Compressed instruction cache architecture for high-performance embedded RISC systems

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/33689

Publisher: © Elena Georgieva Nikolova

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.
Please note that fines are charged on ALL overdue items.

EOR REFERENCE ONLY
Compressed Instruction Cache Architecture for High Performance Embedded RISC Systems

by Elena Georgieva Nikolova

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy

of

Loughborough University

July 2007

© Elena Georgieva Nikolova
The influence of embedded systems is felt in many aspects of our daily lives; being particularly apparent in consumer electronics and automotive products. Customer demand and rapid advances in the complexity of the underlying technology has enabled the introduction of new systems and services that were simply not feasible just a few years ago.

Although the cost of embedded systems is an important design parameter, their development is also affected by performance and functionality. The performance issue is traditionally addressed by the design of faster microprocessors, but more recently by the exploitation of parallelism (for example, vector units and very long instruction word processors), as well as special purpose hardware architectures, such as graphics processing units and network cards. In such systems, however, the main performance bottleneck is often the memory hierarchy, particularly in systems with complex memory access arbitration, where read or write operations to the main memory could result in delays of thousands of cycles. Although the widespread use of cache memories aims to alleviate this effect to some extent, memory access penalties remain a significant drain on performance. Functionality is closely related to the memory capacity available, particularly in portable systems such as mobile phones and hand-held games consoles.

The work described in this thesis includes a comprehensive analysis of code size and performance issues of embedded reduced instruction set computer architectures. The main contribution is a novel lossless compression-based solution (both hardware architecture and software tools) that generates significant reductions both in the memory requirement of the executable and the number of instruction cache misses. The new solution is quantitatively evaluated, demonstrating improvements in system performance for a wide variety of embedded applications, but particularly in high-end, real-time applications, such as high-definition televisions, mobile platforms and embedded media processing systems. The benefits
obtained are twofold. Firstly, code compression gives the opportunity to increase the number of features implemented in the system without the need to increase the memory budget. Secondly, filling the instruction cache with compressed instructions enhances its effective capacity and consequently improves performance due to the increased likelihood of finding the required instruction in the cache.
Acknowledgements

I would like to thank to ARC International Ltd for financing this research and providing the necessary equipment to carry it out.

I am deeply indebted to my supervisor Dr. David Mulvaney for his support, guidance, sound advice and much valued input throughout the extended period of this research. Special thanks to Dr Vassilios Chouliaras for the stimulating suggestions and his technical expertise that he so generously shared with me.

Thanks to my colleagues and friends at Loughborough University for making such a challenging period special and fun.

My deepest gratitude to my parents, Zhika and Georgi, for their love, understanding and constant encouragement. The same goes to my new family, Soledad, Adolfo and Cesar, who have always been so supportive and good to me.

Lastly, and most importantly, I would like to thank to my husband, Dario, for his inspiration and support, and for sharing this and everything else in my life. It is to him that I dedicate this work for he is truly the most remarkable person that I have ever met...

... and, of course, to my little girl who keeps me company while I write these lines, and who is to be born in a few months.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEU</td>
<td>Address Extraction Unit</td>
</tr>
<tr>
<td>ALDC</td>
<td>Adaptive Lossless Data Compression</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic and Logic Unit</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ATU</td>
<td>Address Translation Unit</td>
</tr>
<tr>
<td>BBC</td>
<td>Burst Buffer Controller</td>
</tr>
<tr>
<td>BTC</td>
<td>Branch Target Cache</td>
</tr>
<tr>
<td>CAM</td>
<td>Content Addressable Memory</td>
</tr>
<tr>
<td>CCRP</td>
<td>Code Compression RISC Processor</td>
</tr>
<tr>
<td>CCS</td>
<td>Code Compression Suite</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>CLB</td>
<td>Cache Line Address Lookaside Buffer</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block (Xilinx)</td>
</tr>
<tr>
<td>COF</td>
<td>Change Of Flow</td>
</tr>
<tr>
<td>CPC</td>
<td>Compressed Program Counter</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycles Per Instruction</td>
</tr>
<tr>
<td>CR</td>
<td>Compression Ratio</td>
</tr>
<tr>
<td>DCLZ</td>
<td>Data Compression according to Lempel and Ziv</td>
</tr>
<tr>
<td>DU</td>
<td>Decoding Unit</td>
</tr>
<tr>
<td>DDR</td>
<td>Double Data Rate</td>
</tr>
<tr>
<td>DMC</td>
<td>Dynamic Markov Compression</td>
</tr>
<tr>
<td>ELF</td>
<td>Executable Linkable Format</td>
</tr>
<tr>
<td>EPIC</td>
<td>Efficient Pyramid Image Coder</td>
</tr>
<tr>
<td>FCRAM</td>
<td>Fast Cycle RAM</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>HLL</td>
<td>High Level Language</td>
</tr>
<tr>
<td>IBC</td>
<td>Instruction Based Compression</td>
</tr>
<tr>
<td>IC</td>
<td>Instruction Count</td>
</tr>
<tr>
<td>ICache</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ICE</td>
<td>In-Circuit Emulators</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISS</td>
<td>Instruction Set Simulator</td>
</tr>
<tr>
<td>LAB</td>
<td>Logic Array Block</td>
</tr>
<tr>
<td>LAT</td>
<td>Line Address Table</td>
</tr>
<tr>
<td>LC</td>
<td>Logic Cell (Xilinx)</td>
</tr>
<tr>
<td>LE</td>
<td>Logic Element (Altera)</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last-In First-Out</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LTP</td>
<td>Long Term Prediction</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>LZ</td>
<td>Lempel-Ziv (compression algorithms)</td>
</tr>
<tr>
<td>MIPS</td>
<td>Microprocessor without Interlocked Pipeline Stages</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>PAG</td>
<td>Pre-fetch Address Generator</td>
</tr>
<tr>
<td>PBC</td>
<td>Pattern Based Compression</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PPM</td>
<td>Prediction by Partial Match</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced instruction set computer</td>
</tr>
<tr>
<td>RPE</td>
<td>Residual Pulse Excitation</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating Systems</td>
</tr>
<tr>
<td>SADC</td>
<td>Semi-Adaptive Dictionary Compression</td>
</tr>
<tr>
<td>SAMC</td>
<td>Semi-Adaptive Markov Compression</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SRS</td>
<td>Subroutine Return Stack</td>
</tr>
<tr>
<td>TBC</td>
<td>Tree Based Compression</td>
</tr>
<tr>
<td>TIA</td>
<td>Target Instruction Addresses</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
</tbody>
</table>
Contents

List of Figures x

List of Tables xiii

1 Introduction 1

1.1 Historical overview ..................................................... 2
1.2 Problem identification and existing solutions ......................... 3
1.3 Research aims and objectives .......................................... 5
1.4 Summary of achievements ............................................... 5
1.5 Structure of Thesis .................................................... 6

2 Literature Review 8

2.1 Embedded systems .......................................................... 8
   2.1.1 ARCanel development board [9] .................................. 10
   2.1.2 PNX8550 ............................................................. 10
2.2 Data compression ............................................................ 11
   2.2.1 Modelling .............................................................. 14
   2.2.2 Coding ................................................................. 15
2.3 Compiler optimisations ..................................................... 16
2.4 Hybrid RISC Architectures ............................................... 18
   2.4.1 MIPS16 ............................................................... 18
REFERENCES

A EXECUTABLE AND LINKING FORMAT (ELF) 142
B COMPRESSION RATIO RESULTS 144
C PUBLICATIONS 148
List of Figures

2.1 Data compression taxonomy .............................................................. 12
2.2 An example of compression process ..................................................... 13
2.3 Mapping of hybrid instructions ............................................................ 18
2.4 Code Compression RISC Processor, [15] .............................................. 27
2.5 Line Address Table Entry ..................................................................... 27
2.6 CLB Organisation [27] ........................................................................ 29
2.7 Index table mapping of TIAs to compressed memory [7] ......................... 30
2.8 Basic Spartan-IIIE Family FPGA Block Diagram [31] ............................ 32
2.9 Virtex-II Architecture Overview [32] ..................................................... 34
2.10 Cyclone EP1C6 Device ....................................................................... 35
3.1 Entropy Tool Pseudo Code ................................................................... 40
3.2 Ratio between entropy and actual programs' sizes ................................. 47
3.3 Compression performance of a range of algorithms applied to the test-bench programs ........................................................................... 48
3.4 Compression ratio results for different input block sizes ....................... 50
3.5 Compression ratios for various dictionary sizes ........................................ 51
4.1 Uncompressed to compressed file conversion ......................................... 58
4.2 Codeword format ................................................................................... 59
4.3 Compression ratios achieved for different dictionary sizes ...................... 60
4.4 Relative COF instructions format .......................................................... 62
4.5 PC update and loop detection mechanism [42] ....................................... 63
4.6 Calculating the branch appendixes ....................................................... 64
4.7 Schematic of the hardware/software co-design process including compression 68
4.8 Compression stages ............................................................................... 69
5.1 ARCTangent-A4 pipeline stages ............................................................. 72
5.2 Decompressor interface ......................................................................... 73
5.3 Modification of the pipeline of ARCTangent-A4 for decompression ........... 74
5.4 Architectural overview of the decompressor .......................................... 76
5.5 Functional diagram of the decompressor’s DU ...................................... 77
5.6 Basic diagram of the decoding unit ....................................................... 78
5.7 Operation of the input buffer .................................................................. 79
5.8 The instruction word in the decoder ...................................................... 80
5.9 Decode control unit state diagram .......................................................... 81
5.10 Start up behaviour of the address translation unit .................................... 82
5.11 Functionality of the ATU during sequential execution ............................. 83
5.12 Functionality related to branch and branch and link instructions ............. 85
5.13 Jump related functionality of the ATU ................................................... 86
5.14 Main modules of the ATU ..................................................................... 87
5.15 Factorial calculation illustrating recursive function calls ....................... 89
List of Tables

2.1 Four examples of embedded systems with approximate attributes, Koopman[8] .... 9
2.2 Results achieved by Araujo et al. [18]........................................................ 23
2.3 Main complexity parameters for the selected Spartan-IIIE FPGA............... 32
2.4 Virtex2 FPGA main complexity parameters.............................................. 33
2.5 Selected Cyclone FPGAs main complexity parameters [33].............................. 35
3.1 Entropies for a range of programs compiled for the ARCTangent-A4............................. 46
5.1 Processor interface signals ............................................................................ 73
5.2 Targeted FPGAs ......................................................................................... 94
5.3 Decompressor configuration parameters and values used for synthesis............. 95
5.4 Decompressor's clock frequencies for the targeted devices............................... 95
5.5 Complexity figures for Cyclone EPIC6 device ........................................... 96
5.6 Complexity figures for Xilinx devices .......................................................... 96
6.1 Summary of files that may be found in an application's project directory .......... 108
7.1 $IC_{TOTAL}$ and pipeline stall cycles for the test bench applications ........................... 119
7.2 Number of instruction cache misses for uncompressed memory systems ............... 120
7.3 Number of instruction cache misses for compressed memory systems................. 122
7.4 COF instructions and coverage of BTC hardwired entries.................................. 127
7.5 Performance improvement for systems with different number of ICache refill cycles 131
B.1 Compression ratio results for ADLC.................................................................. 144
B.2 Compression ratio results for LZS................................................................... 144
B.3 Compression ratio results for X-MatchPro for dictionary size of 128 bytes......... 145
B.4 Compression ratio results for X-MatchPro for dictionary size of 256 bytes........ 145
B.5 Compression ratio results for X-MatchPro for dictionary size of 512 bytes........ 145
B.6 Compression ratio results for X-MatchPro for dictionary size of 1024 bytes....... 146
B.7 Compression ratio results for X-MatchPro for dictionary sizes of 2048 and 4096 bytes................................................................. 146
B.8 Compression ratio results for PPMZ......................................................... 146
B.9 Compression ratio results for DCLZ........................................................ 147
B.10 Compression ratio results for CLB......................................................... 147
Chapter 1

INTRODUCTION

Embedded systems play a substantial and ever-increasing part in our lives. Without such systems, most consumer electronics devices would not be feasible and many modern systems, such as those developed in the aerospace and automotive industries would be significantly less advanced in their features. While the physical dimensions of many devices have reduced considerably, the performance and features expected by consumers have increased. Consequently, an ever-increasing quantity of code must be developed by software engineers for inclusion in embedded systems that are limited in terms of their resources, such as space, memory, speed, and power. This research concentrates on RISC processors that are commonly found at the core of embedded systems. In particular, the work focuses on the widely recognised problem [1] of such architectures, which has received significant amount of attention in the past decade: namely the inefficient use of memory by their executables.

Although a number of solutions have been adopted, they usually tend to improve the memory usage, at the expense of performance. This thesis, describes a novel solution based on a holistic approach, where the requirements of modern embedded systems in terms of code size, performance, and power consumption awareness are taken into account. The proposed solution is based on a lossless compression method, where the boundary between compressed and uncompressed address spaces is located between the instruction cache (ICache) and the processor core. In this way, the ICache holds compressed instructions, and therefore its effective size increases, thus reducing the ICache miss ratio.

In this chapter, a brief historical overview of the development and adoption of RISC architectures is presented, followed by the identification of the problem, objectives and aims of this research. The last section of this chapter summarises the structure of the thesis.
1.1. Historical overview

In the late 1960s, due to the significant increase in the code size of the developed applications, for the first time software costs began growing faster than hardware costs [1]. High-level programming languages (HLLs) had already appeared, but still, due to memory limitations and underdevelopment in compiler technology, most programs were written in assembly. In the following 15 years, in order to alleviate the software crisis, the major efforts mainly concentrated on developing new software-oriented processor architectures. Probably, the most significant representative of these architectures was the DEC VAX, with a highly orthogonal instruction set that provides mapping for most high-level language statements into a single instruction [1]. The implementation of architecture of such complexity was a significant advance for its time, but it suited the limited memory resources and state-of-the-art compiler technology available.

In 1975, IBM began a project to build the first reduced instruction set computer (RISC), the 801, which was not announced until 1982. In 1980, at Berkley, the RISC-I and RISC-II processors were designed [1]. At around the same time, at Stanford University, the MIPS (microprocessor without interlocked pipeline stages) processor was developed. These first RISC computers all had the basic features that are still common in current RISC processors, namely a simple load-store architecture, efficient pipelining, fixed-length instructions (typically 32-bit), and compiler-based static scheduling. By the end of the 1980s, most of the major processor manufacturers, including HP, Sun Microsystems, Apple, IBM and Motorola had developed a range of microprocessors based on the RISC architecture. These were rapidly adopted in server and desktop applications and these and their descendants have been widely used ever since. However, the embedded market proved to be rather slow in migrating to the 32-bit RISC architectures, and only in the early 1990s, when demand increased for applications requiring enhanced performance, did the technology become widespread. The emergence of Systems-on-Chip (SoC) technologies, based on RISC cores, in the late 1990s, met the demand from the consumer electronic market for high performance and low power.

Today, almost every modern processor is based on the RISC architectural principles, with the principal exception of the 80x86 families of microprocessors. The advantages and disadvantages of both RISC and CISC (complex instruction set computer) architectures have been thoroughly studied and documented [1]. In today's rather saturated and highly
competitive market, the number of features that can be offered by a product is often seen as one of the main differentiating factors. To support such demand, additional functionality and more applications are under continuous development, requiring more memory and hence increasing cost. Furthermore, in many embedded systems such as mobile phones or real-time control systems, the memory contains a substantial portion of the solution's cost. As additional memory consumes more power, optimising memory usage is of huge importance, especially for portable and handheld devices.

1.2. Problem identification and existing solutions

The RISC computer paradigm has determined, to large extent, the way that microprocessors are currently engineered. Rather than following the design rules of their time, their architectures combined simplicity (e.g. fixed-length instruction encoding), efficiency (e.g. efficiently designed pipelining) and performance. Such a combination of characteristics makes them particularly suitable for embedded system and System-on-Chip (SoC) architectures, where area constraints and power consumption are often as important as the performance of the final solution. However, RISC architectures are not exempt from drawbacks, the principal one being poor code density. Considerable effort has been expended in the improvement of compiler technology, in order to better match the target executable requirements thereby generating smaller and more efficient code, and significant advances have been made in terms of local optimisations (within the scope of functions). Conversely, the quantity of software that needs to be run on modern embedded products (such as MP3 player, mobile phones and gaming platforms) is growing rapidly in order to support the ever-increasing number of features demanded by an extremely competitive market. Thus, applications that only a few years ago required a small number of assembler or C functions for their implementation now require full scale software engineering programs, some of which require the code to be partitioned for simultaneous execution on a number of embedded processors. Current compile technology is not sufficiently advanced for the optimisation of such a parallel set of executables. Bearing in mind that embedded applications are constrained in terms of physical dimensions and unit cost, memory is often a scarce resource in embedded systems. Therefore, efforts to improve the code density of RISC processors date back to the very first embedded
implementations, and have generally followed one of the three main strategies described below.

**Compiler technology improvements**

A range of techniques are available in most compilers when they are configured to optimise the code size of an application. These include *dead or unreachable code elimination* (which strips the programme of redundant code) and *global and local sub-expression elimination* (which replaces two instances of the same computation by a single copy). A more comprehensive discussion of these and other compiler optimisation techniques are presented in Chapter 2. The main advantages of allowing the compiler to reduce the code size are the automation of the process and that no additional run-time support in terms of software or hardware is required. However, the reduction ratios that compilers achieve are rather moderate compared with compression, and some of the techniques, such as sub-expression elimination, can seriously reduce the performance of the generated executable [1, 2].

**Hybrid instruction set architecture (ISA)**

This approach has been taken by most RISC manufacturers [3], and it is based on the development of a 16-bit subset of the previously purely 32-bit instruction set architecture (ISA). The implementation of a hybrid ISA requires the development of a new set of software tools (compiler, linker and debugger), together with the implementation of dedicated hardware to support the decoding of the new 16-bit instructions and a method for switching between ISAs. Some examples of this technology are: Thumb (from ARM Ltd.) [4], MIPS-16 (from MIPS) [5] and ARCompact (from ARC Int.) [6]. However, the 16-bit ISAs do not yield a 50% reduction in code size, nor is the 32-bit performance retained. ARM, for example, claims that the Thumb technology decreases code size by about 30%, but that this comes at the price of a reduction in performance (execution time increase) of up to 26% [4].

**Code compression**

The third approach uses code compression to achieve higher code density. Since the early 1990s, a number of compression algorithms have been adapted for application to embedded code, but no single best approach is apparent at present. This technique achieves a more substantial code reduction than the other two methods discussed above and was exploited by IBM in the development of its CodePack system [7] (see Chapter 2).
1.3. Research aims and objectives

This research is concerned with alleviating the code size inefficiency typically found in embedded RISC implementations. Over the past twenty years, many solutions have been proposed, but in most cases the resulting code density improvements, which often were moderate, came at the cost of serious performance degradation. Therefore, there is a clear need to further investigate this field with the aim of identifying a suitable solution that provides good code density while preserving (or ideally improving) the overall performance of the system. Thus, the aim of this research is to obtain such a solution and in order to achieve this goal, several objectives are apparent.

- To highlight the current state-of-the-art in code reduction, by carrying out a comprehensive literature review of the field, including the previously mentioned techniques of compiler optimisations, hybrid architectures and code compression.

- To quantify the redundancy present in RISC code, by studying the suitability of code compression to be used at the core of a novel code reduction solution for high-performance embedded RISC applications.

- To implement a suitable simulation model that would allow the validation of the proposed solution and to quantify its impact on a system’s performance.

- To develop the necessary software tools for compressing RISC code and the required hardware architectures for supporting the real-time execution of the compressed applications. Such architecture should be power-efficient and performance oriented.

1.4. Summary of achievements

This thesis presents a novel code compression technique particularly well suited to high-performance embedded RISC applications and complex SoC solutions, which not only provides excellent compression ratios (typically about 50% code reduction), but also improves performance by reducing significantly the number of cache misses. The solution employs a dictionary-based compression scheme that is tailored to the characteristics of the RISC ISA.
The decompression process is carried out in real-time by a novel hardware decompression architecture, which is seamlessly incorporated in the pipeline of the processor, extending it by two stages. A number of original design solutions have been developed in the building of the decompressor, including the elimination of large look-up tables usually employed to resolve the mapping between compressed and uncompressed address spaces. The solution is scalable and configurable to suit the characteristics of each application and can be easily tuned to the physical area or execution speed requirements of the project. A desktop software application supported by an intuitive graphical user interface (GUI) has been developed that allows the control and analysis of all the implementation parameters. For instance, this tool allows the user to analyse the entropy and control flow structure of a RISC executable, to compress it selecting from a variety of options (e.g. dictionary size and type, codeword length, etc.), to generate the decompression tables, and to evaluate the performance of different hardware configurations, amongst others.

1.5. Structure of Thesis

The structure of the thesis is as follows.

Chapter 2 provides background information on lossless data and code compression and presents the state-of-the-art in the particular field of code size reduction. The techniques subject to analysis include those enumerated in Section 1.2.

Chapter 3 presents a quantitative study of the entropy of embedded RISC code. A set of test bench programmes compiled for a particular commercial RISC processor (ARCTangent-A4 is used as proof of concept) are used to obtain results that are valid for a wide range of embedded applications, ranging from automotive systems to consumer electronic applications. These executables are compressed with a number of alternative compression algorithms, demonstrating that compression is able to modify the executables in such a manner that their entropies become significantly closer to their intrinsic values.

Chapter 4 identifies the restrictions that embedded systems impose on the candidate algorithms used for compressing the executables, and the criteria for selecting one from the various candidate solutions. The compression scheme developed in this work is then described
in detail, explaining its design considerations and benefits. This chapter also develops the design flow and the mechanisms necessary to perform the translation between compressed and uncompressed instruction addresses.

Chapter 5 describes the hardware implementation of the real-time decompressor, its functionality, interfaces and the main building blocks. The design is synthesised and a layout produced for various FPGA technologies and implementation results are presented.

Chapter 6 identifies the metrics used for analysing the performance of compressed executables. A comprehensive compression tool suite developed in this research is presented and described. Among the main components of this suite are: a system simulator designed to evaluate the impact of code compression on the overall system performance, a compressor tool, and a trace analyser that provides information about the number of delay cycles in the processor that arise from instruction dependencies and pipeline hazards. A configurable simulation model of the instruction cache enables the generation of results for different cache configurations.

Chapter 7 presents and analyses the performance results (that is, the impact of the compression solution) at system level, obtained by the software suite presented in Chapter 6.

Chapter 8 concludes the thesis by summarising the main findings and how they fulfil the aims of the research. Potential future research paths are also outlined at the end of this chapter.

Appendix A introduces the ELF format, commonly used in embedded code. Appendix B presents in full the compression results discussed in Chapter 3, and Appendix C presents a list of the publications based on the outcomes of this work.
Chapter 2

LITERATURE REVIEW

To set the context of the current work, a brief overview of embedded systems is given first. An introduction to data compression is then given, followed by a review of existing methods developed in the past decade that improve the density of the code produced for RISC embedded machines. The most efficient technique, in terms of maximising entropy, is code compression, but other methods such as compiler optimisations and hybrid architectures are widely available. These methods are introduced in this chapter, but as the literature indicates they are unlikely to give significant code reduction improvements, they are not considered in detail. However, code compression methods are considered more thoroughly as the investigation of this area is more likely to lead to efficiency improvements. Finally, an overview of the FPGA devices used for synthesising the developed design presented in Chapter 5 is given.

2.1. Embedded systems

Embedded systems are computer systems with software and hardware architectures tailored to the specific application for which they are designed. Examples of such systems are DVD players, digital TV sets, video cameras, mobile phones, microwave ovens, automotive control, avionics and spacecraft control systems, elevators, arcade games, laser printers and cash registers. These systems typically have one or more microprocessors, memory and peripheral interfaces. Although in some applications the devices are programmable, in many embedded applications the only programming occurs in connection with the initial loading of the application, or during a later software upgrade.
Koopman [8] categorised embedded systems design into four groups, as shown in Table 2.1, where each group represents generalisation of a real system currently in production.

Table 2.1 Four examples of embedded systems with approximate attributes, Koopman [8]

<table>
<thead>
<tr>
<th></th>
<th>Signal Processing (1)</th>
<th>Mission Critical (2)</th>
<th>Distributed (3)</th>
<th>Small (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing Speed</td>
<td>1 GFLOPS</td>
<td>10-100 MIPS</td>
<td>1-10 MIPS</td>
<td>100,000</td>
</tr>
<tr>
<td>I/O transfer rates</td>
<td>1Gb/sec</td>
<td>10Mb/sec</td>
<td>100Kb/sec</td>
<td>1Kb/sec</td>
</tr>
<tr>
<td>Memory Size Units Sold</td>
<td>32-256MB</td>
<td>16-32MB</td>
<td>1-16MB</td>
<td>1 KB</td>
</tr>
<tr>
<td>Development Cost</td>
<td>$20-100M</td>
<td>$10-50M</td>
<td>$1-10M</td>
<td>$100K-1M</td>
</tr>
<tr>
<td>Lifetime</td>
<td>15-30 years</td>
<td>20-30 years</td>
<td>25-50 years</td>
<td>10-15 years</td>
</tr>
<tr>
<td>Cost/Size, weight, power</td>
<td>$1000 $100 $10 $0.05</td>
<td>$100 $100 $10 $0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maintenance Frequency/repairs</td>
<td>Frequent repair/Frequent repair</td>
<td>Aggressive fault detection/Scheduled maintenance</td>
<td>“Never” breaks</td>
<td></td>
</tr>
<tr>
<td>Digital Content Repair</td>
<td>Digital except I/O signals</td>
<td>~1/2 digital</td>
<td>~1/2 digital</td>
<td>Single digital chip; rest is analogue</td>
</tr>
<tr>
<td>Repair time/goal</td>
<td>1-12 hours</td>
<td>30 min</td>
<td>4 min-12 hours</td>
<td>1-4 hours</td>
</tr>
<tr>
<td>Initial cycle time</td>
<td>3-5 years</td>
<td>4-10 years</td>
<td>2-4 years</td>
<td>0.1-4 years</td>
</tr>
<tr>
<td>Product variants</td>
<td>1-5</td>
<td>5-20</td>
<td>10-10 000</td>
<td>3-10</td>
</tr>
<tr>
<td>Other possible examples</td>
<td>Radar/Sonar Video</td>
<td>Jet engines</td>
<td>Trains/Subways</td>
<td>Automotive, Consumer electronics</td>
</tr>
<tr>
<td></td>
<td>Medical imaging</td>
<td>Manned Spacecraft</td>
<td>Manned</td>
<td>Consumer electronics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Air conditioning</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although some of values are now somewhat dated (the table was published in 1999), it is interesting to note that the two first columns, namely military and aerospace embedded applications show performance characteristics of memory size and Input/Output transfer rates are now not uncommon in modern consumer electronic devices, demonstrating the substantial growth in hardware capabilities in the intervening years.

One of the most important design decisions when building an embedded system is the selection of an appropriate hardware architecture. This decision is typically based on two major factors, namely the processing capability requirement and the cost per unit.
Performance is often determined according to the real-time requirements of the application. For example, a domestic television with a single frame buffer must be able to process the current image frame before the next one is received. The production cost on the other hand, depends on a variety of factors, such as implementation technology, number and type of components, manufacturing process. In many embedded systems the cost of memory constitutes a substantial portion of the overall cost, restricting the memory capacity of the design and forcing software engineers to limit the supported features. As a consequence, the memory architecture, capacity and usage must be carefully considered in many embedded applications. Finally, systems with heat dissipation issues and embedded in a portable device may have additional restrictions in terms of power consumption.

As it became clear from Table 2.1 the term 'embedded system' covers a wide variety of systems, ranging from simple electronic devices to complex multi-processor platforms. In order to give a better idea of this variety, this subsection presents two modern systems with very different complexities and performance capabilities. The first is a simple test board implementation, while the second is a complex SoC ASIC media-processor that is featured in a number of high quality multimedia products such as domestic televisions and video cameras.

2.1.1. ARCAngel-1 test board [9]

This FPGA development board is designed to test and evaluate embedded solutions based on the ARCTagent-A4 microprocessor. It includes an Altera Flex 10K200E/250E PLD, which has the capability to hold an entire processor design (including a limited capacity instruction cache). It also implements phase-locked loop (PLL) circuitry (to generate system clock frequencies up to 25MHz), incorporates a range of displays and switches, and provides parallel and serial ports to enable interfacing a host workstation. In terms of memory, the board has 1 MB of fast SRAM, from which 512 kB is used as system memory, and the remaining 512 kB is allocated to the data cache.

2.1.2. PNX8550

Due to their efficient pipelined architecture and small physical area requirements, RISC processors are often used in high performance SoC implementations. Philips Semiconductors' PNX8550 SoC solution [10] is a highly integrated media processor, responsible for video
improvement processing for both analogue and digital sources. It includes integrated dual program conditional access, dual program MPEG2 transport stream de-multiplexers, dual standard definition or single high definition MPEG2 video decode, audio decode and processing, graphics generation, video processing, and image composition and display. Its internal architecture includes more than 25 different hardware blocks that perform core video functions, such as picture-level MPEG2 decoding, scaling, image composition and pixel post processing. In addition, two 32-bit 240 MHz very-long instruction word (VLIW) processors carry out the advanced video improvement processing, as well as all audio operations, and a further 32-bit 250MHz RISC processor core runs the operating system and both accesses and controls all the hardware resources available on the chip.

