. Fast On-Chip Profiler Memory [3]

A hardware device used to profile an executing application in a non-intrusive way is not a simple design. Even the simple part of updating profile counts is difficult due to speed issues. Even the existing fast access memory designs, including advanced content-addressable-memories, have several limitations.

The typical hardware profiler counts only simple and easy to detect events (like a cache hit rate), or in some cases a particular pattern (like a specific address on a bus).

In order to address these issues, a profiler, named “Fast On-Chip Profiler Memory”, is proposed in [3]. This Profiler required the development of a memory architecture designed specifically for profiling. Next subsections describe this profiler in some detail.

Summarizing, this Profiler is a memory with some extra built in logic, linked to a data or instruction bus, and designed specifically to identify, store and count certain pre-programmed patterns.

3.2.1. Problem definition

The profiler memory is capable of counting hundreds of thousands of bus patterns simultaneously, unlike most profiling hardware. Even so, the bus-memory interface is extremely simple, size efficient and can be made of as little as a collection of standard register files or memories embedded with a small amount of logic devices. With the use of novel pipelined binary tree architecture, the profiler-memory is able to achieve single cycle throughput, which translates into complete accuracy, even if the bus being monitored is moving very fast sequences of patterns, like many devices capable of fetching one instruction per cycle.

The binary tree is implemented using a separate module for each of the tree levels, allowing for a more efficient scaling and increasing its efficiency along with its size.

The whole idea behind code profiling is to count how often a given target address is present on the microprocessor address bus, for a period of time. This profiler’s weakness is that the target address must be given. This target address may be statements, blocks, loops, subroutines, variables, arrays, and even combinations of these. This approach is independent of how these addresses are chosen, their meaning, or their purpose. Another assumption is that the bus in question contains the relevant addresses, and that virtual memory is not being used, or in other words, the address in the address bus has not undergone any type of translation (to a cache address, for example) or it does not need to be translated into a virtual address. Another assumption is that a target address can appear at every clock cycle. The profiling circuit is also design in a way that its speed can either be the same as the bus

being monitored, or lower. It would obviously be simpler if the profiler had a higher clock frequency, but this is seldom possible.

The objective is to monitor a bus (B) with a given width (w) for a time span of M clock cycles. In each cycle (t), a pattern (P) emerges on the bus (B), which will be the input Pattern.

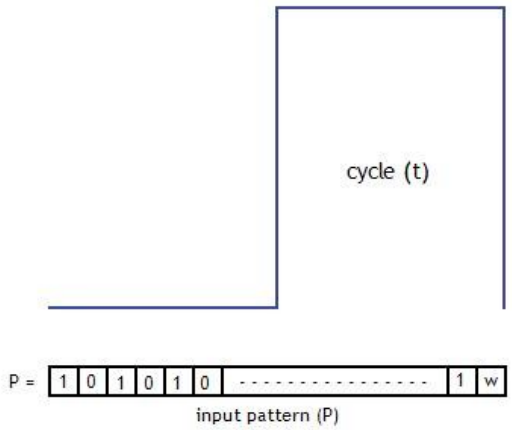

Figure 3-6- Promem, Input Pattern and cycle

A pattern is nothing more than a group of 0s and 1s (whose size is the same as the width of the bus). A target pattern (TP) is provided (which is also a combination of 0s and 1s).

To achieve this goal, a set of pattern counts(C) must be kept.

The following equations represent this information

$$P = \{P_1; P_2; P_3; - - -; P_M\}$$

$$TP = \{tp_1; tp_2; tp_{3;} - - -; tp_n\}$$

$$C = \{ctp_1; ctp_2; ctp_3; - - -ctp_n\}$$

Where ctp_i is the number of times that pattern tp_i was present on B, for M cycles:

$$ctp_i = \sum_{t=1}^{M} \begin{cases} 1, & if\ p_t = tp_i \\ 0, & otherwise \end{cases}$$

3.2.2. Pipelined binary tree

One of the main objectives of this profiler is its use with embedded systems. To make this possible, the profiler must be non-intrusive. This means that no extra clock cycles or pattern changes may occur over the instruction bus.

One key step in the development of the profiler memory was the conclusion that single cycle lookup or single cycle write is not needed. The only requirement is single cycle update throughput, or in other words, the memory has to be able to accept a new pattern per cycle but, there is no need to actually update the count field of a matching target pattern until later.

To allow a one pattern per cycle throughput, the Promem memory structure was implemented using a pipelined binary tree structure (see Figure 3-7). This structure is responsible for finding the location of a given input pattern. The tree was implemented by using a separate module for each separate level. Each module also implements a pipeline stage.

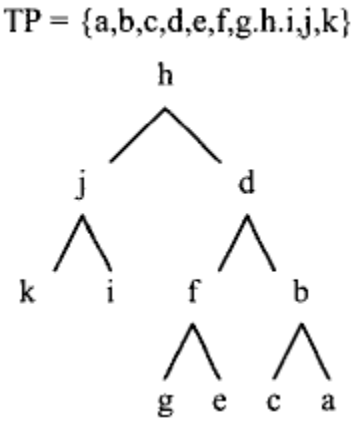

Figure 3-7- Promem, in order binary tree for a TP [3]

Assuming a four stage Promem design, the target patterns are placed in memory, in a way that the following property is followed:

"A node's children have the same high-order address bits as the parent's address, with the low order address bit of each child indicating left (1) or right (0) child."[3]

This property is verified in Figure 3-8, in the bits marked in bold. f has the 01 address in TPM_2, and its left and right children ('f' and 'g') have the 011 and 010 addresses in TPM_3.

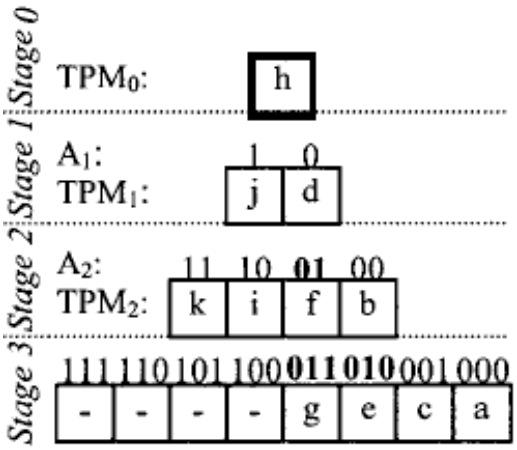

Figure 3-8 - Promem, how target patterns would appear in a memory [3]

Figure 3-9 depicts a search for a single input pattern ('e') for the tree shown in the previous image. 'e' is compared with the root node 'h'. Since 'e'<'h', the address (A_1) passed to stage 1 is 0. At this stage, 'e' is compared with $TPM_1[0]$ ('d'). This time, 'e' >'d', so the address A_2 passed to the next stage (stage 2) is 01. At this stage 'e' is compared with 'f' ($TPM_2[01]$). 'e'<'f', making the passing address 010. In stage 3, a match is finally found, and a count value (associated with value $TPM_3[010]$) is increased. These count values are stored in another memory, in a stage module.

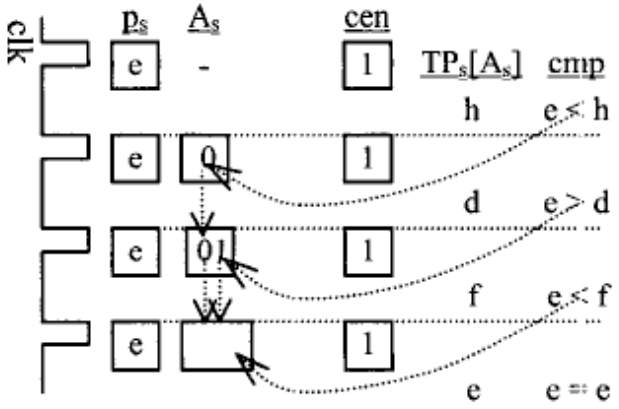

Figure 3-9 - Promem, search algorithm [3]

The previous example illustrates how a search is conducted, in order to locate a single item on the tree. The next image, takes this technique further by continuously supplying input patterns to the root node on each clock cycle, demonstrating how the pipeline works and hence, the single cycle throughput of Promem. The patterns being searched are 'e', 'd' and 'w', which will appear on the bus for consecutive clock cycles. As in the previous example, the same sequence can be seen for 'e', as it progresses through the pipeline. In this image it can also be seen the way in which the ***cen*** (enable signal) is updated. In the second clock cycle, 'd' enters the pipeline at Stage 0, from where it travels to Stage 1 as the right child. Since a match is found in this stage, ***cen*** is set to 0 (disabled) in order to prevent the search from continuing to stages 2 and 3. 'w' progresses in the same way, following the left children until it fails when it hits a node with no target pattern stored.

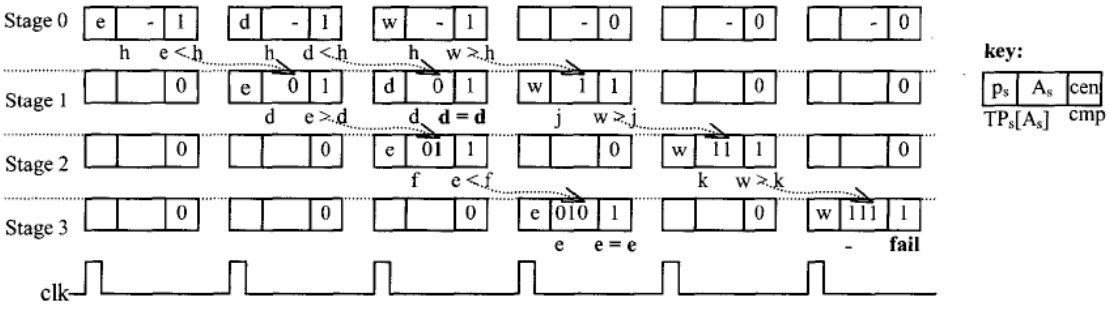

Figure 3-10 - Promem, pipelined search [3]

3.2.3. Memory architecture

As mentioned previously each level consists of its own independent module, made of registers to latch the input pattern, address and enable signals (collectively referred as Pipeline register), a memory containing the target patterns (TPM), a memory that contains the count for each of the target patterns (CM), and logic units responsible for performing the comparing and incrementing operations and other required logic operations.

Figure 3-11, along with the rest of this section helps to understand how the memory works, from a hardware point of view.

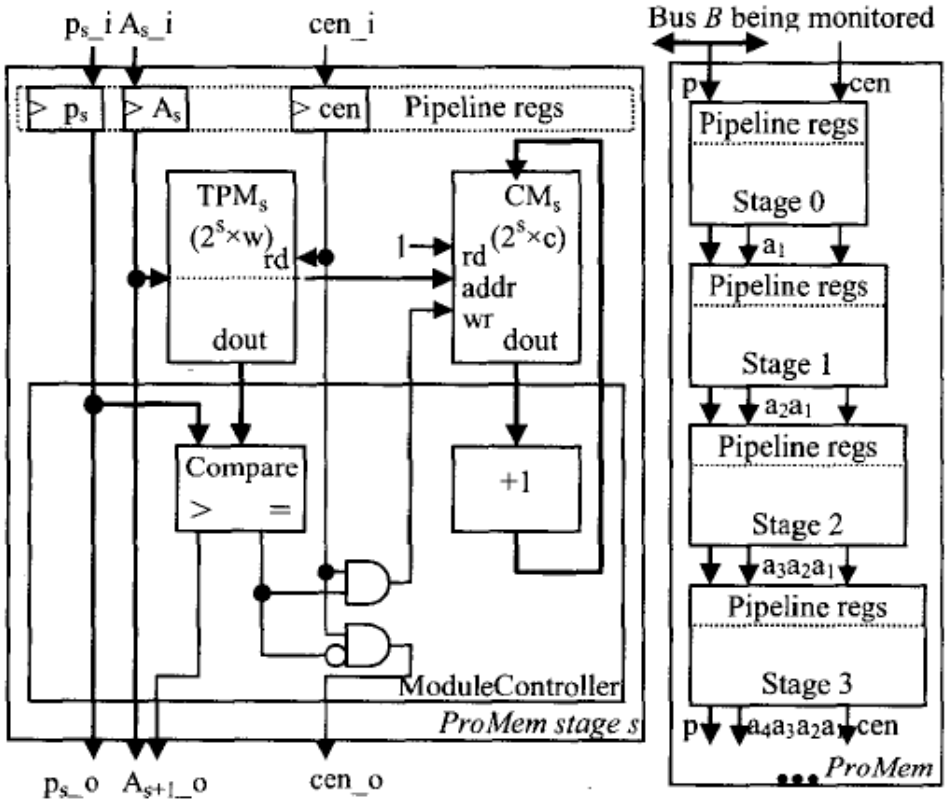

Figure 3-11 - Promem, Memory Architecture [3]

The input pattern is compared against the contents of the TPM_s at the address A_S_i. If the current stage is enabled (*cen_I* signal) and the input pattern matches the target pattern, the *cen_o* signal is set to 0, thus indicating a match and that no more searching is required.

As this happens, the CM write port is enabled, so that the associated count value is increased.

Because this requires that the same memory location is read and written, a memory with independent write and read ports is required. Though this kind of memory is slightly larger than a single port memory, it is still smaller than a dual port memory.

When the input pattern is different from the target pattern, the result from the comparison is concatenated with the input address A_i, to create the address for the next stage (A_{s+1}_i), and *cen_o* is set to 1 (thus insuring that the search continues).

The validity of the addresses and locations within the binary search is an issue that must be addressed. The scheme is to extend the width of the TPM by one bit. Every time a valid target pattern is written to the TPM, its concatenated with 1 (single bit). Then, the input pattern is also concatenated with a 1 bit, before reaching the comparator. Only then the target pattern is compared to the input pattern. After this operation, either a pattern match occurs (ending the search) or the validity of the target pattern must be checked. This is accomplished by checking the most significant bit of the current target pattern. If the bit is 0, then the target pattern is not valid which also causes the search to finish (if the target pattern is invalid, then no valid patterns will be found inside that sub-tree).

This design can be used in every stage of the binary search tree structure, despite its low efficiency on the first stage. If only a single entry exists within both the TPM and CM, to use a memory to implement then cannot be considered a viable solution. Instead, the TPM and CM on the first stage are implemented through two registers. The input to the first stage consists of the *cen-I* signal and the target pattern. This is the only difference between the first and the other stages. All other computation and logic is implemented in the same manner as the other stages.

This describes how an individual model works. The Promem structure, as mentioned previously, is made of several of these modules linked between them, as can be seem on the right side of Figure 3-11. As an example, to create a Promem design to handle a target set with 1023 entries, a binary search tree structures with 10 stages is required. This means that by using the modular design, if the number of target patterns needs to be extended, the solution is to simply add another level to the design.

3.3. Conclusions

This chapter introduced two important schemes for dynamic non-intrusive profiling. Despite their differences they have similar objectives: Identifying a particular section of the program being run by the system they are monitoring. They use different approaches and

have different purposes. DAProf identifies and categorizing loops in a purely dynamic way, while Promem by searches and counts a specific pattern.

While DAProf is within all parameters proposed for this thesis, Promem does not. Despite being a non-intrusive profiler, the pattern it searches for has to be manually introduced (unlike DAProf, which searches for its own patterns). Promem is relevant for its thesis because it introduces a different way to identify and store data.

The analysis of these two papers was critical in assisting the early stages of development of this thesis. Without its assistance, there would have been the need for a lot of testing, as well as the development of different methods (which criteria is relevant in loop identification and how to organize it for example).

Chapter 4

High Level Dynamic Identification of Control-Flow Constructs

This chapter describes in detail the developed solutions and the procedures followed in their implementation. During this project several ideas and possibilities arose for its implementation. A scheme for identifying dynamically control flow structures identification of the executing program can be achieve in many different ways, each of them having their particular set of advantages and disadvantages. In this chapter, we discuss the proposed solutions and analyze possible advantages and disadvantages.

4.1. Problem definition

When attempting to identify control flow structures in a program being executed by an embedded system, the first step is to obtain access to the execution flow. To achieve this trough a dynamic method implies that the system will be monitored in real time, without human intervention and causing no interruption to the original execution flow. This tends to restrict the method used for "probing". It prevents the use of any intrusive techniques, since these will necessarily require that changes be made to either the software being run or to the structure of the system itself.

Based on these assumptions, one can either monitor the instruction bus used by the system if the system keeps its instruction memory on a dedicated bus (in case of a Harvard architecture), or monitor the common bus, should the instructions be on an external memory. Despite the obvious difficulties in the first option, it can be done when using a soft core processor, or when the system allows for an external connection to that bus (highly unlikely).

The solution that can benefit the most cases is the second, however it requires that the profiler filter any un-wanted data (in a bus connecting a microprocessor with memory and GPIO's, for example the unwanted data represents a huge part of all data flowing through the bus). This would typically require that the range of relevant memory addresses be defined by the designer.

As mentioned before, in order to monitor the system, the hardware module will be directly connected to the instruction bus. For practical reasons, the development was designed to be tested by a soft-core processor (MicroBlaze). There are several reasons for this however the main one is that it is simpler to use a soft-core processor, since the module can be "plugged in" in a way far simpler than using a hardware platform.

Considering that the profiler probably will not have the luxury of a higher clock frequency than the system that is profiling, and that the scheme behind the profiling itself can take quite a few clock cycles to complete, a buffer may help to insure the correct and smooth operation of the profiler.

