Huffman-based Code Compression Techniques for Embedded Systems

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Talal Bonny

aus Aleppo

Tag der mündlichen Prüfung: 18.06.2009

Erster Gutachter: Prof. Dr.-Ing. Jörg Henkel

Zweiter Gutachter: Prof. Dr.-Ing. Wael Adi

Talal Bonny
August-Bebel-Str.68
76187 Karlsruhe

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____

Talal Bonny

Acknowledgements

First and foremost, I would like gratefully and sincerely to thank my advisor Professor Jörg Henkel for enabling and supporting the research presented in this thesis and for offering an enjoyable work environment for his PhD students. This dissertation could not have been written without his support and guidance.

I am also very grateful to Professor Wael Adi for accepting to be my co-examiner and providing valuable feedback.

I thank members of our chair "Chair for Embedded Systems" at University of Karlsruhe for their feedback and interesting discussions, especially (in alphabetical order) Mohammad Al Faruque, Hussam Amrouch, Lars Bauer, Thomas Ebi, Fridtjof Feldbusch, Fazal Hameed, Nabeel Iqbal, Semeen Rehman, Muhammad Shafique and Sammer Srouji. And all others that influenced the research of this PhD thesis and were not explicitly mentioned here.

I thank the students I supervised during their Master/Diploma thesis for their contributions to the implementation and evaluation of the code compression techniques.

I am also deeply indebted to my family for believing in me and for giving me the inspiration to apply and complete a doctoral degree. Their love, guidance, and encouragement have been a constant source of strength for me.

And last but not least, my heartfelt thanks go out to my beloved better-half, Amani, who has been so patient and supportive since I started the master program. She has been my inspiration and provided encouragement when research progress was slow, and when research felt like spiraling out of control, she brought serenity. It should be no surprise that this dissertation would be impossible without her. I dedicate this thesis to her.

List of Publications

Conferences

[**C.1**] Talal Bonny and Jörg Henkel:
LICT: Left-uncompressed Instructions Compression Technique to Improve the Decoding Performance of VLIW Processors.
In 46th ACM/EDA/IEEE Design Automation Conference (DAC09), ,
Pages: 903-906, San Fransisco CA, USA, July, 2009 (accepted).

[**C.2**] Talal Bonny and Jörg Henkel:
FBT: Filed Buffer Technique to Reduce Code Size for VLIW Processors.
In 26th IEEE/ACM International Conference on Computer-Aided Design (ICCAD'08),
Pages: 549-554, San Jose CA, USA, November, 2008.

[**C.3**] Talal Bonny and Jörg Henkel:
Instruction Re-encoding Facilitating Dense Embedded Code.
In IEEE/ACM Proceedings of Design Automation and Test in Europe Conference (DATE'08),
Pages: 770-775, Munich, Germany, 2008.

[**C.4**] Talal Bonny and Jörg Henkel:
Instruction Splitting for Efficient Code Compression.
In ACM/EDA/IEEE Design Automation Conference (DAC07),
Pages: 646-651, San Diego CA, USA, June 2007.

[**C.5**] Talal Bonny and Jörg Henkel:
Efficient Code Density Through Look-up Table Compression.
In IEEE/ACM Proceedings of Design Automation and Test in Europe Conference (DATE'07),
Pages: 809-814, Nice, France, April 2007.

[**C.6**] Talal Bonny and Jörg Henkel:
Using Lin-Kernighan Algorithm for Look-up Table Compression to Improve Code Density
In ACM/IEEE 16th, Great Lakes Symposium on VLSI (GLSVLSI'06),
Pages: 259-265, Philadelphia, USA, April 2006.

Journals

[**J.1**] Talal Bonny and Jörg Henkel:
Efficient Code Compression for Embedded processors.
In IEEE Transactions on Very Large Scale Integration Systems (TVLSI),
Volume 16, Issue 12, Pages: 1696-1707, December 2008.