The memory access infrastructure of PNX8550, called the HUB, is based on the pipelined memory access network (PMAN) technology specification. In addition to the network function, the HUB includes a generic arbiter for data flow control within complex memory systems and the PMAN Security block. The PMAN or HUB provides direct memory access (DMA) data paths and control which link the majority of the peripheral devices with the main memory controller, allowing data to be read from or written to main memory at a very high rate.

These two systems are used later in this work as a base for defining some of the parameters used in the performance evaluation study of the proposed compression solution.

2.2. Data compression

Technological developments continue to allow ever more data to be held by the computer storage systems. More efficient use of data storage can be achieved by sophisticated compression algorithms, which are prevalent not only in portable systems, where the reduction in the memory requirement can bring power savings, but are also frequently provided by the memory management units of modern operating systems.

Compression is a means of reducing the number of bits required to represent a given item of data. Not only does this result in an effective increase in the storage capacity of computer memories, but also improves the effective transmission bandwidth over internal busses.
Compression can be lossless or lossy. In lossless compression, algorithms data can be retrieved completely after decompression, while during lossy compression some information is irretrievably lost. In this thesis, the emphasis is on lossless compression, as in computer applications any modifications to instruction code will affect its functionality.

Lossless compression algorithms can be divided into three categories, namely ad hoc, statistical and dictionary, as shown in Figure 2.1. Ad hoc algorithms use the natural redundancies of specific data sets, for example the differential coding where only the difference between the consecutive data symbols is coded, or run-length encoding, where the number of identical values of data items are recorded. Dictionary approaches achieve compression by replacing groups of consecutive characters (or symbols) with indexes into a dictionary that contains a list of symbols that occur frequently. The resultant compressed data contains indices into the dictionary that take less space than the symbol they encode. In statistical methods, a probability is estimated for each character and a code is chosen based on the value of the probability. Ad hoc methods generally only perform well on the data sets they are designed to work on, and on general data sets, statistical methods normally achieve better compression ratios, but require more complex calculations in their implementation.

![Data compression taxonomy](image)

For both dictionary-based and statistical compression, the process is performed in two steps; firstly modelling the input data and secondly coding it (see Figure 2.2). The model
provides the probability distribution of the symbols in the data set, while the coder, once supplied with this prediction, constructs a compressed representation of the data with respect to the probability distribution.

![Diagram](image)

**Figure 2.2. An example of compression process**

The rate of compression is called the compression ratio (CR) and can be calculated as in equation 2.1:

\[
CR = \frac{\text{output size}}{\text{original size}}
\]  

(2.1)

where *input size* is the length of a file before compression (in bits) and the *output size* is the length of the file after compression (in bits). The value of output size depends on the method employed for compression and also on the information content, or entropy, of the data set that is being compressed. The entropy \( H \) of a model can be determined by:

\[
H = -\log_2 p_i \text{ bits}
\]

(2.2)

where \( p_i \) is the probability of symbol occurrence and \( H \) is measured in number of bits as defined by Shannon in [11] (for further details see Chapter 3, Section 3.1). The symbols with greater probabilities of occurrence contain less useful 'information' and the lower the total information content of a data set the greater the redundancy and consequently the higher the potential compression ratio that can be achieved.

It is important to notice that choosing a suitable model for the data set and implementing an efficient coding technique, can result in a compression ratio that approaches the limit that can be determined from the natural entropy of the data [12].
2.2.1. Modelling

The role of the model is to supply the probabilities for the symbols. Based on how they are constructed, models can be divided into three main groups, termed static, semi-adaptive and adaptive. There is no rule of thumb that can be used to select the most suitable model for a particular application. Compared with static and semi-adaptive models, adaptive models are more complex to implement but they provide greater flexibility and better performance by being able to modify their behaviour as the probabilities of symbols change during transmission. The operation of a static model is determined before the transmission of the messages starts. Usually static models are built on analysis of set data representative of the required type, resulting in an assignment of the same probability to any give symbol in the message each time it appears in the stream. Semi-adaptive models require two data parses. In the first, the data stream is parsed and the probabilities of each symbol estimated, and, in the second, each message is assigned the probability determined in the first pass before being encoded. Adaptive models, as the name suggests, are built as the data is being streamed. The assignment of probabilities to symbols is based on the values of the relative frequencies of occurrence at each given point in time. This means that a symbol that occurs frequently near the start of the data stream may be represented by a short codeword early in the transmission, even though its probability of occurrence over the complete data is low. Later, when other symbols begin to occur more frequently, the adaptive modeller is likely to assign to one of these more probable symbols.

In addition to the type, a number of classes of model exist. Finite-state probabilistic models are based on finite-state machines. They have a set of states $S$ and a set of transition probabilities $p$, that give the probability that when the model is in state $S$, the next state will be $S'$. Each transition is labelled and no two transitions from a state will have the same label. A message defines a specific path through the model that follows the sequence of symbols that form the message itself. The probability of a message is computed as the product of the transition probabilities that make up the path. The most popular finite-state model is the so-called order-$n$ fixed-context model, which uses the $n$ preceding characters to determine the probability of the next character. The grammar model, which is built to accommodate nesting depths with probabilities associated with each production, has been used successfully for compressing text in formal languages, such as Pascal programs [12].
2.2.2. Coding

Given a particular message, coding involves the selection of a suitable set of codewords, each member being a unique string of symbols (bits). Codewords can be of fixed length, in which case all symbols contain the same number of bits, or of variable length where symbols may be of different lengths. Fixed-length encoding generally yields relatively low compression ratios, but due to its simplicity it is still used in some applications (including code compression schemes [13]). However, most modern coding techniques employ variable-length codewords. A well-known example of variable-length coding is Huffman coding [14]. In Huffman coding, all symbols are sorted in order of probability of occurrence before the two symbols of lowest probability are combined to form a new composite symbol; eventually a binary tree is built where each node is the probability of all nodes beneath it and the leaves are the original message. Forming a code for any symbol then consists of traversing the tree from the root to the particular leaf. For a given frequency distribution, there are many possible Huffman codes, but the total compressed length will be the same. Huffman codes have the unique property that no code is a prefix of another code; meaning Huffman codes can be unambiguously decoded. A number of different versions of Huffman coding have been used in dictionary-based compression schemes.

Another popular and very efficient coding algorithm is arithmetic coding, which differs from Huffman coding in that it operates incrementally on a stream of symbols, whereas Huffman coding requires the full data set to be available to build its tree. In arithmetic coding, words are not necessarily represented by an integral number of bits, as the input stream is translated into a single floating-point value in the range [0, 1). The algorithm for coding a file using arithmetic coding works conceptually as follows. The model provides the symbols and their associated probabilities to the coder. When the first symbol is processed, the initial interval [0, 1) is narrowed to the probability of the symbol, for example [0.2, 0.5). If the probability of the next symbol is in the range [0, 0.2) it will further narrow the interval by a factor of 1/5, namely to [0.2, 0.26). Each successive symbol then narrows the interval in the same manner in accordance with its probability. If the coding is incremental, the whole data set is divided into blocks of equal size and each of these blocks is encoded separately in the full interval [0,1).
2.3. Compiler optimisations

The reduced number of instructions available on RISC architectures makes them particularly dependent on compiler technology in order to provide efficient solutions. There is considerable research effort into code size reduction and the wide commercial adoption of RISC processors has stimulated the development of new compiler optimisation techniques.

Modern optimising compilers usually provide the option to optimize the code either for ‘speed’ or for ‘space’. Speed optimization techniques, which aim to shorten execution time, include procedure inlining (rather than branching to a subroutine), loop unrolling and inversion, and allocation of live variables to registers. ‘Space’ optimizations, which aim to reduce the memory requirements, usually utilize techniques such as local and global common subexpression elimination (where two instances of the same computation are replaced by a single copy), factoring out invariants, global value numbering, and unreachable and dead code elimination. Due to the nature of these techniques, a program that is compiled with speed optimisations will, generally, result in faster but larger code, while conversely, when compiled for space, the code will be more compact (up to about 18%) [15], but slower. It is important to realize that compilers typically only deal with a small part of the program at any one time (perhaps as much as a compilation unit, but often only a single function), and so are unable to take into consideration optimisation at a higher level of abstraction that could further reduce code size.

Code compaction includes a number of compiler techniques that were developed as a next step of improving code density, building on the base of “classical” compiler optimisations. Initially, compaction was envisaged as a suitable solution to the RISC code density problem as it does not require any further support: no decompression takes place and there is no need to implement additional hardware or provide further run-time software support. Thus the overhead is limited to the development of the special compiler optimizer only.

Much of the earlier work to reduce the memory space occupied by executable code treated it as a simple linear sequence of instructions. Fraser et al. [16] used suffix tree construction to identify repeated instruction sequences for extraction into functions. Applied to a range of Unix utilities on a VAX processor, this technique was found to reduce the code
size by about 7%. A shortcoming is that since it relies on a purely textual interpretation of the program, the approach is sensitive to superficial differences between code fragments, such as the use of different registers in identical operations. This disadvantage was addressed by Cooper and McIntosh who used register renaming to alleviate the problem [17]. Two improvements were made: firstly instructions were rewritten so that instead of using hard-coded register names, the register operands of an instruction are expressed in terms of previous references to that register; and secondly, jump instructions were rewritten in a format where they were compared relative to the Program Counter (PC), where possible. These transformations allowed the suffix tree construction to detect the repetition of similar, but not lexically identical instruction sequences. Code reductions of about 5% were reported using these techniques on classically optimized code.

Debray et al. [15] concentrated their efforts on the control flow of the program. They identified 'similar' basic blocks by adopting a fingerprinting scheme and, by renaming registers locally, similar blocks were translated into identical blocks. Then, these were replaced by a single code block and a number of jump instructions to its location. They called this technique 'code factoring'. By the use of an aggressive, inter-procedural application of the classical compiler techniques and code factoring they reported typical reductions in code size of 30%. They implemented their idea in the form of a binary-rewriting tool based on Alto, a post link-time code optimizer. A derivative of Alto, called 'squeeze' can also be integrated into compilers capable of inter-procedural optimization in order to perform program compaction. The authors do not mention the effect this technique has on performance.

In summary, it can be concluded that most compiler techniques developed for code compaction can only achieve moderate improvement in code density [15]. Moreover, they would seriously detriment the performance due to the nature of the optimisations carried out, for example, limiting the number of registers used implies more (slower) external memory accesses are required and introducing additional functions requires more branch and jump instructions that increase the execution time.

As shown in section 3.3, compression is able to achieve much more substantial reductions in code size, and, although requiring hardware support, has little effect on overall performance.
2.4. Hybrid RISC Architectures

When the RISC architecture was first developed in the early 1980s, memory was not considered a scarce resource. Nowadays, under the strict requirements for low memory usage in embedded systems, most of the 32-bit RISC processor suppliers now provide a hybrid version of their cores, supporting both 32 and 16-bit instructions. The instructions included in the 16-bit subset are typically selected for one of the following reasons:

- Frequency of use
- Size (they may not require the full 32 bits in the 32-bit ISA)
- Are important to the compiler for generating small object code

Several examples of such hybrid architectures are described next.

2.4.1. MIPS16

MIPS16 is a 16-bit instruction encoding "application specific extension" that offers the choice of a mixed mode (16 or 32-bit instruction lengths) and runs on specific MIPS processors designed to accommodate it. MIPS16 instructions are translated on-the-fly using relatively simple hardware [5]. In order to reduce the number of instruction bits by half, all fields in the instruction word (opcodes, register numbers and immediate values) are downsized, as shown in Figure 2.3.

![Figure 2.3 Mapping of hybrid instructions](image)

The opcode field is reduced from 6 to 5 bits limiting the number of instructions available, and only 8 different registers can be accessed (3-bit operand fields). Furthermore, MIPS16 usually permits only 2 register specifiers per arithmetic instruction as opposed to the 3 registers
The most substantial reduction is achieved by restricting (from 16 to 5 bits) the range of immediate values that can be represented. The MIPS16 instruction set specifically excludes coprocessor instructions, floating point instructions and those that reference the 'system' coprocessor. These instructions must be in 32-bit code routines. Switching between MIPS 16 and 32-bit modes of operation is performed by an additional instruction.

2.4.2. ARM Thumb Processor

The ARM Thumb instruction set contains a subset of 36 instructions drawn from the standard 32-bit ARM instructions and recorded into 16-bit format. In the ARM7, the 16-bit instructions are decompressed into their 32-bit equivalents in real time during the first phase of the decode cycle. In the second phase these are decoded as normal 32-bit ARM instructions. A Thumb-aware processor, such as the ARM 9, has separate logic for executing both Thumb and original ARM instructions, switching between them by means of a new instruction. High-level code can be compiled to either native ARM code or Thumb code on a function by function basis [3].

In order to define the Thumb instruction set in 16 bits, several restrictions have been imposed. Like MIPS16, only three bits are used to encode registers (opposed to four in ARM instructions), so only eight registers can be directly accessed by Thumb instructions. The immediate field is also narrower. In addition, instructions can contain two or three operands only, while some ARM instructions have four. Conditional execution is not supported, so Thumb code tends to look much more like a conventional assembler with compares followed by short branches. Yet another disadvantage is the increase in the number of instructions that need to be fetched from memory and executed, which may increase the overall power consumption of the system. Thumb code typically provides a 30% improvement in code density over native ARM code at the expense of a 26% performance reduction [4].

2.4.3. ARCompact

The ARCompact ISA was announced as part of the ARtangent-A5 processor architecture. Its principles are largely the same as those discussed above for Thumb and MIPS16. However, instead of using a decompressor at the head of the instruction pipeline to convert a 16-bit
instruction into its 32-bit equivalent (the technique used by MIPS16), the ARCTangent-A5 instruction decoder processes the 16-bit instructions natively, leaving the basic pipeline unchanged. Thus, no extra instructions are needed to change between compressed and uncompressed modes, since the nature of the instruction is encoded in its most significant bit (0 for 32-bit instructions and 1 for 16-bit instructions). With this approach, it is possible to achieve compression ratios of around 0.7, but the performance may be significantly affected, increasing the execution time by between 15% and 20%. Moreover, the new instruction subset implementation reduces the flexibility of the ISA and results in an increase of the instruction count by up to 40% [6].

2.4.4. Conclusions

Providing a hybrid subset of the ISA is an established technique for code compaction, which has been implemented in a number of commercial RISC architectures. The achieved memory reduction can reach up to 40%, but typically it comes at the expense of performance, which might decrease up to 26% [4]. Performance degradation occurs due to the reduction in the number of registers accessible in the instruction, resulting in the need to include additional instructions to carry out many arithmetic and logic operations. The reduced number of registers available also means that more data needs to be stored in external memory, thereby slowing access and increasing power consumption. An additional disadvantage of the hybrid approach is the need to develop new software tools (compiler, linker, and loader) to support the ISA, which increases the cost of supporting these solutions.

2.5. Code Compression

Shortly after the development of the first RISC architectures, a group at Berkeley University [1] proposed compressing RISC executable code, namely a RISC machine that supported both 16-bit and 32-bit instructions in memory, but which were translated to 32 bits in the cache. Since then, a number of different techniques have been developed based on modified data compression algorithms.
Due to performance constraints, a natural separation arises in the implementation of the compression and decompression stages. Compression can be carried out off-line, and therefore a software approach is most convenient, whereas decompression needs to be executed in real-time immediately before execution and so is better suited to hardware implementation.

The purpose of this section is to give a comprehensive appraisal of the most relevant existing approaches for compressing embedded RISC code in order to clarify the contributions of this thesis to the state-of-the-art.

2.5.1. Algorithms

As discussed in Section 2.1, embedded systems are subjected to many constraints such as low power consumption, high performance and minimal area requirements. In order to be viable in the highly competitive embedded systems market, it is important to make a significant impact in one of these areas, or perhaps mere minor contributions across two or more of these areas. As the initial intention of compressing code has been developed in order to economise on memory area and, to a lesser extent, power consumption, the decompression implementation must consist of relatively little logic, and, at the same time, should not significantly affect the performance of the system. Therefore, the algorithms used in code compression implementations need to take into account a range of factors additional to those considered in general data compression applications.

Specifics of code compression algorithms

Although code compression algorithms have generally been developed on the basis of existing, well-known techniques used for compressing either text or data, the following features are particular to code compression applications:

- **Asymmetric, fast decompression**

In embedded systems, code compression and decompression are not symmetrical processes. Compression is normally carried out off-line on a host development machine and can be rather time consuming due to the optimisations and analysis of the code that need to be performed. After the code segment of the executable has been compressed, it can be loaded in
the memory of the embedded system. Decompression is performed in real-time on the embedded system. The extra hardware required in the decompression stage should remain as simple as possible in order to avoid excessive overheads in silicon area and execution speed.

- **Random access decompression**

The program on the embedded system could not normally be decompressed in its entirety before being executed as this would defeat the whole purpose of compressing the program. The control flow of the program is not sequential as it will contain jump and branch instructions (discussed in more detail in section 4.2), which may require that the execution of the program moves to a new point in the code. Consequently, the decompression should be capable of starting at any point in the code, or, at least, at a suitable block boundary, which indicates that decompression should perhaps be carried out on instruction or block basis. This restricts the algorithmic development, as, in many compression algorithms, the statistical history of the data already seen contributes to the building of the appropriate models. The widely used Ziv-Lempel family of algorithms [12], for example, use pointers to previous occurrences of strings, and consequently the decompression of fixed size blocks is likely to be not well suited to such algorithms.

- **Low buffering requirements**

As the main purpose of introducing compression is to conserve memory usage, it is appropriate to severely restrict the lengths of any buffers required by the compression algorithms. Dynamic Markov compression (DMC) and prediction by partial match (PPM) [12], are both able to achieve excellent compression ratios for certain applications. However, they are not suitable for compressing code as they require considerable memory resources for both compression and decompression, and such large buffers are not normally available in embedded systems.

- **Correct program execution**

As expected of any lossless compression algorithm, the decompressed instructions must be identical to the original ones. However, in code compression, correct execution requires that a number of different mechanisms are in place to resolve certain issues that arise; principally those that result from changes in control flow. In particular, it is often necessary to
incorporate large address translation tables in order to map between the compressed and uncompressed address spaces.

**Code compression algorithms**

Due to the limitations outlined above, many data compression algorithms are not suitable for code compression applications. Code compression algorithms are typically dictionary-based, where the most commonly used instructions or sequences of instructions are held in the dictionary, and replaced in the programme by a single, shorter codeword. In that way, the compressed programme is composed either of codeword only, or of codewords interspersed with non-compressed instructions. The codewords are generally encoded using some variant of Huffman (often used due to its simplicity) or other fixed-to-variable size coding technique. To simplify the decompression process, the number of different lengths available to represent the codewords may be limited (class-based approach) [12].

Araujo et al. [18] developed a code compression algorithm in three variants, which differ only in the granularity of the symbol used for compression. The main principle behind the algorithm is to separate the symbols into classes based on the frequency distribution of the symbols. Symbols in each class are assigned codewords of the same length. The symbols used are whole expression trees (tree based compression - TBC), parts of expression trees (pattern based compression - PBC) or single instructions (instruction based compression - IBC). In TBC and PBC, the expression trees (formed by the same rules used by compilers) are the basic unit of compression, but in PBC the operands are factored out, that is, the expression trees are divided into two parts that are separately encoded (the tree pattern and operand pattern). Table 2.2 shows typical compression ratios achieved by the three approaches described above. The better compression ratio of PBC results from the repetition of patterns of instructions, which differ from each other only in the operands they use.

<table>
<thead>
<tr>
<th>Techniques</th>
<th>tree based compression</th>
<th>pattern based compression</th>
<th>Instruction based compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbols used for compression</td>
<td>whole expression trees</td>
<td>parts of expression trees (patterns)</td>
<td>single instructions</td>
</tr>
<tr>
<td>compression ratio</td>
<td>0.73</td>
<td>0.60</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Table 2.2 Results achieved by Araujo et al. [18]
An algorithm, similar to the ones described by Araujo, was implemented in Motorola’s MPC565 range of microcontrollers, the main difference being the use of instructions and half-instructions as units of compression. The compression ratio achieved was quoted to be as low as 0.5, but the memory used by the dictionary was not taken into account in the supplied figures [19].

Lekatsas et al [20, 21] proposed two methods for code compression. The first method, semi-adaptive dictionary compression (SADC) is ISA dependent. Instructions are divided into streams (groups of bits within the instruction) in a pre-determined fashion according to the properties of the ISA, whereby suitable combinations of instructions and fields within instructions are identified, such as opcodes, opcode-register and opcode-immediate. Semi-adaptive dictionaries (of up to 256 entries) are built for each executable programme, where only the most frequently repeated instructions are included. The compression itself encodes all streams using Huffman coding. The second method is semi-adaptive Markov compression (SAMC), which is ISA-independent. It uses a binary arithmetic coder driven by Markov models as well as the stream division approach of SADC. For each stream, a binary Markov tree with \(2^{k+1} - 1\) states is built and transition probabilities are calculated, which are used by the compressor as predictions of the forthcoming bit. The compressor parses the subject programme a second time to encode the message. The storage requirements for SAMC are the encoded message and the Markov trees for the streams and the reported reduction in code size is 50%. Although the authors acknowledged that the decompression speed is poor, they proposed to improve this by building a decoding table that allows multi-bit arithmetic decoding.

Yoshida et al. [22] described a logarithmic-based method to compress instruction memory, by substituting the original 32-bit instructions by a set of references to a translation table (dictionary) that can be stored in the on-the-chip memory. The main drawback of the scheme is the substantial memory requirement of the large translation tables. However, the approach not only produced good instruction memory size reductions (ignoring the translation tables) with a mean compression ratio of 0.62 for an ARM610, but also registered significant power consumption savings.

Lefurgy et al. [23] proposed two dictionary-based compression techniques differing in the codeword sizes they employed. In this approach, the most frequent sequences were
compressed and additional bits, called escape bits, were used to define the boundaries between compressed and uncompressed instructions. The instructions corresponding to each codeword were stored in a dictionary in the decompression engine and the codeword bits are used to index the dictionary entries. The decompression engine was designed to incorporate sufficient information to be able to expand codewords to reproduce the original instruction sequences. Since the compressed program is composed of codewords and uncompressed instructions, branch targets need to be recomputed so as to reflect their new location in the program. The target address bits are divided into two parts, namely the address of the compressed word and the offset from the beginning of the compressed word. Forming the sum of these values determines the target address. To achieve bit addressing, further modifications to the control unit of the processor are required. The results achieved when using fixed-length codewords were compression ratios of 0.61, 0.66, and 0.74 for the PowerPC, ARM and i386 processors respectively.

An approach described by Liao et al [24] identified frequently appearing strings of instructions (termed mini-subroutines) in a program and replaced each instance of a mini-subroutine by a call instruction. The mini-subroutine appears only once in the text of the program and may themselves contain branch instructions under certain conditions. Although a mean compression ratio of 0.88 was achieved, the large number of branch instructions introduced by this algorithm adversely affects overall performance. An extension that augmented the instruction set with a new call instruction that was able to point to any location in the dictionary was also described, which resulted in the elimination of most of the mini-subroutines. The call instruction consisted of a dictionary address plus a length value specifying the number of instructions to be executed from the dictionary. With these modifications the compression ratio was further reduced to a 0.84.

2.5.2. Decompression implementations

Although the decompression stage could be implemented in either software or hardware, the performance of a software solution makes this option unpractical in most cases, whereas hardware decompression has already been successfully included in commercial products. Commercial microcontrollers that incorporate dedicated hardware decompression engines are the IBM CodePack and the Motorola M5xx series. The decompression architectures of these
devices, together with the one of the first RISC processors to incorporate code compression, are described in more detail later in this section.

Decompression in Software

Lefurgy et al [23] proposed a dictionary compression scheme, where each 32-bit instruction is indexed by a 16-bit codeword. The obvious limitation is that a maximum of $2^{16}$ (65,536) unique instructions can be compressed in this scheme, but if the number of unique instructions is larger, then the less frequently used instructions are left in uncompressed form and the memory is divided into separate compressed and native regions. The decompression software is implemented in only 26 assembler instructions, but they need to be executed each time a compressed instruction is encountered, while the only hardware change is the addition of a special instruction that invokes the decompressor at cache miss. The compression ratio that can be achieved depends mainly on the number of unique instructions in the application and the dictionary.

Bird and Mudge [25] proposed a look-up table method that provided fast decoding. Following code generation, the instruction stream is analyzed for often-reused sequences of instructions from within the program's main blocks. These patterns of multiple instructions were then mapped to single byte opcodes, yielding compression of multiple, multi-byte operations into a single byte. Compressed opcodes are detected during the instruction fetch stage, and expanded within the CPU into the original (multi-cycle) set of instructions. As they operate within the program's basic blocks, branch instructions and their targets are left unaffected by this technique. The authors reported that by incorporating "1kbyte of decode ROM in the CPU" [25], a reduction in the memory occupied by the program code of between 45% and 60% can be achieved.

Kirovski et al. [26] described a system where code is decompressed on a procedure basis from ROM into a procedure cache (a dedicated region of RAM that is explicitly managed by the runtime system). An important issue in this approach is the management of this procedure cache. Any decompression algorithm (both dictionary as well as statistical) can be used, as long as fast software implementations are available. In the authors' implementation, the decompression is performed by software and little or no hardware support is required.
Decompression in Hardware

In 1992, Wolfe and Chanin [27] proposed a new solution, termed the code compression RISC processor (CCRP). The CCRP consists of a standard RISC processor core augmented with a special code-expanding instruction cache. CCRP programs are compiled and linked using standard tools, and the generated executable is then compressed. Fixed 32-byte blocks are considered as the atomic unit of compression. The compression algorithm used is based on bounded (up to 16 bits) static Huffman encoding, where some blocks remain uncompressed if their lengths were to increase during compression. The hardware decompression engine is located between the main memory and the cache, as shown in Figure 2.4. In this architecture, the instruction cache holds uncompressed instructions (those in the original form as generated by the compiler), while the compressed program is stored in the main memory.

![Figure 2.4. Code Compression RISC Processor [27]](image)

At run time, on a cache miss, the ICache (Instruction Cache) refill-engine locates the missed cache line in the main (compressed) memory and expands (decompresses) it into the cache. This approach does not require any modifications to the core processor, but only the implementation of an ICache refill-engine.

![Figure 2.5. Line Address Table Entry](image)

In a CCRP program, the instructions are allocated addresses according to their location (for example, in the main memory or in the cache). A line address table (LAT), stored alongside with the compressed program, is used to map the original (uncompressed) instructions addresses, to the compressed ones. Each LAT entry is 64-bits wide and is used to
map 64 consecutive instructions, which are grouped into eight blocks of eight instructions each. The base address field (see Figure 2.5), represents the (compressed) base address of the first block. The length (in bytes) of each compressed block is stored in the eight subsequent fields. A length of 0 represents an uncompressed block, which is always of length 32 bytes. In this way, the starting address of a particular compressed block can be calculated by adding the length of its preceding blocks to the base address. This operation is performed within the cache line address lookaside buffer (CLB). The CLB contains a small fully associative cache with least recently used (LRU) replacement policy, capable of storing from 4 to 16 LAT entries. The CLB is accessed in the fetch instruction stage, simultaneously with the instruction cache (see Figure 2.6). When an instruction cache miss occurs, if the required LAT entry is found in the CLB, the compressed block can be immediately requested for transfer to the main instruction memory. If, on the other hand, the CLB misses, it has to be refilled before requesting the compressed address, thus incurring delays due to both ICache and CLB refill times.

The CLB’s cache is implemented using a content addressable memory (CAM) to map uncompressed block addresses to LAT entries, and an address computational unit to calculate the compressed address of the selected block. The processor instruction address (24-bit long) is composed of three parts, as shown in Figure 2.6. The 16 most significant bits, the LAT index, are compared with the tags of the entries currently in the CLB to determine whether the required address computation LAT entry is currently in the CLB or has to be fetched from memory. The next 3 bits are the LAT length pointer index that determines which elements of the LAT entry have to be summed in order to compute the compressed block starting address. The 5 least significant bits act as the byte offset into the selected cache line. In terms of resources and performance, the memory required to store the LAT is around 3% of the total memory usage, achieving compression ratios of around 0.73 for MIPS R2000, but clearly additional memory and logic are needed to implement the decompression engine.
Similar in operation to CCRP is IBM’s CodePack code compression system [7, 28] that was designed to compress any PowerPC programme into a format that can be executed by the processor after decompression. During compression, the CodePack compression utility analyses the instructions’ frequency distribution and produces a pair of 2-kbyte lookup tables generated specifically for that particular programme. When the compressed programme is run, a CodePack-equipped processor uses these tables to decompress the code on the fly before execution. Although there is some performance penalty for decompression (typically around 10%), significant extra latency is introduced only during instruction cache misses.

The CodePack compression algorithm operates as follows. Each 32-bit instruction is divided into 16-bit most significant and least significant half-words, which are then translated (using Huffman encoding) to a variable bit codeword whose length is in inclusive range 2 to 19 bits, where the most common half-word values are placed into one of the two look-up tables. The main reason for using two separate look-up tables is the different frequency distributions that characterise the upper half of the PowerPC instruction (which holds the opcodes) and the lower half (which typically holds constants, displacements, or masks). Each group of 16 instructions is combined into a compression block, which is the granularity at which decompression occurs and, when an instruction is requested by the processor, the block in which it is located is fetched and decompressed.