The heart of the profiler is the part responsible for processing the data. It needs to insure that loops are identified as such, accurately counted and has to achieve this in a time frame that insures that no data are lost. These objectives have to be met without requiring complex logic operations.

At this point it is convenient to elaborate on how to identify and classify loops. A loop can be detected when a backward jump (typically identified by the value of the instruction address) occurs. It is a fact that the backward jump may be due to an unconditional jump, the calling of a subroutine, or other options, however this method though not 100% accurate is completed by the remainder of the program and it is fairly accurate. A backward jump is not the only condition in finding a Loop. A random backward jump may eventually occur, but it will only occur once. A loop has a pattern. Identifying that pattern makes the distinction between a random backward jump and a loop. The properties of a loop within a program are its boundaries (the address from where it jumps and the address to where it jumps), its origin and destiny, the number of times the loop is executed during the duration of the program, the number of iterations per execution (maximum, minimum and average) and its relevance (i.e. is it frequently or rarely running?). To correctly classify the detected loops, most of this information is needed. The purpose of the profiler also weighs heavily in how relevant all of these criteria are. Another approach to loop identification can be to use not only the instruction address, but also the instruction itself. This method should have a higher accuracy, however it would also require that the profiler had all jump related instructions in memory.

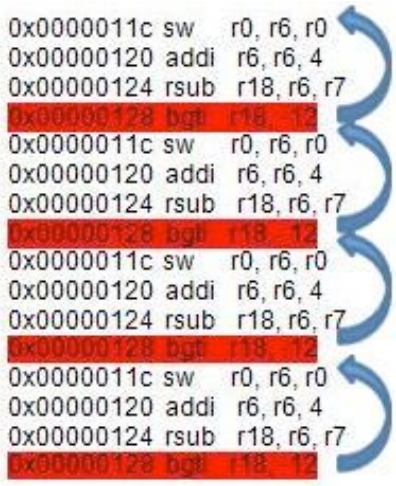

Figure 4-1 - Backward Jump Identification [5]

Figure 4-1 shows how the backward jump identification method works. Using a small loop as an example, it can be observed that the program flow keeps jumping from 0x00000128 to 0x0000011c. The bgti instruction tests the value stored in r18 (in this case) and if the test fails, branches to the address located at “current address” -12”.

4.2. Software Implementation

To evaluate a set of algorithms for the identification of control-flow constructs, software implementations were used. This made possible to test different criteria on how to check for loops, how to store and access the information, how to replace it if needed.

Though most (or almost all) of the restrictions hardware typically presents do not exist on this level of abstraction (timing and memory constraints being the most critical) this implementation still allows for some tweaking of the algorithms.

The algorithms being tested use as input data traces of program execution in the target microprocessor. Those traces were obtained via the simulator proposed in [5].

To be able to test the software, it was necessary to provide input (via a data file) to the program. To make the test more reliable, it would be good if actual addresses could be used. After many attempts, no way could be found to achieve this using EDK, which lead to the use of another software [5]. The “trace” information obtained, consists of the memory address, as well as the assembly code generated by the compiler. A small sample of a trace file can be seen on Figure 4-2.

```
0x000001ac  addk    r8, r8, r5
0x00000180  addk    r3, r7, r7
0x00000184  lhu     r4, r0, r6
0x00000188  lhui    r5, r3, 748
0x0000018c  sext16  r4, r4
0x00000190  sext16  r5, r5
0x00000194  mul     r5, r5, r4
0x00000198  addik   r7, r7, 1
0x0000019c  addik   r6, r6, 2
0x000001a0  addik   r18, r0, 159
0x000001a4  cmp     r18, r7, r18
0x000001a8  bgeid   r18, -40
0x000001ac  addk    r8, r8, r5
0x00000180  addk    r3, r7, r7
0x00000184  lhu     r4, r0, r6
```

Figure 4-2 - Trace sample [5]

The software consists of three main modules. The first module is responsible for parsing through the trace file and storing the relevant data, as well as converting the addresses from their original hexadecimal values to decimal values for processing. To make this as dynamic as possible, and to avoid allocating unneeded space, the program first checks the number of address lines on the data file, and creates an array with the size of the number of addresses. The addresses are then read and stored in an array (Figure 4-3, AddressList). Once this procedure is over, the (hexadecimal) addresses contained in that array are converted to their decimal value.

By now, the addresses contained in the trace file are stored in an array and ready to be processed.

The program now has an array (AddressList, Figure 4-3) that contains all of the addresses in decimal format, and is ready to start the loop identification process. In order to store all the information required to correctly identify and catalogue a Loop (loop boundaries, executions, iterations, and other relevant data) a matrix of arrays was created.

$$\text{AddressList} = \begin{bmatrix} ADDR_0 \\ ADDR_1 \\ ADDR_2 \\ \vdots \\ ADDR_n \end{bmatrix}$$

$$Loops = \begin{bmatrix} Index \\ 0 \\ 1 \\ \vdots \\ SizeOfResultTable - 1 \end{bmatrix} \begin{bmatrix} Origin \\ ADDR_\alpha \\ ADDR_\beta \\ \vdots \\ ADDR_\omega \end{bmatrix} \begin{bmatrix} Destiny \\ ADDR_\delta \\ ADDR_\pi \\ \vdots \\ ADDR_\varphi \end{bmatrix} \begin{bmatrix} \#Executions \\ \gamma \\ \delta \\ \vdots \\ \varepsilon \end{bmatrix} \begin{bmatrix} \#Total\ Iterations \\ \epsilon \\ \theta \\ \vdots \\ \mu \end{bmatrix}$$

Figure 4-3 - Result table and Address List

The #Executions and #Total Iterations arrays store the number of times a particular loop has been executed and how many times it has iterated. The Origin array stores the address from where the loop originated and the Destiny array stores the address to where the loops jumps to. The index array links all arrays. For example, in Figure 4-3, index 1 points to a loop

that jumped from $ADDR_\beta$ to $ADDR_\pi$, has δ executions and θ Total iterations. The number of Iterations per execution can be calculated by simply dividing the number of total iterations by the number of executions. In this case,

$$\frac{\#Total\ Iterations}{\#Executions} = \frac{\theta}{\delta}$$

4.2.1. Loop Identification

The second module is tasked with the detection of backward jumps. The simplest scheme to identify a loop is to compare two consecutive addresses. If the address $ADDR_{n+1}$ is smaller than $ADDR_n$, then a backwards branch has occurred. Since in C there are no timing constraints, there was no need for buffering values between the loop identifying and the information processing parts. Every time a loop is detected and the appropriate sub-routine called, the program simply waits for the result before resuming the scanning of the addresses.

The flow chart in Figure 4-5 illustrates how the detection process takes place. Until the end of the address list (obtained by parsing the trace file) the algorithm searches for backward jumps (by comparing $ADDR_{n+1}$ to $ADDR_{n)}$. When a backward jump occurs, a function is called to catalogue the loop. These functions will be introduced in the next sections.

The pseudo-code in Figure 4-4 also illustrates how this algorithm operates.

```
while(!EndOfAddressList)
{
If(ADDR-1>ADDR)
    {
    //A backward jump has occurred
    CatalogueLoop(ADDR, ADDR-1);
    ADDR-1=ADDR;
    }
else
    {
    //No backward jump detected
    ADDR-1=ADDR;
    }
}
```

Figure 4-4 - Loop Identification algorithm pseudo-code

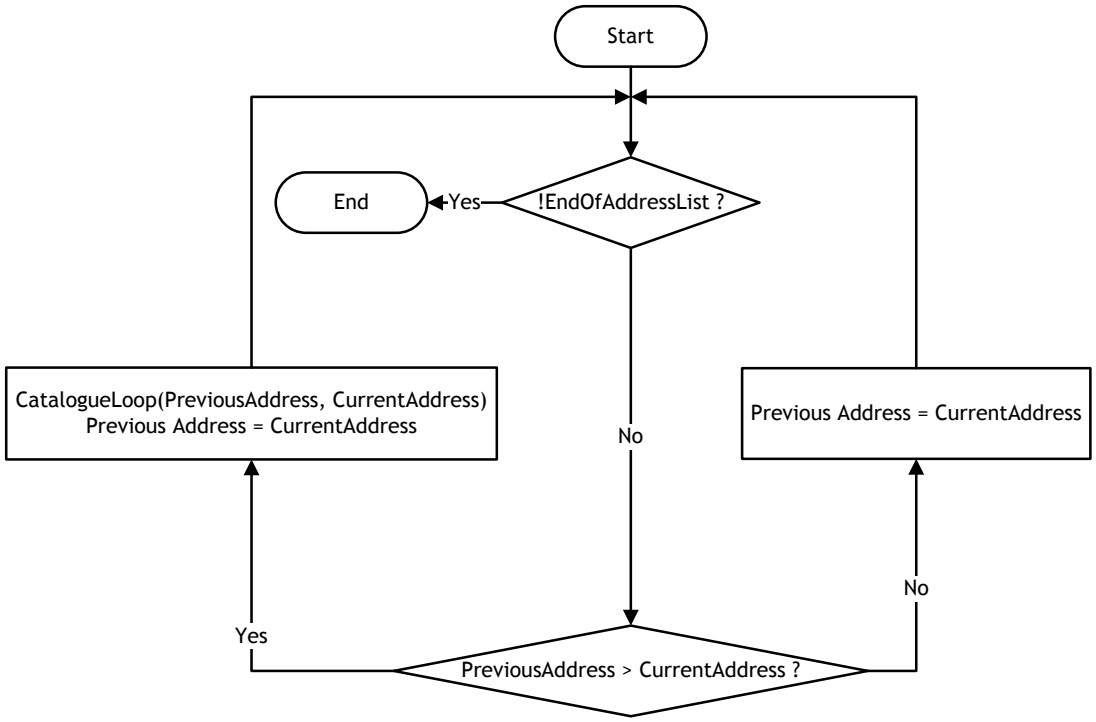

Figure 4-5 - Loop identification flow chart

4.2.2. Search by brute force

As was seen in the previous section, when a loop is identified, the algorithm will have to catalogue it. This means attempting to locate it in the result table (if it is a new execution of an already stored loop), and if this search is unsuccessful, add it to the result table (if it is a new loop).

There are three ways in which this may be accomplished. The simplest way is by using a brute force search (Figure 4-6 and Figure 4-7). The input parameters are ADDR and ADDR+1. If a match is found, the function returns the position index, or if it is a new loop it returns a pointer to the first available position in the result table.

As previously stated, the brute force search is by far the most straightforward way (though hardly the most efficient). The value contained in each position of the array containing the Loop information (the Origin array, which contains the ADDR values) is compared to the input ADDR until a match is found. If that match occurs, the program compares the values stored in the Destiny array for that same index. If the value is equal to ADDRp1, then the Loop identification is complete. The search ends and outputs the index of the Loop location, so that the Loop's Iterations and/or Executions can be increased. A flow chart of this algorithm can be seen in Figure 4-7.

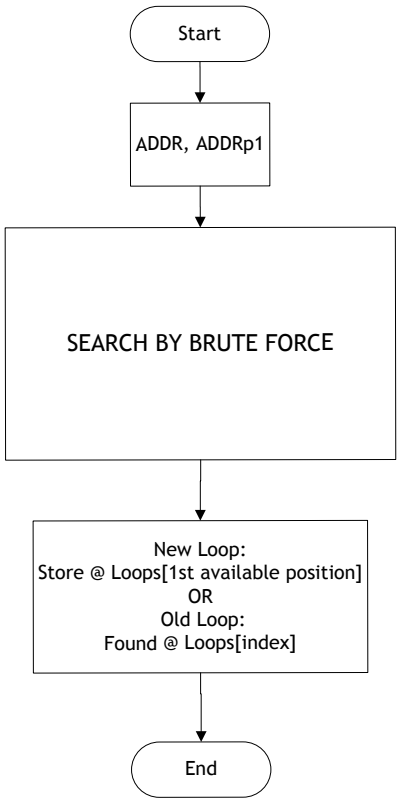

Figure 4-6 – Brute force search

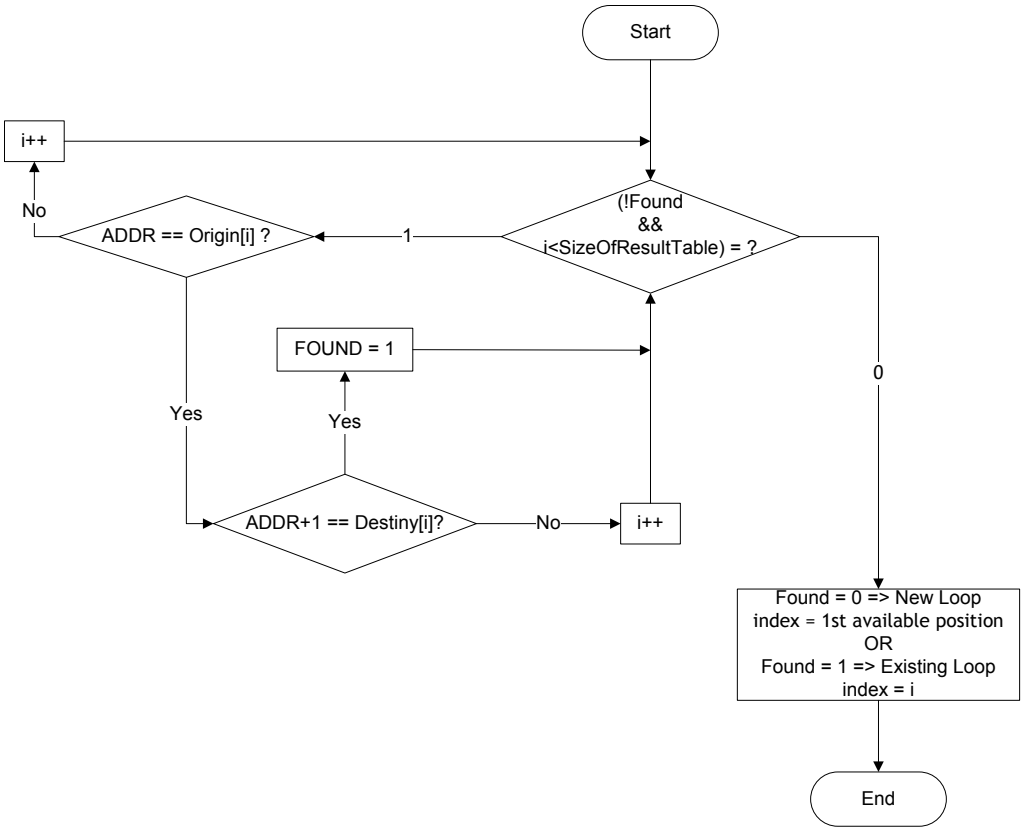

Figure 4-7 - Search by brute force algorithm

Search by brute force is very simple to implement, however it has several severe drawbacks.

Every time a loop is detected, the entire result table has to be checked, which is hardly efficient. It also cannot distinguish between a new loop execution and iteration. In order to differentiate between the two, it would be needed to store the information of the last loop to be identified. The need to differentiate between loop execution and loop iteration, and its subsequent need to store the ADDR, ADDRp1 and index of the last loop to be identified, lead to a second algorithm which is described in the next section.

In order to allow this algorithm to differentiate between execution and iteration, another array was added to the result table (called FLAG).

Whenever a loop is detected and is either inserted into the result table (if it is new) or if it already exists, its FLAG value is checked. If it the FLAG value is one that means that it is a new iteration of the same execution. If the value is zero, then it is a new execution.

When a new execution is detected, the FLAG value for that loop is set to one and the FLAG value for all other stored loops is set to zero.

4.2.3. Search by brute force--

This algorithm is similar to the brute force algorithm. In fact, the search mechanism itself is exactly the same. The differences are as can be seen in the flowchart depicted in Figure 4-8.

The first change is that the function parameters are no longer directly fed to the search mechanism. The first step here is to compare the new values to the last ones that the function analyzed. The first comparison is used to distinguish between loop execution and loop iteration. Should these values not match, another attempt is made. The index to Origin (and Destiny) tables is incremented by one, and the new values are compared with the ones stored in that position. This is done under the assumption that should the program be running continuously (as if under a while(1) cycle), then the loops should appear in the same order (or most of them, at least). By checking if the new loop is the same as in the previous cycle, a lengthy search is avoided. And if this test fails, then the table is searched from top to bottom. This method adds a few extra cycles to the search, but in theory it should prevent unnecessary searches in the result table.