Book Chapter

[**B.1**] S. Parameswaran, J. Henkel, A. Janapsatya, T. Bonny and A. Ignjatovic:
Design and Run Time Code Compression for Embedded Systems.
In **Designing Embedded Processors**.
pp. 97-128, Springer 2007.
ISBN978-1-4020-5868-4 (HB).
ISBN 978-1-4020-5869-1 (e-book).

Abstract

Increasing embedded systems functionality causes a steep increase in code size. For instance, more than 60MB of software is installed in current state-of-the-art cars [9].
It is often challenging and cumbersome to host vast amount of software in an efficient way within a given hardware resource budget of an embedded system. This may be done by using code compression techniques, which compress the program code off-line (i.e. at design time) and decompress it on-line (i.e. at run time).
Among all statistical compression algorithms, Huffman Coding is one of the best compression techniques since it provably provides the shortest average codeword length [36]. When Huffman Coding is used as a compression technique, Look-up Table(s) are generated to store the original instructions. As the size of the tables becomes large, it may negatively affect the final compression ratio (defined as the ratio of compressed code to uncompressed code). Thus, the Look-up Tables diminish the advantages that could be obtained by compressing the code.

This thesis presents different Huffman-based hardware supported code compression techniques, which can efficiently solve this problem and improve the compression ratio. In addition to that, this the presents a new technique to improve the performance of the hardware decoder. The code compression techniques are targeted two processor architectures, namely RISC and VLIW.

In this thesis, the Look-up Table size is reduced by using the "Look-up Table Compression Technique" for RISC processors. This is done by sorting the entries of the table to decrease the number of bit toggles between each two successive entries.
To show the efficiency of the "Look-up Table Compression Technique", we apply it to two compression schemes, i.e. Dictionary-based Compression Scheme and Statistical Compression Scheme.
Using the "Look-up Table Compression Technique" reduces the table size by up to 45% and improves the compression ratio on average by more than 25%.
The evaluations are conducted using a representative set of applications from MiBench [20] and are built for three major embedded processor architectures, namely ARM, MIPS

and PowerPC.

The Look-up Table size may further be reduced if its instructions are encoded efficiently before the "Look-up Table Compression Technique" is applied to it. For that, the second compression technique for RISC processors is introduced, which is called "Instruction Splitting Technique". This technique reduces not only the Look-up Table size but also the size of the encoded instructions generated by using Huffman Coding. It splits the instructions into patterns of varying size before Huffman Coding is applied. Using this technique improves the final compression ratio (including all overhead) by more than 20% compared to known schemes based on *Huffman Coding*. Average compression ratios of 47% and 49% are achieved for ARM and MIPS processors, respectively.

Both our compression techniques "Look-up Table Compression Technique" and "Instruction Splitting Technique" are ISA (Instruction Set Architecture)-Independent, i.e. they can be applied to any processor architecture.
When the ISA is specified, the code compression technique utilizes the information in the opcodes and the instruction format to build the hardware decoder. In this case, the compression ratio will be further improved. For that, ISA-Dependent code compression technique is introduced for RISC processors, which is called "Instruction re-encoding Technique". In this technique, the benefits of re-encoding the unused bits in the instruction format for a specific application are investigated to improve the compression ratio. Re-encoding those bits may reduce the size of decoding table by more than 37%. Compression ratio as low as 44% is achieved (including all overhead that incurs), targeting MIPS and ARM processors.

VLIW processors provide higher performance and better efficiency than RISC processors in specific domains like multimedia applications. The drawback of the VLIW processors is the bloating code size of their compiled applications in comparison to the size of the same applications compiled for RISC processors. Therefore, reducing the size of embedded applications is a key issue for VLIW processors.
For that, the last code compression technique (Deflate Algorithm [107]) in this thesis is targeting the VLIW processors. It significantly reduces the code size compared to state-of-the-art approaches for VLIW processors.
The Deflate Algorithm is enhanced by using a new technique called "Filled Buffer Technique", which can be applied to any Lempel-Ziv family algorithms to improve the compression ratio. This compression technique is independent of the Instruction Set Architecture and can be used by previous compression techniques [82] to further improve the obtained compression ratio. Using the "Filled Buffer Technique" in conjunction with the previous work "V2F" [77] improves the compression ratio by 10%. The evaluations are

conducted using a representative set of benchmarks (from MediaBench and MiBench) and targeting two VLIW processors, namely TMS320C62x and TMS320C64x. Average compression ratios of 61% and 56% are achieved for TMS320C62x and TMS320C64x VLIW processors, respectively.