A further set of tables, termed the index table, is used for mapping between uncompressed and compressed addresses, as shown in Figure 2.7. The index table holds 32-bit
entries that point to the compressed-space addresses of a group of two compression blocks, forming a compression group. During run-time, at a cache miss, the decompression core calculates the address of the index table entry of the compression group containing the target instruction address (TIA) and fetches it from the memory. Next, it calculates the address of the compressed block to which the requested instruction belongs, and as the compression block is read in, the decompression core uses the contents of the decode look-up tables to decompress the block. Finally, once the cache-line refill is completed the processor continues execution. The size of the index table can be up to 2 Mbytes in size, which would cover an entire 64 Mbytes compression region.

![Index table mapping of TIAs to compressed memory](image)

Figure 2.7. Index table mapping of TIAs to compressed memory [7]

The tool chain available for CodePack is similar to that of other PowerPC processors, as no specific compilation or linkage is required, but they are supplemented by a post-processor that compresses the executable and builds the compression tables. IBM claims that the compression scheme of CodePack typically reduces code size by 35 - 40% [7].

Larin and Conte [29] compared Huffman coding compression performed on cache misses with tailored encoding of the ISA for a very long instruction word (VLIW) architecture. Depending on the requirements of a particular program, such as general purpose registers or the number of floating point operations, the register and opcode fields in the instructions are coded to the optimum length needed. For example, if no more than four registers are alive at the same point at a given source code position, the register fields are coded into two bits only. In terms of performance improvement, the tailored ISA compares favourably with Huffman
encoding, while, in terms of code size, compression gives better results. Lecatsas et al. [20] investigated the potential benefits of implementing a modified arithmetic coding technique and claimed that they achieved energy savings between 22% and 82%, were able to reduce chip area and even simultaneously improve performance. Although their savings are attractive, the results were presented only for a theoretical study and no implementation has been carried out. The implementation of the logarithmic algorithm of Yoshida et al [22], briefly described earlier, achieved a reduction in power consumption by up to 42.3% in its implementation.

2.6. FPGA devices

FPGA technologies are becoming increasingly widely used in industry, delivering performance and features that previously only ASIC devices could provide. Their prices are far more competitive and their design tools are often much more user friendly than their ASIC counterparts. FPGAs not only allow for fast prototyping, but they provide reconfigurability, a feature not available in ASICs. However, it has been suggested that FPGAs are still only a first-generation embodiment of the big idea of a general-purpose, reconfigurable substrate for special purpose computing [30]. In any case their advantages are sufficiently important to investigate the role that this new technology can play in current and future control implementations.

This section briefly reviews the main characteristics of a selected number of FPGAs that were used in the implementation of the decompressor in order to understand the implementation figures presented in later subsections.

Spartan-1IE

The Spartan-1IE family of FPGAs is targeted towards very low cost solutions that require a good level of performance. These devices provide system clock rates beyond 200 MHz and they are implemented on a 0.15 μm technology. A Spartan-1IE FPGA is composed of five major configurable elements [31] (see Figure 2.8).

- IOBs (Input/Output Blocks), which provide the interface between the package pins and the internal logic.
- CLBs (Configurable Logic Blocks), which provide the functional elements for constructing most logic.

- Dedicated block RAM memories of 4096 bits (4 Kbytes) each.

- Clock Delay-Locked Loops (DLLs) for clock-distribution delay compensation and clock domain control.

- Versatile multi-level interconnect structure.

The basic building block of the CLB is the logic cell (LC). An LC includes a 4-input function generator (implemented as 4-input LUT), carry logic, and storage element. The output from the function generator in each LC drives the CLB output or the D input of the flip-flop. Each CLB contains four LCs, organised in two similar slices. The main parameters for the selected device are given in Table 2.3.

![Figure 2.8. Basic Spartan-IIe Family FPGA Block Diagram](image)

**Table 2.3. Main complexity parameters for the selected Spartan-IIe FPGA**

<table>
<thead>
<tr>
<th>Part</th>
<th>Logic Cells</th>
<th>Typical System Gate Range (Logic and RAM)</th>
<th>CLB Array (RxC)</th>
<th>CLB Total CLBs</th>
<th>Available User I/O</th>
<th>Distributed RAM Bits</th>
<th>Block RAM Bits</th>
<th>Block RAM Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2S200e</td>
<td>5,292</td>
<td>71,000 – 200,000</td>
<td>28x42</td>
<td>1,176</td>
<td>285</td>
<td>75,264</td>
<td>56K</td>
<td>14</td>
</tr>
</tbody>
</table>
Virtex-II

The Virtex-II devices are platform FPGAs developed for high performance designs that are based on IP cores and customised modules. The family is often used in telecommunication, wireless, networking, video, and DSP applications. The 0.15 μm/0.12 μm CMOS 8-layer metal process and the Virtex-II architecture are optimised for high speed with low power consumption. The main parameters of the Virtex-II device used for implementing the decompressor are summarised in Table 2.4.

Table 2.4. Virtex-II FPGA main complexity parameters

<table>
<thead>
<tr>
<th>Part</th>
<th>System Gates</th>
<th>CLB Array (RxC)</th>
<th>CLB Slices</th>
<th>Distributed RAM KBits</th>
<th>Multiplier Blocks</th>
<th>18 Kbits Blocks</th>
<th>Max RAM KBits</th>
<th>GCLKs</th>
<th>IO Pads</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2V500</td>
<td>500K</td>
<td>32x24</td>
<td>3.072</td>
<td>96</td>
<td>32</td>
<td>32</td>
<td>576</td>
<td>8</td>
<td>264</td>
</tr>
</tbody>
</table>

Its internal configurable logic includes four major elements organised in a regular array (32) (see Figure 2.9).

- CLBs, which provide functional elements for combinatorial and synchronous logic, including basic storage elements.
- BUFTs (3-state buffers) associated with each CLB element, which drive dedicated segmentable horizontal routing resources.
- Block SelectRAM memory modules, which provide large 18 kbit storage elements of dual-port RAM.
- DCM (Digital Clock Manager) blocks.

CLB resources include four slices and two 3-state buffers. Each slice is equivalent and contains: a) two function generators; b) two storage elements; c) arithmetic logic gates; d) large multiplexors; e) wide function capability; f) fast carry lookahead chain and g) horizontal cascade chain (OR gate).
The function generators are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory. In addition, the two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches.

Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources. The block SelectRAM memory resources are 18 kbits of dual port RAM, programmable from 16384 x 1 bit to 512 x 36 bits, in various depth and width configurations. Each port is totally synchronous and independent, offering three “read during write” modes. Block SelectRAM memory is cascadable to implement large embedded storage blocks and memory configurations for dual-port and single-port modes are supported. A multiplier block is associated with each SelectRAM memory block. The multiplier block is a dedicated 18 bit x 18 bit multiplier and is optimized for operations based on the block SelectRAM content on one port. The 18 x 18 multiplier can be used independently of the block SelectRAM resource and both the SelectRAM memory and the multiplier resource are connected to four switch matrices to access the general routing resources.

**Altera’s Cyclone**

The Cyclone field programmable gate array family is based on 1.5 V, 0.13 μm technology. With features such as phase-locked loops (PLLS) for clocking and a dedicated double data rate (DDR) interface to meet DDR SDRAM and fast cycle RAM (FCRAM) memory requirements, Cyclone devices are a cost-effective solution for data-path applications [33]. The
Chapter 2 – Literature Review

main parameters of the Cyclone device used for describing the implementation of the decompressor are summarised in Table 2.5.

Table 2.5. Selected Cyclone FPGAs main complexity parameters [33]

<table>
<thead>
<tr>
<th>Part</th>
<th>LEs</th>
<th>M4K RAM blocks</th>
<th>Total RAM bits</th>
<th>PLLs</th>
<th>Max user IO pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP1C6</td>
<td>5,980</td>
<td>20</td>
<td>92,160</td>
<td>2</td>
<td>249</td>
</tr>
</tbody>
</table>

Cyclone devices contain a two-dimensional row-based and column-based architecture to implement custom logic. Column and row interconnects of varying speeds provide signal interconnects between logic array blocks (LABs) and embedded memory blocks.

The logic array consists of LABs, with 10 logic elements (LEs) in each LAB. An LE is a small unit of logic providing efficient implementation of user logic functions. LABs are grouped into rows and columns across the device.

M4K RAM blocks are true dual-port memory blocks with 4 kbits of memory plus parity (4,608 bits). These blocks provide dedicated true dual-port, simple dual-port, or single-port memory up to 36-bits wide at up to 250 MHz. These blocks are grouped into columns across the device in between certain LABs [33].

Each Cyclone device input/output pin is fed by an I/O element located at the ends of LAB rows and columns around the periphery of the device.

![Figure 2.10. Cyclone EP1C6 Device](image)
2.7. Conclusions

The problem of poor code density of RISC processors has attracted considerable attention and research effort since the introduction of RISC processors into embedded systems. A number of different techniques targeting this problem have been described in this chapter, including compiler optimisations, hybrid ISAs and code compression.

Improvements in compiler technology have made it possible to eliminate much of the redundant application code and to provide optimisations that reduce memory requirements, such as local and global common subexpression elimination, as well as unreachable and dead code removal. In addition, code compaction techniques have been developed, consisting of a set of mechanisms, such as register renaming, to further maximise the effects of compiler optimisations. Although some of these techniques achieve relatively good results in terms of code reduction (30% code reduction is reported in [15]), they have a detrimental affect on performance. Similar improvements have been achieved following the introduction of hybrid ISAs by a number of RISC suppliers. However, the use of hybrid 16-bit instructions can result in a decrease in performance of up to 26% [4], due to the increase in the number of instructions that need to and the limiting of the number of registers that can be used making additional accesses to slower memory necessary.

Code compression, often achieves better reductions compared with the compiler optimisations and hybrid ISA approaches, and a number of existing implementations, both in software and hardware, have been presented in this chapter in order to reflect the state-of-the-art in the field. A number of authors found significant improvements in the density of the code when using compression and a number of the previous works also report decreases in the power consumption as well as performance improvements.
Chapter 3

ENTROPY OF EMBEDDED RISC CODE

The problem of poor code density of RISC processors is commonly acknowledged in the field of computer architecture and, as the literature review showed (Chapter 2), considerable effort has been invested in its resolution, both in academic and industrial circles. The very idea on which the RISC ISA is based, namely the uniformity of the instruction formats, implies the existence of significant redundancy in its encoding. This chapter investigates the nature and the extent of this redundancy and its amenability for compression.

The first part of this chapter describes an experimental framework and methodology for quantifying the information content of data in terms of its entropy. A set of representative RISC programs is used in order to obtain an indication of the entropy of data that will be studied in this thesis. Once the entropy is known, suitable techniques can be identified to reduce the redundancy in the code while typically increasing its execution performance. Although a number of techniques exist that are capable of decreasing the code size, section 1.2., code compression significantly outperforms other approaches (in terms of code size reduction), such as compiler optimisations and hybrid architectures. This chapter will evaluate relevant modern compression techniques in order to set the context for the final solution presented in later chapters.

3.1. The concept of entropy in information theory

In communication theory, one of the fundamental problems is to determine efficient representations of the data to be transmitted in order to increase the information bandwidth
of the communication channel. This requires reproducing at one point (the destination), either exactly or approximately, a message selected at another point (the source). The task of the current study is to a large extent similar to the fundamental communication problem. It can be assumed that the executable program in its original format is the communication source, while the instruction fetch stage of the processor pipeline is the destination. The communication channel can be seen to be a combination of the memory resources and the instruction data bus, where both of these will benefit from an improved representation. This assumption would allow us to employ entropy, as presented by Shannon in his classical work “Mathematical Theory of Communication” [11], as a method of identifying the actual information content of the program's code. In the second theorem of Shannon's paper, known as the noiseless channel encoding theorem, he postulates that if there were a measure of the information content of a data set (entropy), $H(p_1, p_2, \ldots, p_n)$, where $p_i, i = 1, \ldots, n$, accounts for how much choice is involved in the selection of the event or of how uncertain we are of the outcome (in our case the probability of occurrence of a selected symbol in the code), the following three conditions must be satisfied:

- $H$ must be continuous in the $p_i$.
- If all the $p_i$ are equal, $p_i = 1/n$, and $H$ should be a monotonically increasing function of $n$.
- If a choice can be broken down into two successive choices, the original $H$ should be the weighted sum of the individual values of $H$ at each stage.

And the only equation for $H$ that satisfies the three conditions is of the form

$$H(p_1, p_2, \ldots, p_n) = -K \sum_{i=0}^{n} p_i \log(p_i),$$

(3.1)

where $K$ is a positive constant, which amounts to the choice of a unit of measure. The minus sign preceding $K$ merely reflects the desire for entropy to be a positive quantity, whereas, always being less than unity, probabilities always have negative logarithms [11]. When the unit is bits, $K = 1$, the logarithms are taken with base 2 and $H$ is the entropy of the set of probabilities $p_i$, then
Chapter 3 – Entropy of Embedded RISC Code

\[ H = -\sum_{i=0}^{n} p_i \log_2 p_i \text{ bits} \]  \hspace{1cm} (3.2)

As defined by Shannon [11], entropy applies only to a probability distribution, that is a set of choices with probabilities summing to one. In some cases, it can be more useful to know the information content of a particular choice. If the probability of a choice is \( p_i \), its information content or entropy can be defined as a negative logarithm of its probability

\[ H_i = -\log_2 p_i \text{ bits} \]  \hspace{1cm} (3.3)

This means that those symbols more likely to occur contain ‘less’ information. The term entropy as defined in Equation (3.2) will be further used in this thesis as a measure of the entropy in a program’s code, and the entropy of an individual symbol will be defined as in Equation (3.3).

3.2. Experimental framework

The main purpose of this section is to describe the experimental framework used to identify the entropy of a set of programmes that represent adequately a wide variety of embedded RISC applications, and the improvements, in terms of code size, achieved by the use of different compression techniques. These programs (selected mainly from the MiBench [34] test suite) have been compiled for the ARCTangent-4 microprocessor using the space optimisation option. The software tool developed in this research for measuring the entropy of the code is described along with a number of compression algorithms that were evaluated against this code, in order to highlight the state-of-the-art in the field of lossless data compression.

3.2.1. Entropy measurement tool

As stated at the beginning of the chapter, a scientific study that aims to improve the code memory utilisation of RISC processors requires a quantitative analysis of the actual entropy of the code, thereby providing an insight into the degree of redundancy present. This analysis is
automatically performed by a software tool developed in this research, whose main structure is presented in pseudo code in

Figure 3.1. In the code, the tool takes as an input parameter the name of the executable file for which it builds a probability distribution table for symbols of three different lengths, namely 8, 16 and 32 bits. The ELF headers of the executable file are processed in order to extract the code segment's size and offset. Analysis of the code segment identifies the elements of the data set and builds the frequency distribution tables for each of the three symbol lengths. The probability of occurrence of each symbol is calculated and its individual entropy is obtained according to Equation (3.3). Finally the entropy of the whole code segment is calculated using Equation (3.2) and the results are sent to a statistical log file.

```plaintext
inFile = fopen(inputFile, "rb");
readHeaderInfo(spCodeSegment, codeSegmentSize);
findElfCodeSegment(inFile, pCodeSegment);
for(i = 0; i < codeSegmentSize; i++)
{
    fread(bye, 1, 0);
    frequencyDistributionTable(byte, 1, byteFrDistrilation);
    calculateProbabilities(byteLevelProbabilities, byteFrDistribution);
    calculateEntropies(byteLevelProbabilities);
}

for(i = 0; i < codeSegmentSize; i++)
{
    fread(halfWord, 2, 0);
    frequencyDistributionTable(halfWord, 2, halfWordFrDistribution);
    calculateProbabilities(hWordLevelProbabilities, halfWordFrDistribution);
    calculateEntropies(hWordLevelProbabilities);
}
```

Figure 3.1 Entropy Tool Pseudo Code

3.2.2. Test Material

Ten representative embedded RISC applications (eight of them taken from the MiBench suite [34]) were selected in order to provide the typical redundancy present in embedded RISC code. While many test bench suites target only specific areas of computation (such as integer
or floating point arithmetic) or application (for example, MediaBench is specifically designed for multimedia applications), MiBench is a free, portable benchmark suite, specially designed to evaluate compiler and ISA performance for embedded implementations and based on a wide variety of representative commercial embedded applications. These applications are divided into the following five categories (with flexible boundaries), according to their market segments: control and automotive, consumer electronics, telecommunications, networking and security. The first category (automotive and industrial control) probably represents the majority of the current embedded system applications, including air bag controllers, image recognition, engine performance monitors, industrial controllers and sensor systems. The test programmes contained in this category are focused on assessing performance in basic mathematical operations, including bit manipulation, data input/output and simple data organization. The consumer electronics sector is currently experiencing rapid expansion, opening the arena to many new embedded technologies, and the selected test programmes for this category are intended to represent the many consumer electronic devices that have grown in popularity during recent years such as scanners, digital cameras and personal digital assistants (PDAs). Another relatively new and growing sector powered by modern embedded system applications is telecommunications. With the explosive growth of the Internet, many portable consumer devices incorporate integrated wireless communications. The benchmarks from this category consist of voice encoding and decoding algorithms, frequency analysis and a checksum algorithm. The network category covers embedded systems that support network devices, such as switches and routers. The benchmark designed for this category focuses on shortest path calculations, lookup tables, search tree generation and processing, and data input/output. Finally, the data security group includes a range of encryption algorithms. A brief description of each of the selected applications is provided below.

**Basicmath** performs simple mathematical calculations that often don’t have dedicated hardware support in embedded processors, such as cubic function solving, integer square root and angle conversions from degrees to radians. The input data is a set of constants.

The **bitcount** algorithm tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers. Five methods are used, including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, non-recursive bit count by bytes using a table look-up and shift and count bits. The input data is an array of integers with an equal number of 1's and 0's.
The CRC32 benchmark performs a 32-bit cyclic redundancy check (CRC) on a file, an approach often used to detect errors in data transmission.

Dijkstra is a benchmark that constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm, a well-known solution to the shortest path problem.

The display application was supplied by ARC International for performing simple tests on ARCAngel, the ARCTangent-A4 development board. It drives a set of user interface indicators, including an LCD display, a 7-segment display and a number of LEDs. It was selected as a part of the test set as its functionality is commonly found in many consumer electronic devices.

The FFT/IFFT benchmark performs a fast Fourier transform (FFT) and its inverse (IFFT) on an array of data.

The qsort program sorts a large array of strings into ascending order using the well-known quick sort algorithm.

Sha is a secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures.

Shine is an MP3 encoder, whose source code is freely available from the web [35]. The baseline source code contains no processor-specific optimizations. The encoder's code includes primarily 16-bit integer and double precision floating-point code.

The susan image recognition package was developed for identifying corners and edges in magnetic resonance images of the brain. It is also capable of image-smoothing and has adjustments for threshold, brightness, and spatial control. In this implementation, the input data is a black and white image of a rectangle.
3.2.3. Lossless data compression algorithms

As discussed in Chapter 2, data compression algorithms can be divided into three categories: ad hoc, dictionary and statistical. Ad hoc algorithms are not used in this study, as they are not suitable for compressing 32-bit RISC code as there is not sufficient regularity in the op-codes and the symbols would vary widely between programs. Dictionary algorithms are the most commonly used as they present a good combination of processing speed and compression efficiency. Statistical techniques are more complex to implement, but achieve typically the highest compression ratios. Some of the most popular compression algorithms from both the dictionary and statistical categories have been selected for evaluation and they are briefly described next.

LZ Family of Algorithms

LZ encoding [36] is a popular adaptive dictionary technique commonly used in text compression applications, which replaces phrases with pointers to their previous occurrences. A phrase might be a word, part of a word, or several words. The pointer is usually a pair \((m, \ell)\), where \(m\) is the position of the input string and \(\ell\) its length. References can be recursive, that is, they can point to other pointers. There is a whole family of algorithms based on LZ technique, each member of which reflects different design decisions. The two main distinguishing characteristics are the size of the history buffer (that is how far back in the file a pointer can access), and which substrings are allowed as targets for a pointer. The history buffer can be unrestricted (growing window) or limited to (typically) several thousands of words (fixed-size window). The choice of characters can be also unrestricted or limited to a set of phrases that are chosen according to some heuristics. Each combination of these choices represents a trade-off between performance (execution time), memory requirement and compression ratio. Decoding is quite simple – the decoder replaces the pointer with its corresponding phrase.

Three different commercial algorithms based on LZ are used in this study. The **LZS** algorithm [37] (designed and commercialised by Hifin Inc.) has a fixed history window of 2kbytes. The special characteristic of this implementation is the ability to specify the level of searching effort carried out to find the best matching (longest) string, allowing the compression ratio to be traded for execution time.
A more complex implementation of the LZ algorithm, which is used in this study, is data compression according to Lempel and Ziv, (DCLZ) produced by Advanced Hardware Architectures Inc. [38]. The algorithm allows the resetting of the dictionary if a poor compression ratio is detected and makes use of a number of control codes. The dictionary entries have lengths in the range 2 and 128 bytes and uncompressed bytes in the output string are also encoded so that they fall into a specified range (8..263). The algorithm has been used for compressing data in a range of commercial devices, including high speed data communication systems, high resolution laser printing devices and SCSI host-bus adaptors. Adaptive Lossless Data Compression (ALDC) is another implementation of the LZ family of algorithms developed by IBM [39]. The two major differences between ALDC and DCLZ are the variable size of ADLC’s history buffer, and the use of control codes in DCLZ.

Prediction by partial match (PPM)

PPM is a compression technique based on probability estimation. It uses an adaptive statistical modelling technique that blends together different length context models to predict the next character of the input. The models record the frequency of characters that have followed each of the contexts. For example, if a particular context happens to be "thei", then all the characters that have followed this context are counted and the next time the context "thei" occurs in the text, these counts are used to estimate the probability of the next character. A feature of PPM is that it operates for different context lengths (for example, "hei" as well as "ei"), to arrive to an overall probability distribution, and that arithmetic coding can then be used to optimally encode the character with respect to this distribution. Variants of PPM have been developed for different context lengths and local order estimation (that is, selecting a particular context model from all the current context models). The executable used in this study (PPMZ2) includes a number of coders based on PPM techniques, including order-12 PPMdet, order-8 PPMZ, order-5-4-PPMQ and order-3-2-1 PPMC [40].

X-MatchPro

X-MatchPro is an adaptive dictionary based algorithm that allows partial matching of incoming data with the data stored in the dictionary [41]. The data is read by tuples of 4 bytes, and each incoming tuple is compared with the dictionary entries. A full match occurs when all the bytes of the incoming tuple match a dictionary entry and a partial match occurs when at least two of the bytes of the tuple exactly match a dictionary entry. In the case of a partial match, the bytes that do not match are transmitted literally. The code prefixes of each
compressed tuple indicate its match location in the dictionary and the match type, thereby specifying which bytes matched the specified dictionary entry. In the case of miss, a single bit is added to the incoming tuple and it is placed in the dictionary. The dictionary is maintained using a move-to-front strategy, whereby the current tuple is placed at the front of the dictionary and other tuples move down by one location. XMatchPro is a high bandwidth algorithm, allowing very fast real-time processing, but the compression ratios achieved are moderate.

Among the set of algorithms discussed above, PPM provides, in general, the best compression ratios, but its performance comes at the expense of a highly complex implementation that makes this algorithm unsuitable for real-time commercial embedded applications. Conversely, XMatchPro's architecture is targeted towards high-speed implementation, and, although it offers moderate compression ratios, it is the fastest compression implementation currently available [41]. The LZ family has many commercial features allowing the user to select an appropriate compromise between compression performance and implementation complexity, making it very suitable for a wide range of applications.

3.3. Experimental trials

The test applications described in the previous section were compiled for ARCTangent-A4 microprocessor, using the hcarc compiler supplied by ARC International, and the entropy of each measured using the entropy measurement tool described in subsection 3.2.1. The following section presents the results of the experiments described to determine the entropy of the executable codes.

3.3.1. Entropy measurements

The entropy of a program's code can be defined as the minimum number of bits necessary to encode it. It is important to note that entropy can be evaluated only relatively to an estimate of the probabilities of occurrence of the symbols that comprise the program. Thus, the entropy
depends not only on the choice of model but also on the set of symbols selected for generating the probabilities.

Although, in general, better entropies can be achieved by applying more sophisticated models (for example, using units of compression with variable lengths), the practical overheads (such as complexity of hardware, slower execution speed, realizing in a larger physical area and the requirement for greater power usage), required to support the decompression process, makes this approach unrealistic in most cases.

As mentioned in Chapter 1, this research has, as primary objective, the design and development of a compression architecture suitable for demanding embedded RISC applications where memory requirements are intimately coupled with real-time performance. Therefore, the solution developed in this thesis (see Chapter 4) aims to balance three major aspects: compression ratio, performance (increasing significantly the cache-hit ratios and ensuring that the decompression engine can be clocked at least at the same frequency as the host RISC processor) and flexibility.

An important part of the solution is its compression model. A zero-order fixed-context model was selected because it provides good entropy results at low (hardware) costs (see section 2.2.1). Once the model is identified, the next step is to determine the appropriate number of bits for the input symbols. Table 3.1 present the entropy results of the test programmes included in the test bench (compiled for ARCTangent-A4), for three different symbol sizes (8, 16 and 32 bits).

Table 3.1. Entropies for a range of programs compiled for the ARCTangent-A4

<table>
<thead>
<tr>
<th>Programs</th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>Actual size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>299594</td>
<td>220038</td>
<td>128439</td>
<td>405632</td>
</tr>
<tr>
<td>bitents</td>
<td>250696</td>
<td>182378</td>
<td>106432</td>
<td>341216</td>
</tr>
<tr>
<td>crc</td>
<td>239291</td>
<td>173923</td>
<td>101144</td>
<td>325088</td>
</tr>
<tr>
<td>djkstra</td>
<td>293427</td>
<td>214615</td>
<td>123899</td>
<td>396256</td>
</tr>
<tr>
<td>display</td>
<td>65248</td>
<td>45106</td>
<td>25291</td>
<td>91520</td>
</tr>
<tr>
<td>fft</td>
<td>252044</td>
<td>184730</td>
<td>107903</td>
<td>341888</td>
</tr>
<tr>
<td>qsort</td>
<td>294425</td>
<td>215415</td>
<td>126036</td>
<td>398048</td>
</tr>
<tr>
<td>sha</td>
<td>241043</td>
<td>176015</td>
<td>102552</td>
<td>327520</td>
</tr>
<tr>
<td>shine</td>
<td>492695</td>
<td>374779</td>
<td>226969</td>
<td>662656</td>
</tr>
<tr>
<td>susan</td>
<td>445214</td>
<td>333075</td>
<td>195031</td>
<td>606976</td>
</tr>
</tbody>
</table>
Figure 3.2 shows, that for 32-bit symbols, the original code appears to have a redundancy of up to 69%. Clearly, embedded RISC code has significant redundancy, and an appropriate compression model should be able to reduce the number of bits in the program.

![Figure 3.2. Ratio between entropy and actual programs' sizes](image)

As stated before, the selection of a model that matches the requirements of a particular compression application is certainly necessary in order to achieve good compression. However, as Table 3.1 demonstrates, another decisive factor is the selection of the appropriate number of bits for the input symbols. During the experiments the smallest individual entropy achieved for 32-bit input symbols (which corresponds to a instruction word size) was 4.3, while the largest was 14.63. This not only means that the code size reduction that could be achieved is more than 50%, but it also provides a suitable range for use when determining the appropriate lengths of the codewords that should be employed in the compression algorithm.

### 3.3.2. Evaluation of data compression algorithms

To take advantage of the redundancy in the RISC programs as found in the previous section, compression can be used to produce more compact executable code. Section 3.2.3 introduced the most popular compression algorithms in the field and in this section, the same test bench
used for entropy evaluation is used to assess the code reduction that these general-purpose compression techniques can achieve on RISC code.

The compression results presented in Figure 3.3 are obtained using history and input buffers (for the LZ family and XMatchPro) set to 2 kbytes and a dictionary length for XMatchPro of 256 bytes (64 entries).

As expected, PPMZ provides the greatest reduction in code size, generating compressed executables that are typically 34% of the original length. The compression results of the LZ-family followed closely, with ALDC (0.43 mean compression ratio) consistently outperforming the other two algorithms. The worst mean compression ratios were achieved by DCLZ (0.55) and XMatchPro (0.53).

![Figure 3.3. Compression performance of a range of algorithms applied to the test-bench programs](image)

These results can be improved by fine-tuning the settings and parameters of each algorithm, such as input buffer or dictionary size. The effect that the values of these two parameters have on the final compression ratios is discussed below for three different compression algorithms for which it is relevant.

LZS
Figure 3.4 (a) shows the effect of varying the input buffer size on an LZS implementation. As can be seen there is a clear trade-off between buffer size (which is directly related to the hardware complexity and decompression performance) and compression ratio. As is expected, a small buffer size (such as 32 or 64 bytes) provides almost no improvement on the final code size. On the other hand, in code-compression applications, big buffer sizes will often introduce significant performance overheads related with the decompression and processing of jump instructions.

DCLZ
This implementation shows no significant response to a change in size of the history buffer. As Figure 3.4 (b) shows, all the test programmes present almost no variation in their compression ratios for all the range of history buffers tested. Unfortunately, IBM's implementation does not allow the setting the buffer to sizes smaller than 512 bytes, thus preventing the evaluation of the algorithm at these very small block sizes.