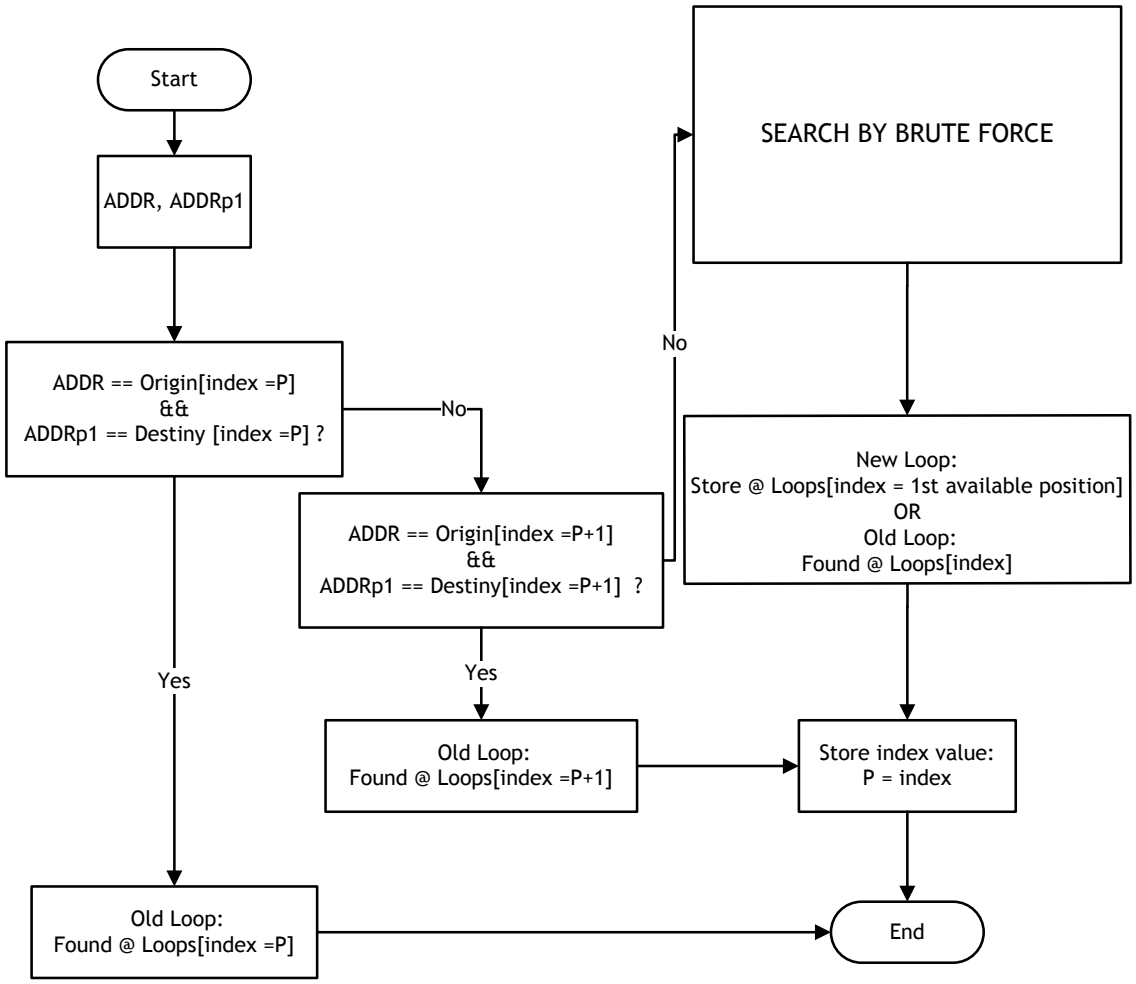

Figure 4-8 - Brute force-- algorithm

4.2.4. Binary Search

Brute force search can be an effective method if the result table is small. If the program is designed to store twenty or thirty Loops, then a brute force search can be the best method, despite its disadvantages. Considering a larger result table, for example, a table with over a hundred positions, then the brute force approach becomes impossible.

To be able to effectively search a result table that size (more than one-hundred positions), a different approach must be used. The main difficulty here is the fact that any approach more efficient than brute force has one basic need. The array must be ordered. It's simply not possible to attempt to apply an effective search algorithm without having the array ordered in any way. The first step is then to order the array. The best approach is to order it when a new value is being introduced. In this way lengthy searches can be avoided. To maintain an ordered array, every time a new value is to be added, it is always inserted in the first empty slot. Then, its value is compared to the one occupying the previous slot. If

smaller, the values are swapped, if not, than it has reached its position. The flow chart in Figure 4-9 shows the way in which this process occurs

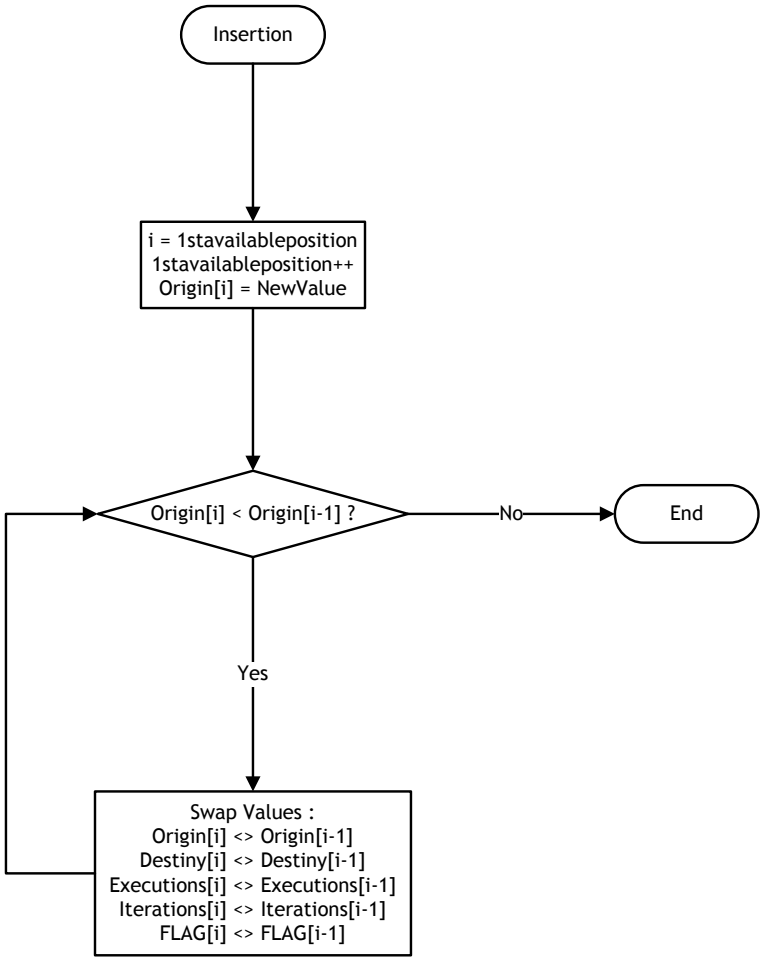

Figure 4-9- Flowchart for data insertion to allow the use of a binary search algorithm

Considering an ordered result table, the binary search algorithm can then be applied. The binary search algorithm is depicted in Figure 4-10. Unlike the previous algorithms where the search was conducted in a sequential way (by incrementing the result table index and comparing the stored value of ADDR with the new value of ADDR) this method searches the result table by "guessing" where the stored ADDR value is. The algorithm always starts at the median value (the value located in the middle of the result table size). It compares the stored value to the target value and tests if the stored value is equal, greater or less than the target value. Depending on the result, it redefines the boundaries for its search. If the value is greater that position becomes the upper boundary of the result table span, if it is smaller, it becomes the lower boundary, and if it is equal it means that it has found the value it was looking for. This process is repeated until the new value of ADDR is equal to the stored value of ADDR or it reaches the end of the search grid. This method reduces by a factor of two per comparison the number of positions where to search.

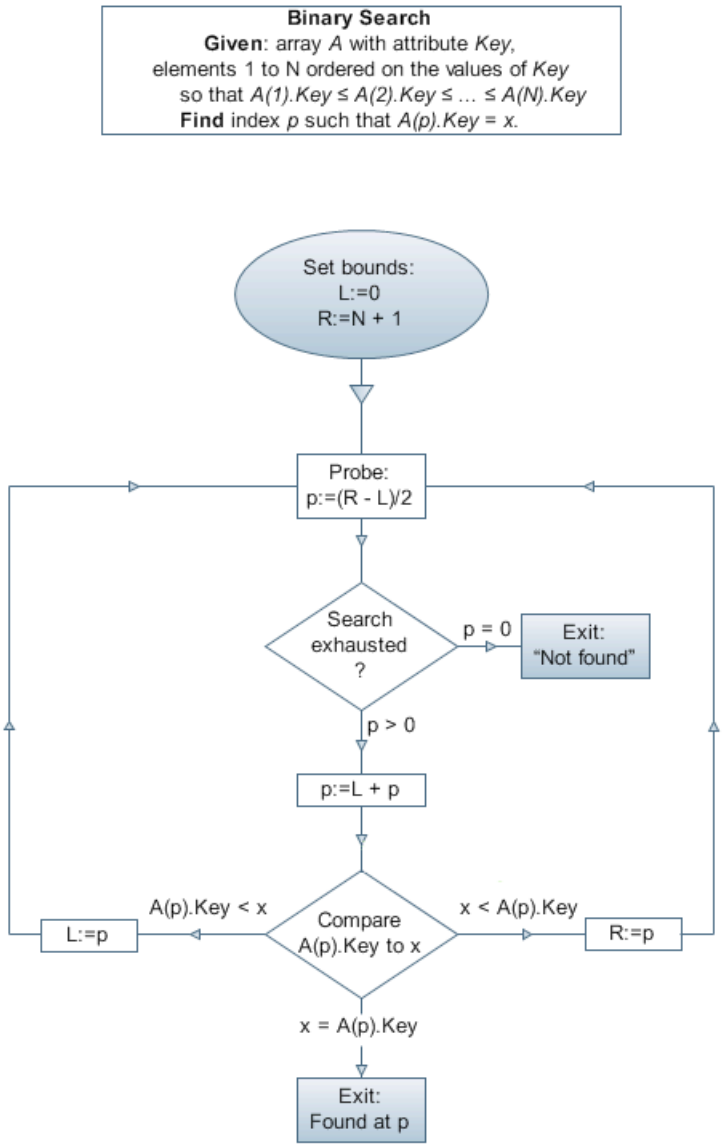

Figure 4-10 - binary search algorithm [6]

4.2.5. Hash indexing

Using a hash function to create a unique index to introduce directly values into a table is an extremely efficient way to access data. However, it is extremely rare when a hash function can map each possible key to a different slot index.

Instead of using a hash function an attempt to use a similar method was developed, by using part of the address value as the index for insertion in the result table. The principle is simply to use a number of LSB of the address value to directly insert or access a loop in the result table. For example the address 0x00001010 using the four LSB would be inserted in position 10 of the result table. The obvious disadvantage is that the size of the result table grows exponentially with the number of LSB required to minimize collisions. The advantage however is that this method would be perfect for a hardware implementation, given its low cost in resources.

Instead of implementing this insertion method directly (since it could lead to erratic results), it was decided to check the addresses of the loops identified by the brute force and brute force – algorithms (Figure 4-11) and them analyse those addresses to find out how many LSB would need to be used, so that all relevant loops are stored in the result table at the end of runtime.

```
loop @ADDR 0x164 to 0x064 with 1 executions, 1 total iterations and 1 iterations per execution
loop @ADDR 0x1ac to 0x180 with 16 executions, 2544 total iterations and 159 iterations per execution
loop @ADDR 0x1cc to 0x174 with 1 executions, 16 total iterations and 1 iterations per execution
loop @ADDR 0x1f4 to 0x15c with 1 executions, 1 total iterations and 1 iterations per execution
=================================================================================================
MASK = 4 LSB
  Decimal   |    Hexa     |  Unmasked binary   |         Masked binary          |Masked Decimal
ADDR = 356  | 0x00000164  |  0000000101100100  | MASKED ADDR = 0000000000000100 | 4
ADDR = 428  | 0x000001ac  |  0000000110101100  | MASKED ADDR = 0000000000001100 | 12
ADDR = 460  | 0x000001cc  |  0000000111001100  | MASKED ADDR = 0000000000001100 | 12
ADDR = 500  | 0x000001f4  |  0000000111110100  | MASKED ADDR = 0000000000000100 | 4
=================================================================================================
MASK = 8 LSB
  Decimal   |    Hexa     |  Unmasked binary   |         Masked binary          |Masked Decimal
ADDR = 356  | 0x00000164  |  0000000101100100  | MASKED ADDR = 0000000001100100 | 100
ADDR = 428  | 0x000001ac  |  0000000110101100  | MASKED ADDR = 0000000010101100 | 172
ADDR = 460  | 0x000001cc  |  0000000111001100  | MASKED ADDR = 0000000011001100 | 204
ADDR = 500  | 0x000001f4  |  0000000111110100  | MASKED ADDR = 0000000011110100 | 244
```

Figure 4-11 - Example output file, showing the addresses of identified loops and the number of LSBs necessary to identify them

To turn this policy into an efficient way of indexing the result table, a specific replacement policy would have to be designed, but even so, since the main purpose behind this concept would be a hardware implementation, the minimum number of required bits still is very big.

The main issue with this method is that it is unable to distinguish between loops. As long as the same LSB's are detected, the algorithm will simply assume that is a new execution/Iteration of an existing loop, which will lead to an erroneous count of Executions/Iterations.

A possible solution for this issue (not disregarding a replacement policy) might be to use an associative memory (Figure 4-12), where n bits of the address of the instruction (not being used to address the result table) to create a tag of sorts, and therefore allowing for more results to be stored under the same entry.

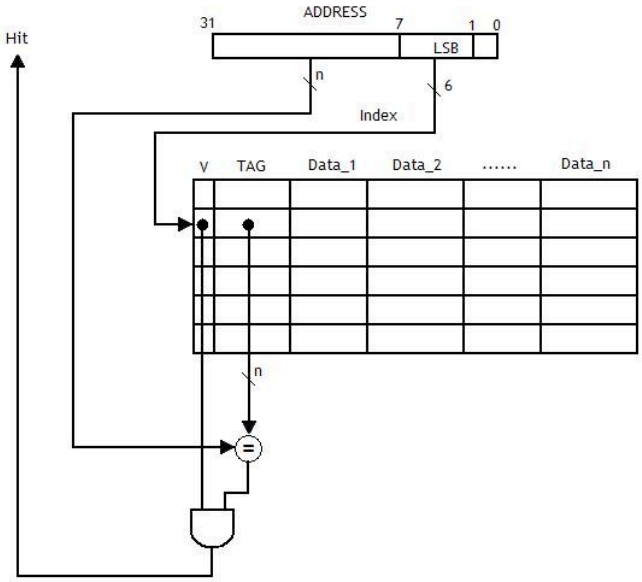

Figure 4-12 Associative memory

4.3. Replacement Criteria:

Depending on the number of entries in the result table a number of loops might have to be discarded. The size of the result table has to be determined.

Unless a very large result table is used, it is only a matter of time until it becomes completely full. Since the objective of the profiler is to identify the critical Kernel, or in other words the loops contribute the most to global execution time, a criteria to determine which loops shall be discarded to allow for the new one to be stored.

The first step is to determine which of the stored loops are more and less important.

Going back to the definition of critical kernel, the more important loops are the ones which executed often. So, a good starting point might be the number of executions. However if two loops have the exact same number of executions, how can this tie be broken?

When two loops have the same number of executions, the loop with the highest number of iterations is a more suitable candidate for parallel processing (which is one of the possible applications for the profiler: identification of loops suitable for parallel processing, leading to an increase in processing speed or lower energy consumption).

Thus, the factors that determine the importance of a loop, the conclusion are its executions and iterations.

4.3.1. ReplaceExecIter algorithm

```
int i = 1,j=0;
minExecutions = Executions[0];
minIterations = Iterations[0];
while(i<SizeOfResultTable){
    if(Executions[i]<= minExecutions)
        if(Iterations[i]<= minIterations)
            {
            minIterations = Iterations[i];
            minExecutions = Executions[i];
            j=i;
            i++;
            }
        else
            i++;
    else
        i++;

            }
    return j;
```

Figure 4-13 - ReplaceExecIter code

This simple algorithm shown in Figure 4-13 scans through the Executions and comparing the stored values with each other. Every time a new low for the Executions is found, it then checks to see if that loop also has a lower iteration count (comparing to previous values). By doing this, the algorithm insures that the lowest Executions and Iterations pair is the one that will be replaced.

Although this method protects the loops with the higher Execution and Iteration count, it also has a rather severe drawback. It prevents new loops from reaching a high Execution and Iteration count. By constantly electing the loops with less Executions and Iterations for replacement, there is a high probability that a loop that has just been added will be replaced (and therefore, lost) which might lead to an erroneous count. Even if the new loop is critical, the profiler may never realize that. Thus, a new factor is needed: Age.

By adding a new characteristic to each loop, Age, it becomes possible to have an idea of when was the last time this loop was executed. Making this new characteristic the main criteria for determining loop replacement, the following algorithm was developed.

4.3.2. ReplaceAgeSimple algorithm

In order to implement this algorithm, certain changes had to be made to the Brute Force and Brute Force-- algorithms. Every time a new loop was found, or a new execution was detected, the Age field in the result table for that particular loop was set to maximum (the value used was 16) and the Age for all other stored loops was decremented by one. For the

test purposes, the minimum value of Age for any stored loop was set to zero to insure that a minimum common level would exist.

The following code was added to perform that function:

```
for(j=0;j<SizeOfResultTable;j++)
        {
        if(Age[j]!=0)
            Age[j]--;
        else
            Age[j]=0;
        }
    Age[i]=16;
```

Figure 4-14 - Code modification for the Brute Force and Brute Force-- algorithm to allow the use of the Age criteria for the ReplaceAgeSimple

To find the lowest Age dor a given loop the following code was used:

```
int minAge=Age[0];
    for (j=1;j< SizeOfResultTable;j++)
        {
        if(Age[j]<=k)
            {
            minAge=Age[j];
            i=j;
            }
        }
return i;
```

Figure 4-15 – ReplaceAgeSimple code

This system accounts for how recent a loop is, but since no other criteria is being used, the importance of a loop is measured only by how recently it was executed. It would seem that a more accurate solution would be to merge this algorithm with the ReplaceExecIter algorithm.