The main disadvantage of any code compression technique is the system performance penalty because of the extra time required to decode the compressed instructions during run time.
For that, we improve the performance of decoding compressed instructions by using our novel compression technique (*LICT: Left-uncompressed Instruction Technique*) which can be used in conjunction with any compression algorithm. Using our *LICT* in conjunction with the Burrows-Wheeler (*BW*) algorithm [107] improves the performance explicitly (2.5x) with a little impact on the compression ratio (only 3% compression ratio loss).

Zusammenfassung

Erweiterungen der Funktionalität eingebetteter Systeme führen zu deutlich größeren Programmgrößen. Beispielsweise sind mehr als 60MB Software in einem aktuellen Auto installiert [9]. Es ist oft herausfordernd und mühsam in einem eingebetteten System mit beschränkten Hardwareressourcen eine große Menge an Software effizient bereitzustellen. Dies kann jedoch durch Codekomprimierungstechniken erreicht werden, die das Programm off-line (d.h. zur Entwurfszeit) komprimieren und es on-line (d.h. zur Laufzeit) dekomprimieren.
Unter allen statistischen Komprimierungsalgorithmen ist die Huffmankodierung eine der besten Komprimierungstechniken, da sie wohl die kürzeste durchschnittliche Codewortlänge liefert [36]. Zur Verwendung der Huffmankodierung als Komprimierungstechnik werden Nachschlagetabellen erzeugt, um die originalen Befehle zu speichern. Da diese Nachschlagetabellen groß werden, kann die endgültige Komprimierungsrate, die durch das Größenverhältnis von komprimiertem Code zu unkomprimiertem Code definiert ist, negativ beeinflusst werden. Daher verringern die Nachschlagetabellen die Vorteile, die durch Codekomprimierung erreicht werden könnten.

Diese Arbeite präsentiert Huffman-basierte hardwareunterstützte Codekomprimierungstechniken, die dieses Problem effizient lösen können und die Komprimierungsrate verbessern. Die Codekomprimierungstechniken sind auf zwei Prozessorarchitekturen ausgerichtet, nämlich RISC und VLIW.

Wir reduzieren die Größe der Nachschlagetabellen durch die “Look-up Table Compression Technique” für RISC Prozessoren. Dies wird durch eine Sortierung der Tabelleneinträge erreicht, mit der die Anzahl der Bitunterschiede zwischen zwei aufeinander folgenden Einträgen verringert werden.
Um die Effizienz der “Look-up Table Compression Technique” zu zeigen, wenden wir sie auf zwei Komprimierungsschemata an: Wörterbuch-basierte und statische Komprimierungsschemata.
Die Verwendung unserer “Look-up Table Compression Technique” reduziert die Tabellengröße um bis zu 45% und verbessert die Kompressionsrate im Durchschnitt um mehr

als 25%. Die Evaluierungen wurden unter Verwendung einer repräsentativen Menge an Applikationen aus MiBench [20] für drei bedeutende eingebettete Prozessorarchitekturen (ARM, MIPS und PowerPC) durchgeführt.