XMatchPro
This algorithm behaves in a similar way to LZS with respect to changes in the input buffer length (see Figure 3.4 (c)). Although the LZ family of algorithms are formally categorised as 'dictionary' techniques, their implementation does not include a separate dictionary. Thus the only algorithm (among the ones described in this chapter), which can be used to evaluate the influence of the dictionary size on the compression performance is XMatchPro, which features dictionary sizes between 128 bytes and 4 kbytes. Figure 3.5 shows the compression ratios obtained for an input buffer of 4 kbytes for a range of different dictionary sizes. As intuition would suggest, an increase in the dictionary size does indeed bring an improvement in the compression ratio. However, as the dictionary size is increased and approaches the size of the input buffer, the proportionate improvement diminishes.
Figure 3.4. Compression ratio results for different input block sizes
Several conclusions can be drawn from the results presented in this subsection. Firstly, it is clear that compression algorithms can significantly reduce the redundancy of RISC code achieving an average compression ratio of 0.46. Secondly, the compression results are highly dependent on the selected parameters, where larger input buffer and dictionary sizes significantly improve the compression performance of the algorithms. This suggests that the compression algorithm to be developed by this work should allow for certain flexibility in a number of parameters to be finely tuned to the characteristics of the program's code.

3.4. Conclusions

This chapter presented a study of the entropy of the embedded RISC code, together with an investigation of the performance of a number of state-of-the-art lossless data compression algorithms.

The entropy of the code was numerically quantified by the use of several real-life applications, representative of those found in embedded systems compiled for commercial RISC processors. The results obtained demonstrated the presence of high levels of...
redundancy in the executable RISC code, whose representation was found to frequently exceed the most highly compressed form by a factor of three. Apart from demonstrating the feasibility of the investigation of RISC code compression, the entropy study provided a number of additional useful outcomes. For example, the results suggested that a suitable unit of compression is the 32-bit instruction word, while the lengths for the codewords, ranging between 5 bits for the instructions with higher probabilities and 14 bits for those with lower probability of occurrence, might result in the most efficient representation.

In the second part of this chapter, the suitability of compression as means of removing redundancy from the code was examined. A number of the most popular compression algorithms were presented and evaluated, achieving excellent compression ratio results (0.46 on average for the test bench programmes). This demonstrated that lossless compression can be successfully and usefully applied to RISC executable code.
Chapter 4

COMPRESSION ALGORITHM, TOOLS AND DESIGN FLOW

The experiments described in Chapter 3 provided insight regarding the degree and nature of the redundancy in embedded RISC code, as well as indicating the relative performance of different compression algorithms when applied to embedded code, while disregarding hardware implementation issues. As described in Section 2.5.1 in order to be suitable for embedded code compression, an algorithm should:

- allow for a highly efficient hardware implementation that not just avoids performance degradation (lengthening of execution time), but enables improvement. This algorithm will need to differ from those ones presented in Chapter 2, since the code size reduction achieved by them was at the cost of performance;

- be able to work from a limited context, where the program is decompressed in small blocks of length typically no greater than that of a cache line;

- provide for real-time non-sequential decompression, so that it can cope with changes to the execution flow of the programme.

The above issues are considered in this chapter. The chapter describes a novel algorithm suitable for high-performance real-time embedded applications, whose seamless hardware implementation does not come at the expense of degradation in the compression ratios relative to the algorithms presented in Chapter 3, and that, in most of the benchmark problems considered, improved the overall system performance.
4.1. Overview of the proposed solution

This section provides an overview of the design decisions leading to the compression solution developed in this study. Details of the algorithm's implementation are presented in later sections.

One of the particular characteristics of code compression is its asymmetric nature, where compression and decompression take place in separate environments and, therefore, have to comply with different constraints. The compression process takes place off-line, on a host development machine, and is, therefore, substantially free from memory and performance restrictions. Generally, the compressor can be implemented in software, and can make use of relatively complex and time-consuming code analysis methods in order to achieve optimum encoding. On the other hand, decompression occurs in the embedded environment and needs to be carried out in real-time, as embedded products are often targets of strict performance requirements. Nevertheless, overheads in terms of area usage, power consumption and execution time are inevitable as decompression involves additional processing of the instructions fetched from the memory at run-time. The hardware implementation will need to not only minimize overheads, but also counterbalance commensurate drawbacks by providing an overall performance gain as well as a reduction in memory requirements of executables.

This study has achieved both a significant reduction in the size of embedded RISC executables and an increase in the system performance by:

- Using code compression in order to obtain executables with sizes close to their entropy levels, thus reducing significantly the redundancy levels of the original code.

- Relocating the boundary between compressed and uncompressed spaces (in order to obtain a significant increase in the instruction cache hit ratio), hence improving the overall performance of the system. Alternatively the cache memory could be reduced in capacity while maintaining the same hit ratio.

In most of the compression schemes described in Chapter 2, the boundary between compressed and uncompressed space is located between the ICache and the instruction memory, the decompression process being triggered by a cache miss. Only the instructions in memory are in compressed format, while in the ICache they remain in their original
uncompressed form. Relocating the boundary between the ICache and the processor core allows the ICache to hold compressed instructions, enlarging its effective size. As will be seen in the results presented in Chapter 7, this virtual increase in the ICache size has a significant effect on the performance of the system resulting from substantial improvements to the ICache hit ratio, and amply over-compensating, in the majority of the cases, the decompression-related overheads. Furthermore, this scheme does not require any changes to the processor core or to the ICache architecture and the whole compression/decompression system (IP-core and support software) could be implemented as a plug-in for System-on-Chip development tools.

4.2. Compression Algorithm

The initial intention for the compression algorithm developed in this work was to mirror the instruction encoding lengths typical in x86 processors, in order to obtain the code representation benefits of its CISC ISA. This implied replacing the most commonly-used instructions with codewords of length 8, 16 or 24 bits, thus preserving memory alignment and keeping address translation relatively simple. However, after performing a series of tests, it became clear that the algorithm should provide greater flexibility, by allowing a wider range of classes and codeword lengths. This flexibility resulted in significant improvements in compression ratio, at the cost only of losing the byte-alignment of instructions. This section presents the algorithmic design decisions and the implementation details of the developed compression scheme.

4.2.1. Algorithmic design considerations

Code compression is performed in two stages, namely modelling and coding. In the first stage, a model well suited to the characteristics of the code is selected and used to obtain the probabilities of the symbols in the embedded program. The modelling requires that three important parameters, namely the unit (or symbol) of compression, the type of the model and its order, are determined. These parameters, together with the defined encoding algorithm, specify the compression algorithm. Due to the influence that these modelling decisions have
on compression performance, the process involved in obtaining suitable parameters are now
considered in detail.

In a number of the code compression studies, the instruction words are divided into
smaller building blocks (half-words), which are compressed separately. Examples of this
approach can be found in IBM’s CodePack system [7] and the compression scheme
implemented in Motorola MPC5xx microcontrollers [19]. This approach has been
demonstrated to be very successful when used with code that incorporates only a small
number of unique half-words that have high probabilities, or where the data contains some
half-words that have a much higher probability than others. However, as the entropy study
proved (see Section 3.3.1), such a scheme does not produce the best compression
performance for ARCTangent-A4’s code (or for other RISC processors), due to the large
number of unique half-words normally present in the code. Studies that utilised compression
symbols longer than a single instruction, such as expression trees or instruction patterns (see
Section 2.5.1. for description of TBS and PBS), achieve compression results that are poor in
comparison with using the whole instruction word. Therefore, based on the outcomes of the
previous studies described in Chapter 2 and the entropy results presented in Figure 3.2, single
instruction words are used as compression units in this work.

The remaining two parameters defining the model, namely the type and order of model,
are determined mainly by the environment in which the compression process is carried out.
Modelling of the embedded code, as already discussed, takes place on the host development
machine and is not normally subject to strict timing constraints. This allows for a static model
to be used, in which the application to be compressed can be parsed many times in multiple
passes in order to determine accurately the probability of each symbol. This is likely to
improve compression ratio in comparison with dynamic modelling, as the probabilities of all
the symbols are more accurately known before encoding starts.

The model chosen for implementation in this work can be classified as a fixed-context 0-
order model. This means that the probabilities of individual compression symbols will be
estimated without taking into account either the preceding or subsequent symbols, and the
probabilities are entirely independent of the location of the symbols in the source. This choice
is dictated according to the following line of reasoning. The code is unlikely to contain a
sufficient number of repeated blocks of two or more consecutive instructions, where all the
opcodes, register names and immediate values match precisely. The absence of such repeating blocks means additional complexity introduced in the hardware decompressor by implementing the support for higher-order models, is not warranted. Moreover, should such repetitions of blocks of instructions appear frequently in the code, they would be normally identified and optimised by the compiler when a space optimisation option is set.

Following the definition of the parameters of the model, the encoding algorithm also needs to be specified. There are a number of encoding algorithms available, originally developed mainly for text and data compression, but which can be adapted for compressing code. However, due to the restrictions imposed by real-time embedded processing, many of these algorithms proved to be unsuitable. A clear example is arithmetic coding that requires extensive use of multiplications and divisions. Other encoding techniques, such as Huffman and Shannon-Fano [12] coding, produce variable-length codes, where the number of classes (sets of codes of the same length) is dictated by the number of symbols and their probability distribution, and not by the designer. Hence, if this technique were to be used for code compression, the decompressor needs to be able to manage an undetermined (and often large) number of different classes, which would result in highly complex solution that is unsuitable for real-time hardware implementation. Should variable-length encoding be employed by the compression algorithm, then the following two requirements should be followed in order to achieve an efficient real-time-based hardware decompressor. Firstly, the number of classes should not be determined by the characteristics of the programme to be compressed (such as its length and probability distribution of instructions). Secondly, the number of classes should be small.

The following section describes in detail the compression algorithm developed in this study in order to achieve an efficient hardware decompressor architecture that is particularly suited to operation in demanding real-time embedded applications.

4.2.2. General Description

The compression algorithm that has been developed for this study can be described as a class-based dictionary algorithm, with a 32-bit instruction word as the symbol of compression. The compression process consists of the following two stages. Firstly, the executable file produced by the tool chain (compiler/linker) is analysed and the frequency of occurrence of each
instruction is determined. Secondly, the instructions with the highest frequencies are assigned codewords and placed in a dictionary table. As Figure 4.1 shows, during compression the instructions are replaced by their corresponding codewords and a new, compressed executable file is generated that includes uncompressed instructions not assigned a codeword.

![Figure 4.1. Uncompressed to compressed file conversion](image)

Codewords are divided into two classes. The first (class 1) has a shorter word length and therefore is used to represent the instructions most frequently present (thus, achieving higher compression ratios), while the remainder of the instructions are represented using class 2 codewords that are of longer length. The word length of each class not only determines the number of instructions that can be represented using this class, but also affects the compression results. Furthermore, two programs using classes of the same length will, in general, yield different compression ratios. To achieve the best compression performance, it is important to determine the optimum size of each class for a particular program, and to have a decompressor architecture capable of dealing with parameterisable class sizes. In embedded applications, there will need to be a practical limit placed on the dictionary size, and will consequently place a constraint on the class sizes. Once the dictionary size is fixed, there will be a finite number of class size combinations and the one that provides the best compression results can then be chosen. After selecting the size of the classes, the next step is to map the most frequently used instructions to a unique set of codewords. The manner in which a codeword is generated is shown in Figure 4.2.

1. As not every instruction in the original program is compressed (due to the finite size of the dictionary), this work uses the most significant bit (MSB) of an
instruction (in the compressed space) to signify whether it is a compressed instruction (codeword) or an uncompressed one. In compression terminology, this bit is called the Escape Bit. This bit was chosen as the ARCTangent-A4 processor used for proof-of-concept has all its base-case instructions’ MSB set to 0. Consequently, this bit is available, assuming that no extensions to the ISA are required.

2. The second MSB of a codeword is used to identify its class, namely 0 for class 1, and 1 for class 2).

3. The remainder of the codeword is an index (the position of the codeword in the class). Note that, in general, the smaller the index the more frequently the codeword will be found in the compressed file. During decompression, this index is used to locate the original uncompressed instruction in the dictionary.

For example, the following instruction 1100010 in the compressed space would represent a codeword (MSB = 1), of Class 2 (second MSB = 1), with index 2 (010).

During the decompression process, an instruction is fetched from the cache and its MSB is decoded in order to determine whether it is a codeword or an uncompressed instruction. In the latter case, the instruction is directly sent to the processor, otherwise the codeword is fetched and its class information bit and index are read. Depending on the class to which it belongs an appropriate different offset is added to the index in order to calculate the precise address of the original (uncompressed) instruction in the dictionary. The offsets are calculated off-line as follows: for class 1, the offset is always 0, while class 2’s offset is equal to the size (the number of codewords) of class 1. Therefore, in the example above (codeword 1100010), if the size of class 1 is 4, the corresponding uncompressed instructions for this particular codeword would be located at position $2 + 4 = 6$ in the dictionary.
This form of codeword representation has proved very efficient during the
decompression of sequential sections of code. However, it will be necessary to implement
mechanisms to support the control flow changes that occur during the compressed program’s
execution. Such mechanisms are discussed in detail in the next subsection of this chapter,
while their hardware implementation is presented in Chapter 5.

In order to evaluate the compression ratios achieved by the proposed algorithm, the same
test-bench presented in Chapter 3 for assessing different compression algorithms has been
used, with the results summarised in Figure 4.3 for six different dictionary sizes. For this test,
class 1 was allowed to vary in the range 4 to 8 bits, and class 2 in the range 8 to 12 bits giving
dictionary sizes from 256 to 4 Kbytes. As can be seen Figure 4.3, the compression results are
very promising (close to 0.5) for dictionary sizes of 4 Kbytes.

![Figure 4.3 Compression ratios achieved for different dictionary sizes](image)

Although increasing the dictionary size initially improves the compression ratio,
eventually, a saturation point is reached where any further increase does not provide any
further substantial compression improvement. The selection of the dictionary size has to take
in consideration also the total size of the code. Unreasonably large dictionary would not
improve memory utilisation, as instructions would be stored in the decompressor’s dictionary
instead of instruction memory. Therefore, although that 8Kbyte dictionary gives better
compression ratio results for the programs included in the test-bench, the dictionary size used for further experiment is 4 Kbytes.

In summary, this subsection has presented a novel compression scheme that provides similar compression ratios to those obtained by the standard text and data compression algorithms presented in Chapter 3. However, the principal advantage of the new class-based method is that it is amendable to an efficient hardware implementation in real-time embedded systems.

4.2.3. Compressed to uncompressed address space mapping

The instructions of a computer programme are normally executed sequentially, except when control flow statements, which will change the point of execution, are encountered. Examples of control flow statements are:

- *Conditional statements*, which may only be executed under certain conditions (typically specified flag states);

- *Loops*, a group of statements that may be executed repeatedly;

- *Subroutines*, a group of remote statements that may be executed before control returns to the point from where the statements were called.

Conditional statements are not further regarded by this study, unless the statements themselves are jumps or branches, as they are always fetched and decoded by the processor in order to evaluate the condition. Consequently, even when such statements are not executed, their own and their following instruction's addresses are consecutive and thus they do not represent true change of flow (COF). The other two cases, namely loops and subroutines, would normally interrupt the sequential fetch process, forcing the recalculation of the address in real-time (by adding an offset to the current address, or by using a different address supplied as immediate data in the instruction). Since, in the solution presented in this work, the processor is unaware of compression, it always receives uncompressed instructions. If these instructions generate a COF, the target address will be always relative to the uncompressed space, and therefore a mapping mechanism needs to exist between the compressed and uncompressed
address spaces in order to allow a compressed instruction to be obtained from an uncompressed target address.

In previous studies, this problem is usually solved by the use of address look-up tables, where each original address (or the address of a block of instructions) has a corresponding entry in the compressed address space [7, 27]. This solution guarantees correct mapping between compressed and uncompressed memory spaces, but requires large buffers, which has a significant detrimental effect on the compression results. Other implementations have modified the processor's architecture in such a way that the programme counter (PC) value corresponds to the address in the compressed space [4]. Since one of the aims of this study is to implement the compression system such that its presence is unknown to the processor, the latter needs to continue to operate entirely in uncompressed space, while it is the decompressor that takes care of locating instructions in compressed space. The mechanisms developed in this research for solving this problem are described in the following subsection and are highly innovative while avoiding the use of large look-up tables and processor modifications. This section will present COF instructions available in the ISA of ARCTangent-A4, thus clarifying the requirements for this scheme.

Relative changes of flow instructions

In ARCTangent-A4's ISA there are three instructions whose execution can result in relative changes of flow, and each has the same format, as shown in Figure 4.4. These instructions are: conditional branch (Bc), conditional branch and link (BLc), and conditional loop set up (LPc) [42].

![Figure 4.4 Relative COF instructions format](image)

For Bc, the displacement value (or offset) is encoded into the 20-bit RELATIVE OFFSET field, which, if the specified condition is met, will be added to the value of the current PC in order to calculate the target instruction address. BLc is executed as Bc, but in addition, it places the address of the instructions that follows it (nextPC) in a special Branch and Link (BLINK) register. The LPc instruction, together with three special purpose registers (LP_COUNT, LP_START and LP_END) provides a mechanism for performing loops
without any delays being incurred by the count decrement or the end address comparison. The functionality of the loop mechanism is illustrated in Figure 4.5.

![Figure 4.5 PC update and loop detection mechanism](image)

If the loop condition is true, then the address of the next instruction is loaded into the LP_START register and LP_END is loaded with the address PC + RELATIVE OFFSET. If the condition is evaluated to false, then a branch occurs to the address PC + RELATIVE OFFSET. The loop mechanism comes into operation only if there are no pipeline stalls, interrupts, branches or jumps. The number of loops that need to be executed is held in another register, the loop counter, LP_COUNT. If LP_COUNT is not equal to 1, then the PC is loaded with the contents of LP_START and LP_COUNT is decremented. Otherwise (LP_COUNT=1), the PC is allowed to increment normally and LP_COUNT is decremented.

To produce a correct address translation during decompression, branch instructions are processed by the compression tool as follows.

1) On detection of a branch, its target address (in uncompressed space) is calculated.

2) The branch is compressed, its codeword is written in the compressed executable and the 16 bits that immediately follow are bookmarked and left empty for a branch appendix to be written.

3) The location of the branch target in the compressed space is available once the entire executable has been compressed. The branch appendix can now be calculated by subtracting the compressed target address from the original
(uncompressed) target address (see Figure 4.6), and written back in its bookmarked space.

As illustrated in Figure 4.6, the first 11 bits of the appendix represent the memory offset (difference) between compressed and uncompressed target addresses. Since the compressed space is not word-aligned, 5 bits of the appendix are used to indicate the offset of the compressed instruction in the 32-bit memory word. At run time, the inverse process takes place: the decompressor calculates the branch target in compressed space by simply adding the appendix to the target (uncompressed) address supplied by the microprocessor.

![Figure 4.6. Calculating the branch appendixes](image)

Adding an appendix after each branch instruction will usually marginally degrade the compression ratio (depending on the application), yet it provides a simple and efficient mechanism for handling relative COF instructions, namely branches, loops and branch-and-links. It is important to notice that these are the vast majority of COF instructions found in typical embedded applications.

**Jumps – absolute changes of flows**

Jump instructions are COF instructions that take a memory address as an argument and, upon execution, modify the PC to this new address. There is small number of different jump types that can be classified, as follows, according to the format of the address found in the instruction:

- *direct jumps*, where the target address is given as an immediate value;

- *indirect jumps*, where the target is calculated at run time and is held in a register;
• jump to a link register value that holds the return address of a subroutine.

How these different types of jump can be dealt with in the compression architecture developed in the current work is described below.

Direct Jumps

Direct jumps are normally the least frequently occurring of the jumps being considered, although they are the easiest to resolve. The target address is either part of the jump instruction word (as a short 16-bit immediate value, which is expanded by the processor to the required 24-bit address form) or is stored in the 32-bit memory word that immediately follows the instruction. During compression, the target addresses are stored in a table together with the corresponding addresses in compressed format. Later, this table, (which also holds the targets of indirect jumps) will be required to form part of the decompressor’s branch management.

Indirect Jumps

An indirect jump (also known as a computed jump) does not specify the address of the next instruction to execute, but rather the argument (register) where the address is held. The actual target address of the jump is generally not known at assembly or compile time, as it is only computed when the instruction is executed. Indirect jumps are mainly used to make conditional, multi-way jumps, and might be generated by a compiler in the following cases [1]:

• switch statements, which select one of several alternatives;

• function pointers that hold the address of a function to be called;

• virtual functions in object oriented languages such as C++ and Java;

• dynamically shared libraries, which allow a library to be loaded at run time and its functions called.

In embedded systems the last case is rarely encountered, while the virtual function addresses are located in a virtual function address table (vtable) by the compiler and thus can be easily resolved. Therefore, this research focuses on resolving the mapping between compressed and uncompressed spaces for switch statements and function pointers. For both of these cases, compilers normally generate a jump table that holds a list of addresses of a set
of routines that can be selected by index. For the specific purposes of this research, compiler engineers at ARC International generated a special code section in the executable that provides the address of the jump table and indicates the number of entries it contains. With this information, a look-up table is built that holds the target addresses of the indirect jumps and their corresponding addresses in the compressed instruction memory. In those cases where the compiler is unable to generate the required jump table information, it is possible to exhaustively determine all the targets of indirect jumps by simulating the application.

Jumps to Blink Register
In the ARCTangent-4 processor, there are two special-purpose registers dedicated to holding the subroutine return addresses, namely BLINK and ILINK. In order to change the control flow to the beginning of a subroutine, the two available instructions are conditional branch-and-link BLex and conditional jump-and-link (JLec). The final instruction of a subroutine is usually Jex %blink (jump-to-blink), whose execution changes the control flow to the address stored in the blink register. As the decompressor must be able to track the return addresses in the compressed space, the solution proposed uses a small, configurable stack, managed by the decompressor at run-time. This stack stores the return address (i.e. PC+4) of a taken branch of the type BLex or JLex. Its size is determined off-line by simulation and the number of entries corresponds to the maximum nesting of subroutines in the application.

Using the above approaches, the mapping from uncompressed to compressed space can be achieved without the need for large look-up tables; details of the hardware implementation can be found in Chapter 5.

4.3. Design flow and development tools

Compiler-based and hybrid architecture techniques for reducing code size require significant modifications to the standard development tools and often to the whole development process. Code compression solutions, including the one proposed by this research, introduce only minimal changes to the whole system, since not only the original hardware architecture (processor and memory system) is unchanged, but also the standard development tools (compiler and linker) remain unaltered. The proposed code compression scheme is thus, effectively, a code-size reducing and performance-improving plug-in module, which might be
easily configured and used where a need for additional performance or extra memory has been identified.

4.3.1. Design flow

Embedded systems design is a complex paradigm of hardware and software development and their integration. With the ever-increasing complexity of embedded programs and pressing time-to-market constraints, software development very often starts far before there is any reliable target hardware in place for its verification. Thus, the quality of the development tools, their effectiveness and ease-of-use, together with the completeness of the design flow, play a key role in the successful fulfilment to time and within budget.

A hardware/software co-design process that includes the integration of a soft-IP processor would have the following software-development stages.

- Software application writing (typically in C and, perhaps, a small proportion of assembly language)
- Compilation and linkage of the software for the target hardware platform, often resulting, for embedded RISC architectures, in a single executable ELF (executable and linkable format) file.
- Testing, debugging and final implementation of the software run from non-volatile memory (internal or external to the embedded system).

In parallel, the following hardware development stages would take place.

- Customisation and simulation of the soft-IP processor core (in the case of this research, a VHDL model of ARCTangent-A4).
- Integration with other hardware modules required by the embedded solution.
- Synthesis and place-and-route of the hardware system for a particular technology and vendor.
Integrating the developed compression scheme in the design flow highlighted above is relatively straightforward. Figure 4.7 shows the hardware/software co-design process including the additional compression stage. In the software process, the only extra step required is to run the linked executable file through the compressor tool in order to generate all the outputs (compressed executable, dictionary and jump tables) necessary for decompression and correct execution of the code in real-time. Based on these outputs, the hardware decompressor can be configured and synthesised as a part of the processor’s architecture. The software-hardware verification can be performed, for instance, using an FPGA development board.

Figure 4.7 Schematic of the hardware/software co-design process including compression
4.3.2. Compressor tool description

The analyses that need to be carried out by the compressor in the development path in order to enable the run-time decompression are shown in Figure 4.8.

First, the code segment of the executable file is parsed in order to gather the necessary statistics regarding the frequency distribution of the instructions. Each unique instruction is identified for sorting according to its static frequency distribution, while the size of the static dictionary, and the length of the codewords for each class are determined by exhaustive analysis. Next, the static dictionary is generated; its entries containing those uncompressed instructions assigned a codeword. Two outputs are produced at this stage: the static dictionary and a configuration file, which includes the optimum codeword lengths. The codewords are encoded using two classes, with class 1 codewords being assigned the most frequently occurring instructions and encoded using a smaller number of bits than those in class 2. Note
that the dictionary generated by the compressor is termed static because its entries contain the most frequently occurring instructions found in the executable file. However, as Chapter 6 will show, the final dictionary used to compress the executable file may contain also a selection of the most frequently executed instructions.

In the second parse, the code segment is compressed. This again involves reading each instruction, searching for an identical entry in the dictionary and, if it is found, replacing it in the compressed executable with its corresponding codeword. At this stage, the instructions are partially decoded, so when a branch or branch-and-link instruction is detected, its offset in the compressed executable is stored and an empty 16-bit entry appended for completion in the final parse. An uncompressed-compressed address mapping table is generated, which holds each instruction’s original address and its corresponding address in the compressed file. This table, together with the jump table information required to address indirect-jump cases, is used to generate the jump-table look-up table.

In the third and final parse, the compressor calculates the branch appendices as described in section 4.2.3 and writes them at their defined (bookmarked) locations in the compressed executable. Finally, the ELF headers are adjusted in accordance with the new sizes and offsets of the sections and segments of the executable.

4.4. Conclusions

This chapter has presented a novel compression algorithm targeted at high-performance real-time embedded systems that would benefit from a significant reduction in the size of their executable code. In contrast with current state-of-the-art compression algorithms discussed in Chapter 3, the new algorithm allows operation from limited context, avoids the use of large buffers to hold decompressed instructions and removes any dependency on previously decompressed instructions. Non-sequential execution mechanisms have also been designed in order to permit compression when there are changes of execution flow. The hardware blocks developed to support these mechanisms, the rest of the components of the decompressor and its operation at run-time will be described in the following chapter.
Chapter 5

DECOMPRESSOR FUNCTIONALITY,
ARCHITECTURE AND IMPLEMENTATION

Chapter 1 put into context this research, introducing the widely acknowledged problem of poor code density inherent to the RISC architecture, while Chapter 2 presented an overview of the different approaches that have been proposed to address this issue. Chapter 3 focussed on the entropy analysis of embedded RISC code in order to quantify the amount of redundancy present in an executable file. Based on these results and the analysis of the compression algorithms described in the previous chapter, a suitable code compression algorithm was developed in this research (as presented in Chapter 4).

The main objective of this chapter is to describe the functionality and architecture of the decompression hardware designed to support real-time high-performance embedded applications. The chapter also presents, for a number of FPGA devices, the hardware implementation results in terms of physical area usage and speed.

5.1. Overview

As discussed in chapter 4, in order to (over)compensate for the decompression overheads (delays), the decompression module is located between the ICache and the core of the processor. This design increases the effective size of the ICache, since its lines will contain compressed instructions, and so will improve the cache hit ratio. This section provides a brief overview of the operation of the ARCTangent-A4 processor’s pipeline (used in this study for proof-of-concept) and the integration of the decompressor within its first stage.
5.1.1. ARCTangent-A4 architecture

ARCTangent-A4 is a 32-bit RISC processor with four-stage pipeline implementation (see Figure 5.1). In the first stage (instruction fetch or $IFetch$), the requested instruction is fetched from the ICache. The request is done by placing the instruction's address on the address bus and setting the request signal, $IFetch$, high. In the event of a cache hit, the instruction is placed on the instruction bus and the $invalid$ signal validates it. Otherwise, on a cache miss, the ICache sends a fetch request to the instruction memory and the processor is stalled until the cache is refilled.

![Diagram of ARCTangent-A4 pipeline stages]

In the second stage ($Decode$), the instruction is decoded and any operands required are fetched from the register file. Where the instruction is a branch or jump, its conditional codes are evaluated. If the condition is true, the jump or branch is taken, the control flow is changed and the PC is updated with the value of the operand supplied. Otherwise, the PC is incremented normally ($PC=PC+4$). In the third stage ($Execute$), the appropriate arithmetic or logic operations are carried out in the arithmetic logic unit (ALU). Finally, at the $Write Back$ stage, the output of the ALU or the result of a load instruction is written to the register file. Load and store instructions use the ALU in stage three to calculate the data transfer address and make the access request to the memory management unit.

Although the basic operation of the ARCTangent-A4 is very similar to other RISC architectures, its configurability, and the fact that it is a commercial processor, supplied as a synthesisable (VHDL) soft IP core, make it an excellent platform for experimentation. With the tools ARC provides to accompany the processor, design engineers can customise its architecture by adding or removing specified hardware blocks. Following the philosophy of
Chapter 5 — Decompressor specification, architecture and implementation

this modular approach, the decompression hardware developed by the author is designed to operate as a plug-in module that can be easily configured and incorporated as a part of the processor system.