4.3.3. ReplaceAgeExecIter algorithm

This algorithm combines the ability of the ReplaceExecIter algorithm to select the loop with the least executions and iterations with the ReplaceAgeSimple algorithm capability of allowing recently added loops to remain in the result table long enough to allow for the possibility of being relevant to the program flow. Age was selected as the most important criteria because of that possibility.

```
k=Age[0];
        l=Executions[0];
        m=Iterations[0];

    for(j=1;j< SizeOfResultTable;j++)
            if(Age[j]<=k)
                if(Executions[j]<=l)
                    if(Iterations[j]<=m)
                        {
                        k=Age[j];
                        l=Executions[j];
                        m=Iterations[j];
                        i=j;
                        }
return i;
```

Figure 4-16 - ReplaceAgeExecIter code

4.4. Summary

In this chapter the algorithms that control the Profiler were described.

To determine if a Loop has occurred, the Profiler will monitor the instruction addresses for small backward jumps. Since the profiler will be interfacing with the instruction bus no filtering will be required.

There are four approaches on how to search the result table and either update the data concerning stored Loops or insert new ones into storage. Brute Force and Brute Force-- are similar on how they search the result table but have different approaches on how to differentiate Loop Executions from Loop Iterations. The binary search method is only effective for a big result table (because of the time lost ordering it), and the hash indexing algorithm is particularly useful for a hardware implementation but it requires very thorough testing to insure that is able to keep data loss to a minimum.

Since a result table is bound to be smaller than the total number of Loops identified in any given program, three replacement policies were developed. Each of them searches for a minimum value of a given criteria. The chosen criteria were number of Executions, number of Iterations and AGE (the age of a loop is a measurement of how often it is executed) of a Loop. ReplaceAgeSimple chooses the Loop with the lowest AGE, Replace ExecIter the Loop with the lowest Execution and Iteration count (Execution being the main criteria) and ReplaceAgeExecIter determines which Loop is the oldest and has the lowest number of Executions and Iterations.

Chapter 5

Results

Replacement algorithms were tested individually (always for both search algorithms).

The information available in the program output is the number of backward jumps detected, the number of loops each algorithm identified, the maximum number of loops that could be stored and the loops identified by the profiler (along with information concerning executions and iterations).

To measure the success rate of the profiler, it is compulsory to compare the results to the original file. Table 1 and Table 2 contain the number of loops and their parameters from the C source code, as well as the number of Loops identified in the assembly generated file for the benchmarks under test obtained without complier optimization and with -O2 respectively.

For the trace files containing the instructions and code obtained with the level two optimization it isn't possible to simply read the source code and fill the data. It was necessary to confirm the number of loops by comparison. This was achieved by comparing the assembly files (generated by the mb-gcc) of the optimized version with the non-optimized version. While this allowed for an estimate of how many loops existed, it was impossible to discover their parameters (Executions and Iterations). However, after running all the tests, and confirming the accuracy for the profiler (this was done by comparing the results obtained with the non-optimized code with the original C code), a pattern clearly emerged.

	#Loops Identified in		Loop ID	#Executions (EX)	#Iterations per Loop (IT)	#Total Iterations (#EX x #IT)
	C source code	Assembly code				
FDCT	3	32	1	1	4	4
			2	4	8	32
			3	1	32	32
Autocorrelation	2	32	1	1	1	16
			2	16	160	2560
ADPCM - Coder	1	31	1	1	1024	1024
ADPCM - Decoder	1	31	1	1	1024	1024

Table 1 - Benchmark characteristics extracted from C source code with default parameters obtained without compiler optimization

	#Loops Identified in		Loop ID	#Executions (EX)	#Iterations per Loop (IT)	#Total Iterations (#EX x #IT)
	C source code	Assembly code				
FDCT	N/A	13	1	3	1	3
			2	4	7	28
			3	1	31	31
Autocorrelation	N/A	12	1	1	15	15
			2	16	159	2544
ADPCM - Coder	N/A	13	1	1	1024	1024
			2	1	1024	1024
			3	1	512	512
ADPCM - Decoder	N/A	13	1	1	1024	1024
			2	1	1024	1024
			3	1	1024	1024

Table 2 - Characteristics for the O2 version of the benchmarks

The tests were conducted in the following manner. There were a total of eight trace files to profile. Four benchmarks were chosen and compiled without optimization and with level 2 optimization (hence the eight files). Two search and insertion and three replacement algorithms were tested, using four values for the size of the result table where the results were stored (four, eight, sixteen and one hundred and twenty-eight). Each output file contains the result of the analysis performed by the two search algorithms with one of the

replacement policies and for one result table size. This lead to a total of twenty-four output files.

```
 2773 backward jumps detected
106 backward jumps were classified as loops according to the BruteForce method
106 backward jumps were classified as loops according to the BruteForce-- method
The maximum number of loops that could be stored is 4
Backward jumps which occured only once(One execution with only one iteration) do not qualify as Loops


================================================================================================================

By BruteForce:
loop @ ADDR 0x0128 to ADDR+1 0x011c with 1 executions, 88 total iterations and 88 iterations per execution
loop @ ADDR 0x0228 to ADDR+1 0x01bc with 16 executions, 2560 total iterations and 160 iterations per execution
loop @ ADDR 0x0164 to ADDR+1 0x0064 with 1 executions, 1 total iterations and 1 iterations per execution
loop @ ADDR 0x0258 to ADDR+1 0x01a4 with 1 executions, 16 total iterations and 16 iterations per execution

================================================================================================================

By BruteForce--:
loop @ ADDR 0x0128 to ADDR+1 0x011c with 1 executions, 88 total iterations and 88 iterations per execution
loop @ ADDR 0x0228 to ADDR+1 0x01bc with 16 executions, 2560 total iterations and 2560 iterations per execution
loop @ ADDR 0x0164 to ADDR+1 0x0064 with 1 executions, 1 total iterations and 1 iterations per execution
loop @ ADDR 0x0258 to ADDR+1 0x01a4 with 1 executions, 16 total iterations and 16 iterations per execution


================================================================================================================
```

Figure 5-1 - Sample output

As can be observed in Figure 5-1, the results for Brute Force and Brute Force are the same. This was a constant throughout the tests, so the displayed results in this section do not specify which search algorithm was used in obtaining them. This is because both algorithms output the same values.

For all the benchmarks both Brute Force and Brute Force-- yielded the same results (as can be seen in the example output Figure 5-1 - Sample output Figure 5-1). This means that the same values (#Loops, #Executions, #Iterations) are obtained regardless of which of these algorithms is used.

The section shows the results obtained for the FDCT benchmark. Since both search and insertion algorithms have equal results, no distinction was made between them.

5.1. FDCT

In this section, the results gathered for the FDCT benchmark are presented in Table 3 through to Table 12.

In each table, results are organized by optimization and include the following characteristics:

- #Original Relevant Loops: Number of loops found in that benchmark, for the corresponding level of Optimization (none or O2)
- #Loops detected by the algorithm: Total number of backward jumps that the algorithms considers Loops

- # Relevant Loops stored w\ result table: From all Loops detected by the Algorithm, the ones who match the characteristics of the ones present on the benchmark found at the result table at the end of runtime
- #Executions: Number of time a given Loop is executed
- #Iterations for execution: Number of times a given loop iterates each time it is executed

As can be seen by analysing Table 3, for the trace file obtained by execution of the assembly code for FDCT without optimization, there are three Loops in the original code (as can be seen in Table 1), whereas the search algorithms found 32 Loops. From those 32 Loops, only three are relevant. Relevant means that the number of Executions and Iterations match the values from the original code (Table 1). In Table 3, since the result table is bigger than the total number of Loops, #The Relevant Loops stored w\ result table field was omitted.

Code	#Original Relevant Loops	#Loops detected by the algorithm	Loop ID	#Executions	#Iterations per Execution
Not Optimized	3	32	1	1	4
			2	4	8
			3	1	32
Optimized	3	13	1	3	1
			2	4	7
			3	1	31

Table 3 - FDCT, no replacement policy, Result Table size = 128

The data in Table 3 was gathered using a 128 entry result table. Since the maximum number of Loops found in the FDCT benchmark, using the Brute Force (or Brute Force--) algorithm is 32 (or 13 for the compiler optimized version), no replacement policy was needed and the data related to all loops (relevant or otherwise) was stored.

Next the results obtained when considering a smaller result table are shown. By selecting a size that is smaller than the total number of loops detected by the algorithm, the objective is now to test the efficiency of the three replacement policies implemented.

As can be seen in Table 4, despite the reduction in the size of the result table (from 128 to 16) all relevant loops are still stored. The replacement algorithm ReplaceExexITer had a 100% success rate for a result table with 16 entries.

The results in Table 6 show that for the first time not all of the Relevant Loops were in the result table at the end of runtime.

5.1.1. Using the ReplaceExecIter algorithm:

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 4 - FDCT, Array size of 16, using the ReplaceExecIter algorithm

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 5 - FDCT, array size of 8, using the ReplaceExecIter algorithm

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	2	1	1	4
			2	4	8
Optimized	3	2	1	3	1
			2	4	7

Table 6 - FDCT, array size of 4, using the ReplaceExecIter algorithm

5.1.2. Using the ReplaceAgeSimple algorithm:

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 7 - FDCT, array size of 16, using the ReplaceAgeSimple algorithm

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 8 - FDCT, array size of 8, using the ReplaceAgeSimple algorithm

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	0	0	0	0
Optimized	3	0	0	0	0

Table 9 - FDCT, array size of 4, using the ReplaceAgeSimple algorithm

5.1.3. Using the ReplaceAgeExecIter algorithm:

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 10 - FDCT, array size of 16, using the ReplaceAgeExecIter algorithm

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 11 - FDCT, array size of 8, using the ReplaceAgeExecIter algorithm

Code	#Original Relevant Loops	#Relevant Loops stored w\ result table	Loop ID	#Executions	# Iterations per Execution
Not Optimized	3	3	1	1	4
			2	4	8
			3	1	32
Optimized	3	3	1	3	1
			2	4	7
			3	1	31

Table 12- FDCT, array size of 4, using the ReplaceAgeExecIter algorithm

The ReplaceAgeExecIter algorithm is the only one that manages to achieve a 100% success rate for a size 4 result table.

5.1.4. Result analysis:

For the presented replacement policies and for result table sizes of 8 and 16, all algorithms manage to have a 100% success rate (all relevant loops are present in the result table at the end of runtime). For the size 4 table the ReplaceAgeSimple had a 0% success rate the ReplaceExecIter a 66% success rate and the ReplaceAgeExecIter still had a 100% success rate.

After analysis of this section, particularly Table 4, Table 5, Table 7, Table 8, Table 11 and Table 10 it becomes apparent that the success rate is rather constant, and most of the information is redundant.

In the next section, instead of continuing to present the individual results for each benchmark, it seemed preferable to present the remaining results in the form of graphs, and for all benchmarks to allow a more direct comparison.

5.2. Overall Results

In this section, the results for the FDCT, ADPCM-Coder, ADPCM-Decoder and Autocorrelation profiling are shown in the form of bar charts.

The charts are divided into two categories. There is one set for the results obtained with the optimized trace files and another set for the values obtained with the non-optimized trace files. Within each of these sets there are three charts, one for each replacement policy.

Each chart contains the values for the number of Loops found in the assembly generated file, the relevant loops found in the C source code, the number of loops the algorithms detected and the number of relevant loops found in the result table (for different table sizes) at the end of runtime per benchmark and for each of the replacement criteria.

5.2.1. Experimental results obtained without code optimization:

The chart in Figure 5-2 shows that for the FDCT, 32 Loops exist in the assembly code, despite the fact that only 3 exist in the C source file (similar differences can be found in the other benchmarks). The profiler, using the ReplaceExecIter policy, is able to keep the 3 relevant loops in the result table until the end of runtime, even with a result table that can store only one eighth of the total number of Loops. The same success rate is achieved for the other benchmarks

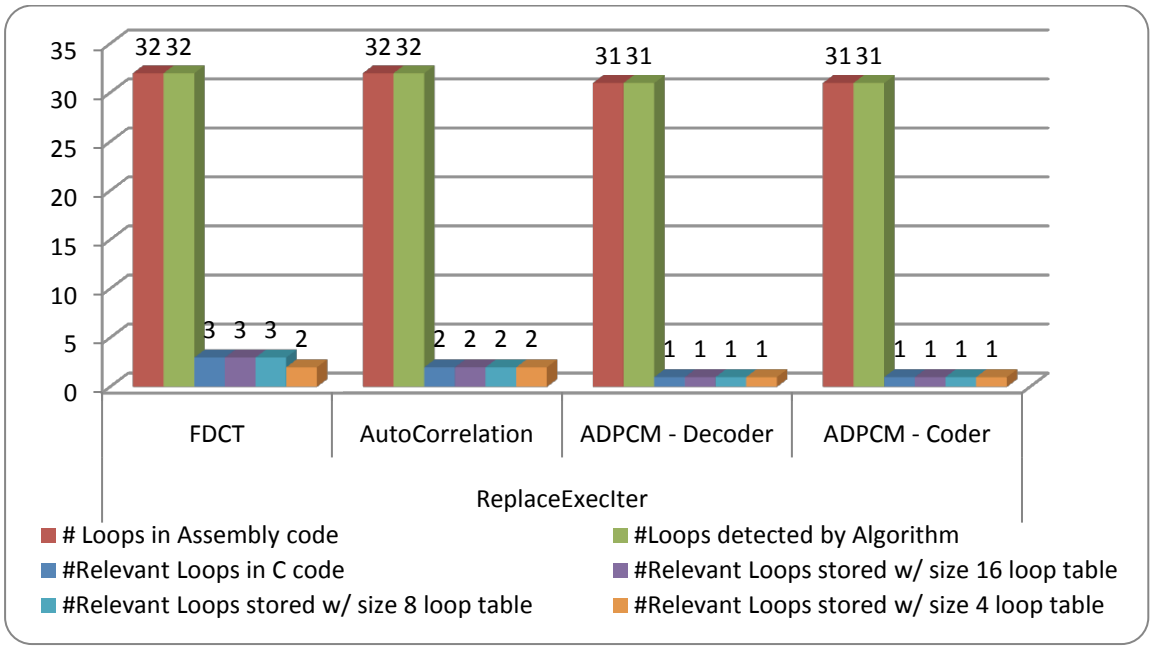

Figure 5-2 Result Chart for the benchmarks compiled without optimization, and stored with the ReplaceExecIter policy

Figure 5-3 shows the success rate for the ReplaceAgeSimple policy is lower than for the ReplaceExecIter. With this policy, none of the relevant loops are found in the result table at the end of execution. Since the relevant loops are a small percentage of the total number of Loops and this method replaces Loops stored within the result table based on how often a Loop is executed, the results are not unexpected.

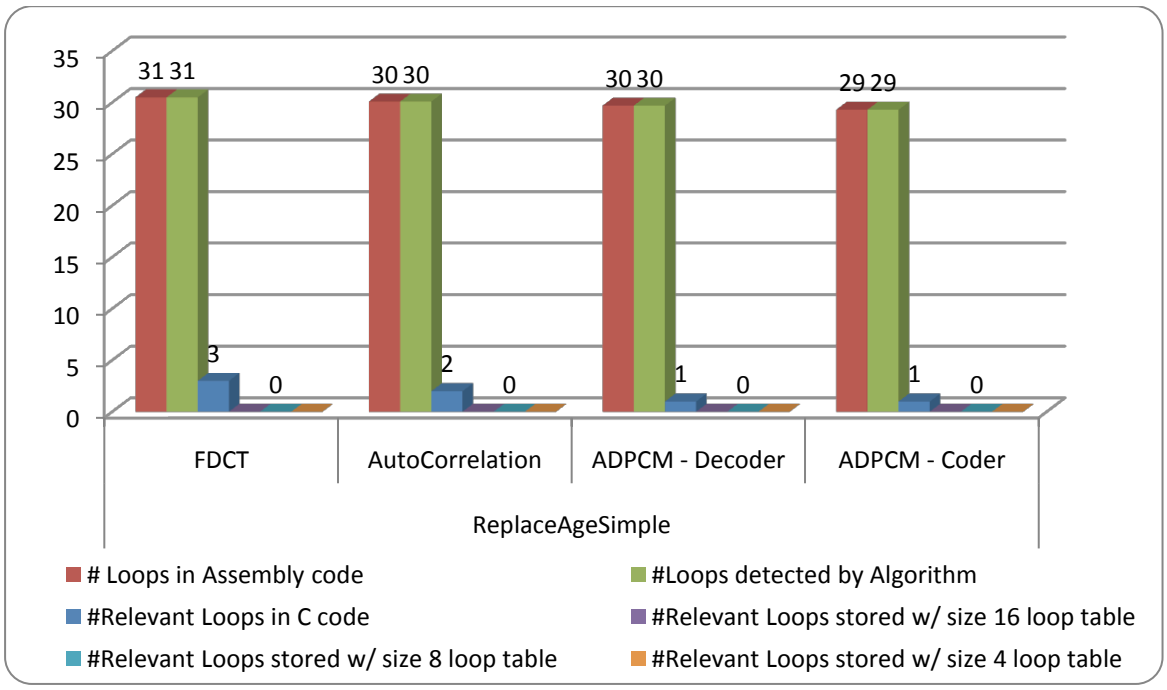

Figure 5-3 Chart for the benchmarks compiled without optimization, and stored with the ReplaceAgeSimple policy

The results in Figure 5-4 despite having a higher success rate than the ones in Figure 5-3 are still not as good as the ones for the ReplaceExecIter. Since AGE is the prime criteria, and considering the results obtained with the ReplaceAgeSimple algorithm, these results confirm that with added criteria, a balance between the age of a Loop and its number of executions (and iterations) can be achieved, allowing for a replacement policy that takes all loop characteristics into account.