Die Größe der Nachschlagetabellen kann weiter reduziert werden, wenn die Instruktionen effizient enkodiert werden, bevor die "Look-up Table Compression Technique" angewendet wird. Dafür präsentieren wir unsere zweite Codekomprimierungstechnik für RISC Prozessoren, die "Instruction Splitting Technique" heißt. Diese Technik reduziert nicht nur die Größe der Nachschlagetabelle, sondern außerdem auch die Größe der enkodierten Instruktionen, die durch die Huffmankodierung erzeugt werden. Sie unterteilt die Instruktionen in Muster unterschiedlicher Größe, bevor die Huffmankodierung angewendet wird. Durch die Verwendung dieser Technik wird die endgültige Komprimierungsrate (inklusive aller Extraaufwendungen) im Vergleich zu bekannten, auf Huffmankodierung bestehenden Verfahren, um mehr als 20% verbessert. Wir erreichen eine durchschnittliche Komprimierungsrate von 47% und 49% für ARM, beziehungsweise MIPS Prozessoren.

Beide Komprimierungstechniken ("Look-up Table Compression Technique" und "Instruction Splitting Technique") sind ISA unabhängig, d.h. sie können für jede Prozessorarchitektur verwendet werden.
Wenn die ISA bekannt ist, verwendet die Codekomprimierungstechnik die Information in den Opcodes oder dem Instruktionsformat, um den Hardwaredekodierer zu erstellen. In diesem Fall wird die Komprimierungsrate weiter verbessert. Dafür präsentieren wir unsere ISA-abhängige Kodekomprimierungstechnik für RISC Prozessoren, die "Instruction re-encoding Technique" heißt. In dieser Technik untersuchen wir die Vorteile, die unbenutzten Bits im Instruktionsformat für eine bestimmte Applikation neu zu kodieren, um so die Komprimierungsrate zu verbessern. Diese Bits neu zu kodieren kann die Größe der Kodierungstabelle um mehr als 37% verringern. Wir erreichen Komprimierungsraten (inklusive aller Extraaufwendungen) bis runter zu 44% für MIPS und ARM Prozessoren.

VLIW Prozessoren erbringen in bestimmten Domänen wie Multimediaapplikationen höhere Performanz und bessere Effizienz als RISC Prozessoren. Der Nachteil der VLIW Prozessoren ist die deutlich vergrößerte Codemenge der kompilierten Applikationen im Vergleich zu der Größe derselben Applikation die für RISC Prozessoren übersetzt wurde. Daher ist die Verringerung der Codegröße ein Hauptanliegen für VLIW Prozessoren.
Dafür führen wir unsere letzte Codekomprimierungstechnik (Deflate Algorithmus [107]) in dieser Arbeit speziell für VLIW Prozessoren ein. Sie verringert die Codegrößen im Vergleich zu aktuellen Ansätzen für VLIW Prozessoren deutlich.
Wir erweitern den Deflate Algorithmus durch die neue "Filled Buffer Technique", die auf

jeden Algorithmus der Lempel-Ziv Familie angewandt werden kann, um die Komprimierungsrate im Durchschnitt um mehr als 13% - im Vergleich zur ausschließlichen Anwendung des Deflate Algorithmus - verbessern. Diese Komprimierungstechnik ist unabhängig von dem Befehlssatz (ISA) und kann von früheren Komprimierungstechniken [82] benutzt werden, um die erreichte Komprimierungsrate weiter zu verbessern. Die Verwendung unserer "Filled Buffer Technique" zusammen mit "V2F" [77] verbessert die Kompressionsrate um 10%. Wir haben Evaluierungen anhand einer repräsentativen Menge an Benchmarks (aus MediaBench und MiBench) durchgeführt und unser Schema auf zwei VLIW Prozessoren angewandt, nämlich TMS320C62x und TMS320C64x. Wir erreichten bei Verwendung des Deflate Algorithmus eine gesamtKompressionsrate bis runter zu 44% (im Durchschnitt 61% bzw. 56% für den TMS320C62x bzw. TMS320C64x).

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The size of embedded software is marching at a rapid pace. It is often challenging and cumbersome to fit an amount of required software functionality within a given hardware resource budget. Code compression is a means to alleviate the problem by providing substantial savings in terms of size. In this thesis, efficient code compression techniques are presented for different processor architectures.