5.1.2. Integration of the decompressor

The interface between the processor core and the instruction cache supports a simple, synchronous handshake protocol, described in the previous subsection (stage one). The interface signals for the two modules are summarised in Table 5.1.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc_next</td>
<td>out [23..0]</td>
<td>Address of the instruction to be fetched.</td>
</tr>
<tr>
<td>ifetch</td>
<td>out</td>
<td>Instruction fetch signal. Indicates to the cache controller that a new instruction is required. The instruction will be fetched from the memory at address pc_next.</td>
</tr>
<tr>
<td>ic_busy</td>
<td>in</td>
<td>Instruction cache busy. Set active during tag clearing and line loads from memory.</td>
</tr>
<tr>
<td>pliw</td>
<td>in [31..0]</td>
<td>The 32-bit instruction fetched from the ICache.</td>
</tr>
<tr>
<td>ivalid</td>
<td>in</td>
<td>Qualifying signal for pliw[31:0]. A low value would indicate that the requested instruction is not yet available, and therefore the programme counter should not be incremented.</td>
</tr>
</tbody>
</table>

In order to achieve a seamless integration of the decompressor in the processor's pipeline (one that avoids any changes in the original interface of the processor), the decompressor mirrors the signals presented in Table 5.1. The incorporation of the decompressor into the processor system is shown in Figure 5.2.

![Figure 5.2. Decompressor interface](image-url)
This scheme implies that the decompressor should directly receive the instruction request from the core, translate the instruction's address (expressed in uncompressed space) to the corresponding one in the compressed space, and raise a request for the compressed instruction. Once the (compressed) instruction is available, it is decompressed and sent to the processor's core.

The decompressor itself has a two-stage pipelined architecture (described in detail in the next section), and therefore after being integrated into the processor's architecture, it extends by two stages the processor's pipeline (see Figure 5.3). During the decompressor's first stage (Decompressor Instruction Fetch or Dec IFetch), the compressed 32-bit instruction word is fetched from the ICache and stored in a 64-bit input buffer. Due to its size, the buffer can hold a number of compressed instructions and is refilled independently of the processor's requests. In the second stage (Decompressor Decode or Dec Decode), the instruction is decompressed and sent to the processor. During normal, sequential execution no delays are incurred by the extension of the processor's pipeline. However, at start-up and when a COF occurs, filling the decompressor's pipeline will take four additional cycles, that can be compensated by the decompressor's Branch Management Unit (BMU). The BMU's functional specification and architecture, together with those of the rest of the decompressor's components are described in the following section.

Figure 5.3. Modification of the pipeline of ARCTangent-A4 for decompression
5.2. Functionality and architecture of the decompressor

The role of the decompressor is to make the decompression process completely transparent to the processor, which will run unaware of the fact that the executable code is compressed. One of its main tasks of the decompression unit is, therefore, the handling of misaligned (compressed) memory. The instruction memory, which normally would hold 32-bit instruction words with 4-byte alignment, will, when compressed, contain instructions which start at any bit position within that memory word and may spread across the boundary between two words. In this way, the memory word, which for 32-bit memories is identical to the instruction word, in compressed space might hold a number of instructions, or, alternatively, only parts of two instructions. The same holds true for the instruction cache, where instructions may reside in two consecutive words or be spread across the boundary between adjacent cache lines, the latter requiring two cache line fills to fetch a single, compressed instruction.

The decompressor (shown in Figure 5.4) contains two parts, namely the Decoding Unit (DU) and the Address Translation Unit (ATU). The DU carries out the buffering and decompression of compressed instructions and generates the necessary control signals for the interface the processor and the ICache. The ATU performs the mapping between compressed and uncompressed memory spaces and provides bit memory support. It contains the following modules: Branch Target Cache (BTC), Jump Table, Subroutine Return Stack (SRS) and Compressed PC (CPC) extraction logic. The ATU performs all the address conversion operations, supplies the compressed address to the ICache when a instruction request is made (ensuring that the memory is accessed following the original 4-byte address alignment) and handles the bit-addressing operations internally. Both units (ATU and DU) are synchronised with the processor's operations in order to ensure appropriate execution in the event of a COF or a stall in the processor's core or ICache.

The remainder of this section provides a detailed description of the functionality and hardware architecture of the main components of the decompressor.
5.2.1. Decoding unit (DU)

A functional description of the DU is provided first and then the architecture and design are given.

Functional description
The main tasks performed by the DU are:

- buffering the compressed instructions fetched from the ICache;
- bit-address alignment management of the buffer;
- instructions decompression and dispatching to the processor’s core.

The functional operation of the DU is depicted in Figure 5.5. If the generated dictionary (see Section 4.2.2) is not implemented as a ROM component in the design, it is loaded into the DU’s SRAM. Once the dictionary is in place, the DU waits for a request signal from the microprocessor to start its normal execution cycle. During normal execution, the DU monitors the status of the input buffer, which might be either ‘empty’ or ‘full’. In this particular context, these words mean the following: ‘empty’ signifies that the input buffer still
does not hold 32 bits of valid data, while 'full' means the contrary. As soon as the input buffer becomes 'empty', the ATU places the correct address on the address bus, and sends a fetch request (dec_fetch) to the ICache. When the ICache is ready to supply the requested memory word (which may contain more than one compressed instruction), it enables the che_invalid signal and the decompressor stored the word into its input buffer. Once the buffer is 'full', the decoder starts processing the first 32 valid bits available, and, depending on whether the instruction is compressed, either the relevant uncompressed instruction is retrieved from the dictionary or the instruction is send directly to the processor's core.

![Diagram](Image)

**Figure 5.5. Functional diagram of the decompressor's DU**

**Architecture and design**

The decoding unit consists four main blocks, namely input buffer, decoder, dictionary and control unit. A schematic diagram of these blocks is given in Figure 5.6.
Chapter 5 — Decompressor specification, architecture and implementation

The input buffer is a 64-bit register, tailored to operate as a circular buffer, which stores the 32-bit memory words fetched from the ICache. Note that these memory words may contain between zero and four compressed instructions. The contents of the register are updated by alternating writes to the upper half of the register (from bit 32 to bit 63) or to the lower one (from bit 0 to bit 31), when 32 or fewer bits are valid.

The address logic of the input buffer (see Figure 5.6) keeps track of the number of valid bits that remain as well as a start pointer to the location of the first one of those bits. This pointer is used by a small select logic unit to calculate the select signals for an array of 32 64-to-1 multiplexers, the outputs of which form the input to the decoder.

A second pointer (end pointer) designates the end of the 32-bit word, which will be sent to the decoder. The values of both, the start and end pointers are updated after the decoder has determined the class and the length of the first codeword (or uncompressed instruction) contained in the 32 bits provided. Figure 5.7 shows an example of the operation of the input buffer. In the first cycle, there are 40 valid bits and the start pointer is set to bit 39. Thirty-two
bits (delimited by the start pointer and the end pointer) are fetched to the decoder, which analyses the class information and updates the pointers with the length of the codeword A, which, in this example, is 8-bits in length. In cycle 2, the next 32 bits (from 31 to 0) are output to the decoder and codeword B is extracted. The start pointer is updated to point to bit 23 and, as there is an insufficient number of valid bits available in the buffer (the end pointer falls into a region of non-valid data), a request for a refill is sent to the ICache. While the subsequent 32 bits are being decoded by the decoder, the upper half of the register is filled with valid data, as shown in cycle 3. As the word is longer than the number of valid bits currently being held in the register, the output word from the decoder is invalidated and the pointer positions are not updated. The processing can continue only after the register has been loaded with the remainder of the instruction. However, when the length of the codeword is smaller than the number of valid bits currently available in the buffer (as is the case with codeword D), the pointers are updated and the word is sent to the dictionary.

![Figure 5.7 Operation of the input buffer](image)

It is important to note that the 64-bit register is shadowed, that is, its previous value is stored for one cycle, so that its contents can be recovered when a branch instruction is detected at the next stage of the pipeline. This allows the correct retrieval of the 16-bit branch appendices, which are otherwise treated as normal instructions.

**Decoder**

After receiving 32 bits from the input buffer, the decoder takes the first instruction stored in these 32 bits (instruction A in cycle 1 of Figure 5.7). The MSB is analysed to determine whether the instruction is compressed. When an uncompressed instruction is detected it is stored in a 32-bit pipeline register, to be accessed in the next cycle by the processor. When the
Chapter 5 — Decompressor specification, architecture and implementation

Decompressor detects a codeword (indicating a compressed instruction), its class offset is added to the index field (see Figure 4.1) in order to obtain the address of the corresponding uncompressed instruction in the dictionary. Finally, the decoder transmits the length of the codeword (obtained from the class information) both, to the input buffer that uses this information to move the pointer to the beginning of the next instruction stored in the buffer (see Figure 5.8), and to the ATU, to allow it to calculate the bit-address of the next instruction in the ICache.

![Figure 5.8 The instruction word in the decoder](image)

**Dictionary**

The dictionary module holds the uncompressed instructions that have been replaced by codewords in the compressed programme memory. The dictionary can be implemented either in non-volatile memory or as on-chip RAM (typically SRAM on an FPGA device). Due to its significant effect on compression ratios, performance and memory requirements, the size of the dictionary is configurable to adapt to the requirements of each application.

**Decode control unit (DCU)**

The DCU is responsible for synchronising the interaction between the decoding unit, the ICache and the processor's core. At the heart of the DCU is a finite state machine (FSM), whose structure is presented in Figure 5.9.

At start-up (or after a soft reset) the DCU is in RESET state. When a request signal from the processor arrives (fetch), it moves to COF state. There the input buffer is flushed and a request for refill (dec_fetch) is sent to the ICache. In case of a cache miss, the state becomes STALL and when stage one of the decompressor is valid (the input buffer is 'full'), the transition is made to the CONTINUOUS_PROCESSING state. In this state and as long as the input buffer is 'empty', instructions are continuously decompressed as they are fetched from the cache. This occurs until either the processor or the ICache stall (the later due to an ICache miss), in which case, the DCU re-enters the STALL state. When both the processor and the ICache are again available, the DCU returns to the CONTINUOUS_PROCESSING
state. When a change of execution flow occurs (indicated \( br\_taken = 1 \)), the DCU returns to the COF state.

![State Diagram](image)

**Figure 5.9. Decode control unit state diagram**

### 5.2.2. Address Translation Unit (ATU)

In order to avoid the need for large look-up tables to map the address of each instruction from uncompressed to compressed memory space, a more elegant and area-efficient solution based on resolving each type of COF targets independently has been developed in this work. Functional and architectural description of the hardware required to support these mechanisms is presented in this section.

**Functional description**

Allowing misalignment of instructions within words for the purpose of compression, while retaining word aligned accesses to the instruction memory and ICache, requires that the entire address handling and mapping between compressed and uncompressed memory space is performed by the decompressor. The implementation of such a demanding task requires a number of provisions to be made in hardware, which include:

- extraction of the bit-aligned compressed programme counter (CPC) address;
calculation of the compressed memory branch target addresses;
translation of the addresses of indirect jump targets from uncompressed to compressed space;
keeping track of subroutine return addresses in compressed space.

These functions are performed by the ATU of the decompressor.

The ATU has three modes of operation: start-up, sequential and non-sequential execution. The start-up mode, as the name suggests, takes place during system initialisation. The sequential execution mode of operation corresponds to the normal system operation, where instructions are fetched from consecutive memory locations and executed in an orderly fashion. When a COF instruction has been executed and the address supplied from the processor is not consecutive, that is a jump takes place, the ATU switches to non-sequential execution mode.

**Start-up behaviour**

Figure 5.10 shows the start-up sequence of the address translation unit.

First, the jump table, which is generated by the compressor (see Chapter 4) and holds the target addresses of jump instructions, is loaded into the jump table memory (see Figure 5.4). The decompressor then waits for a request signal from the processor, takes the instruction’s
uncompressed address, searches the jump table and outputs the corresponding compressed address. Once the instruction has been fetched, the decompressor enters into sequential execution mode.

*Sequential execution mode*

The decoder located in the ATU is responsible for detecting instructions capable of changing the control flow, prior to being sent to the processor. Whenever such an instruction is found, the ATU's processing mode changes in accordance with the type of the instruction. If the instruction decoded is not COF, the ATU operates in sequential execution mode (see Figure 5.11), during which the ATU calculates and stores the 27-bit address (the bit-aligned address of the instruction in compressed memory) of the instruction being currently decompressed. When the input buffer is 'empty', the ATU updates the decompressor’s PC \((\text{decPC} = \text{decPC} + 4)\) and raises a request for a memory word fetch \((\text{dec_fetch})\) to the ICache.

![Flowchart](image)

*Figure 5.11 Functionality of the ATU during sequential execution*

*Execution of COF instructions*

Once a codeword has been decoded, its related uncompressed instruction is found in the dictionary and send to the ATU’s decoder in order to check if it is a COF instruction. The ATU must handle all the addressing modes supported by the processor, which are:
- displacement addressing mode (used by branch and branch and link instructions);
- immediate addressing mode (used by direct jumps);
- indirect register addressing mode (resulting from the execution of indirect jumps).

As tailored mechanisms have been developed for handling the different addressing modes, these are presented separately below.

In the event a conditional branch or a branch and link instruction being detected, the immediately following uncompressed instruction output from the dictionary will be discarded, for each branch codeword is followed not by another codeword, but by a 16-bit appendix (see Chapter 4), which will be used to calculate the target address of the branch. In addition, the start pointer of the input buffer (see Figure 5.7) is updated to point to the next buffered codeword (that is, it is increased by 16 in a circular fashion). The course of action that follows is shown in Figure 5.12 and depends on whether the branch is taken (as informed by the \textit{br\_taken} signal shown in Figure 5.4).

- If the branch is taken, then, as the ARCTangent-A4 conditionally executes the instruction that directly follows a branch (delay slot instruction), the codeword pointed by the updated start pointer of the input buffer needs to be decompressed and sent to the processor. Then, the input buffer is reset (flushed) and a fetch instruction request, \textit{dec\_fetch} on Figure 7.4., with the address of the calculated branch target is sent to the ICache. If the branch instruction is of the type branch and link, then, additionally, the address of the instruction following the delay slot instruction is placed on the SRS, which is a stack that holds the return addresses from subroutines (i.e. generated by \textit{JLex} and \textit{BLex} instructions).

- If the branch is not taken, the decompressor continues with its normal (sequential) execution.
Figure 5.12. Functionality related to branch and branch and link instructions

Figure 5.13 shows the decompressor's functionality for the case when a jump instruction is detected by the ATU's decoder. First, the decompressor waits for the br_taken signal from the processor core, and where this indicates the jump is not executed, the CPC is saved and the processing continues in its normal sequential mode. Conversely, if the jump is executed, the instruction in the delay slot is decompressed. At this stage, the actions taken by the ATU depend on whether the jump instruction is direct or indirect.

- For a **direct jump**, the target address in uncompressed space is held in the form of a long immediate value in the 32 bits directly following the jump instruction. Therefore, the operation required is simply to send the next 32 bits stored in the input buffer to the Jump Table in order to convert the target address from uncompressed to compressed address space.
Figure 5.13. Jump related functionality of the ATU

- For an **indirect jump**, the target address (in uncompressed space) is supplied by the processor and the same process is carried out as for direct jumps. If a **jump to BLINK** is being decoded, the address at the top of the SRS is popped from the stack and used as target address (in compressed space). Finally, in the case of a **jump and link** instruction, the target address is retrieved in accordance with the type of jump, and the address of the instruction (in compressed space) following the delay slot instruction is pushed onto the SRS.

The mechanism to handle **loop** instructions mirrors that of the processor (see Figure 4.5): the start and end loop addresses are stored, together with the number of loops that have to be
performed. A counter is decremented in each loop, and, when it reaches zero, sequential execution resumes.

**ATU architectural implementation**

The architecture of the modules of the ATU are presented in Figure 5.14 and described in the following subsections.

![Diagram of ATU modules](image)

**Figure 5.14 Main modules of the ATU**

**Decoder**

The instructions output to the processor are decoded partially in the ATU’s decoder in order to detect COF instructions at as early stage as possible, sending the relevant information, which includes the type of COF detected, to both control units of the decompressor. Based on the decoder’s output, the ATU’s control unit activates the COF circuitry and determines the correct output address.

**CPC**

As previously discussed, ARCTangent-A4 has a 24-bit instruction address bus, where the two least significant bits are not used due to the instruction word-address alignment common in
32-bit microprocessor ISA. This study, however, makes extensive use of these two bits (byte-address alignment), and adds three further bits in order to implement the addressing that permits instructions in the compressed space to start at any bit position in a memory word. The CPC’s architecture is based on a three-entry 27-bit shift-register file, whose write enable signal is the fetch signal from the processor. Reg A holds the address (in compressed space) of the codeword being decoded (by the DU). Its value is periodically incremented by the size of the codeword, unless a COF instruction is executed, in which case the register value is updated with the target address of the branch or jump. The increment can be of $x$, $y$, 16 or 32 bits, where $x$ and $y$ are the lengths of the codewords of the two different classes, 16 corresponds to the length of the branch appendices and 32 to the length of uncompressed instructions. The purpose of having a three-entry shift register is to be able to push into the SRS the correct return address that the subroutine execution requires. Note that subroutines are initiated by JLecc or BLLecc instructions, and terminated by a Jump-to-Blink instruction (Jcc %blink), since the return address of a subroutine (in uncompressed space), is stored by the processor in its BLINK register.

The SRS is a 27-bit wide dual port stack (of configurable size) that holds the return values of all the pending subroutines (ones that have been called but have not yet returned). The minimum size of the stack corresponds to the maximum number of nested subroutine calls that can be issued by the executable, a figure that can be found during simulation or by the use of profiling tools. A special case of subroutine nesting is recursion, that is, where a subroutine calls itself. Although recursive functions are not in widespread use in embedded applications (mainly due to their aggressive and statistically unpredictable use of the system’s stack that is liable to result in stack overflow errors), they are legal syntactical constructs and need to be considered for completeness. Each time a recursive functions call is made, not only its return address, but also its local variables, return value (if such is present) and its parameters are pushed into the stack. Should the number of recursive calls exceed a certain limit, overflow of the stack will result. This research addresses recursion by implementing a simple but effective mechanism in the SRS that, instead of pushing the function’s return address on its stack every time the function is called, keeps track of the number of calls made. The operation is depicted in the following example of the calculation of the factorial of a given number (see Figure 5.15). For this particular example, the SRS would behave as follows (see Figure 5.16).
```c
int main(...) {
    int a;
    func1(...);
    func2(...);
    a = Factorial(4);
    // Return Address [A]
    func3(...);
    ...
}

int Factorial(int number) {
    int intermediate_result;
    if(number > 1) {
        intermediate_result = number * Factorial(number - 1);
        // Return Address [B]
        return intermediate_result;
    } else {
        return 1;
    }
}
```

**Figure 5.15. Factorial calculation illustrating recursive function calls**

Initially, before Factorial(4) is called, the SRS stack is empty (disregarding for the purpose of clarity the return address of the main() function), and the recursive counters are set to zero (Figure 5.16 (a)). After Factorial(4) is called by the main() function, its return address [A] is pushed into the SRS (Figure 5.16 (b)) and the counter is incremented. During the execution of the function, a recursive call occurs (Factorial(3)), and its return address [B] is pushed on to the SRS (Figure 5.16 (c)). The subsequent Factorial(2) call has the same return address as that currently on the top of the stack, and here the address is not pushed again, but instead, its counter is increased (Figure 5.16 (d)). At this point, there are no further calls to the factorial function and as it reaches its return point, its counter is decremented. Return addresses are only popped out when their associated counters are equal to zero and therefore, no address is removed from the stack at this stage (Figure 5.16 (e)). Finally, the factorial function finishes its execution and the stack returns to its initial state Figure 5.16 (f,g)).
Chapter 5 – Decompressor specification, architecture and implementation

Figure 5.16 SRS stack operation

Jump Table

The Jump Table (see Figure 5.17) is implemented as a content addressable memory (CAM), which stores the target addresses of direct and indirect jump instructions in both compressed and uncompressed spaces. The table ensures that the target address provided by the processor (for a jump instruction) is translated from its original uncompressed space into compressed space.

Figure 5.17. Jump table architecture

CAM architectures search the indexed entries in parallel, thus obtaining the entry sought in a single cycle (outperforming any software-based search algorithm). In this implementation,
the index is the target address of a jump in uncompressed space, and the output is its corresponding address in compressed space. In order to save power, the CAM is disabled (cam_enable signal) when not required.

**Branch Target Cache**

The calculation of jump and branch instructions' targets (in the compressed space) requires four additional cycles: one cycle for translating the target address into compressed space, two cycles for loading the target instruction into the input buffer, and one cycle to perform decompression. In order to compensate for these overheads, a configurable branch target cache unit (BTC) is implemented, as shown in Figure 5.18.

![BTC entry fields](image)

**Figure 5.18. BTC entry fields**

The Branch Table holds a list of target addresses (in uncompressed space), while the Instruction 1, Instruction 2 and Instruction 3 tables hold respectively the first, second and third uncompressed instructions originally located at the target address. The last table (Next Address) holds the address of the instruction following Instruction 3, but this time in compressed space. In this way, when the target address of a branch or jump, provided by the processor, is found in the Branch Table, the target instruction is sent without delay to the processor. In parallel, the corresponding entry in the Next Address table is output to the ICache in order to fill the input buffer with the correct instruction (to be output to the processor two cycles later). In the next two cycles, the processor will be given the instructions contained in the Instruction 2 and Instruction 3 tables. In the final cycle, the instruction requested by the Next Address table to the ICache will already have been decompressed and ready to be sent to the processor.

The BTC is divided into two modules of similar architecture (see Figure 5.19). The first is the Static Branch Table that contains the target addresses of the jumps and branches most often...
found in the code and that can be determined following a simulation. The second is the Dynamic Branch Table that is configured in a cache-like fashion to dynamically update the tables, using a Least Recently Used (LRU) replacement algorithm. A trace-based, software tool, which is described in Chapter 6, has been developed to evaluate the required number of static and dynamic entries for each particular application, thus ensuring the best performance for the given area resources. The benefits of introducing the BTC module on the overall system performance will be quantified in Chapter 7.

![Figure 5.19. BTC configurability allows for hardwired and dynamic branch tables](image)

**Figure 5.19. BTC configurability allows for hardwired and dynamic branch tables**

*ATU Control Unit and Output Address (dec_pc_next) Select Logic*

The ATU control unit is implemented as a finite state machine (FSM) that controls the SRS activities, enables the Jump Table's CAM, and determines the values of the `dec_pc_next` and `fetch` signals. This FSM is graphically described in Figure 5.20.
Figure 5.20. ATU control unit finite state machine diagram

The initial state of the FMS is RESET. The ATU waits until a request from the processor arrives and then enters into COF address calculation state, where the BTC and the Jump Table are searched in parallel in order to supply the correct starting address to the cache. If the address is found in the BTC, the FSM enters into BTC state, and will remain in this state for three cycles. If none of the instructions supplied by the BTC to the processor is a jump, then the FSM will move into the Sequential Address Calculation state, where the CPC is incremented and the Jump Table circuitry is disabled. If at any point in this state a branch or jump instruction is detected (equally if a jump instruction is detected in the BTC state), the FSM again enters into COF state. In the COF state, if there is no match in the BTC, then the target address using the mechanisms explained earlier.

During sequential execution, the CPC and dec.pe_next are incremented. In the event of a change of flow, dec.pe_next will be updated with the value of the target address as shown in
Figure 5.21. In case of a processor or ICache stall, the FSM will transit into IDLE state, which will be left as soon as processing resumes.

![Diagram](image)

**Figure 5.21. Output Address Select Logic**

### 5.3. Decompressor implementation

The architecture of the decompressor, as described in Section 5.2, has been implemented in VHDL, is vendor independent and syntheses can be achieved in both FPGA and ASIC (Application Specific Integrated Circuit) technologies. The results provided in this section are related to the particular FPGA technologies listed in Table 5.2.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Technology</th>
<th>Part</th>
<th>Package</th>
<th>Speed Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx</td>
<td>Spartan-IIE</td>
<td>XC2S200E</td>
<td>fg456</td>
<td>-6</td>
</tr>
<tr>
<td>Xilinx</td>
<td>Virtex-II</td>
<td>XC2V500</td>
<td>fg256</td>
<td>-6</td>
</tr>
<tr>
<td>Altera</td>
<td>Cyclone</td>
<td>EP1C6</td>
<td>T144</td>
<td>-6</td>
</tr>
</tbody>
</table>

The synthesis tool used was Synplify Pro v.7.3.1 from Synplicity. The place & route tools used were Xilinx’s ISE (Integrated Software Environment) v.6.1 for Xilinx the devices and Altera’s Quartus II v. 4.1 for the Altera Cyclone.
5.3.1. Configuration parameters

The implementation results depend on the particular configuration of the decompressor. The summary of these parameters and the values for which the decompressor architecture has been synthesised are shown in Table 5.3.

Table 5.3. Decompressor configuration parameters and values used for synthesis

<table>
<thead>
<tr>
<th>Configuration Parameters</th>
<th>Range</th>
<th>Implementation Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jump table entries</td>
<td>1 – 256</td>
<td>32</td>
</tr>
<tr>
<td>Number of hardwired BTC entries</td>
<td>1 – 32</td>
<td>1</td>
</tr>
<tr>
<td>Number of dynamic BTC entries</td>
<td>1 – 32</td>
<td>1</td>
</tr>
<tr>
<td>Dictionary size (bytes)</td>
<td>16 – 8192</td>
<td>16</td>
</tr>
<tr>
<td>SRS depth</td>
<td>1 – 256</td>
<td>4</td>
</tr>
<tr>
<td>Number of codewords</td>
<td>1 – 4</td>
<td>2</td>
</tr>
<tr>
<td>Codeword lengths (bits)</td>
<td>8 – 16</td>
<td>8, 14</td>
</tr>
<tr>
<td>Branch offset size (bits)</td>
<td>8 – 32</td>
<td>16</td>
</tr>
<tr>
<td>Flag bit for uncompressed instructions</td>
<td>0 or 1</td>
<td>1</td>
</tr>
</tbody>
</table>

5.3.2. Implementation Results

The results provided in the following subsections are related to the particular technologies listed in Table 5.2.

Clock rates

The clock frequency achieved by the decompressor depends largely on implementation technology. The clock frequencies obtained after place and routing for the targeted FPGA technologies are summarised in Table 5.4.

Table 5.4. Decompressor's clock frequencies for the targeted devices

<table>
<thead>
<tr>
<th>Technology</th>
<th>Part</th>
<th>Clock Period (ns)</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRTEX-2</td>
<td>x2v500</td>
<td>17.928</td>
<td>55.778</td>
</tr>
<tr>
<td>SPARTAN-IIe</td>
<td>x2s200e</td>
<td>35.437</td>
<td>28.2</td>
</tr>
<tr>
<td>CYCLONE</td>
<td>EP1C6</td>
<td>17.637</td>
<td>56.7</td>
</tr>
</tbody>
</table>

Complexity figures

The complexity figures for the different configurations of the decompressor (see Table 5.3) are presented in Table 5.5 (Cyclone) and
Table 5.6 (Xilinx devices).

Table 5.5. Complexity figures for Cyclone EPIC6 device

<table>
<thead>
<tr>
<th>LEs</th>
<th>M4K RAM blocks</th>
<th>PLLs</th>
<th>Max user I/O pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>3531 (59%)</td>
<td>0</td>
<td>0</td>
<td>119</td>
</tr>
</tbody>
</table>

Table 5.6. Complexity figures for Xilinx devices

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Spartan-IE</th>
<th>Virtex-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Flip Flops</td>
<td>1,637 (34%)</td>
<td>1,610 (26%)</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>2,616 (55%)</td>
<td>2,119 (34%)</td>
</tr>
<tr>
<td>Logic distribution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of occupied slices</td>
<td>2,028 (86%)</td>
<td>1,785 (58%)</td>
</tr>
<tr>
<td>Total 4 input LUTs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...used as logic</td>
<td>2,616</td>
<td>2,119</td>
</tr>
<tr>
<td>...used as a route-thr.</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>118 (41%)</td>
<td>119 (69%)</td>
</tr>
<tr>
<td>Number of GCLKs</td>
<td>1 (25%)</td>
<td>1 (6%)</td>
</tr>
</tbody>
</table>

The decompressor in its standard configuration can be successfully implemented in low-cost FPGA families such as Cyclone and Spartan-IE, including its low gate count model XC2S200E. As the dictionary size used is rather small the memory resources of the targeted FPGAs are not utilised. The utilisation of the RAM blocks of the devices will increase according to the size of the dictionary.

5.4. Conclusions

This chapter presented the functional specifications, hardware architecture and implementation figures (speed and area) of the decompressor unit, which is designed as an application-specific, reconfigurable IP module that extends the pipeline of the processor by two extra stages. The COF issues typical to compressed code execution architectures were introduced and addressed using novel mechanisms that avoid the implementation of large address translation tables (in accordance with most of the work reported to date), thus providing a more refined, scalable and area-aware solution. These mechanisms translate to an
additional delay of up to 4 cycles per executed COF, which commonly are overcompensated by the significant increase in the ICache hit ratio, for modern embedded systems may incur penalties of up to hundreds of cycles on every cache miss. Finally, this chapter introduced the idea of compression-transparent architecture, where the decompressor module can be seamlessly integrated and synthesised on a host processor, without the need of any architectural changes in the processor, which is completely unaware that the instruction memory is compressed.
Chapter 6

METRICS AND TOOLS FOR SYSTEM PERFORMANCE ANALYSIS

In order to develop and improve a particular design or architecture, it is necessary to measure and analyse its performance. The results (and their usefulness) obtained by such an analysis will be directly related to the measurement framework used, including the parameters and metrics selected, as well as the measurement method and the dataset.

This chapter focuses on the description of the measurement framework developed to evaluate the impact of the compression scheme (algorithm and hardware implementation), presented in this thesis, on the performance of embedded RISC systems. The measurement results obtained for the selected test applications are presented in Chapter 7.

The software application developed to automate the compression of RISC executables, and the range of performance measurements applied are described at the end of this chapter.