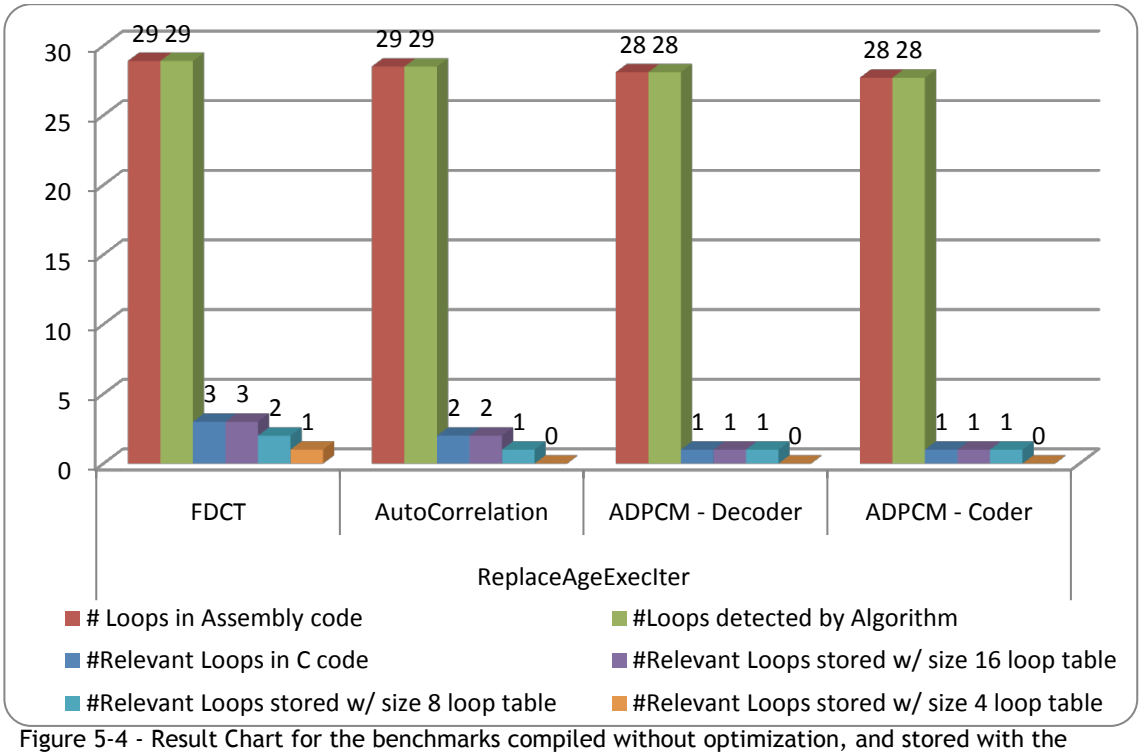

Figure 5-4 - Result Chart for the benchmarks compiled without optimization, and stored with the Replace AgeExecIter policy

5.2.2. Experimental results obtained with code optimization:

Figure 5-5, Figure 5-6 and Figure 5-7represent the results for the compiler optimized trace files. The main difference between these results and the ones obtained with the non-optimized trace files is the total number of loops in all of the benchmarks (approximately half of the total number of loops in the non-optimized version) and that for the ADPCM-Coder and Decoder, the number of relevant loops increased (from 1 to 3). The results are very similar to the ones obtained for the non-optimized version of the trace files (as was expected).

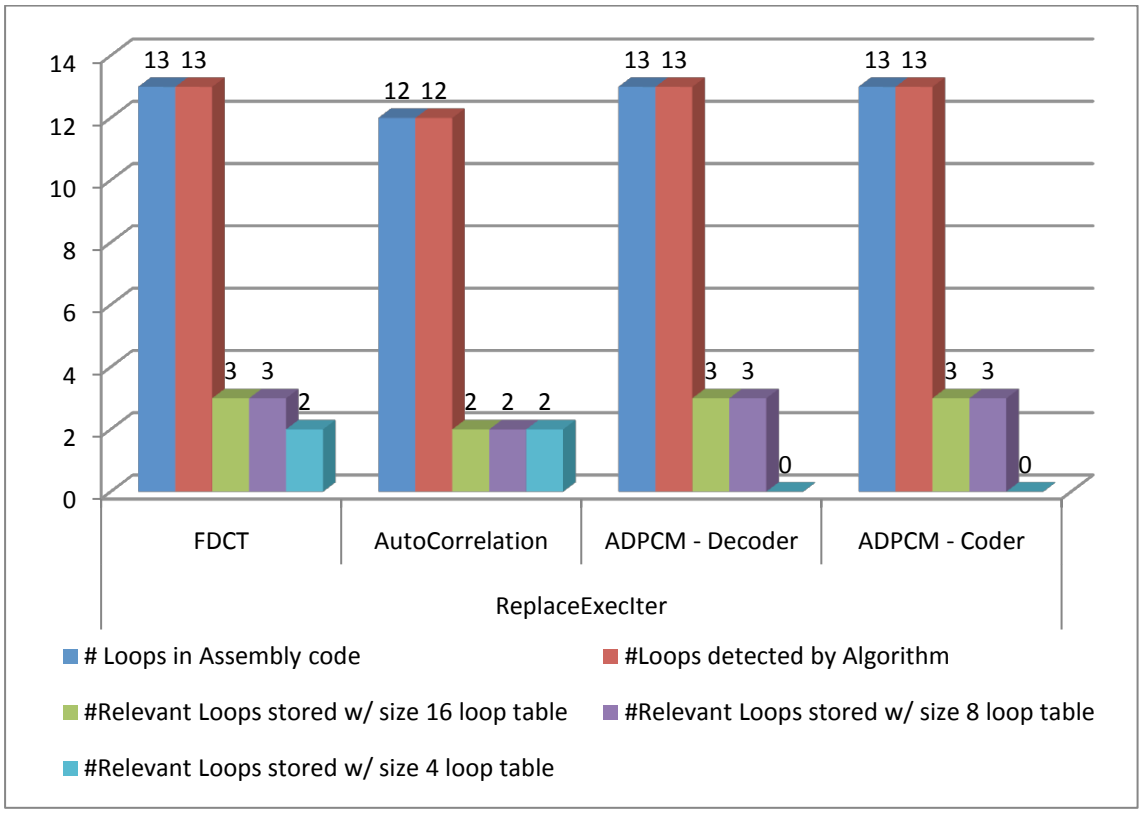

Figure 5-5 - Result Chart for the benchmarks compiled with optimization, and stored with the ReplaceExecIter policy

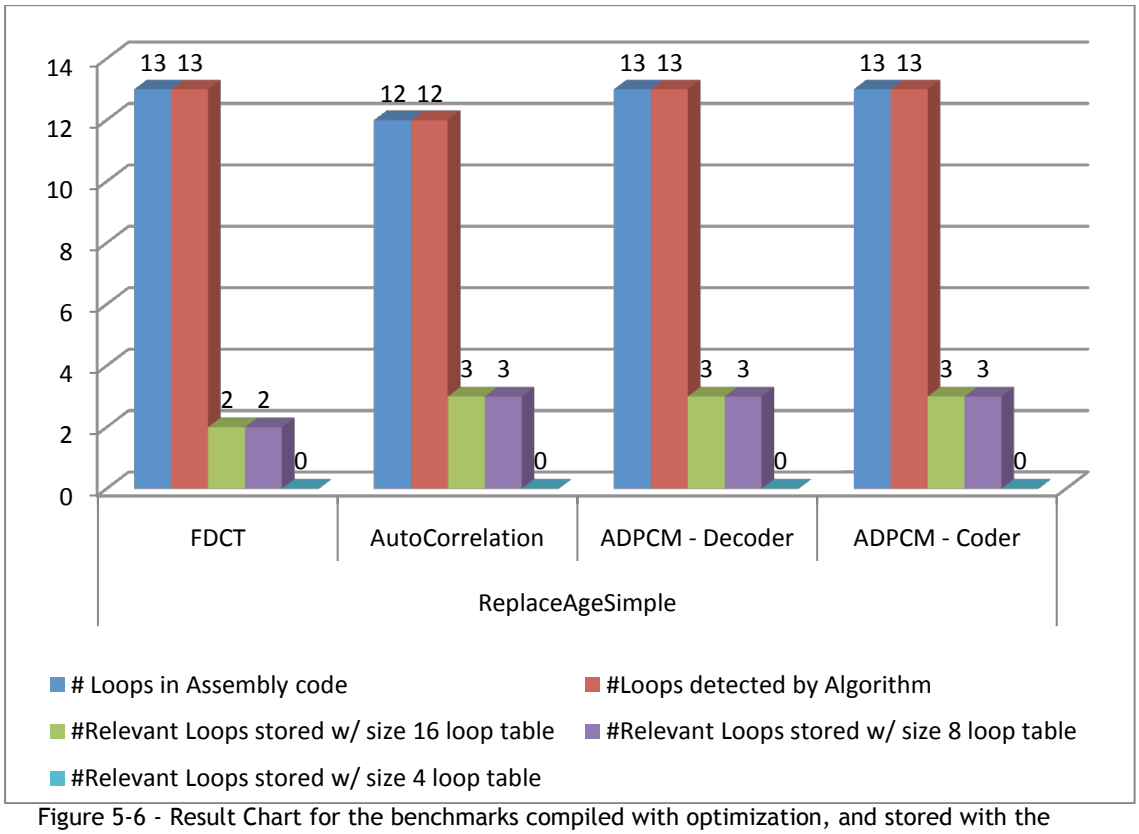

Figure 5-6 - Result Chart for the benchmarks compiled with optimization, and stored with the ReplaceAgeSimple policy

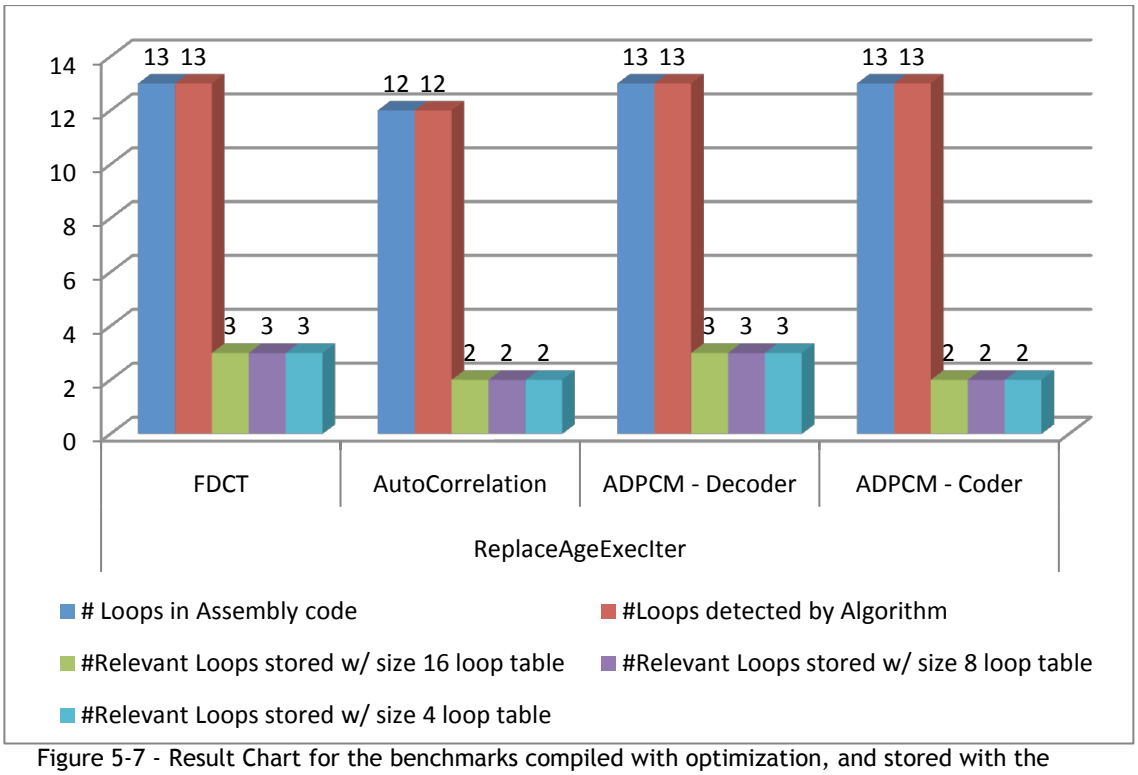

Figure 5-7 - Result Chart for the benchmarks compiled with optimization, and stored with the ReplaceAgeExecIter policy

The charts in section 5.2 (containing the results for all four benchmarks) match the results found for the FDCT benchmark (section 5.1). Given their size, and the fact that these benchmarks are non-recursive, the profiler is capable of correctly identifying and storing the relevant loops and storing until the end of execution, even if the result table can store only a fourth of all detected loops (for the non-optimized code, considering an average of 32 identified loops, and for a size 8 result table), regardless of which replacement policy is being used. When using the ReplaceExecIter policy, this fact merely points out that the relevant loops have a higher Execution/Iteration count.

The performance of the ReplaceAgeSimple policy, despite being the one with the lowest overall success rate (success rate is the number of relevant loops stored in the result table at the end of execution) also lifts an interesting question. If the relevant loops are executed in the beginning of runtime, they will always be replaced, and if the opposite occurs (the relevant loops are executed close to the end of runtime) the need to replace them may never arise. This policy is therefore inadequate.

The ReplaceAgeExecIter policy is the most complete, and also yields the best results for the benchmarks considered in this work.

The effect the result table size has on results reflects that for this set of benchmarks, and regardless of replacement policy the only size that shows different results is the size 4 result table.

5.3. Hash indexing results

After analysing and parsing the values outputted by the brute force and brute force--algorithms, in order to collect the necessary data for analysis and building the charts in Figure 5-8 to Figure 5-17 the following parameters were selected:

- #Total ID -> the total number of Loops that would be found in the result table at the end of runtime
- #Relevant ID -> The number of Relevant loops that would be found in the result table at the end of run time
- #Overlapping Values -> Number of loops that would be overwritten due to index conflicts

5.3.1. Charts representing the values obtained without code optimization:

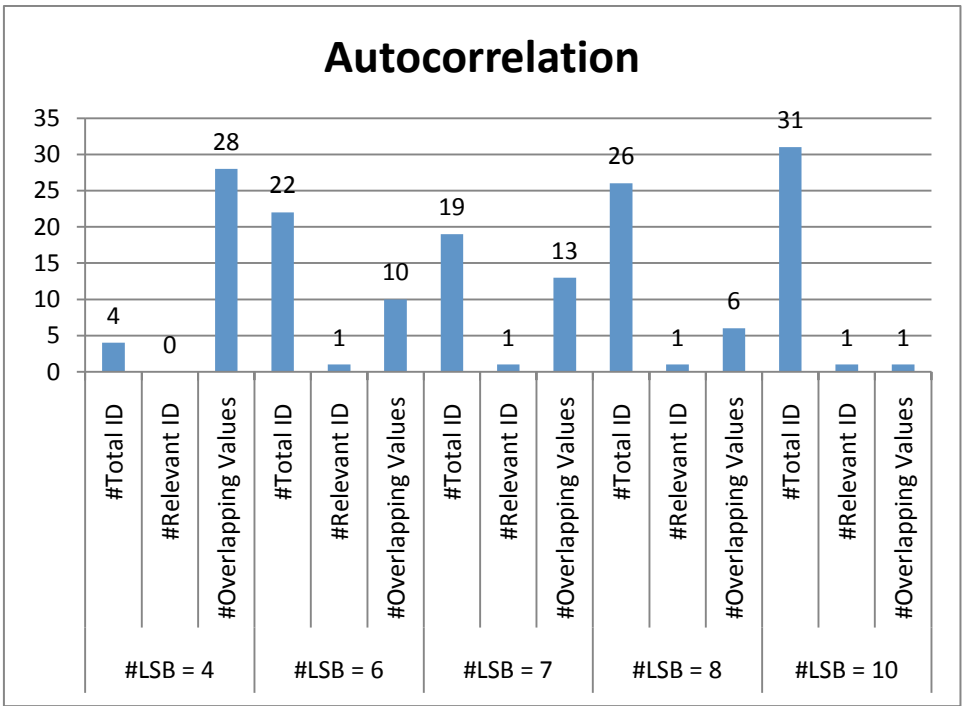

Figure 5-8 - Hash indexing result for the Non-optimized Autocorrelation benchmark