1.1 Motivation

The aim of this thesis is to explore different Huffman-based hardware supported code compression techniques in order to reduce the size of compiled code and consequently the memory size.
By compressing the program, and including only a small decompression unit to decompress the instructions on-the-fly, the processor can continue to operate without any change in its architecture. Researches have shown [19, 65, 72] that this sort of technique applied to a single issued processors has the potential not only for program size reduction, but also power savings and performance enhancement.

For each of our code compression technique, the reduction in the code size (compression ratio) and the performance overhead (performance ratio) are investigated. Furthermore, the Instruction Set Architecture (ISA) dependability of the code compression techniques is analyzed, i.e. the impact of using the ISA-Dependent and ISA-Independent code compression techniques on the code size.
Our ISA-Independent code compression techniques are orthogonal to any ISA and can be applied to any processor architecture. In order to demonstrate the orthogonality, we have conducted evaluations using single-issue (usually RISC) processor architectures

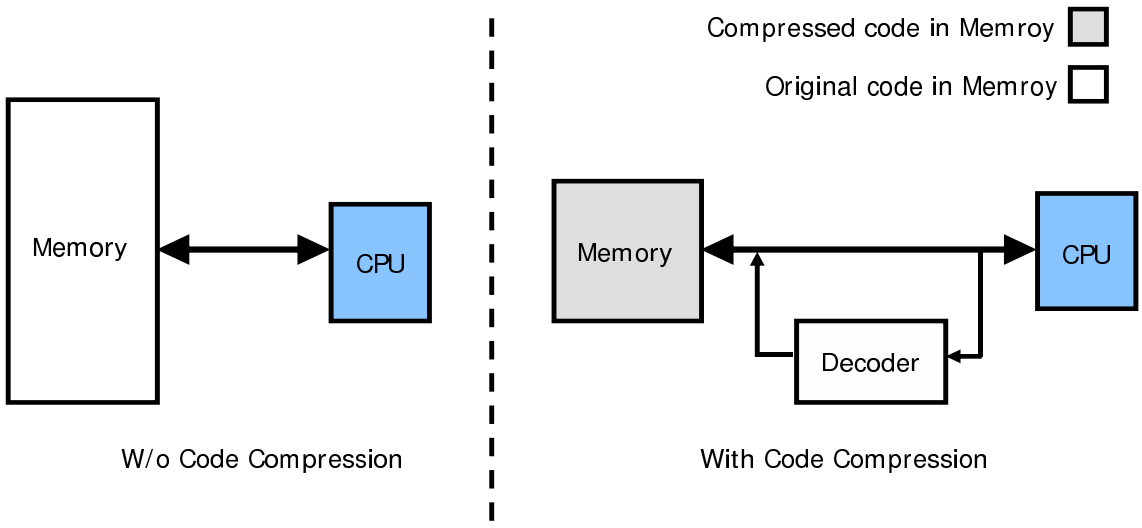


FIGURE 1.1: Overview of an embedded system

like MIPS, ARM and PowerPC and also multi-issue (VLIW) processor architectures like TMS320C62x and TMS320C64x.

1.2 Introduction and Background

Nowadays more than 98% of all programmable processors run in embedded mode [9]. According to the World Semiconductor Trade Statistics Blue Book, there are an estimated 5 billion embedded microprocessors in use today [10]. The reason for the growing popularity of embedded system-driven devices, such as PDAs (Personal Digital Assistants) and web-enabled cell phones, is the sustainable growth in demands of their applications. For instance, the world market for embedded software will grow from about $1.6 billion in 2004 to $3.5 billion by 2009, at an Average Annual Growth Rate (AAGR) of 16% [11] and nowadays, we can find more than 60MB of software installed in current state-of-the-art cars [9].
Since the memory size of the embedded system must be small according to the demands of the embedded market, a key challenge in designing high volume and cost effective embedded systems is to host this vast amount of software in an efficient way. This can be accomplished by deploying code compression which beside memory size reduction, may also reduce the power consumption [12, 13, 14] since memory consumes a significant amount of an embedded system's power [17, 18, 113]. The beginning of this trend had already been recognized in the early 1990s, when the first approaches for code compression for embedded applications arose [19].