6.1. Performance metrics

The development of modern consumer electronics products must temper production costs with ever-increasing performance requirements, often leading to solutions with an acceptable level of performance at the lowest possible price. In such an environment, it is important to identify suitable performance metrics that correctly reflect the processing capabilities of the system with respect to a particular set of design requirements. Usually, the performance of a microprocessor-based embedded solution is directly related to the time (in terms of CPU cycles) that processor takes to execute a particular task or program. Although in the marketing literature MIPS (million instructions per second), GFLOPS, TFLOPS (Giga and Tera Floating Point Operations per Second, respectively) are commonly-used metrics, they do not properly reflect the performance of the entire system, since they do not take into account factors such as memory bandwidth, complex hierarchical memory architectures, cache and page misses,
and pipeline issues. Thus, a performance metric that takes into account the whole system and not only the number of instructions that can be executed on ideal circumstances is needed. Such a metric could be defined as [1]:

\[ T_{CPU} = \frac{CPU \text{ clock cycles per program}}{clock \text{ frequency}} \]  \hspace{1cm} (6.1)

where \( T_{CPU} \) is the CPU execution time, defined as the time necessary for the processor to execute a predefined task. As a first approximation, \( T_{CPU} \) could be expressed as the (ideal) number of clock cycles required to execute such a task, and therefore Equation (6.1) could be written as:

\[ T_{CPU,\text{ideal}} = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\text{clock frequency}} \]  \hspace{1cm} (6.2)

where \( IC_i \) (instruction count) is the number of repetitions of instruction \( i \), and \( CPI_i \) is the number of clock cycles taken to execute instruction \( i \) and \( n \) is the number of instructions in the ISA. The number of instructions required to execute a given \( \sum_{i=1}^{n} IC_i \) program is clearly dependant on the processor's ISA, while the CPI\( i \) depends on its architectural design. This model, however, ignores effects on the performance that arise due to memory access and pipeline hazards and, in general, any situation that causes the processor to stall. Pipeline hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle and so reduce the performance from the ideal speedup gained by pipelining. There are three classes of pipeline hazard [1]:

- **Structural hazards**, which arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

- **Data hazards** arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- Control hazards arise from the pipelining of branches and other instructions that change the PC.

Most pipelined processors incorporate additional hardware structures designed to reduce stalling, but these are not effective for all instruction combinations. Stalls effectively reduce the mean CPI of an instruction, thus adversely affecting the processor (and hence, the system) performance.

Memory-related stalls are common in microprocessor-based systems, especially in complex bus hierarchies where a cache miss can easily lead to hundreds of stall cycles when accessing either instruction or the data memory. Data memory stalls that occur when executing load or store instructions are not considered in this study, since only the code is compressed and this has no affect on the behaviour of data memory (or the data cache) performance. The memory stalls considered here result from ICache misses and the number of memory stall cycles can be expressed as the product of the number of cache misses and the cache miss penalty in terms of cycles:

\[
\text{memory stall cycles} = N_{\text{cache misses}} \times \text{miss penalty}
\]  

(6.3)

The actual miss penalty that is obtained when accessing instruction memory can vary significantly. Determining the cache miss penalty for a particular system depends on many parameters, such as the type and access times of the memory, the priority of instruction loads in the memory arbiter, the status of the memory (such as busy or available), the memory controller implementation, etc. The choice of the values for cache miss penalties used in this research for the performance analysis of the compressed memory system is based on two alternative processor implementation scenarios, as presented in Chapter 2. The first, the ARCAngel-A4 evaluation board, is an example of a simple embedded system, in which the processor can directly access the memory. The second, the PNX8550, represents a complex SoC ASIC commercial system, where the memory accesses may be significantly delayed due to a highly complex bus hierarchy with multiple arbitrations; several hundreds or even several thousands cycles could be needed to recover from a cache miss.
Taking into account the additional execution times caused by the presence of hazards and memory access delays results in a more realistic (and therefore useful) representation of the processor execution time. Equation (6.1) can now be rewritten as:

$$T_{CPU} = T_{CPU\_ideal} + T_{delays},$$

(6.4)

where

$$T_{delays} = \frac{\text{system stall cycles}}{\text{clock frequency}}$$

(6.5)

and

$$\text{system stall cycles} = \text{memory stall cycles} + \text{pipeline hazard cycles}$$

(6.6)

Finally, combining Equation (6.4) and (6.5), the following result is obtained

$$T_{CPU} = \frac{\sum_{i=1}^{n} IC_i \times CPI_i + \text{system stall cycles}}{\text{clock frequency}}$$

(6.7)

Substituting Equation (6.6) in Equation (6.7), the final expression for the CPU execution time is found:

$$T_{CPU} = \frac{\sum_{i=1}^{n} IC_i \times CPI_i + N_{\text{cache missed}} \times \text{miss penalty} + \text{pipeline hazard cycles}}{\text{clock frequency}}$$

(6.8)

Equation (6.8) will be used for the analysis of the comparative performance of the original with the compressed memory system.

In order to analyse how the developed compression scheme affects the overall performance of the system, Amdhal's law [1] will be used. Amdhal's law, which is commonly used for calculating the performance gain that can be obtained by improving a particular portion of a computer system, states that if an improvement can be made to a portion of a
system, the increase in performance (in terms of $T_{CPU}$) when using that improvement is the ratio:

$$speedup = \frac{T_{CPU_{ORIGINAL}}}{T_{CPU_{ENHANCED}}}, \quad (6.9)$$

where $T_{CPU_{ENHANCED}}$ and $T_{CPU_{ORIGINAL}}$ are the execution times of a task run on a system with and without the enhancement, respectively.

Assuming that the processor in both the compressed and the original systems runs at the same clock frequency, the impact of compression on the system performance can be calculated by substituting Equation (6.8) in Equation (6.9), leading to:

$$speedup = \left( \frac{\sum_{i=1}^{n} IC_i \times CPI_i + N_{cache\_misses} \times miss\_penalty + pipeline\_hazard\_cycles}{\sum_{i=1}^{n} IC_i \times CPI_i + N_{cache\_misses} \times miss\_penalty + pipeline\_hazard\_cycles} \right)_{ENHANCED}$$

Hence, in this study the main parameters required to calculate the performance improvement of the compressed system are:

- $CPI_i$ of each instruction, which requires the knowledge of the instructions execution cycles.

- $IC_i$ of each instruction executed.

- Number of ICache misses (for the selected ICache configuration) of both the original and the compressed memory systems.

- Instruction cache miss penalty for the particular system architecture.

Section 6.2 presents suitable methods to obtain the values of the above parameters.
6.2. Performance analysis techniques

Techniques for computer performance analysis can be divided into three broad classes: performance measurement (benchmarking), analytical modelling and simulation modelling. Performance measurement (including the use of benchmarks) does not use models but instead relies on direct observation of the system of interest, or a similar representative system. Benchmarking is difficult to perform without actual hardware, which not always is available. Additionally, the measurements that can be collected can be rather limited, depending on the instrumentation installed. All analytical and simulation techniques require the construction of a model, that is, an abstract representation of the real system, but while an analytical performance model is a mathematical construct, a simulation model is a specialized computer program that emulates (at different levels) the behaviour of the real system. Analytical modelling provides a quick way to compare different product configurations and it is cheapest in terms of effort and cost. Its main disadvantage and inaptness for this work is that it does provide only the mean values of the basic performance metrics, such as throughput, latency and utilization. It is not generally possible to obtain estimates of metrics such as number of cache misses or instruction counts, and it does not account for the particularities of each executed programme. Simulations, on the other hand, can be programmed to collect any metric required (up to the level of accuracy provided by the simulation system) and to allow for different system configurations to be evaluated (such as different cache sizes in a processor or different dictionary sizes in a dictionary-based compression scheme). Clearly, the more parametrisable and accurate the model is, the more effort its design and implementation requires.

Only simulation modelling will be discussed further as it stands out as the most suitable technique for measuring the performance of the compressed system developed in this work. Benchmarking is not feasible due to the absence of real hardware (that is a complex embedded SoC with its design fully available to be freely analysed and modified), while analytical modelling is generally limited to providing a guide of some particular macroscopic properties of the system highly averaged over time.

The behaviour of a computer system can be typically simulated using one of the following three techniques [1].
• Profile-based static modelling

• Trace-driven simulation

• Cycle-accurate execution simulation

Profile-based static modelling is the simplest technique to implement since it disregards memory access behaviours and complex pipeline architectures are generally only approximately modelled. The main purpose of this technique is to provide a statistical report (profile) of various well-defined parameters (for example, instruction repetition, total instruction count, cache hits and misses), and often to generate a trace file with the memory addresses that have been accessed. Its simplicity allows for a relatively fast execution compared with the trace-driven and cycle-accurate simulations. Trace-driven simulation is particularly useful for modelling memory system performance and when combined with a static (profile-based) analysis of the pipeline it leads to a reasonably accurate performance model of the processor system. Finally, cycle-accurate execution simulation performs simultaneously both, a detailed simulation of the memory system and the pipeline. Although this technique is the most accurate one, it requires substantial effort to develop such a detailed model of a complex system. Furthermore, the simulation time can be very long (particularly for complex architectures) and, depending on the parameters under observation, a profile-based technique could provide similar results (within a known percentage of error) in much shorter time.

In this study, the uncompressed system (incorporating the ARCTangent-A4 processor) was simulated using ARC's instruction set simulator (ISS), which falls into the profile-based simulator category. The influence of the compression scheme on the system was analysed using a trace-driven simulator developed during this research. Both simulation systems will be presented in detail in the following section.

6.3. Simulation tools

This section presents the tools used for evaluating the performance of the original (uncompressed) and modified (compressed) systems. Initially, the uncompressed system performance is analysed by means of a profile-based simulation tool, namely ARCTangent-A4 ISS, which executes the original ELF executable file. This instruction simulator provides, among other information, the number of instructions executed, the number of cache misses,
and generates a trace file. Once the execution traces are available, they are processed by a trace analysis tool developed in this research, which obtains various statistics such as a list of the most frequently executed instructions and jumps, and counts of instructions according to the type, as well as producing a instruction-address table (IAT). This table is used by a configurable instruction cache simulator (ICS), which by processing the trace file, calculates the number of cache misses. This ICS can be configured to analyse different cache configurations, where the cache type, size and line size can be varied. All the code compression and analysis tools developed in this research have been gathered under a common visual interface (the Code Compression Suite) developed by the author using C++ for Windows platforms. This development suite is described in detail later in this section.

6.3.1. ARCTangent-A4 ISS

The ARCTangent-A4 ISS was provided by ARC International Ltd. in support of the author's research. The ISS is designed to simulate, at instruction level, the execution of ARCTangent-A4 applications, providing a wide range of statistical information such as instruction count, ICache and DCache misses, and whether branches are taken, and is able to generate instruction trace files (see Figure 6.1). The trace files record the activity on the instruction memory bus, recording the number of instructions executed (in decimal format) and the actual instruction executed (in hexadecimal format), its virtual address and its assembler representation.

Each time a conditional instruction is executed, the contents of the flag register is shown on a separate line and the outcome of the conditional test is shown in brackets (either 'execute' or 'do not execute'). The details of data load and store operations are also written on a separate line. The generated trace files do not record the execution of the processor's initialisation code that is common to all ARCTangent-A4 applications, and which usually accounts for the first few thousand instructions. The number and range of instructions to be simulated is set by the user through the use of an ISS command. As a reference, the simulation of 2.5 million instructions results in a trace file of length of about 2GBytes.
6.3.2. Code Compression Suite (CCS)

The CCS provides visual configuration, displays the analysis results and manages the integration and the flow of data and information between the various development and analysis tools. Figure 6.2 shows the main interface of the CCS. As can be seen in this figure, the interface is divided into four main areas, namely Command Bar Area, Working Directory Area, Applications Area and Results Area. The Command Bar allows the user to control the operation of the tool (e.g., compress a file, process a trace-file, view results, configure parameters, etc.). The meaning of each button is summarised in the dialog window shown on Figure 6.3, and this can be obtained at run-time by pressing the help button located on the Command Bar. The Working Directory Area allows manual modification of the working directory, whose structure and meaning will be described in detail later. The user can also browse to this directory using the Command Bar. The Applications Area displays which programs are stored in the working directory, these determining the operations available. Finally, the Results Area displays the statistical information extracted during the processing of the trace files.
Figure 6.2. CCS main interface window. In Applications List window T stands for trace file, E for uncompressed executable, D for dictionary file and IAT for Instruction-Address Table file.

Figure 6.3. Summary of the Command Bar functionalities.

The Working Directory

The working directory holds the applications’ projects for all the applications that the user wishes to process (that is compress and analyse). Each application’s project is located under its own subdirectory, and the list of the files it may contain is summarised in Table 6.1.
Table 6.1. Summary of files that may be found in an application’s project directory

<table>
<thead>
<tr>
<th>File</th>
<th>Extension</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace File</td>
<td>.trace</td>
<td>ISS-generated trace file</td>
<td>X</td>
</tr>
<tr>
<td>Uncompressed Executable</td>
<td>.elf</td>
<td>Original (uncompressed) executable file</td>
<td>X</td>
</tr>
<tr>
<td>Compressed Executable</td>
<td>.comp</td>
<td>Compressed executable file</td>
<td></td>
</tr>
<tr>
<td>Address Map</td>
<td>.map</td>
<td>Compress to uncompressed address mapping file generated by the compressor.</td>
<td></td>
</tr>
<tr>
<td>Static Dictionary</td>
<td>.static.dict</td>
<td>Dictionary based on the most frequently repeated instructions in the code</td>
<td></td>
</tr>
<tr>
<td>Dynamic Dictionary</td>
<td>.dynamic.dict</td>
<td>Dictionary based on the most frequently executed instructions</td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td>.dict</td>
<td>Dictionary used to generate the compressed executable</td>
<td></td>
</tr>
<tr>
<td>Instruction-Address Table</td>
<td>.iat</td>
<td>Instruction-address table file generated after analysing the trace file.</td>
<td></td>
</tr>
<tr>
<td>Dynamic Instruction Analysis Results</td>
<td>.dinst.results</td>
<td>File containing dynamic code analysis statistics</td>
<td></td>
</tr>
<tr>
<td>Dynamic Branch Analysis Results</td>
<td>.branch.results</td>
<td>File containing dynamic branch analysis statistics</td>
<td></td>
</tr>
</tbody>
</table>

The first two files of Table 6.1, namely the trace file and the uncompressed executable, are the only inputs that CCS requires in order to generate the remainder according to the parameters selected by the user. As this table shows, three different dictionaries are generated. The static dictionary is generated based on the most frequently occurring instructions found in the original executable file. The dynamic dictionary contains the most frequently executed instructions found from the analysis of the trace file; clearly its usefulness will depend on how well the generated trace file is able to characterise a typical execution of the application. The third dictionary is a combination of the static and dynamic two dictionaries, with the constituent proportions selected by the user. As shown in Figure 6.2, the Applications Area displays information about the existence or otherwise of a number of the key files presented in Table 6.1.

System Parameters

The following system parameters are accessible to the user through the Settings dialog window, see Figure 6.4.

- Branch Target Cache (BTC) configuration. This area allows the selection of the number of entries in the static and dynamic BTC tables.
Chapter 6 — Metrics and Tools for System Performance Analysis

- **Codewords and dictionary configuration.** The lengths of the codewords of each class can be specified; note that the size of the dictionary is determined by these lengths (see Chapter 4). The proportion of static and dynamic dictionary entries used to generate the final dictionary can also be set.

- **ICache configuration.** The number of ways and ICache lines, together with the ICache size, can be determined by the user.

- **Executables’ Path.** This allows the user to select the location of the modules that comprise the CCS.

![Figure 6.4 Settings dialog window](image)

**Trace file analysis**

After the system parameters have been set, the CCS is ready to process the trace file, following which the following files are obtained.

- The Instruction-Address Table (IAT) file, which contains the executed instructions and their virtual addresses in uncompressed space. This file is used by the Dynamic ICache simulator.
• The Dynamic Branch Analysis Results file shows the COF instructions sorted by their frequency of execution. This file also stores the number of hits and misses in the BTC module.

• The Dynamic Dictionary file holds the most frequently executed instructions and is generated from the Instruction Analysis Results for the selected and for the selected dictionary size.

• The Dynamic Instruction Analysis Results file, stores the instructions sorted by their frequency of execution. This file also provides other statistical information such as the instruction count, number of executed cycles, number of loads and stores, executed COF instructions, and pipeline stalls. The results stored in this file can be visualised using the Instruction Analysis button of the Command Bar (see Figure 6.5).

![Figure 6.5. View of the dynamic instruction (left) and branch (right) analysis windows](image)

In the ARCTangentA-A4 processor, there are three main causes of pipeline stalls (apart from cache misses), namely long immediate data (LIMM) fetch, flag setting before conditional execution, and data interdependency. The first occurs because some ARCTangent-A4
instructions use 32-bit long immediate (LIMM) data, stored in the instruction memory. Thus, when the processor needs to access such data, it has to fetch it from the ICache, so introducing an additional cycle during which an instruction cannot be fetched. The second occurs when the instruction that modifies the processor flags is followed by a conditionally executed instruction. As flags are set in stage three of the pipeline, whereas the conditional tests that use these flags are carried out in stage two, the pipeline has to stall for one cycle until the flags are updated. The third cause of pipeline stalling occurs when the instruction in the execution stage requires a register whose value has not yet been obtained by a previous instruction. An example of this is the execution of an arithmetic instruction, which has an operand that depends on the result of a previous instruction whose execution is not yet completed.

The first two stall cases are relatively straightforward to analyse during the processing of the traces and the CCS includes their statistics in the Instruction Analysis Results file. However, to account for the data-interdependency stalls, an accurate model of the microarchitecture of ARCTangent-A4 processor would be required. The development of a model of such complexity is out of the scope of what is feasible to achieve in a postgraduate research project. However, in the current work such a model is not required as the compression of instructions affects neither the order in which they are executed, nor the occurrence of data cache misses, and consequently has no effect on the data-interdependency-related stalls.

**ICache Simulator (ICS)**

The ARCTangent-A4 implementation allows the user to choose the type and size of instruction cache (1, 2, 4 or 8 Kbytes of instruction and data cache and cache line lengths of 16, 32, 64, 128 and 256 bytes) according to the requirements of the target application [43]. All instruction cache architectures essentially perform the same operations, in that, on a cache miss, the processor is halted while cache line containing the missing instruction word is fetched to the cache. The main differentiating factor between instruction cache architectures is the organisation of the RAM.

The direct-mapped instruction cache is the simplest supported architecture, where, following a cache miss, the fetched instruction can be copied into only one of the cache lines. The cache line address is derived from number of the least significant valid bits of the address (the index), to point to the physical location in the cache. An n-way set associative instruction
cache requires more complex logic in its implementation, allowing an instruction word to be placed into any one of \( n \) locations in the cache. A set refers to a collection of cache line locations where the instruction can be placed; thus, a set that has four lines is referred to as a 4-way set associative cache. Figure 6.6 shows a 2-way set associative cache.

![Diagram of a 2-way set associative instruction cache architecture](image)

Figure 6.6. Two-way set associative instruction cache architecture

When the cache line is filled with instructions after a cache miss, the tag part of the address is stored into the tag RAM of the cache. Following an instruction fetch request, the tag of the address is compared with the RAM entry pointed by the index tag. In the example, assume that a cache line is made of four 32-bit words, and a four-to-one multiplexer (with a select signal coming from the address) is employed to output the correct word from the line. When the tag of the requested address equals the one pointed to by the index tag entry, a hit is detected and the output instruction is validated by the hit signal. As the figure shows, the cache architecture is duplicated for a 2-way cache and four units would be needed for a 4-way cache. When a miss occurs, the fetched instruction word can be placed in either of the two ways and a replacement algorithm is needed to determine the value stored in the way pointer (wp), which in turn dictates which of the ways receives the data. The replacement algorithms supported by ARCTangent-A4 for the multi-way set associative caches, are round robin (RR) and a pseudo-random algorithm. In RR the wp value is incremented after each instruction cache miss, until
the maximum number of ways is reached when the value of the \( wp \) becomes zero. In the pseudo-random algorithm the \( wp \) is incremented after each successful hit and so it is not easily possible to predict the value held in the \( wp \) when a miss occurs.

The ICache Simulator (ICS) has been specifically developed in the current work to provide an accurate model of all the ARC-supported ICache configurations, together with a functional model of the decompressor hardware. The ICS's flow diagram is depicted in Figure 6.7.

![Figure 6.7 Functional diagram of the cache simulation module](image)

The ICS requires two files to start its execution, namely the IAT (generated during the trace analysis) and the uncompressed-to-compressed address mapping (generated by the compressor). The execution starts by reading a request (in uncompressed space) address from the IAT, which, in a physical embedded system, would be supplied by the processor. This address is then translated into a compressed space address using the address mapping file and appropriate bits of the address are compared against the relevant entries in the tag. If there is a hit, an additional check takes is needed in order to determine whether the requested instruction is only partially present in the current cache line. This is possible when the cache line contains compressed instruction words, as they are not guaranteed to be aligned on 32-bit
boundaries, and so can be stored across memory word boundaries. In case of a miss, a cache miss counter is incremented and the tag is updated with the new address. The ICS repeats these steps for each IAT entry until the end of the file is reached. The number of instructions that the ICS can process is limited only by the memory capacity and processing power of the host machine. These parameters determine the simulation time required to process the whole trace file. As an example of execution time to obtain the results in this study, a 3GHz Pentium4 dual-processor with 1 GB RAM required over 50 hours to process a 2.5 million instructions trace file.

**Code Compressor**

The code compression tool developed in this research is based on the compression scheme described in Chapter 4, and its basic architecture was described in Section 4.3.2. As mentioned there, the compressor tool generates a static dictionary that contains the most frequently occurring instructions. The analysis of the trace file allows the development of a dynamic dictionary that contains the most frequently executed instructions. Finally, the CCS allows the user to select the proportion of static and dynamic instructions present in the final dictionary that will be used by the compressor in order to generate the compressed executable file. This is illustrated in Figure 6.8, where the interdependency of the different modules is depicted.

![Figure 6.8. CCS modules interdependency](image-url)
6.4. Summary

This chapter has presented the metrics and tools used in the current study for analysing and comparing the performance of the compressed and uncompressed systems presented in this thesis. In order to automate and simplify the development and tuning of the compression scheme, the Code Compression Suite (CCS) was engineered. This chapter has presented an overview of its functionality and use, which allows the user to compress executables, generate specific types of dictionary, analyse trace files, obtain instruction and jump/branch-related statistics (including BTC performance) and simulate the different types of compressed ICache system, all under a common and user-friendly interface. The results provided by the CCS for the test bench programmes introduced in Chapter 3, will be presented and discussed in Chapter 7.
Chapter 7

SYSTEM PERFORMANCE – RESULTS AND ANALYSIS

The mathematical foundation and software tools described in the previous chapter are suitable for measuring the performance of processor-based embedded systems. In this chapter, these tools are used to quantify the impact that the code compression scheme proposed in this thesis has on the system performance. Producing accurate performance improvement figures for the scheme is rather difficult, as most of the parameters considered in a study of this type (such as the processor clock frequency, memory, cache and ISA configurations, and implementation complexity) vary significantly from one system implementation to another. Consequently, rather than drawing general conclusions of the benefits of compression of instruction memory, the approach taken is to collect results with the purpose of presenting the effects of compression for a variety of typical embedded system configurations. From these results, it is possible to identify some trends and those are also presented in this chapter.

7.1. ICache simulation cases

The cache size and configuration of the cache influence the performance of a computer system, affecting the execution time of the applications. In order to illustrate this, and how the proposed compression scheme can be seamlessly adapted to the particular needs of an embedded application, the test bench applications have been simulated with a set of 24 different ICache configurations (see Figure 7.1), covering the ICache spectrum supported by ARCtangent-A4.

In total, for the eleven benchmark programs (section 3.2.2), 528 cache simulations were run, 48 for each program, covering 24 different cache configurations for each of the compressed and uncompressed executables. In order to isolate the influence of the ICache configuration on the system performance, the compression parameters were kept constant for all the
programs, with a fully static dictionary of 1024 entries (4 Kbytes) with codewords of 8 and 12 bits being used throughout.

<table>
<thead>
<tr>
<th>CACHE TYPES</th>
<th>CACHE SIZES</th>
<th>LINE LENGTHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped</td>
<td>1 Kbyte</td>
<td>16 bytes</td>
</tr>
<tr>
<td></td>
<td>2 Kbytes</td>
<td>32 bytes</td>
</tr>
<tr>
<td>2-Way Set Associative</td>
<td>4 Kbytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>4-Way Set Associative</td>
<td>8 Kbytes</td>
<td>128 bytes</td>
</tr>
</tbody>
</table>

Figure 7.1 ICache simulation cases

7.2. Performance parameters

Based on the metrics presented in Chapter 6, this section will introduce a new metric, purposefully devised to determine the overall performance improvement that the compression scheme proposed in this thesis provides.

In general, the performance improvement between two processor-based systems can be calculated as shown in equation (6.10).

In the particular case of an ARCTangent-A4-based system, \( \text{CPI} = 1 \) (for all the instructions), i.e. all the instructions supported by its ISA have single cycle execution, and therefore:

\[
\frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\sum_{i=1}^{n} IC_i} = IC_{\text{TOTAL}}
\]  

(7.1)

where \( IC_{\text{TOTAL}} \) represents the number of instructions executed during the simulation.

As explained in Chapter 5, due to the particularities of the decompressor module, every COF instruction requires four cycles to calculate the target address (in compressed memory
space) and to fill the decompressor's pipeline. The BTC module, introduced in Chapter 5, was designed to mitigate these overheads, and it is only the BTC misses themselves that give rise to decompressor overheads. These overheads need to be included in equation (6.10) in order to reflect the compressed system's particular architecture, and the equation (6.10) can be rewritten as:

\[
\text{Speedup} = \frac{(IC_{\text{TOTAL}} + N_{CM} \times MP + PLH)_{\text{ORIGINAL}}}{(IC_{\text{TOTAL}} + N_{CM} \times MP + PLH + BTC_{\text{MISSES}} \times 4)_{\text{COMPRRESSED}}},
\] (7.2)

where $N_{CM}$ is the number of ICache misses of each system, $MP$ is the ICache miss penalty, $PLH$ is the pipeline hazards cycles, and $BTC_{MISSES}$ are the number of misses of the BTC table.

The remainder of this section determines the parameters of equation (7.2) for the test bench programs and the performance improvement of the compressed system is quantified.

### 7.2.1. $IC_{\text{TOTAL}}$ and Pipeline Hazards cycles

For the test bench programs, the value of $IC_{\text{TOTAL}}$ (the number of instructions simulated) was found to fall in the approximate range of one to three million. Neither the ISS nor the CCS imposes a particular limit on the number of instructions that can be simulated; the only limiting factors being the simulation time and memory requirements that a very large $IC_{\text{TOTAL}}$ would impose to the host machine. Note that the compression scheme does not affect the number of instructions in the executable.

In order to calculate the number of pipeline stall cycles that occur on ARCTangent-A4, it is necessary to determine the flags set prior to the execution of conditionally executed instructions and to identify the instructions requiring long immediate data (LIMMS). Table 7.1 summarises the pipeline analysis results and the $IC_{\text{TOTAL}}$ for each test bench application.
Table 7.1. $IC_{TOTAL}$ and pipeline stall cycles for the test bench applications

<table>
<thead>
<tr>
<th>Application</th>
<th>$IC_{TOTAL}$</th>
<th>Pipeline stall cycles due to Conditional instructions</th>
<th>LIMMS</th>
<th>Number of different instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>basemath</td>
<td>1015046</td>
<td>4747</td>
<td>61049</td>
<td>1849</td>
</tr>
<tr>
<td>bitents</td>
<td>959351</td>
<td>418</td>
<td>30211</td>
<td>1578</td>
</tr>
<tr>
<td>crc</td>
<td>1042150</td>
<td>20916</td>
<td>41987</td>
<td>706</td>
</tr>
<tr>
<td>djikstra</td>
<td>1068193</td>
<td>22512</td>
<td>52464</td>
<td>897</td>
</tr>
<tr>
<td>display</td>
<td>791781</td>
<td>24</td>
<td>196839</td>
<td>456</td>
</tr>
<tr>
<td>fft</td>
<td>2376902</td>
<td>17597</td>
<td>121515</td>
<td>2026</td>
</tr>
<tr>
<td>fft</td>
<td>2307810</td>
<td>17518</td>
<td>114512</td>
<td>1889</td>
</tr>
<tr>
<td>qsort</td>
<td>1579600</td>
<td>5561</td>
<td>14679</td>
<td>443</td>
</tr>
<tr>
<td>sha</td>
<td>1000343</td>
<td>4</td>
<td>75740</td>
<td>777</td>
</tr>
<tr>
<td>shine</td>
<td>2023763</td>
<td>15795</td>
<td>103727</td>
<td>1885</td>
</tr>
<tr>
<td>susan</td>
<td>3000130</td>
<td>107821</td>
<td>136649</td>
<td>2065</td>
</tr>
</tbody>
</table>

It can be observed that a substantial number of delay cycles is fairly consistently introduced into all the test bench programs as a result of fetching long immediate data (32-bit data) stored in the instruction memory, accounting for, in many cases, for more than 5% ($iff$, $fft$, $sha$, $susan$) of the total number of execution cycles. While fetching these data the execution stage of the pipeline does not perform any useful work. The number of stalls arising due to flag dependencies varies considerably, being quite significant in some cases ($crc$, $djikstra$, $susan$), but has a minimal influence in others ($bitents$, $display$, $sha$).