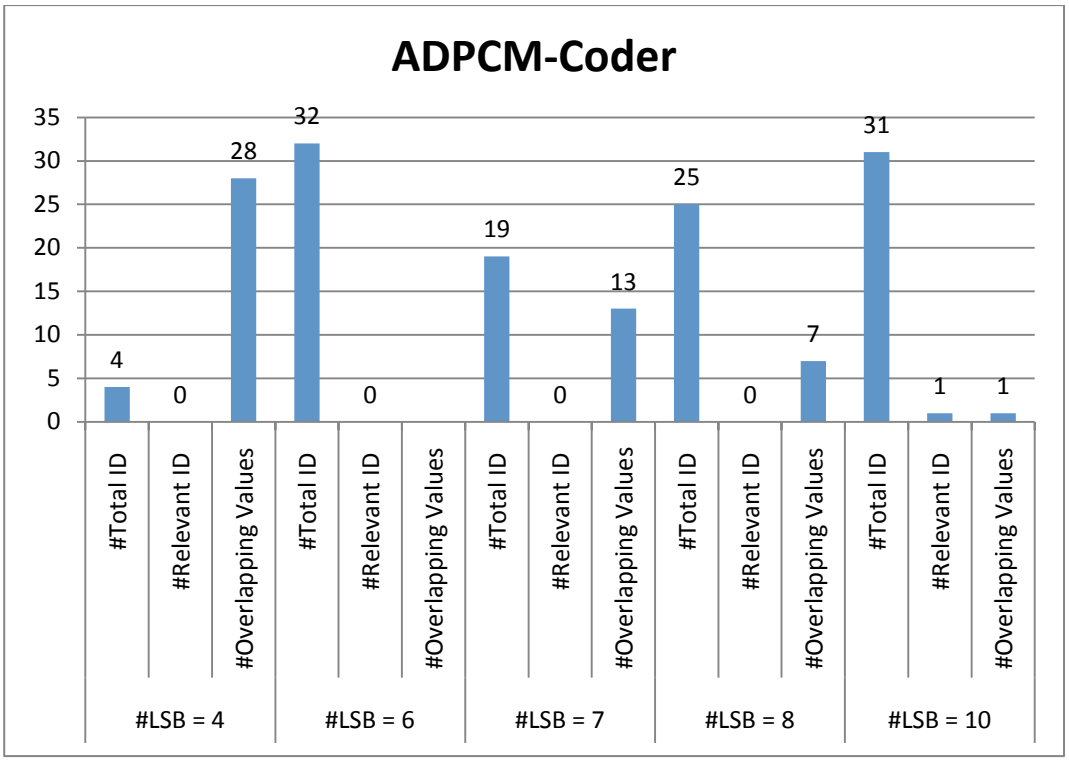

Figure 5-9 - Hash indexing result for the Non-optimized ADPCM-Coder benchmark

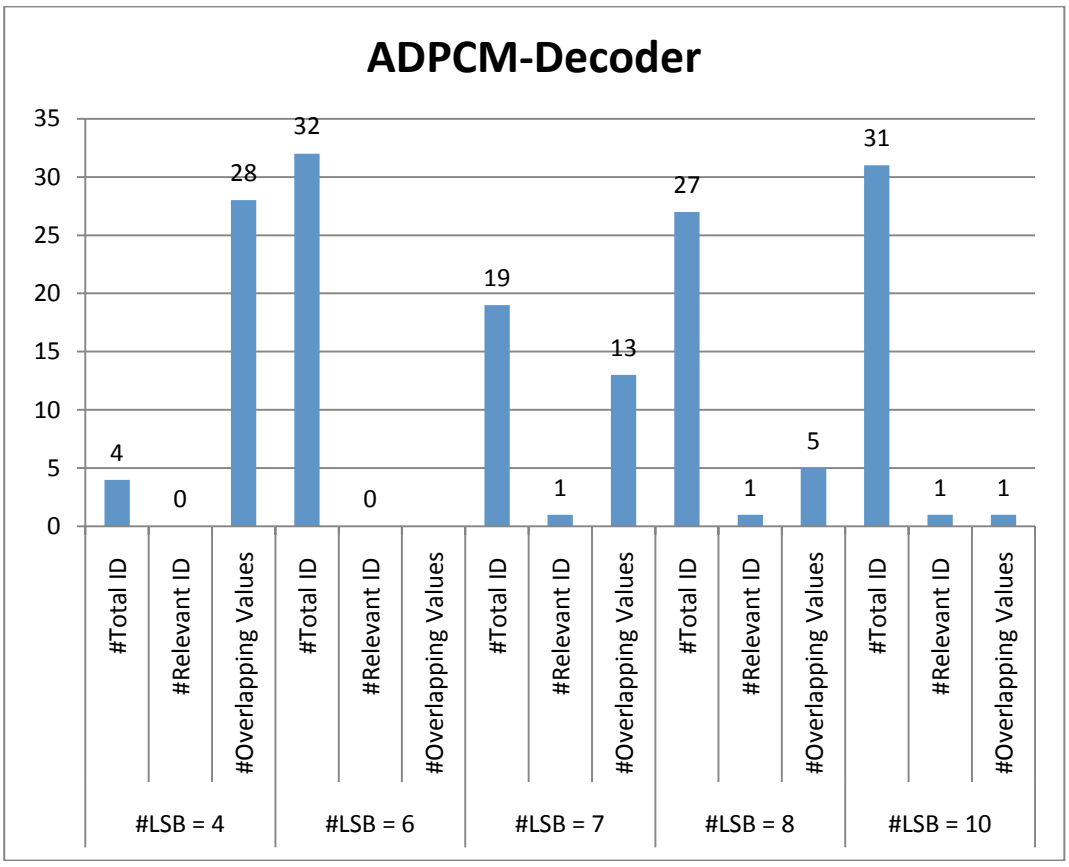

Figure 5-10 - Hash indexing result for the Non-optimized ADPCM-Decoder benchmark

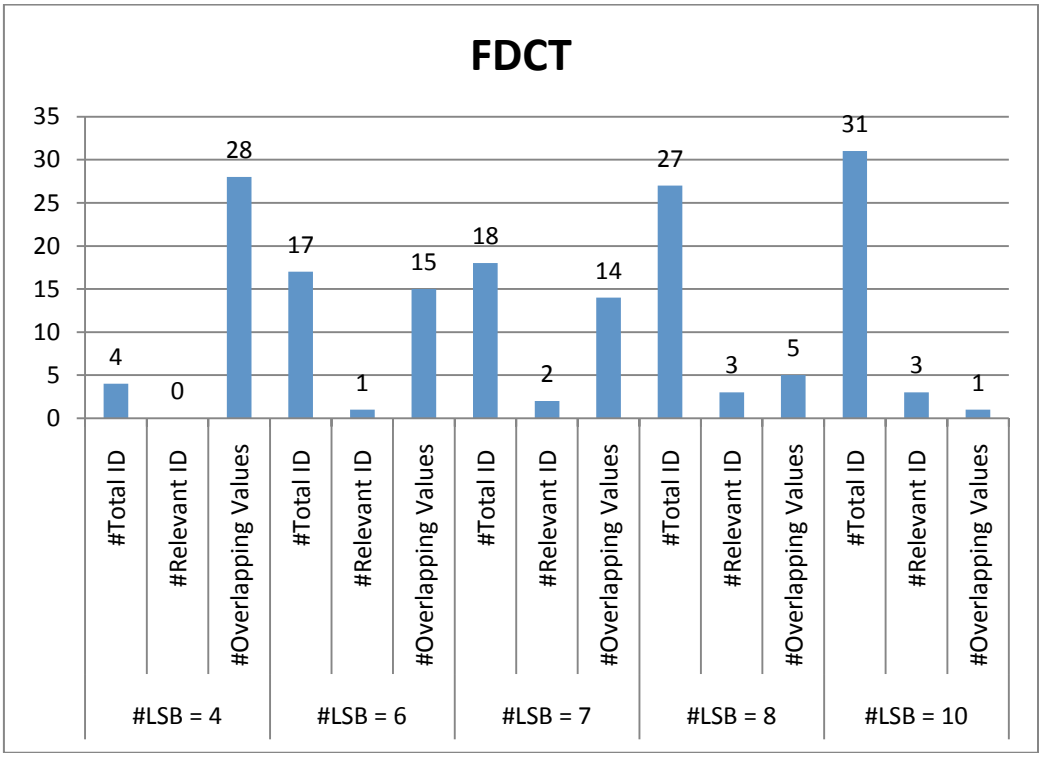

Figure 5-11 - Hash indexing result for the Non-optimized FDCT benchmark

5.3.2. Charts representing the values obtained with code optimization:

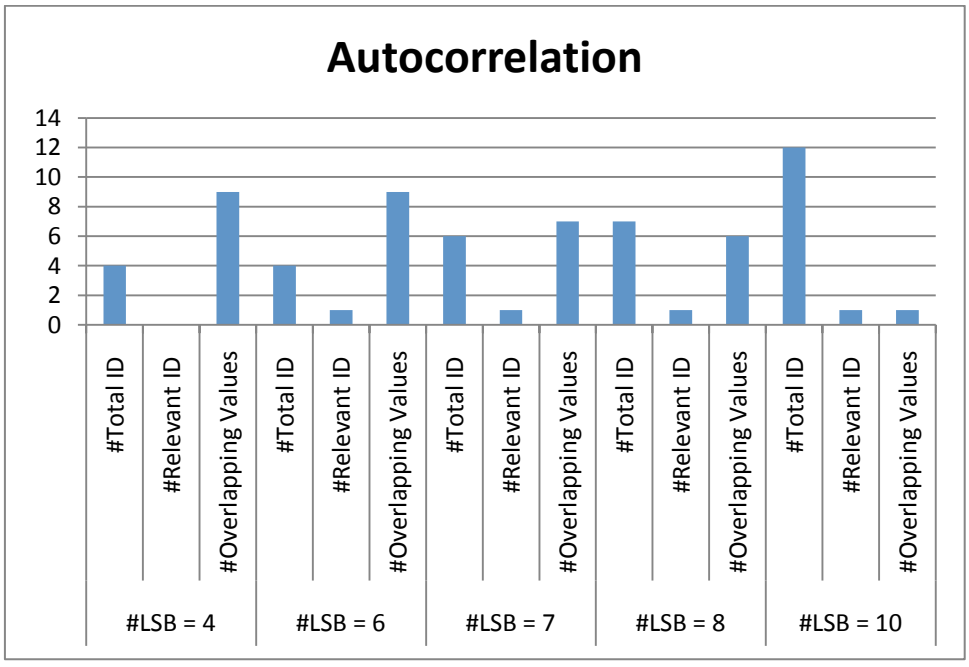

Figure 5-12 - Hash indexing result for the optimized Autocorrelation benchmark

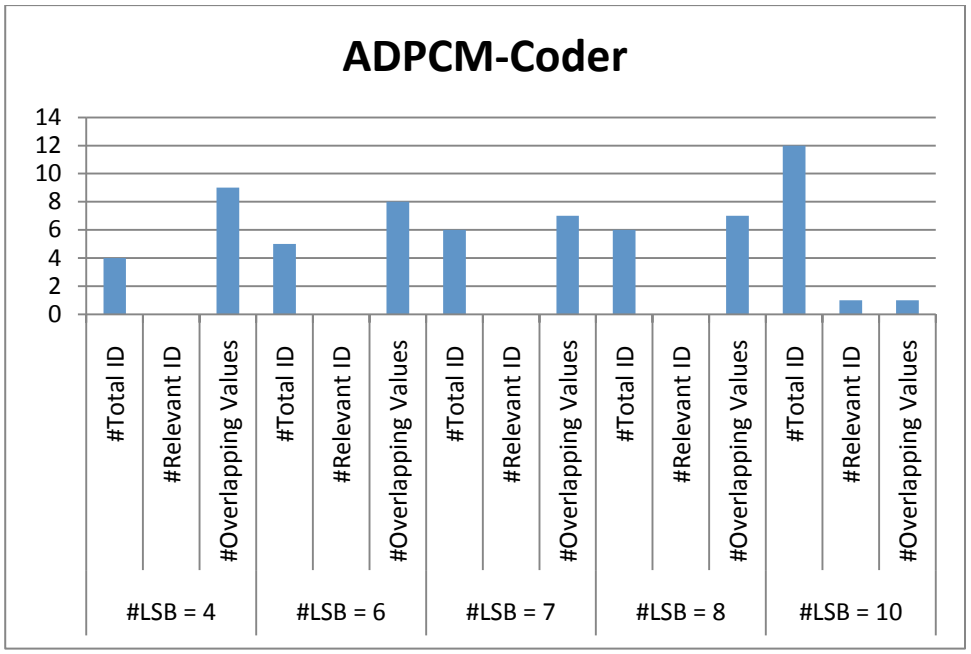

Figure 5-13 - Hash indexing result for the optimized ADPCM-Coder benchmark

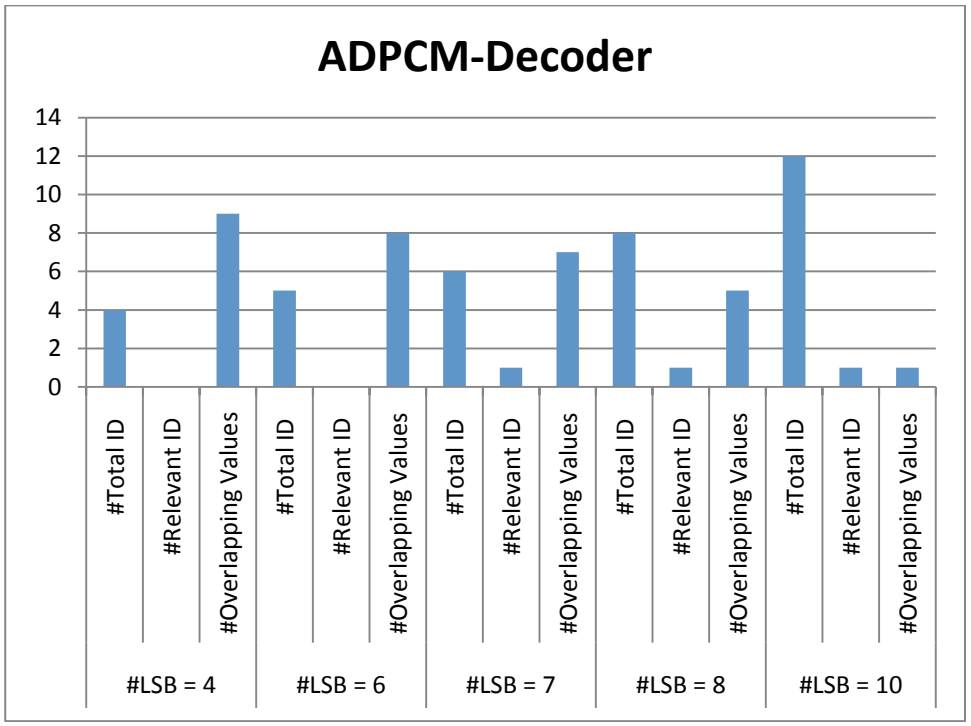

Figure 5-14 - Hash indexing result for the optimized ADPCM-Decoder benchmark

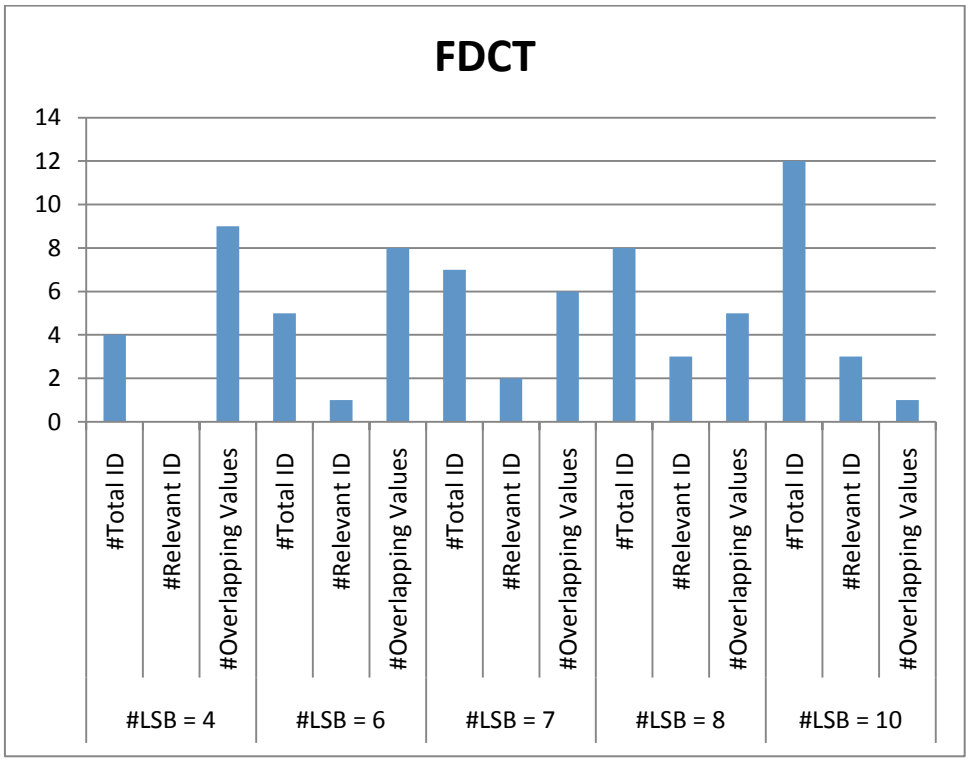

Figure 5-15 - Hash indexing result for the optimized FDCT benchmark

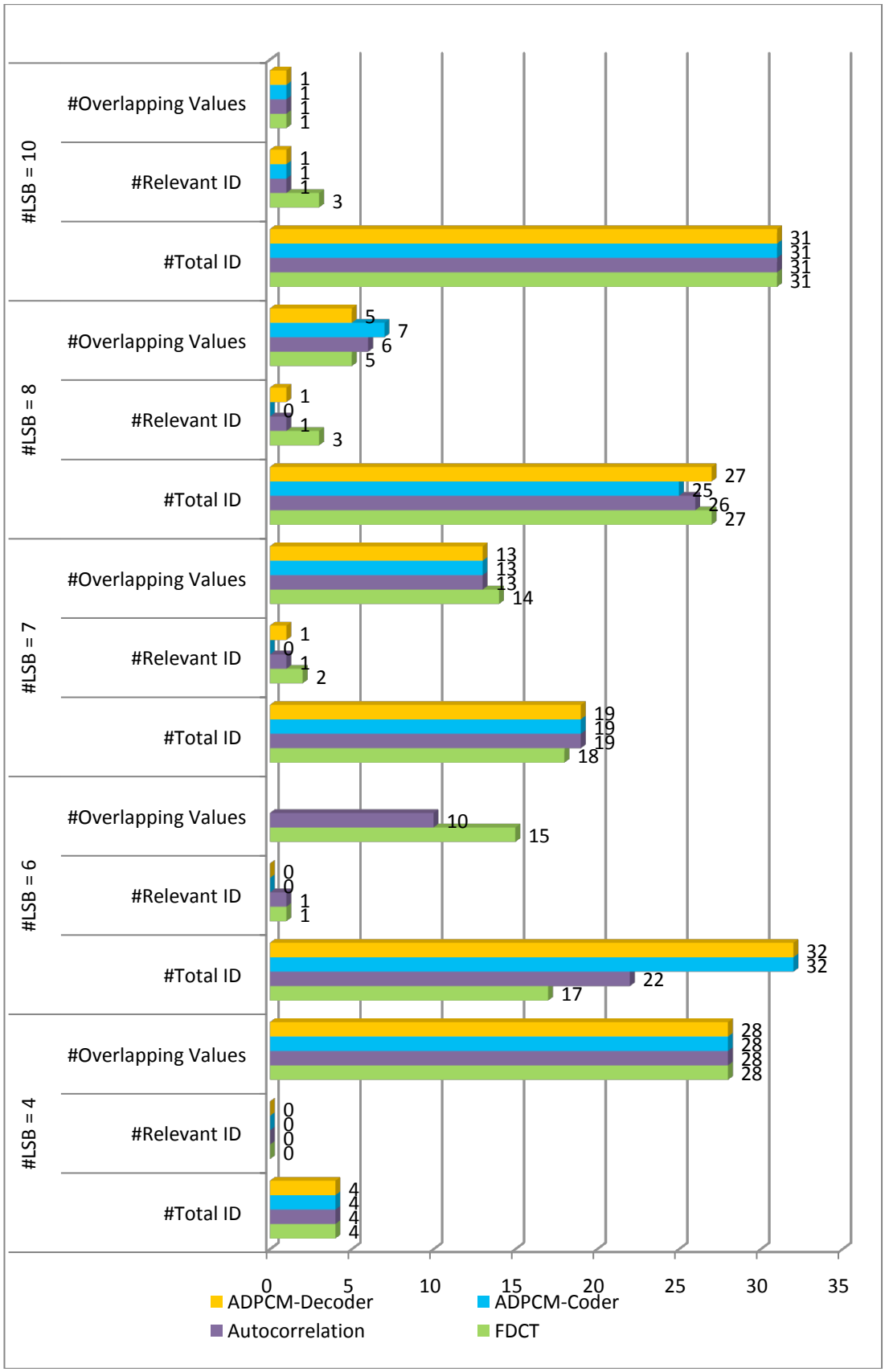

Figure 5-16 - Comparative chart for all non-optimized trace files

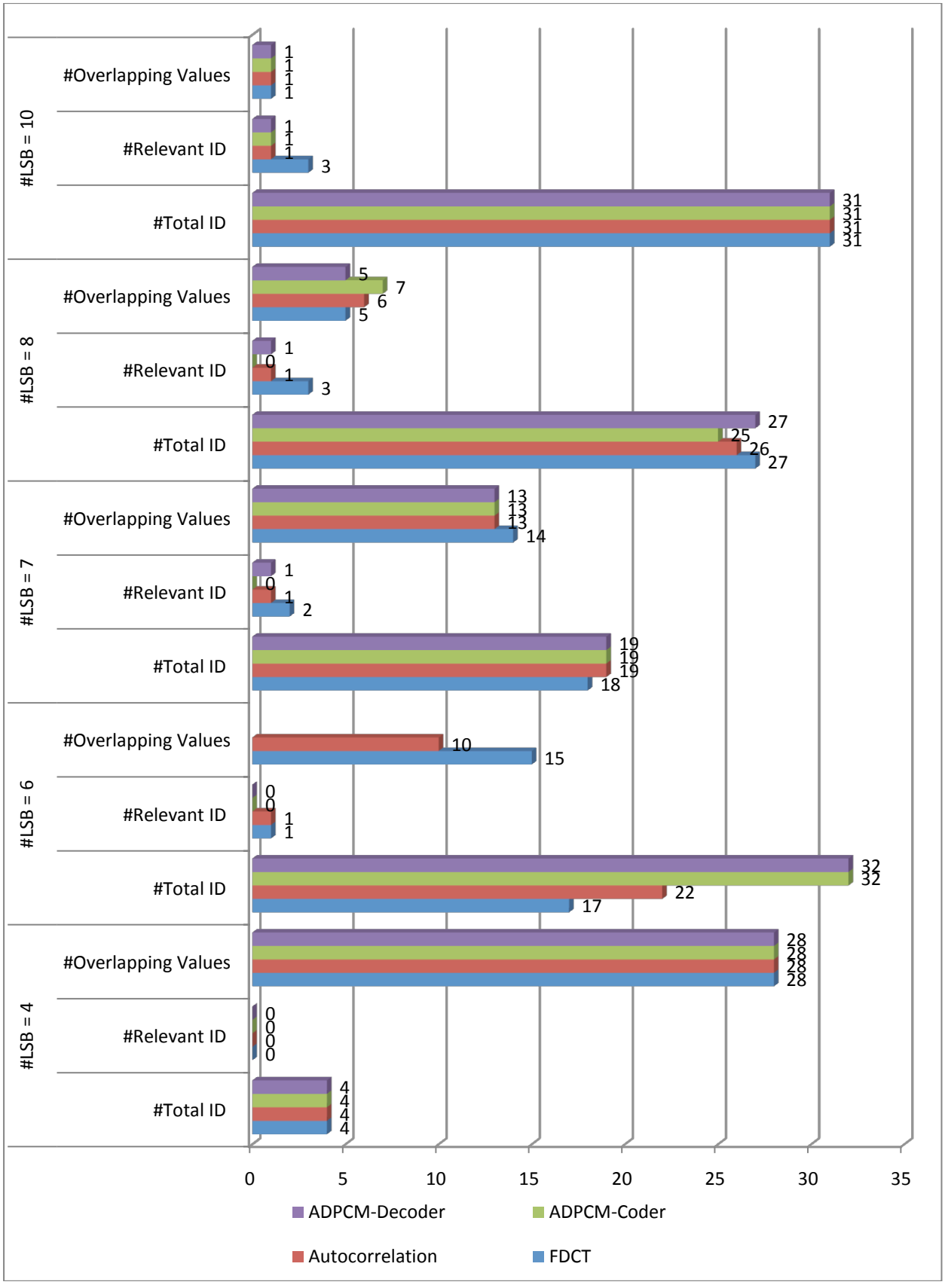

Figure 5-17 - Comparative chart for the optimized version of the trace files

The charts in Figure 5-16 and Figure 5-17 allow for a direct comparison of all the data gathered relating to hash indexing (section 4.2.5). It quickly becomes apparent that in order to assure a minimum level of accuracy at least 8 bits will be necessary, and even so clashes still occur. It is convenient to recall that the 2 LSBs never change (as stated in 4.2.5) and can be discarded. Even so, leaves a result table with a 64 entry size (twice the maximum number of total loops identified by the brute force and brute force-- algorithms).

Chapter 6

Hardware Implementation

In this chapter an idea for a possible hardware implementation will be presented. Despite lacking the time to implement and validate the algorithms in hardware, the base architecture had already been laid in the early stages of this work.

6.1. System integration

As stated in section 4.1 this system was designed considering that it would be directly connected to an instruction bus. This allows the Profiler to read directly from the instruction bus (without the need to filter unwanted data). The block diagram in Figure 6-1 shows how the profiler connects to the microprocessor instruction bus, taking advantage of the Harvard architecture, so that it can monitor the instructions addresses as the microprocessor accesses the BRAM containing the program instructions.

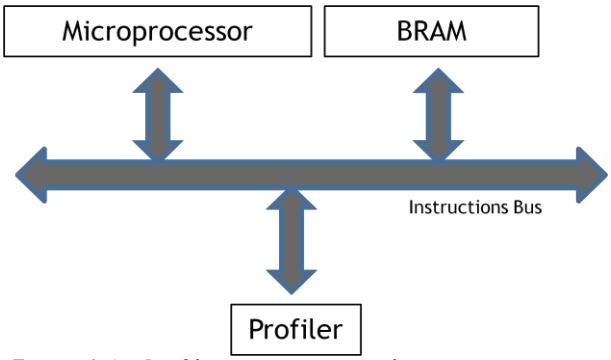

Figure 6-1 - Profiler integration with a microprocessor

6.2. Profiler Hardware Modules

The profiler is made of four modules and a BRAM (Figure 6-2). This separation allows for the identification, buffering, and search algorithm and replacement policy to be interchangeable without the need to alter the whole system.