7.2.2. Instruction cache performance

Instruction cache misses often give rise to the major source of pipeline stalls in a computer system. The cache miss ratio is application specific. If, for example, the code consists mainly of loops containing a sufficiently small number of instructions that they can be contained in the instruction cache, the cache miss ratio will be relatively low and cache refill times would not have a major influence on the overall system performance. Conversely, if the code contains few repeated sequences, the cache miss ratio would increase and the cache refill delays might impair significantly the system performance. Both of these cases can be found in the results of simulated test applications (see Table 7.2). Another major factor that influences the cache miss ratio is the instruction cache configuration, that is, its size, type and architecture.
## Table 7.2. Number of instruction cache misses for uncompressed memory systems

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Direct Mapped Cache (1 way)</th>
<th>2 Way Set-associative Cache</th>
<th>4 Way Set-associative Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache size</td>
<td>1K</td>
<td>2K</td>
<td>4K</td>
</tr>
<tr>
<td>Line lengths</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>basicmath</td>
<td>77721</td>
<td>49087</td>
<td>26926</td>
</tr>
<tr>
<td>bitcants</td>
<td>16446</td>
<td>10936</td>
<td>6000</td>
</tr>
<tr>
<td>crc</td>
<td>168572</td>
<td>115854</td>
<td>63509</td>
</tr>
<tr>
<td>dijkstra</td>
<td>238232</td>
<td>150756</td>
<td>108812</td>
</tr>
<tr>
<td>display</td>
<td>804</td>
<td>501</td>
<td>365</td>
</tr>
<tr>
<td>ifft</td>
<td>147401</td>
<td>95118</td>
<td>55030</td>
</tr>
<tr>
<td>fft</td>
<td>204095</td>
<td>131149</td>
<td>72466</td>
</tr>
<tr>
<td>qsort</td>
<td>3073</td>
<td>1781</td>
<td>348</td>
</tr>
<tr>
<td>sha</td>
<td>2800</td>
<td>1753</td>
<td>563</td>
</tr>
<tr>
<td>shine</td>
<td>162279</td>
<td>97020</td>
<td>46032</td>
</tr>
<tr>
<td>susan</td>
<td>147031</td>
<td>89082</td>
<td>51650</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>1K</th>
<th>32</th>
<th>32</th>
<th>64</th>
<th>32</th>
<th>64</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line lengths</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>basicmath</td>
<td>76928</td>
<td>48968</td>
<td>26910</td>
<td>18232</td>
<td>14910</td>
<td>9697</td>
<td>6102</td>
<td>4023</td>
</tr>
<tr>
<td>bitcants</td>
<td>10655</td>
<td>6923</td>
<td>5495</td>
<td>3944</td>
<td>4479</td>
<td>3348</td>
<td>2039</td>
<td>1463</td>
</tr>
<tr>
<td>crc</td>
<td>199934</td>
<td>131503</td>
<td>47740</td>
<td>52773</td>
<td>592</td>
<td>397</td>
<td>264</td>
<td>182</td>
</tr>
<tr>
<td>dijkstra</td>
<td>255545</td>
<td>166252</td>
<td>78722</td>
<td>69072</td>
<td>27687</td>
<td>23555</td>
<td>6517</td>
<td>8444</td>
</tr>
<tr>
<td>display</td>
<td>760</td>
<td>473</td>
<td>355</td>
<td>256</td>
<td>271</td>
<td>163</td>
<td>135</td>
<td>78</td>
</tr>
<tr>
<td>ifft</td>
<td>142436</td>
<td>92250</td>
<td>56157</td>
<td>41046</td>
<td>29919</td>
<td>20430</td>
<td>9644</td>
<td>8540</td>
</tr>
<tr>
<td>fft</td>
<td>196531</td>
<td>126669</td>
<td>75821</td>
<td>55861</td>
<td>38243</td>
<td>26143</td>
<td>10998</td>
<td>9683</td>
</tr>
<tr>
<td>qsort</td>
<td>738</td>
<td>445</td>
<td>325</td>
<td>213</td>
<td>279</td>
<td>176</td>
<td>136</td>
<td>78</td>
</tr>
<tr>
<td>sha</td>
<td>3739</td>
<td>2347</td>
<td>503</td>
<td>354</td>
<td>456</td>
<td>307</td>
<td>231</td>
<td>145</td>
</tr>
<tr>
<td>shine</td>
<td>160785</td>
<td>98302</td>
<td>47666</td>
<td>41422</td>
<td>15005</td>
<td>11863</td>
<td>3395</td>
<td>2331</td>
</tr>
<tr>
<td>susan</td>
<td>148560</td>
<td>92311</td>
<td>54092</td>
<td>35410</td>
<td>22484</td>
<td>15594</td>
<td>3759</td>
<td>3460</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>1K</th>
<th>32</th>
<th>32</th>
<th>64</th>
<th>32</th>
<th>64</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line lengths</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>basicmath</td>
<td>83484</td>
<td>53619</td>
<td>24948</td>
<td>17274</td>
<td>15555</td>
<td>9873</td>
<td>6147</td>
<td>4040</td>
</tr>
<tr>
<td>bitcants</td>
<td>10761</td>
<td>7353</td>
<td>5418</td>
<td>3910</td>
<td>4216</td>
<td>2879</td>
<td>2150</td>
<td>1464</td>
</tr>
<tr>
<td>crc</td>
<td>152971</td>
<td>149834</td>
<td>780</td>
<td>26709</td>
<td>481</td>
<td>377</td>
<td>269</td>
<td>192</td>
</tr>
<tr>
<td>dijkstra</td>
<td>263630</td>
<td>167594</td>
<td>84502</td>
<td>67072</td>
<td>17793</td>
<td>21662</td>
<td>3215</td>
<td>2048</td>
</tr>
<tr>
<td>display</td>
<td>635</td>
<td>412</td>
<td>339</td>
<td>233</td>
<td>271</td>
<td>161</td>
<td>135</td>
<td>78</td>
</tr>
<tr>
<td>ifft</td>
<td>151701</td>
<td>100516</td>
<td>53376</td>
<td>37769</td>
<td>29045</td>
<td>18934</td>
<td>9602</td>
<td>7651</td>
</tr>
<tr>
<td>fft</td>
<td>211088</td>
<td>140062</td>
<td>71164</td>
<td>50150</td>
<td>38333</td>
<td>50150</td>
<td>10619</td>
<td>8464</td>
</tr>
<tr>
<td>qsort</td>
<td>675</td>
<td>425</td>
<td>332</td>
<td>214</td>
<td>276</td>
<td>167</td>
<td>137</td>
<td>78</td>
</tr>
<tr>
<td>sha</td>
<td>4070</td>
<td>2543</td>
<td>548</td>
<td>307</td>
<td>408</td>
<td>262</td>
<td>233</td>
<td>144</td>
</tr>
<tr>
<td>shine</td>
<td>171798</td>
<td>107190</td>
<td>41039</td>
<td>33940</td>
<td>15669</td>
<td>12750</td>
<td>1776</td>
<td>1616</td>
</tr>
<tr>
<td>susan</td>
<td>157421</td>
<td>98013</td>
<td>54821</td>
<td>35137</td>
<td>25964</td>
<td>17863</td>
<td>4509</td>
<td>4372</td>
</tr>
</tbody>
</table>
In the following subsections the performance of the test bench programs run on a range of instruction cache configurations are assessed for both the original and the compressed memory systems.

**Instruction cache misses for the uncompressed memory systems**

The number of cache misses for the original, uncompressed memory system was determined by running the test programs on the ISS of ARCTangent-A4 for all the use-cases specified in Figure 7.1. Note that although the $IC_{\text{TOTAL}}$ for some of the programs is very similar, the number of cache misses, presented in Table 7.2, varies significantly. This confirms that as each application has specific execution behaviour, the best way to determine the most suitable cache configuration, as well as appropriate compression parameters, is by simulation.

From the data in Table 7.2, trends in the performance resulting from specific configurations can be derived. Firstly, increasing the size of the cache clearly results in improvement of the cache hit ratio. However, it should be notified that in some cases doubling the cache line length for a fixed cache size gives better results than doubling the size of the cache RAM while maintaining the same cache line length (see the results for *bitents* and *display* for a 2 kbytes cache with 64 bytes line length and for a 4 kbytes cache with 32 bytes length). As far as cache architecture is concerned, for the smaller cache sizes (1 kbyte or 2 kbyte), the simpler direct mapped or two-way set associative caches would, in most cases, perform better in terms of cache misses than the four-way set associative cache type. For the larger cache sizes, however, the results were in favour of the four-way set associative cache.

**Instruction cache misses for the compressed memory systems**

In the proposed memory compression scheme, the instructions in the instruction cache, as well as in the main memory, are in compressed format. This results in an effective increase in instruction cache capacity (and in the length of the cache line), thereby improving memory resource utilization. It is expected, therefore, that the cache miss results of the compressed memory system will improve on those of the uncompressed memory system.

The cache performance figures for the compressed memory system were obtained by use of the trace-based simulation model described in Chapter 6. These are presented in Table 7.3 and, as can be seen, the results for the different cache configurations for the test applications follow closely the trends found in the uncompressed memory system results.
Table 7.3. Number of instruction cache misses for compressed memory systems

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Direct Mapped Cache (1 way)</th>
<th>2 Way Set-associative Cache</th>
<th>4 Way Set-associative Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache size</td>
<td>1K</td>
<td>2K</td>
<td>4K</td>
</tr>
<tr>
<td>Line lengths</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>basicmath</td>
<td>31314</td>
<td>21775</td>
<td>10624</td>
</tr>
<tr>
<td>bitcnts</td>
<td>6591</td>
<td>4514</td>
<td>3801</td>
</tr>
<tr>
<td>crc</td>
<td>73856</td>
<td>63188</td>
<td>31727</td>
</tr>
<tr>
<td>dijkstra</td>
<td>120698</td>
<td>96745</td>
<td>56500</td>
</tr>
<tr>
<td>display</td>
<td>360</td>
<td>231</td>
<td>154</td>
</tr>
<tr>
<td>fft</td>
<td>56426</td>
<td>38043</td>
<td>25080</td>
</tr>
<tr>
<td>qsort</td>
<td>316</td>
<td>202</td>
<td>141</td>
</tr>
<tr>
<td>sha</td>
<td>55189</td>
<td>38029</td>
<td>20746</td>
</tr>
<tr>
<td>susan</td>
<td>87741</td>
<td>54899</td>
<td>31007</td>
</tr>
<tr>
<td>bitcnts</td>
<td>5736</td>
<td>3919</td>
<td>3005</td>
</tr>
<tr>
<td>crc</td>
<td>767</td>
<td>52782</td>
<td>371</td>
</tr>
<tr>
<td>dijkstra</td>
<td>98997</td>
<td>79320</td>
<td>23300</td>
</tr>
<tr>
<td>display</td>
<td>271</td>
<td>196</td>
<td>115</td>
</tr>
<tr>
<td>fft</td>
<td>55051</td>
<td>37062</td>
<td>21908</td>
</tr>
<tr>
<td>qsort</td>
<td>65275</td>
<td>47310</td>
<td>24981</td>
</tr>
<tr>
<td>sha</td>
<td>375</td>
<td>246</td>
<td>218</td>
</tr>
<tr>
<td>susan</td>
<td>64692</td>
<td>45335</td>
<td>14744</td>
</tr>
<tr>
<td>susan</td>
<td>93198</td>
<td>59318</td>
<td>32635</td>
</tr>
</tbody>
</table>
In particular, the number of misses generally decreases as the cache size, the cache line size or the complexity of the cache architecture are increased, and a doubling of the cache line length often benefits more the performance than doubling cache capacity.

Using the results presented in Table 7.2 and Table 7.3 for the number of cache misses for each system, the improvement of cache hit ratio, achieved by the proposed compression scheme can be quantified in percentage by:

\[
\text{cache hit rate improvement} = \left(1 - \frac{N_{\text{comp}}}{N_{\text{original}}}\right) \times 100, \tag{7.3}
\]

where \(N_{\text{comp}}\) and \(N_{\text{original}}\) are the number of cache missed for the compressed and uncompressed memory systems respectively. From the equation, it can be seen that a 60 percent improvement in the cache hit ratio means the number of misses for the compressed cache constitutes only 40 percent of the total number of misses for the original system.

The achieved cache performance improvement for the test applications is visualised in Figure 7.2 and Figure 7.3. The proposed compression technique improves performance for all of the simulation use cases from Figure 7.1, with the degree of improvement ranging from 50 % (half the number of cache misses) to 90 % (ten times fewer misses). For some programs, such as the test application ets, the cache performance improvement is more marked when the cache size is small and divided into more ways (as in the case of 2- and 4-way set associative caches). For others, such as swan, the most significant improvement is achieved for larger cache memories, with longer line lengths. Other applications, such as basicmath, display and fft, show similar improvement regardless of cache configuration. Figure 7.3 presents the mean improvement in cache miss ratio achieved by the use of compression for the selected test applications. The mean improvement falls between 45 and 65 percent for all the simulated cases, with the cache miss ratio of the 4-way set-associative caches being the most dramatically decreased by a mean of around 60 %.
Chapter 7 – System Performance: Results and Analysis

### Basic Math
- **basicmath**
  - **Cache and length sizes**
  - **Improvement [%]**

### Bitents
- **bitents**
  - **Cache and length sizes**
  - **Improvement [%]**

### CRC
- **CRC**
  - **Cache and length sizes**
  - **Improvement [%]**

### Diaketra
- **Diaketra**
  - **Cache and length sizes**
  - **Improvement [%]**

### Display
- **Display**
  - **Cache and length sizes**
  - **Improvement [%]**

### FFT
- **FFT**
  - **Cache and length sizes**
  - **Improvement [%]**

### Fit
- **Fit**
  - **Cache and length sizes**
  - **Improvement [%]**

### Quot
- **Quot**
  - **Cache and length sizes**
  - **Improvement [%]**
Figure 7.2. Improvement in cache hit ratio for a number of the test applications (DM is the direct-mapped cache, 2W is the 2-way set associative cache and 4W is the 4-way set associative cache)

Figure 7.3. Average improvement in cache hit ratio

In order to improve cache performance even further, the dictionary used for compressing the executable code can be built using a dynamic frequency distribution (that is, the most frequently executed instructions are incorporated) instead of a static one. Alternatively, a
suitable mix of both static and dynamic dictionaries could be used. The introduction of a
dynamic dictionary would likely lead to some degradation of the compression ratio, as the
most frequently executed instructions are not always the ones most commonly found in static
code. Experiments showed that using an equal combination of the most frequent static and
dynamic instructions (50 per cent each) can improve the cache performance typically by
further 5%, while decreasing the compression ratio with around 0.02.

The tuning of the compression parameters, such as the capacity and type of dictionary,
allows the compression scheme to be targeted towards achieving either better performance or
an improved compression ratio and can be determined by the user of the compression scheme
using simulation.

7.2.3. COF targets calculation

As discussed in Chapter 6, in the ARCTangent-A4 embedded systems, effect on execution
time of the COF target calculation are minimised by the delay slot instruction execution,
which utilises the extra cycle required for calculating the target address to fetch the next
instruction and conditionally execute it. Consequently, the number of stall cycles has already
been minimised and, therefore, need not be considered any further in this study. However,
when the instruction memory is being compressed, the decompressor recalculates the correct
target addresses in the compressed address space and this process introduces stall cycles into
the extended processor pipeline. In order to decrease the number of such stalls, a dedicated
hardware unit, the BTC, is incorporated in the design to hold a number of target addresses
and the necessary instructions to compensate for the delay. As the capacity of BTC might be
practically limited by silicon area, usage and power consumption constraints, the following
results investigate the effect of the number of BTC entries both on the percentage of COF
instructions that can be determined and on the overall system performance.

The BTC is configurable both in the number of hardwired and dynamically written entries
for a given application, and so simulations need to be carried out to determine the appropriate
BTC configuration that gives satisfactory results for each testbench program. To determine
the effects of altering the capacity and configuration of the BTC, the code compression suite
can be used as described in chapter 6. While processing the traces, the tool builds a table of
the dynamic frequency distribution of the executed COF instructions and logs the number of hits and misses of the BTC for a preset number of hardwired and dynamic entries.

The dynamic branch analysis window of the tool, shown in Figure 6.5, facilitates the BTC configuration definition process by showing the frequency distribution table of the target addresses, their number, and the coverage in terms of percentage of the total number of COF targets.

Table 7.4. COF instructions and coverage of BTC hardwired entries

<table>
<thead>
<tr>
<th>Program</th>
<th>Total COF instructions</th>
<th>COF instructions</th>
<th>50% coverage</th>
<th>75% coverage</th>
<th>90% coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>159869</td>
<td>655</td>
<td>3</td>
<td>6</td>
<td>37</td>
</tr>
<tr>
<td>bitents</td>
<td>156530</td>
<td>516</td>
<td>5</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>crc</td>
<td>163052</td>
<td>216</td>
<td>14</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>dijkstra</td>
<td>143511</td>
<td>263</td>
<td>18</td>
<td>32</td>
<td>50</td>
</tr>
<tr>
<td>display</td>
<td>791781</td>
<td>149</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ifft</td>
<td>263014</td>
<td>674</td>
<td>3</td>
<td>11</td>
<td>57</td>
</tr>
<tr>
<td>fft</td>
<td>366132</td>
<td>674</td>
<td>3</td>
<td>11</td>
<td>57</td>
</tr>
<tr>
<td>qsort</td>
<td>7315</td>
<td>142</td>
<td>4</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>sha</td>
<td>258186</td>
<td>230</td>
<td>4</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>shine</td>
<td>313307</td>
<td>693</td>
<td>2</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>susan</td>
<td>370596</td>
<td>719</td>
<td>6</td>
<td>18</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 7.4 presents the number of hardwired BTC entries for each test program that would be required to store specific fixed percentages of the total number of executed COF instructions. For example, for one of the test applications used (display), a single hardwired entry would provide around 50 per cent coverage of all the targets and two entries would provide coverage of at least 90 per cent. For most of the benchmarks, 16 hardwired entries were insufficient to cover even 75 per cent of the COF targets. The trade-off between silicon area usage and the number of BTC entries is effectively a trade-off between area and performance, as the greater the number of uncovered targets, then the larger the number of pipeline stalls and the further the degradation in system performance. The use of dynamic entries in some cases can, although rarely, improve the coverage. This can be seen in Figure 7.4, which presents the number of BTC misses for several different BTC configurations.
Figure 7.4. Number of BTC misses for various configurations

The results include all combinations of 0, 4, 8 and 16 dynamic entries and 0, 4, 8, 16, 32 and 64 hardwired entries, for each of the benchmark programs. For the programs *bitenst, sha* and *susan*, a small number of dynamic entries can be used together with a few hardwired ones to achieve coverage almost equivalent to a much greater number of solely hardwired entries. For most of the programs a larger number of hardwired entries were more effective at reducing the cache misses and consequently in the next section the performance at system level is carried out with the inclusion of 64 hardwired BTC entries.

In general, it can be concluded that the BTC provides a significant improvement in target coverage, diminishing the effect of the delays due to COF target calculation when the memory of the embedded system is compressed.
7.3. System level performance

The performance of an embedded system depends on a number of application-specific factors, such as instruction execution times, memory access times and cache miss ratios. The importance of instruction cache performance on the total system performance can vary from minor, for systems where the number of cache line fills are few and refill times are short, to being a significant bottleneck when the refill times are longer or more fills are needed, such as when the code contains many branches. As the proposed compression scheme decreases the cache miss ratio, a suitable evaluation of the effect on system performance is to investigate a range of systems with different cache refill penalties. The video processing unit (PNX8550) was described in section 2.1.1 as an example of a complex SoC ASIC that incorporates elaborate networks of busses and arbitration units. The PNX's instruction cache fetches for the integrated RISC processor competes for bus accesses with continuous data and instruction memory transactions of two DSPs as well as a number of video and audio processing hardware blocks. For such complex architectures, an instruction cache miss might take from several thousand up to hundreds of thousands of cycles, depending on the load of the system. At the other end of the complexity scale is ARCAngel-l test board (section 2.1.1), in which the processor is implemented on an FPGA and runs at much lower clock frequency, typically around 50 MHz, than would an ASIC realisation. The memory on the ARCAngel-l board runs at such a clock frequency that refilling the cache line after a miss would take only 14 to 18 processor clock cycles. There are no arbitration units and no multiple bus systems, making the system simpler and the operation of the processor itself more efficient than the corresponding PNX processor, with a reduced number of cache miss penalties and cache performance having a reduced impact on total system performance.

The improvement in performance, achieved by implementing the compressed instruction cache developed in has been obtained for systems with cache line refill penalties ranging from 16 to 2048 cycles, is presented in Table 7.5. A result of 1.0 means that the performance of the system is not changed while a result of 2.13 shows that the system with compression runs about 2.13 times faster than the original.

The results obtained for the test applications demonstrate that the instruction memory compression technique can significantly improve overall system performance. Most of the test programmes show steady increase in the improvement factor when the cache refill time
increases. There are some cases, such as the test programme display, for which only a minimal performance improvement could be achieved. The main reason for this can be either very low cache miss ratio of the original system, or the relatively high number of COF instructions in the code. As display has a small number of cache misses (as low as 76 in the original system), even though the ICache performance has increased with around 50%, there is little scope for improving performance, and in particular to compensate for the COF target calculation delays introduced by the decompressor.

Table 7.5 Performance improvement for systems with different number of ICache refill cycles

<table>
<thead>
<tr>
<th>Program</th>
<th>Cache miss penalties (cycles)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>64</td>
<td>512</td>
<td>2048</td>
</tr>
<tr>
<td>basicmath</td>
<td>1.13</td>
<td>1.47</td>
<td>2.13</td>
<td>2.33</td>
</tr>
<tr>
<td>bitcnts</td>
<td>1.02</td>
<td>1.12</td>
<td>1.56</td>
<td>1.86</td>
</tr>
<tr>
<td>crc</td>
<td>1.23</td>
<td>1.52</td>
<td>1.80</td>
<td>1.85</td>
</tr>
<tr>
<td>dijkstra</td>
<td>1.15</td>
<td>1.45</td>
<td>1.69</td>
<td>1.73</td>
</tr>
<tr>
<td>display</td>
<td>1.00</td>
<td>1.01</td>
<td>1.06</td>
<td>1.20</td>
</tr>
<tr>
<td>ifft</td>
<td>1.14</td>
<td>1.51</td>
<td>2.14</td>
<td>2.31</td>
</tr>
<tr>
<td>fft</td>
<td>1.14</td>
<td>1.52</td>
<td>2.17</td>
<td>2.34</td>
</tr>
<tr>
<td>qsort</td>
<td>1.05</td>
<td>1.19</td>
<td>2.84</td>
<td>3.76</td>
</tr>
<tr>
<td>sha</td>
<td>1.01</td>
<td>1.02</td>
<td>1.18</td>
<td>1.49</td>
</tr>
<tr>
<td>shine</td>
<td>1.13</td>
<td>1.45</td>
<td>2.12</td>
<td>2.32</td>
</tr>
<tr>
<td>susan</td>
<td>1.02</td>
<td>1.17</td>
<td>1.47</td>
<td>1.56</td>
</tr>
<tr>
<td>basicmath</td>
<td>1.11</td>
<td>1.41</td>
<td>1.98</td>
<td>2.13</td>
</tr>
<tr>
<td>bitcnts</td>
<td>1.01</td>
<td>1.08</td>
<td>1.39</td>
<td>1.62</td>
</tr>
<tr>
<td>crc</td>
<td>1.25</td>
<td>1.59</td>
<td>1.95</td>
<td>2.02</td>
</tr>
<tr>
<td>dijkstra</td>
<td>1.21</td>
<td>1.66</td>
<td>2.10</td>
<td>2.18</td>
</tr>
<tr>
<td>display</td>
<td>1.00</td>
<td>1.01</td>
<td>1.06</td>
<td>1.21</td>
</tr>
<tr>
<td>ifft</td>
<td>1.12</td>
<td>1.43</td>
<td>1.96</td>
<td>2.09</td>
</tr>
<tr>
<td>fft</td>
<td>1.12</td>
<td>1.44</td>
<td>1.97</td>
<td>2.10</td>
</tr>
<tr>
<td>qsort</td>
<td>1.01</td>
<td>1.04</td>
<td>1.29</td>
<td>1.65</td>
</tr>
<tr>
<td>sha</td>
<td>1.01</td>
<td>1.05</td>
<td>1.35</td>
<td>2.10</td>
</tr>
<tr>
<td>shine</td>
<td>1.12</td>
<td>1.43</td>
<td>2.06</td>
<td>2.25</td>
</tr>
<tr>
<td>susan</td>
<td>1.02</td>
<td>1.17</td>
<td>1.49</td>
<td>1.59</td>
</tr>
<tr>
<td>basicmath</td>
<td>1.14</td>
<td>1.50</td>
<td>2.23</td>
<td>2.45</td>
</tr>
<tr>
<td>bitcnts</td>
<td>1.02</td>
<td>1.18</td>
<td>1.49</td>
<td>1.81</td>
</tr>
<tr>
<td>crc</td>
<td>1.39</td>
<td>2.79</td>
<td>4.67</td>
<td>5.61</td>
</tr>
<tr>
<td>dijkstra</td>
<td>1.26</td>
<td>3.09</td>
<td>2.46</td>
<td>2.58</td>
</tr>
<tr>
<td>display</td>
<td>1.00</td>
<td>1.01</td>
<td>1.05</td>
<td>1.19</td>
</tr>
<tr>
<td>ifft</td>
<td>1.15</td>
<td>2.13</td>
<td>2.32</td>
<td>2.54</td>
</tr>
<tr>
<td>fft</td>
<td>1.15</td>
<td>2.16</td>
<td>2.33</td>
<td>2.55</td>
</tr>
<tr>
<td>qsort</td>
<td>1.01</td>
<td>1.05</td>
<td>1.32</td>
<td>1.75</td>
</tr>
<tr>
<td>sha</td>
<td>1.01</td>
<td>1.06</td>
<td>1.38</td>
<td>2.22</td>
</tr>
<tr>
<td>shine</td>
<td>1.14</td>
<td>1.97</td>
<td>2.38</td>
<td>2.66</td>
</tr>
<tr>
<td>susan</td>
<td>1.04</td>
<td>1.60</td>
<td>1.59</td>
<td>1.71</td>
</tr>
</tbody>
</table>
Consequently, systems with relatively low cache miss ratios and systems with very large number of COF calculation delays, might benefit from compression only in terms of memory area, while only a limited improvement in performance can be achieved.

Overall, it can be seen that the mean improvement factor is 1.10 for a cache miss penalty of 16 cycles, 1.46 for a penalty of 64 cycles, 1.87 for a penalty of 512 cycles, and 2.14 for a penalty of 2048. Consequently, there will be some improvement even for less complex systems that have a small number of cache refill cycles, especially if the compression parameters are tuned to target performance. For the more complex SoC architectures, in which cache refills may take thousands of cycles, the results show that the performance could be several times better than when using the uncompressed cache. Further improvement in performance can be achieved by (a) the use of larger dictionary, which would ensure that higher compression ratios are achieved and so resulting in larger segments of code being contained within a single cache line, (b) ensuring the dictionary is better configured for a specific target by adopting dynamic trace analysis, or (c) incorporating a larger BTC so that more of the delays that result from COF target calculation can be avoided.

7.4. Summary

This chapter has presented the results of studies investigating the effects of cache compression on both processor and system performance. A study of performance-related parameters, such as the CPI, and instruction count was followed by investigation of the COF target coverage provided by BTC for compensating the target address calculation. An improvement in cache performance was used as an indicative measure of implementing compression, and the simulation outcomes produced a mean reduction of more than 50 per cent in cache misses was achieved. In an attempt to characterise the interrelationship between the improvement in performance for a range of practical processor-based systems, figures were obtained for a range of cache misses and cache types. The results showed that, in the majority of the cases, the compression scheme implemented in this study can improve considerably the systems performance, reaching a peak improvement of 5.6 (see Table 7.5) times faster than the original programme execution. The limitations of the scheme were analysed and cases where compression might lead to deterioration of performance were considered.
Chapter 8

CONCLUSIONS

This chapter reviews the aims of the thesis, summarises the results of each chapter, analyses how well these results meet the thesis objectives and discusses the limitations of the current research in order to identify potential future work.

8.1. Summary of aims

In recent decades the features offered by consumer electronics products have become a major differentiator in the market. This demand for additional functionality often requires commensurate development of evermore complex applications, which in turn requires larger memories and better performing systems on which to run; all having an impact on the cost of the product. The main aim of this study is to address these issues with respect to 32-bit RISC processors. RISC processors are popular choice in embedded design due to their small die size, high performance architecture and low power consumption. A major disadvantage of RISC processors compared to its CISC counterpart is code inefficiency, which, due to additional memory requirements, can significantly increase cost.

In previous work, compression techniques have been applied to improve RISC memory performance, yet there is no holistic solution available. This thesis aimed to provide such a solution, by systematically studying the use of compression in RISC processors in order to improve system performance. Furthermore, inspired by the modular approach taken in the design of the processor used for proof-of-concept, ARCTangent-A4, the solution was
developed as an independent add-in module which can be easily configured and incorporated into the processor pipeline.

8.2. Research overview

At the beginning of this work the research scope and objectives were identified (Chapter 1) and a literature review (Chapter 2) was performed. The literature review focused on the following aspects.

- Identification of techniques for reducing code redundancy and their analysis in terms of efficiency and performance. At this stage, compression was identified as the most suitable approach to provide code reduction.

- An analysis of algorithms used for code compression identified which were the most suitable for implementation in embedded environments.

- For high-performance operation on a target, evidence from the literature was that the decompression should be implemented in hardware rather than in software.

It became apparent from the initial stage that a unified approach to the problem of reducing embedded code redundancy can benefit system performance. Existing work in the field targeted only certain aspects of the problem, for example, some concentrating only on algorithm development, others on only power consumption reduction. Secondly, a large number of solutions found in the literature were unsuitable for embedded systems, such as the use of large look-up tables to map compressed to uncompressed address spaces or the placing of the decompressor after the ICache which seriously impacts the performance of the system.

To substantiate the basis of the current work, the entropy of embedded code for a commercial RISC processor, ARCTangent-A4, was studied in Chapter 3. For this purpose, a number of applications from MiBench were selected in order to give a representative sample from embedded market segments. The programs were compiled for ARCTangent-A4 and their entropies at different levels of resolution (instruction word, byte, etc.) were studied as well as other aspects such as the effect of dictionary size on compression ratio. Advanced data
compression algorithms were then applied in order to verify their suitability for resolving embedded code redundancy problems. The results showed that a number of algorithms such as PPMZ almost fully achieve the code's entropy and so entirely remove the redundancy.

Chapter 4 described a dictionary, class-based compression algorithm, especially developed for this study, to fully comply with the restriction of embedded systems implementation, giving compression ratio results comparable to some of the most advanced text compression algorithms. The compression tool takes the linked executable as input, determines the most suitable number of classes and dictionary size and generates the compressed executable and the other necessary components to achieve correct execution of the compressed code on the target system at run time. To obviate the need for large look-up tables for address mapping, novel approaches were developed to deal with the various classes of COF instructions. Taking into account the dictionary overheads, the achieved mean compression ratio for the test bench programmes was around 0.55.

Once the application code has been compressed off-line, it is loaded in the instruction memory and run on the embedded target. The processor and its memory interface have remained unaltered in the current work and are unaware of the addition of the newly designed dedicated hardware decompressor that is able to carry out decompression in real-time. The functionality, architecture and implementation of the decompressor were presented in Chapter 5. It was developed as a soft IP plug-in module, written in VHDL, which, in a practical solution, can be synthesised together with the ARCTangent-A4 processor. The decompressor is a configurable extension of the original pipeline of the processor, adding two additional stages and residing between the processor and the ICache. The decompressor in conjunction with the cache handles the entire functionality related to compression, including translation between uncompressed and compressed memory address spaces and decoding of dictionary information. The decompressor was synthesised for two different FPGA technologies, namely Xilinx and Altera. As the decompressor is configurable, its area requirement depends to large extent on the nature of the particular application. For Altera's Cyclone device, the clock frequency achieved was 56 MHz, which is similar to that which can be obtained for the processor alone on FPGA devices.

As one of the main objectives of this research is to produce a solution which not only does not degrade the system's performance but generally improves it, the performance
efficiency of the compressed memory system has been one of the main considerations throughout the whole design process. In order to evaluate the achievement of this objective, Chapter 6 described the development of a trace analysis tool and an ICache simulation model, both operating in conjunction with the compressor. Such a tool set was able to simplify the selection of the configurable compression parameters and provided the necessary results for calculating performance improvements, both for the ICache and the full system.

Chapter 7 presented results obtained from the benchmark programs for ICache configurations, of different capacity, type and line length and they demonstrated that the proposed compression scheme can lower the cache miss ratio by between 45 and 65 per cent on average. A decrease in cache miss ratio translates into an improvement in system performance. Programs with shorter refill times, particularly when this is combined with a small number of cache misses, do not result in substantial improvement in performance when the cache is compressed. However, for larger number of cache misses and longer refill times, the performance improvement gains significance, and particularly in large systems with severe penalties, such as SoC architectures with complex bus hierarchies and multiple arbitration units in which cache misses cost thousands of cycles, may execute several times faster than a system without compression.

8.3. Summary of achievements

This research has addressed the problem of code size inefficiency in 32-bit RISC processors. It has focused on the development of a holistic solution, which has considered not only code density, but also performance improvements and power efficiency. The original contributions of the thesis are as follows.

- Development of a dictionary class-based compression technique that is able to provide a substantial saving of relatively expensive cache memory cost savings, while complying with the constraints imposed by embedded environments. The mean compression ratio for the test applications was about 0.51 with 4 Kbytes dictionaries.

- Implementation of a set of simple-to-use software tools, which allow the developer to incorporate memory compression in a fast and efficient manner.
• Development of hardware decompressor IP that extends the processor pipeline and performs all related decompression functions, such as decoding and address translation. The module is configurable and allows tuning of several compression parameters to deliver a suitable balance between higher compression ratio and higher performance for the targeted application.

• A novel approach for resolving COF targets in the compressed address space has been implemented, which providing a tailored solution for each type of COF instruction and eliminating the need for large address look-up tables, as used in earlier work. Furthermore, the COF calculation delays which would normally hinder performance improvement are greatly reduced by operations of the BTC.

• A decrease in the ICache miss ratio of embedded applications of about 55 per cent on average has been produced as a result of the above, which especially for systems with larger cache refill penalties, translates into substantial overall performance improvements.

• A modular approach has been followed to ensure that the compression hardware is a plug-in solution and no changes, either to the memory system, or to the processor architecture and the supplied development tools, are required.

8.4. Future work

The work presented in this thesis can be developed further in several ways.

• The research has focused on a single RISC architecture. Although a certain level of generalisation has been provided for in the current work, other aspects that differentiate RISC architectures could be covered, for example, the opcode instruction field of ARCTangent-A4 is only 5 bits long, while that of MIPS is 6 bits. Having the ability to support different opcode lengths, addressing modes and other ISA specific parameters can make the solution more general and applicable to any RISC machine.
• More sophisticated algorithms for COF management could be analysed in order to further improve performance.

• The transfer of data between different levels of memory hierarchy is one of the most power-hungry operations in embedded systems. As the use of a compressed cache decreases the number of transfers, it is likely that the power dissipation of the system can be reduced. The benefits in terms of reduction in power consumption could be quantified in future studies.

• To further simplify the design flow, the CCS software tool could be extended to automatically configure the VHDL modules that make up the decompressor.

8.5. Summary

This chapter has concluded this thesis by reviewing the research effort and by summarising its objectives and achievements. Finally, a number of potential future developments were outlined.
REFERENCES


References


Appendix A

EXECUTABLE AND LINKING FORMAT (ELF)

The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as a part of the Application Binary Interface (ABI). The Tool Interface Standards committee (TIS) has selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems. The following three main types of object files comply with ELF.

- A relocatable file holds code and data suitable for linking with other object files to create an executable or shared object file.

- An executable object file holds a program suitable for execution; the file specifies how exec (BA_OS) creates a program's process image.

- A shared object file holds code and data suitable for linking in two contexts. First, the link editor may process it with other relocatable and shared object files to create another object file. Second, dynamic linker combines it with an executable file and other shared objects to create a process image.

As the objective of this research was to compress executable code, the structure of an executable file is presented in Figure A.1 and a brief description of its main building blocks follows.

The ELF header resides at the beginning and describes the file's organization. It gives information on type of machine used, the actual sizes of the other tables and their offsets from the beginning of the file, the number of sections and segments, different processor specific flags, and the virtual address to which the system first transfers control, thus starting the
process. A section header table, which is optional for executable files, contains information about file sections. It looks like an array of structures, which hold information for all the sections in the ELF file. The program header table tells the system how to create a process image. Its structure is very similar to the section header table, but it describes file's segments. A segment is a block of information that holds all sections of the same data type.

![Figure A.1. Structure of ELF file](image)

The sections of main interest are those that hold executable instructions, namely the `.text`, `.init` and `.fini`, and those are usually combined into a single segment. The ELF and section header tables have to be changed after compression in order to reflect the new sizes and offsets of the sections and segments. The program header table, as well as the actual relocation information have to be adapted as well.
Appendix B

COMPRESSION RATIO RESULTS

This appendix presents the full set of compression results gathered from the tests described in Section 3.3.

Table B.1 Compression ratio results for ADLC

<table>
<thead>
<tr>
<th>Program</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.476</td>
<td>0.457</td>
<td>0.437</td>
<td>0.437</td>
</tr>
<tr>
<td>bitcnts</td>
<td>0.469</td>
<td>0.448</td>
<td>0.429</td>
<td>0.429</td>
</tr>
<tr>
<td>crc</td>
<td>0.498</td>
<td>0.463</td>
<td>0.442</td>
<td>0.442</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.483</td>
<td>0.457</td>
<td>0.427</td>
<td>0.427</td>
</tr>
<tr>
<td>display</td>
<td>0.442</td>
<td>0.427</td>
<td>0.418</td>
<td>0.418</td>
</tr>
<tr>
<td>fft</td>
<td>0.474</td>
<td>0.455</td>
<td>0.435</td>
<td>0.435</td>
</tr>
<tr>
<td>qsort</td>
<td>0.481</td>
<td>0.452</td>
<td>0.422</td>
<td>0.422</td>
</tr>
<tr>
<td>sha</td>
<td>0.469</td>
<td>0.450</td>
<td>0.429</td>
<td>0.429</td>
</tr>
<tr>
<td>shine</td>
<td>0.498</td>
<td>0.481</td>
<td>0.469</td>
<td>0.469</td>
</tr>
<tr>
<td>susan</td>
<td>0.474</td>
<td>0.450</td>
<td>0.427</td>
<td>0.427</td>
</tr>
<tr>
<td>average</td>
<td>0.476</td>
<td>0.454</td>
<td>0.434</td>
<td>0.428</td>
</tr>
</tbody>
</table>

Table B.2 Compression ratio results for LZS

<table>
<thead>
<tr>
<th>Program</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.952</td>
<td>0.840</td>
<td>0.746</td>
<td>0.654</td>
<td>0.585</td>
<td>0.529</td>
<td>0.495</td>
<td>0.467</td>
</tr>
<tr>
<td>bitcnts</td>
<td>0.943</td>
<td>0.820</td>
<td>0.719</td>
<td>0.637</td>
<td>0.575</td>
<td>0.521</td>
<td>0.481</td>
<td>0.457</td>
</tr>
<tr>
<td>crc</td>
<td>0.943</td>
<td>0.826</td>
<td>0.730</td>
<td>0.649</td>
<td>0.581</td>
<td>0.532</td>
<td>0.490</td>
<td>0.474</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.952</td>
<td>0.833</td>
<td>0.735</td>
<td>0.658</td>
<td>0.592</td>
<td>0.535</td>
<td>0.493</td>
<td>0.467</td>
</tr>
<tr>
<td>display</td>
<td>0.893</td>
<td>0.769</td>
<td>0.676</td>
<td>0.588</td>
<td>0.526</td>
<td>0.488</td>
<td>0.457</td>
<td>0.439</td>
</tr>
<tr>
<td>fft</td>
<td>0.935</td>
<td>0.820</td>
<td>0.725</td>
<td>0.641</td>
<td>0.578</td>
<td>0.529</td>
<td>0.493</td>
<td>0.465</td>
</tr>
<tr>
<td>qsort</td>
<td>0.952</td>
<td>0.840</td>
<td>0.741</td>
<td>0.658</td>
<td>0.592</td>
<td>0.535</td>
<td>0.493</td>
<td>0.461</td>
</tr>
<tr>
<td>sha</td>
<td>0.935</td>
<td>0.820</td>
<td>0.725</td>
<td>0.641</td>
<td>0.578</td>
<td>0.526</td>
<td>0.485</td>
<td>0.461</td>
</tr>
<tr>
<td>shine</td>
<td>0.893</td>
<td>0.787</td>
<td>0.699</td>
<td>0.633</td>
<td>0.578</td>
<td>0.538</td>
<td>0.505</td>
<td>0.488</td>
</tr>
<tr>
<td>susan</td>
<td>0.962</td>
<td>0.847</td>
<td>0.741</td>
<td>0.662</td>
<td>0.588</td>
<td>0.532</td>
<td>0.490</td>
<td>0.465</td>
</tr>
<tr>
<td>average</td>
<td>0.9360</td>
<td>0.8204</td>
<td>0.7237</td>
<td>0.6421</td>
<td>0.5773</td>
<td>0.5264</td>
<td>0.4881</td>
<td>0.4643</td>
</tr>
</tbody>
</table>
### Table B.3 Compression ratio results for X-MatchPro for dictionary size of 128 bytes

<table>
<thead>
<tr>
<th>Program</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.863</td>
<td>0.800</td>
<td>0.728</td>
<td>0.669</td>
<td>0.585</td>
<td>0.542</td>
<td>0.518</td>
<td>0.507</td>
</tr>
<tr>
<td>bitcnts</td>
<td>0.867</td>
<td>0.789</td>
<td>0.712</td>
<td>0.673</td>
<td>0.604</td>
<td>0.555</td>
<td>0.540</td>
<td>0.504</td>
</tr>
<tr>
<td>crc</td>
<td>0.858</td>
<td>0.796</td>
<td>0.723</td>
<td>0.653</td>
<td>0.591</td>
<td>0.548</td>
<td>0.527</td>
<td>0.519</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.870</td>
<td>0.795</td>
<td>0.730</td>
<td>0.665</td>
<td>0.608</td>
<td>0.541</td>
<td>0.516</td>
<td>0.505</td>
</tr>
<tr>
<td>display</td>
<td>0.811</td>
<td>0.743</td>
<td>0.691</td>
<td>0.612</td>
<td>0.553</td>
<td>0.510</td>
<td>0.496</td>
<td>0.484</td>
</tr>
<tr>
<td>fft</td>
<td>0.841</td>
<td>0.789</td>
<td>0.740</td>
<td>0.671</td>
<td>0.602</td>
<td>0.556</td>
<td>0.535</td>
<td>0.527</td>
</tr>
<tr>
<td>qsort</td>
<td>0.863</td>
<td>0.801</td>
<td>0.738</td>
<td>0.659</td>
<td>0.589</td>
<td>0.538</td>
<td>0.512</td>
<td>0.500</td>
</tr>
<tr>
<td>sha</td>
<td>0.852</td>
<td>0.781</td>
<td>0.715</td>
<td>0.649</td>
<td>0.597</td>
<td>0.534</td>
<td>0.509</td>
<td>0.498</td>
</tr>
<tr>
<td>shine</td>
<td>0.821</td>
<td>0.766</td>
<td>0.715</td>
<td>0.659</td>
<td>0.602</td>
<td>0.567</td>
<td>0.550</td>
<td>0.536</td>
</tr>
<tr>
<td>susan</td>
<td>0.878</td>
<td>0.809</td>
<td>0.725</td>
<td>0.668</td>
<td>0.589</td>
<td>0.541</td>
<td>0.516</td>
<td>0.503</td>
</tr>
<tr>
<td>average</td>
<td>0.852</td>
<td>0.787</td>
<td>0.722</td>
<td>0.658</td>
<td>0.592</td>
<td>0.543</td>
<td>0.522</td>
<td>0.508</td>
</tr>
</tbody>
</table>

### Table B.4 Compression ratio results for X-MatchPro for dictionary size of 256 bytes

<table>
<thead>
<tr>
<th>Program</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.863</td>
<td>0.800</td>
<td>0.728</td>
<td>0.669</td>
<td>0.585</td>
<td>0.541</td>
<td>0.505</td>
<td>0.487</td>
</tr>
<tr>
<td>bitcnts</td>
<td>0.867</td>
<td>0.789</td>
<td>0.712</td>
<td>0.673</td>
<td>0.604</td>
<td>0.551</td>
<td>0.523</td>
<td>0.484</td>
</tr>
<tr>
<td>crc</td>
<td>0.858</td>
<td>0.796</td>
<td>0.723</td>
<td>0.653</td>
<td>0.591</td>
<td>0.547</td>
<td>0.511</td>
<td>0.500</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.870</td>
<td>0.795</td>
<td>0.730</td>
<td>0.665</td>
<td>0.608</td>
<td>0.539</td>
<td>0.502</td>
<td>0.484</td>
</tr>
<tr>
<td>display</td>
<td>0.811</td>
<td>0.743</td>
<td>0.691</td>
<td>0.612</td>
<td>0.553</td>
<td>0.509</td>
<td>0.484</td>
<td>0.471</td>
</tr>
<tr>
<td>fft</td>
<td>0.841</td>
<td>0.789</td>
<td>0.740</td>
<td>0.671</td>
<td>0.602</td>
<td>0.554</td>
<td>0.522</td>
<td>0.506</td>
</tr>
<tr>
<td>qsort</td>
<td>0.863</td>
<td>0.801</td>
<td>0.738</td>
<td>0.659</td>
<td>0.589</td>
<td>0.536</td>
<td>0.501</td>
<td>0.478</td>
</tr>
<tr>
<td>sha</td>
<td>0.852</td>
<td>0.781</td>
<td>0.715</td>
<td>0.649</td>
<td>0.597</td>
<td>0.532</td>
<td>0.497</td>
<td>0.480</td>
</tr>
<tr>
<td>shine</td>
<td>0.821</td>
<td>0.766</td>
<td>0.715</td>
<td>0.659</td>
<td>0.602</td>
<td>0.564</td>
<td>0.538</td>
<td>0.522</td>
</tr>
<tr>
<td>susan</td>
<td>0.878</td>
<td>0.809</td>
<td>0.725</td>
<td>0.668</td>
<td>0.589</td>
<td>0.540</td>
<td>0.503</td>
<td>0.484</td>
</tr>
<tr>
<td>average</td>
<td>0.852</td>
<td>0.787</td>
<td>0.722</td>
<td>0.658</td>
<td>0.592</td>
<td>0.541</td>
<td>0.509</td>
<td>0.490</td>
</tr>
</tbody>
</table>

### Table B.5 Compression ratio results for X-MatchPro for dictionary size of 512 bytes

<table>
<thead>
<tr>
<th>Program</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.863</td>
<td>0.800</td>
<td>0.728</td>
<td>0.669</td>
<td>0.585</td>
<td>0.541</td>
<td>0.504</td>
<td>0.481</td>
</tr>
<tr>
<td>bitcnts</td>
<td>0.867</td>
<td>0.789</td>
<td>0.712</td>
<td>0.673</td>
<td>0.604</td>
<td>0.551</td>
<td>0.521</td>
<td>0.478</td>
</tr>
<tr>
<td>crc</td>
<td>0.858</td>
<td>0.796</td>
<td>0.723</td>
<td>0.653</td>
<td>0.591</td>
<td>0.547</td>
<td>0.510</td>
<td>0.495</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.870</td>
<td>0.795</td>
<td>0.730</td>
<td>0.665</td>
<td>0.608</td>
<td>0.539</td>
<td>0.502</td>
<td>0.478</td>
</tr>
<tr>
<td>display</td>
<td>0.811</td>
<td>0.743</td>
<td>0.691</td>
<td>0.612</td>
<td>0.553</td>
<td>0.509</td>
<td>0.483</td>
<td>0.469</td>
</tr>
<tr>
<td>fft</td>
<td>0.841</td>
<td>0.789</td>
<td>0.740</td>
<td>0.671</td>
<td>0.602</td>
<td>0.554</td>
<td>0.522</td>
<td>0.501</td>
</tr>
<tr>
<td>qsort</td>
<td>0.863</td>
<td>0.801</td>
<td>0.738</td>
<td>0.659</td>
<td>0.589</td>
<td>0.536</td>
<td>0.500</td>
<td>0.473</td>
</tr>
</tbody>
</table>
### Table B.6 Compression ratio results for X-MatchPro for dictionary size of 1024 bytes

<table>
<thead>
<tr>
<th>Program</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.863</td>
<td>0.800</td>
<td>0.728</td>
<td>0.669</td>
<td>0.585</td>
<td>0.541</td>
<td>0.504</td>
<td>0.481</td>
</tr>
<tr>
<td>bitcents</td>
<td>0.867</td>
<td>0.789</td>
<td>0.712</td>
<td>0.673</td>
<td>0.604</td>
<td>0.551</td>
<td>0.521</td>
<td>0.478</td>
</tr>
<tr>
<td>crc</td>
<td>0.858</td>
<td>0.796</td>
<td>0.723</td>
<td>0.653</td>
<td>0.591</td>
<td>0.547</td>
<td>0.510</td>
<td>0.495</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.870</td>
<td>0.795</td>
<td>0.730</td>
<td>0.665</td>
<td>0.608</td>
<td>0.539</td>
<td>0.502</td>
<td>0.478</td>
</tr>
<tr>
<td>display</td>
<td>0.811</td>
<td>0.743</td>
<td>0.691</td>
<td>0.612</td>
<td>0.553</td>
<td>0.509</td>
<td>0.483</td>
<td>0.469</td>
</tr>
<tr>
<td>fft</td>
<td>0.841</td>
<td>0.789</td>
<td>0.740</td>
<td>0.671</td>
<td>0.602</td>
<td>0.554</td>
<td>0.522</td>
<td>0.500</td>
</tr>
<tr>
<td>qsort</td>
<td>0.863</td>
<td>0.801</td>
<td>0.738</td>
<td>0.659</td>
<td>0.589</td>
<td>0.536</td>
<td>0.500</td>
<td>0.472</td>
</tr>
<tr>
<td>sha</td>
<td>0.852</td>
<td>0.781</td>
<td>0.715</td>
<td>0.649</td>
<td>0.597</td>
<td>0.532</td>
<td>0.496</td>
<td>0.475</td>
</tr>
<tr>
<td>shine</td>
<td>0.821</td>
<td>0.766</td>
<td>0.715</td>
<td>0.659</td>
<td>0.602</td>
<td>0.564</td>
<td>0.537</td>
<td>0.516</td>
</tr>
<tr>
<td>susan</td>
<td>0.878</td>
<td>0.809</td>
<td>0.725</td>
<td>0.668</td>
<td>0.589</td>
<td>0.540</td>
<td>0.503</td>
<td>0.479</td>
</tr>
<tr>
<td>average</td>
<td>0.852</td>
<td>0.787</td>
<td>0.722</td>
<td>0.658</td>
<td>0.592</td>
<td>0.541</td>
<td>0.508</td>
<td>0.485</td>
</tr>
</tbody>
</table>

### Table B.7 Compression ratio results for X-MatchPro for dictionary sizes of 2048 and 4096 bytes

<table>
<thead>
<tr>
<th>Program</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.863</td>
<td>0.800</td>
<td>0.728</td>
<td>0.669</td>
<td>0.585</td>
<td>0.541</td>
<td>0.504</td>
<td>0.481</td>
</tr>
<tr>
<td>bitcents</td>
<td>0.867</td>
<td>0.789</td>
<td>0.712</td>
<td>0.673</td>
<td>0.604</td>
<td>0.551</td>
<td>0.521</td>
<td>0.478</td>
</tr>
<tr>
<td>crc</td>
<td>0.858</td>
<td>0.796</td>
<td>0.723</td>
<td>0.653</td>
<td>0.591</td>
<td>0.547</td>
<td>0.510</td>
<td>0.495</td>
</tr>
<tr>
<td>djikstra</td>
<td>0.870</td>
<td>0.795</td>
<td>0.730</td>
<td>0.665</td>
<td>0.608</td>
<td>0.539</td>
<td>0.502</td>
<td>0.478</td>
</tr>
<tr>
<td>display</td>
<td>0.811</td>
<td>0.743</td>
<td>0.691</td>
<td>0.612</td>
<td>0.553</td>
<td>0.509</td>
<td>0.483</td>
<td>0.469</td>
</tr>
<tr>
<td>fft</td>
<td>0.841</td>
<td>0.789</td>
<td>0.740</td>
<td>0.671</td>
<td>0.602</td>
<td>0.554</td>
<td>0.522</td>
<td>0.500</td>
</tr>
<tr>
<td>qsort</td>
<td>0.863</td>
<td>0.801</td>
<td>0.738</td>
<td>0.659</td>
<td>0.589</td>
<td>0.536</td>
<td>0.500</td>
<td>0.472</td>
</tr>
<tr>
<td>sha</td>
<td>0.852</td>
<td>0.781</td>
<td>0.715</td>
<td>0.649</td>
<td>0.597</td>
<td>0.532</td>
<td>0.496</td>
<td>0.475</td>
</tr>
<tr>
<td>shine</td>
<td>0.821</td>
<td>0.766</td>
<td>0.715</td>
<td>0.659</td>
<td>0.602</td>
<td>0.564</td>
<td>0.537</td>
<td>0.515</td>
</tr>
<tr>
<td>susan</td>
<td>0.878</td>
<td>0.809</td>
<td>0.725</td>
<td>0.668</td>
<td>0.589</td>
<td>0.540</td>
<td>0.503</td>
<td>0.479</td>
</tr>
<tr>
<td>average</td>
<td>0.852</td>
<td>0.787</td>
<td>0.722</td>
<td>0.658</td>
<td>0.592</td>
<td>0.541</td>
<td>0.508</td>
<td>0.484</td>
</tr>
</tbody>
</table>

### Table B.8 Compression ratio results for PPMZ

<table>
<thead>
<tr>
<th>basicmath</th>
<th>bitcents</th>
<th>crc</th>
<th>djikstra</th>
<th>display</th>
<th>fft</th>
<th>qsort</th>
<th>sha</th>
<th>shine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.345</td>
<td>0.336</td>
<td>0.353</td>
<td>0.335</td>
<td>0.360</td>
<td>0.363</td>
<td>0.332</td>
<td>0.369</td>
<td>0.315</td>
</tr>
</tbody>
</table>
Table B.9 Compression ratio results for DCLZ

<table>
<thead>
<tr>
<th>Program</th>
<th>History buffer size (bytes)</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td></td>
<td>0.565</td>
<td>0.569</td>
<td>0.558</td>
<td>0.552</td>
</tr>
<tr>
<td>bitcnts</td>
<td></td>
<td>0.554</td>
<td>0.552</td>
<td>0.553</td>
<td>0.539</td>
</tr>
<tr>
<td>crc</td>
<td></td>
<td>0.565</td>
<td>0.564</td>
<td>0.555</td>
<td>0.560</td>
</tr>
<tr>
<td>djkstra</td>
<td></td>
<td>0.567</td>
<td>0.564</td>
<td>0.559</td>
<td>0.549</td>
</tr>
<tr>
<td>display</td>
<td></td>
<td>0.543</td>
<td>0.538</td>
<td>0.519</td>
<td>0.532</td>
</tr>
<tr>
<td>fft</td>
<td></td>
<td>0.561</td>
<td>0.558</td>
<td>0.556</td>
<td>0.572</td>
</tr>
<tr>
<td>qsort</td>
<td></td>
<td>0.566</td>
<td>0.562</td>
<td>0.557</td>
<td>0.548</td>
</tr>
<tr>
<td>sha</td>
<td></td>
<td>0.559</td>
<td>0.557</td>
<td>0.553</td>
<td>0.550</td>
</tr>
<tr>
<td>shine</td>
<td></td>
<td>0.613</td>
<td>0.609</td>
<td>0.609</td>
<td>0.605</td>
</tr>
<tr>
<td>susan</td>
<td></td>
<td>0.562</td>
<td>0.563</td>
<td>0.559</td>
<td>0.552</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>0.565</td>
<td>0.564</td>
<td>0.558</td>
<td>0.556</td>
</tr>
</tbody>
</table>

Table B.10 Compression ratio results for CLB

<table>
<thead>
<tr>
<th>Program</th>
<th>Dictionary size (bytes)</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8191</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td></td>
<td>0.730</td>
<td>0.663</td>
<td>0.621</td>
<td>0.572</td>
<td>0.529</td>
<td>0.386</td>
</tr>
<tr>
<td>bitcnts</td>
<td></td>
<td>0.729</td>
<td>0.664</td>
<td>0.624</td>
<td>0.574</td>
<td>0.524</td>
<td>0.378</td>
</tr>
<tr>
<td>crc</td>
<td></td>
<td>0.729</td>
<td>0.664</td>
<td>0.623</td>
<td>0.572</td>
<td>0.522</td>
<td>0.384</td>
</tr>
<tr>
<td>djkstra</td>
<td></td>
<td>0.744</td>
<td>0.679</td>
<td>0.634</td>
<td>0.582</td>
<td>0.534</td>
<td>0.390</td>
</tr>
<tr>
<td>display</td>
<td></td>
<td>0.667</td>
<td>0.598</td>
<td>0.552</td>
<td>0.488</td>
<td>0.402</td>
<td>0.325</td>
</tr>
<tr>
<td>fft</td>
<td></td>
<td>0.742</td>
<td>0.677</td>
<td>0.634</td>
<td>0.583</td>
<td>0.533</td>
<td>0.389</td>
</tr>
<tr>
<td>qsort</td>
<td></td>
<td>0.742</td>
<td>0.677</td>
<td>0.634</td>
<td>0.583</td>
<td>0.533</td>
<td>0.389</td>
</tr>
<tr>
<td>sha</td>
<td></td>
<td>0.735</td>
<td>0.671</td>
<td>0.630</td>
<td>0.577</td>
<td>0.526</td>
<td>0.369</td>
</tr>
<tr>
<td>shine</td>
<td></td>
<td>0.777</td>
<td>0.721</td>
<td>0.684</td>
<td>0.639</td>
<td>0.598</td>
<td>0.447</td>
</tr>
<tr>
<td>susan</td>
<td></td>
<td>0.758</td>
<td>0.699</td>
<td>0.654</td>
<td>0.603</td>
<td>0.557</td>
<td>0.434</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>0.735</td>
<td>0.671</td>
<td>0.629</td>
<td>0.577</td>
<td>0.526</td>
<td>0.389</td>
</tr>
</tbody>
</table>