Module 1 is responsible for the detection of backward jumps. As in the software implementation, whenever an address value is lesser than the previous address value, the module signals the FIFO (module 2) and sends it both address values for storage.

Whenever the search and insertion module (module 3) is available for processing, it downloads the address values from the FIFO. The algorithm chosen for this implementation was the Brute Force--. The advantage of this algorithm over the Brute Force one is that it requires fewer accesses to the BRAM, thus allowing for a faster data processing. Once the address values are located within the BRAM, it updates their values (#Executions, #Iterations, #Age, #FLAG). If the address values aren't in the BRAM, it inserts them into the BRAM on the position pointed by module 3.

In the software implementation, the Replacement Policy module was only activated after the result table became full and was called every time a replacement had to be made. Now module 3 is active from the start and is responsible for pointing to the BRAM address where a new found loop is to be inserted, even when the BRAM is not filled to capacity. This method insures that the waiting time to insert a new Loop in memory is reduced (or null).

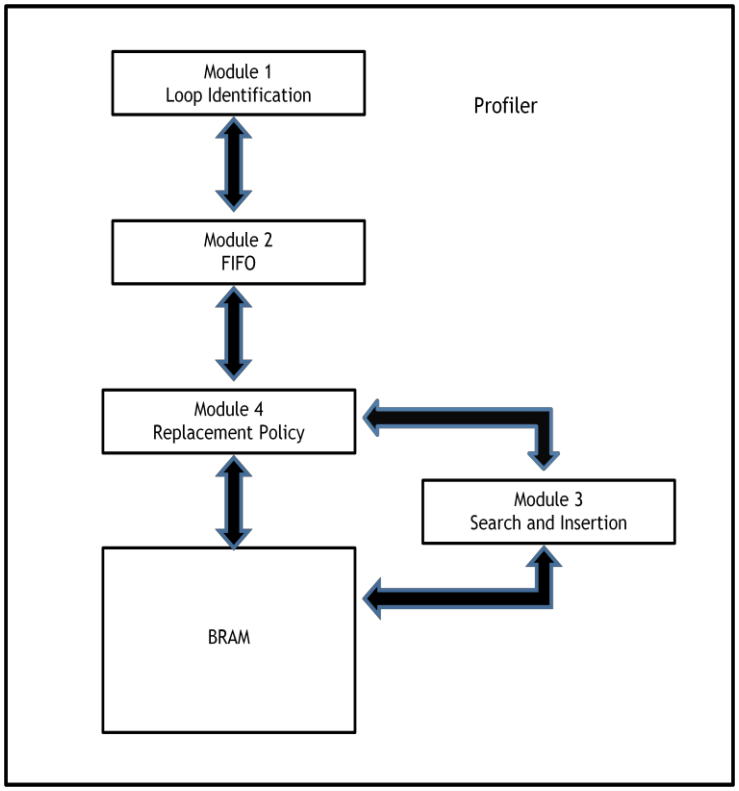

Figure 6-2 - Profiler Block Diagram

6.3. Module specification

In this section an individual description of the modules presented in the previous section is made. This specification is only one of the many possible ones and had not been validated at the time of writing.

6.3.1. Module 1 – Loop Identification

This module implements the algorithm described in section 4.2.1 and depicted in Figure 4-4.It stores the address value currently on the instruction bus (ADDR), compares it with its previous value (ADDRp1). If the current value is lesser than the previous one, then the sbb output goes high and the ADDR and ADDRp1 values are concatenated, so that they can be stored by the FIFO and then to the next module for processing. The sbb output is used to trigger the FIFO Write signal, thus storing the address values on this module output.

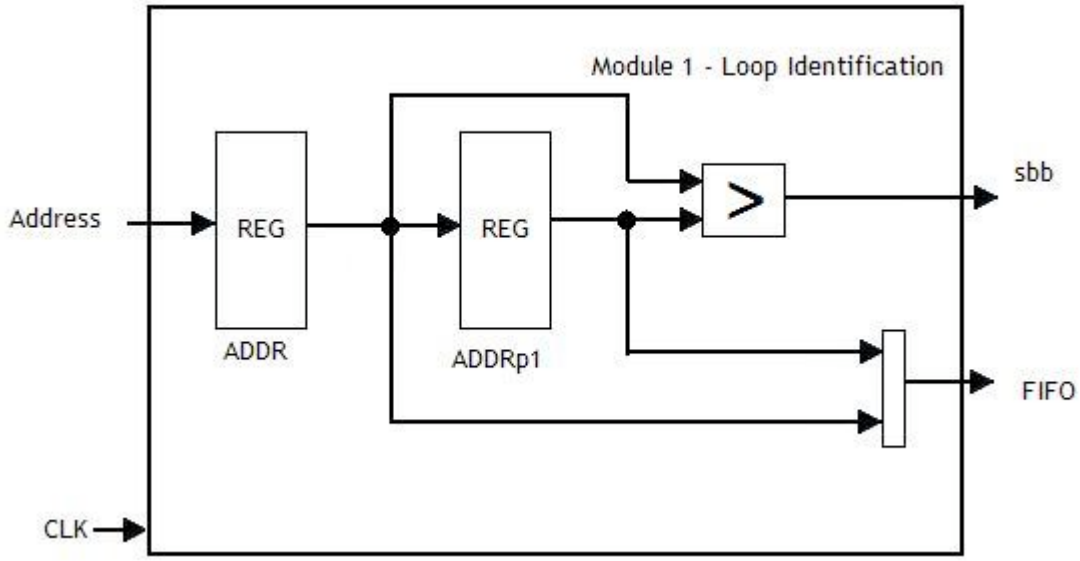

Figure 6-3 - Hardware module 1, Loop identification

6.3.2. Module 2 – FIFO

This module acts as a buffer between the loop identification module and the search and insertion module. Since a new address value appears per clock cycle (though not all are backward jumps) there is very little time for module 3 to search, insert and/or update the information in the BRAM. A First-in-First-out buffer stores the values it receives and then outputs those values (when module 3 requests it) in the same order that the values were received.

Instead of implementing a FIFO, the Xilinx Core Generator was used, to insure a valid and reliable design. The generated IP (Figure 6-4) allows for several degrees of customization, but only the most basic FIFO features were needed.

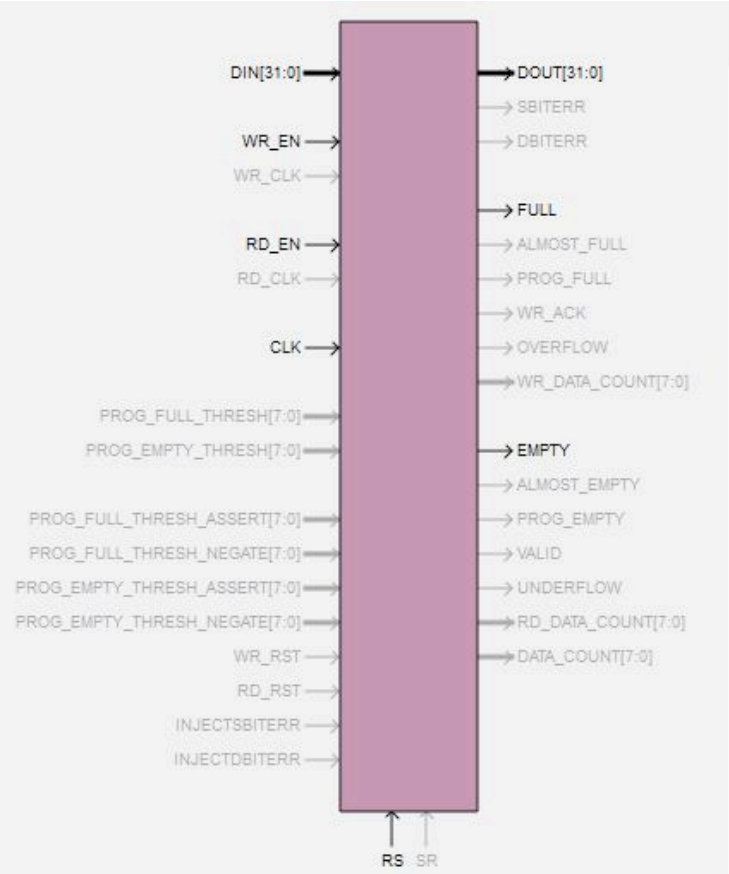

Figure 6-4 - FIFO I/O

6.3.3. Module 3- Search and Insertion

Module 3 is tasked with searching the BRAM for an existing Loop (using the address values as comparison) and either update the values for an existing loop (increment #Executions, #Iterations, etc), or in case of a new loop to insert the new data. The state machine in Figure 6-5 describes how this module operates.

Whenever the FIFO is empty the systems remains in the IDLE state. When a value is available, the state changes to ADDR_WAIT.

The ADDR_WAIT state is needed every time the BRAM address is changed, since there is a one cycle delay between the request and when the actual data is available.

The SAME_IT state checks if the Loop Identified by module 1 is a new Iteration of the current execution. If it is the data is updated and the state changes back to IDLE. If the address values don't match the ones on the last memory position to be accessed, the machine jumps to the SEARCH state (after passing the ADDR_WAIT state).

In the SEARCH state, the entire BRAM is searched to determine if the values are from a new or from an existing Loop. If a match is found (received values are equal to stored values) the execution, Iteration and AGE bits are updated for that Loop. If not the new Loop is inserted in memory. Regardless of being an existing or a new loop, once the search is finished the state is changed to CLEAR_ADDR.

The CLEAR_ADDR state resets the BRAM address to zero to allow the AGE state to alter the AGE bits for all stored Loops. After the BRAM address is set to zero, the state is changed to the ADDR_WAIT state, and set to jump to the AGE state afterwards.

The AGE state is used to decrement the AGE bits of all stored Loops (except for the one just inserted/updated) by one. In each iteration, it decrements (if the stored value is greater than zero), and then jumps to the ADDR_INCREASE state.

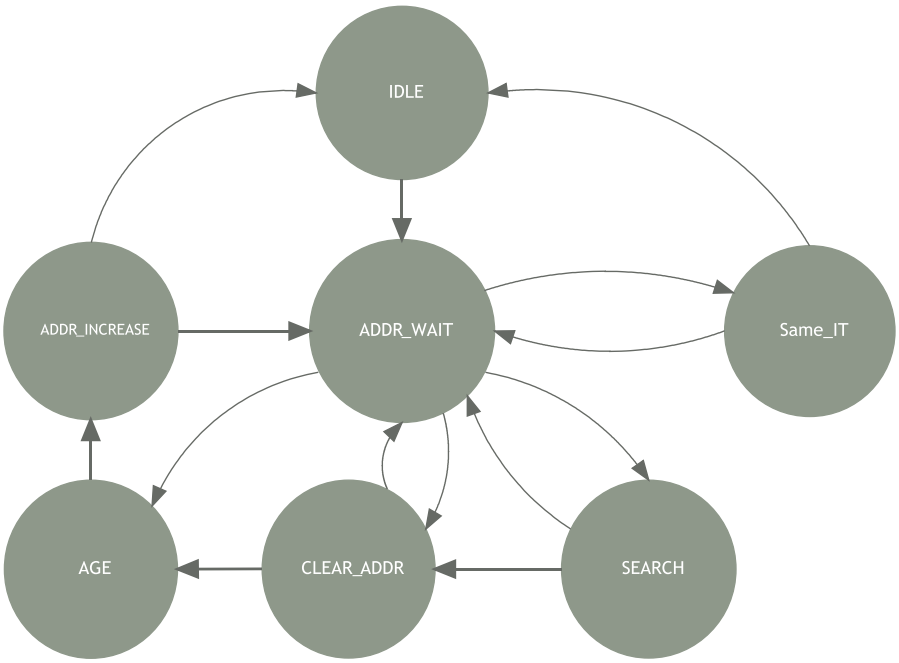

Figure 6-5 - Module 3 State Machine

The ADDR_INCREASE state increases the BRAM address (and jumps to the ADDR_WAIT state before returning to the AGE state) unless the end of the BRAM filled positions have been reached. If so, it returns to IDLE.

6.3.4. Block RAM

In the Software Implementation the result table was stored within a set of arrays. With hardware the BRAM was the most obvious and simple solution. The used BRAM has a width of 64 bits. The depth of the BRAM varied depending on which test parameters were being used (the values were 8, 16 and 32 bits).

The Loop data was stored in the following fashion:

Bits	Data
63:48	Origin
47:32	Destiny
31:21	#Executions
20:5	#Iterations
4:0	Age

Table 13 - Data placement in the BRAM

The data on Table 13 is an example and does not mean that the data is always organized like that. To increase the performance of the replacement policy, the AGE value has to have the same number of bits as the BRAM address (i.e. a 32 depth BRAM has a 5 bit address, so the AGE would be a 5 bit value for this case). The MSB are, in this order, the Origin address, followed by the Destiny, number of Executions, number of Iterations. The LSB are always the AGE of the Loop.

Like the FIFO the BRAM was not implemented, but obtained using the Xilinx Core Generator. Figure 6-6 shows a schematic for the BRAM I/O ports.

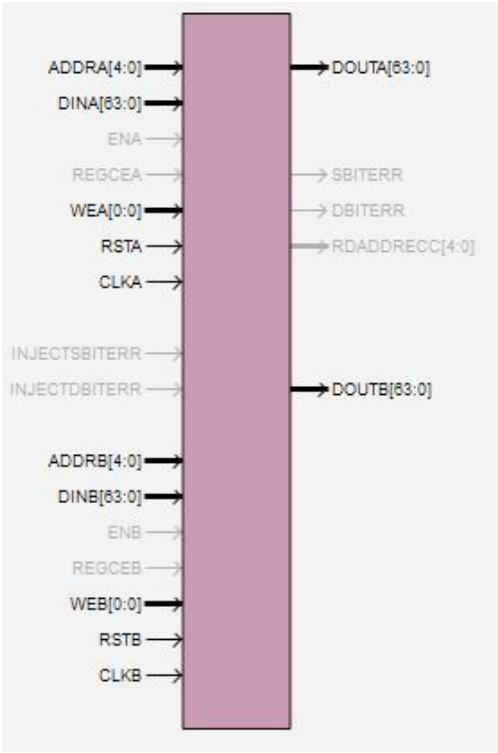

Figure 6-6 - BRAM I/O

To allow module 4 to browse the values stored in memory to locate the position where the next new Loop will be stored while module 3 searches for known Loops to update data a dual port BRAM was used.

6.3.5. Module 4 – Replacement policy

Module 4 is responsible for determining where in memory a new loop is to be inserted. In the software implementation the module responsible for the replacement policy was only activated for the first time after the result table was full and after that occurred, whenever a new loop was detected, the module would search the memory to find the best candidate for replacement (the one with the lowest AGE value).

The ReplaceAgeSimple criteria was the one with the less accurate criteria, however it is also the smallest, and the one that takes up less resources when searching the memory, making it the most suitable one for hardware implementation.

To further improve the ReplaceAgeSimple policy, a few changes were made. Instead of looking for the lowest AGE value, it would be simpler to merely search for a Loop with 0 AGE. However, it would be necessary to insure that at least one Loop would need to have 0 AGE at any given time. By setting the AGE number of bits to the same number of bits of the BRAM address, that objective is achieved.

Even when the BRAM is still not full, this module is responsible for determining the next position where to store a new Loop.

The main reason behind using a different module for the replacement policy is that it allows the search for the new writing position to go on without hampering the "regular" search of the BRAM for Loops. By using separate modules and a dual port BRAM, the system has a smaller latency.

6.4. Hardware Implementation Requirements

The hardware requirements for modules 1, 3 and 4 are in Table 14:

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	178	1920	9%
Number of Slice Flip Flops	174	3840	4%
Number of 4 input LUTs	325	3840	8%
Number of bonded IOBs	18	173	10%
Number of GCLKs	1	8	12%

Table 14 - Device Utilization Summary [17]

The Spartan 3 FPGA has an 18 Kb BRAM that can be configured to house the FIFO and BRAM needed in this project. The ratio between memory and FIFO size depends on several parameters. If the Profiler clock speed is lower than the microprocessor a larger FIFO will be required. Or if a very big program is being profiled, it might be wiser to configure most of the BRAM to store the results.

6.5. Hardware Implementation Results

In this section, similarly to Chapter 5, the results obtained with the Hardware implementation are displayed. The effect that the FIFO and BRAM depth have on the success rate is also displayed.

The data on Table 15 was obtained with a 32 write depth BRAM, a 16384 write depth FIFO and with a clock of 100 MHz for the Profiler and the microprocessor. The purpose of this

experiment was to insure a 100% success rate on all benchmarks (all relevant Loops had to be present in memory at the end of runtime).

Code\Benchmark	FDCT	Autocorrelation	ADPCM - Coder	ADPCM - Decoder
Optimized	4	6	1474	1474
Not Optimized	72	76	74	74

Table 15 - FIFO Sizes required by each of the benchmarks for a 32 write depth BRAM

Table 15 suggests that in order to insure that no data is lost due to the latency caused by modules 3 and 4 a 2048 write depth FIFO is required.

The results on Table 16 were obtained with a 8 write depth BRAM, a 2048 write depth FIFO and with a clock of 100 MHz for the Profiler and the microprocessor. This experiment still yielded a 100% success rate.

Code\Benchmark	FDCT	Autocorrelation	ADPCM - Coder	ADPCM - Decoder
Optimized	4	6	1980	1395
Not Optimized	45	52	50	50

Table 16 - FIFO Sizes required by each of the benchmarks for an 8 write depth BRAM

When using a BRAM with a 4 write depth BRAM none of the relevant Loops was identified (as expected).

6.6. Hardware implementation result analysis

When using a smaller BRAM, the FIFO size can also be downgraded, since the time spent searching the BRAM is smaller as well. The results found for the optimized ADPCM-CODER and ADPCM-Decoder were surprising, however the relevant loops are located closer to the end of runtime, they have the largest number of iterations(when comparing to the other benchmarks), and have only 4 instructions per iteration. These might be the reasons for the apparent disparity between the values obtained for these benchmarks when comparing with the rest.

The Spartan has several BRAMS available. The BRAM and FIFO module use the space that one (or several, depending on size) of those BRAM provide however the FPGA allocates the entire BRAM regardless of how much of the 18Kb BRAM is being used by the design. The most efficient design would be to use an entire BRAM, dividing the space between the FIFO and the BRAM. The Spartan's 18Kb BRAM is made of two 9Kb blocks where 1K is for parity, leaving two blocks of 8Kb available. A 4K FIFO and a 4K BRAM would use only one block, and provide ample buffering and storing space.

Chapter 7

Conclusions and future work

The work conducted during this thesis had the objective to identify program control flow structures in a dynamic non-intrusive fashion. This chapter discusses the achievement of the proposed objectives of this thesis and suggests on the future work that can be developed in this area.

7.1. Achieved objectives

As was emphasized in Chapter 1 an embedded system has to have excellent performance given today's market. Given the choice between developing a new system or optimizing an existing one, optimizing a system allows for significant improvement. This improvement gives an existing system an edge that can be used to compete with more recent systems (performance wise). With the wide variety of embedded systems and their sensitivity concerning time constraints, a non-intrusive dynamic profiler allows for a portable, efficient way of detecting critical regions of the program being run and choosing the ones better suited to be optimized.

Considering that the critical regions are typically made of loops, the first objective was to determine if those loops could be detected. By monitoring the instruction addresses in the instruction bus, loops were successfully identified by detecting when a backward jump occurred. Not all backward jumps were loops. Some were unconditional branches necessary to the program flow. A benchmark that contained 3 loops in the C source file had 32 after compiling without optimization and 13 with level 2 compiler optimization.

Once it loops were being correctly identified, the next objective was to store and collect relevant data concerning them. Not all loops are critical, and to make the distinction between them it was necessary to determine their parameters.

The implemented algorithms successfully identified the loops contained in the tested benchmarks. Both brute force and brute force-- algorithm performed as expected, showing consistent results.

In order to determine the viability of hardware portability, it was necessary to develop replacement policies. The reason for this is that the memory requirements to store all identified loops would probably be too high.

From the three tested replacement policies, the one that proved to be the most effective was the ReplaceAgeExecIter (for a result table the size of a quarter of the total number of loops in one of the benchmarks it was the only one that kept all relevant loops stored until the end of runtime.

The implemented methods proved that gathering information on loops during the execution of a program is possible and can be made in an efficient manner.

The results obtained with the hardware implementation confirmed that with a little overhead it is possible to accurately profile a running program in a microprocessor. With the proper balance between the FIFO and BRAM size the Profiler can accurately identify and store Loops even when working at half the clock speed of the microprocessor.

Despite being the less accurate of the replacement policies, the ReplaceAgeSimple has the advantage of being very “light”, particularly the hardware version, that just looks for the Loop(s) with zero AGE. Testing showed that the Replacement Policy managed to have the address of the Loop to be replaced ready before it was needed on most cases.

7.2. Future work

One of the main difficulties in profiling Loops is that the way in which the program is structured plays a heavy role on how easy it can be profiled. If many small loops are close to the end of runtime, the profiler will have difficulties in accurately identifying and storing all the data before the program ends. This does not occur with a recursive algorithm, but it can still be troublesome.

One of the methods suggested in this thesis, but that was not implemented was the hash indexing algorithm. This method would be the extremely fast what makes it a very good candidate for a Hardware implementation. A new replacement policy specifically designed for this method, along with the use of an associative memory may be enough to avoid the data loss from discarding the MSB of the address.

Since only four benchmarks were used, and none of them were recursive, it would be interesting to widen the testing field, to see how the Profiler behaves when profiling a

recursive program. By using more benchmarks, it would allow to develop a more generic size for the FIFO and BRAM, since that with only four benchmarks having been used, there is no assurance that the system may accurately profile a wide spectrum of programs with the FIFO and BRAM sizes proposed.

Another application might be to implement and test the organized insertion and hashing index algorithms. In the particular case of the hashing index, which has the most advantages for Hardware, a well thought replacement policy might be able to improve the efficiency of the method.

References

[1] Ajay Nair and Roman Lysecky, "Non-Intrusive Dynamic Application Profiler for Detailed Loop Execution Characterization", Oct. 2008

[2] Ann Gordon-Ross and Frank Vahid, "Frequent Loop Detection Using Efficient Nonintrusive On-Chip Hardware", Aug. 2005

[3] Roman Lysecky, Susan Cotterell, Frank Vahid, "A Fast On-Chip Profiler Memory", Jun. 2002

[4] On-Line Xilinx Data Sheet DS003
http://www.datasheetpro.com/771762_download_DS003_datasheet.html

[5] João Carlos dos Santos Ferreira, "Ferramentas para Simulação e Depuração de Sistemas Baseados em Processadores FireWorks," Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores, Instituto Superior Técnico (IST), Universidade Técnica de Lisboa (UTL), 2009 (em preparação).

[6] http://en.wikipedia.org/wiki/Binary_search

[7] Microblaze reference manual,
http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[8] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, 2000

[9] 18 S.C. Govindarajan, G. Ramaswamy, and M. Mehendale, "Area and Power Reduction of Embedded DSP Systems Using Instruction Compression and Re-Configurable Encoding," Proc. Int'l Conf. Computer Aided Design, 2001

[10] 22 Y. Ishihara and H.A. Yasuura, "A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors," Proc. Design Automation and Test in Europe, Mar. 2000

[11] 10 N. Bellas et al., "Energy and Performance Improvements in Microprocessor Design Using a Loop Cache," Proc. Int'l Conf. Computer Design (ICCD), pp. 378-383, 1999

[12] 17 A. Gordon-Ross, S. Cotterell, and F. Vahid, “Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example,” IEEE Computer Architecture Letters, vol. 1, Jan. 2002

[13] Gordon-Ross, A., F. Vahid. Frequent Loop Detection using efficient Non-Intrusive On-Chip Hardware. IEEE Transaction on Computers, Vol 54, October 2005

[14] Embedded Development Kit - http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/edk_ctt.pdf

[15] Ebcioglu, K., E. Altman, M. Gschwind, S. Sathaye, Dynamic Binary Translation and Optimization. Transactions on Computers, Vol 50, June 2001.

[16] Warp Processing: Dynamic Translation of Binaries to FPGA Circuits, F. Vahid, G. Stitt, and R. Lysecky. IEEE Computer, Vol. 41, No. 7, July 2008, pp. 40-46.

[17] ISE - http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/manuals.pdf