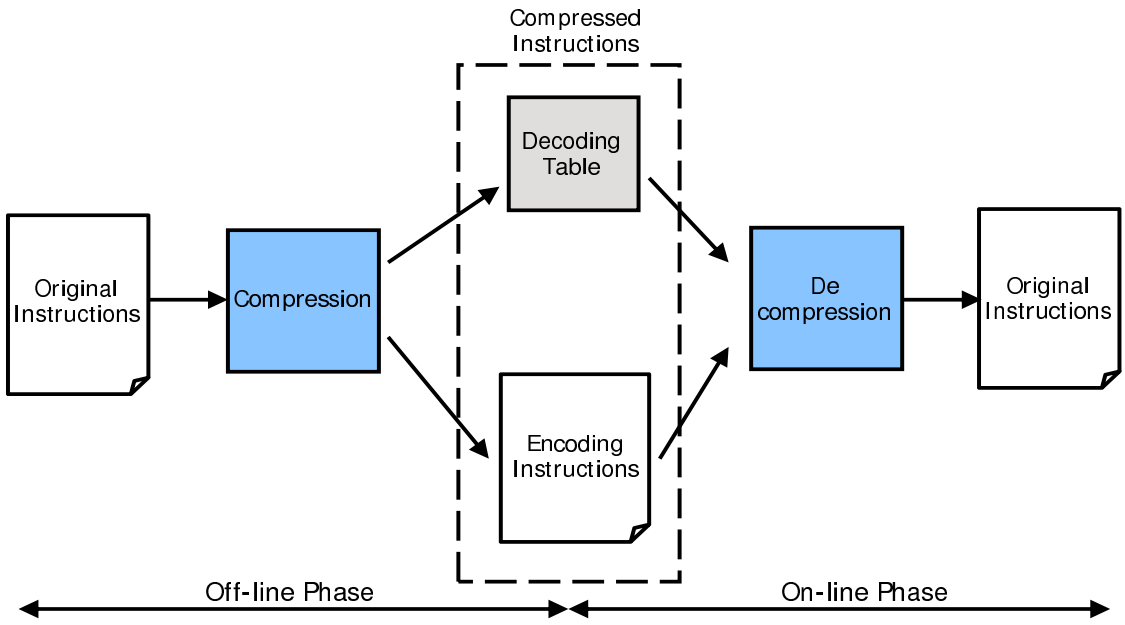


FIGURE 1.2: Compression (off-line) and decompression (on-line) phases

Fig. 1.1 shows an overview of an embedded system without using code compression (left side): Large memory is required as the application is stored in the memory without compression. The same embedded system is presented in Fig. 1.1 (right side) but by using code compression technique; The size of required memory is reduced as the application is compressed and stored in it. Hardware decoder between the memory and the CPU is required to decode the compressed instructions and retrieve the original ones.

The process of compressing and decompressing the program code is carried out through off-line (i.e. design time) and on-line (i.e. run-time) phases (Fig. 1.2). In the off-line phase, the original instructions are compressed (typically after compilation) and the encoded instructions are stored along with the decoding table(s). During the on-line phase, the original instructions are retrieved from the compressed ones by using the decoding table(s) along with some decoding hardware or software.

1.2.1 Measurements and Terms

To measure the efficiency (in terms of code size reduction) of any compression technique, the compression ratio CR is used, which is the size required to store the *compressed instructions* in memory divided by the size required to store the *original instructions* [1]:

$$CR = \frac{size(compressed\ instructions)}{size(original\ code)} \text{ x } 100 \quad (\%) \tag{1.1}$$

[1]Note: a smaller compression ratio means a better compression as it denotes the percentage of the code size after compression.

The term *compressed instructions* includes the instructions after they have been encoded "*encoded instructions*" plus all types of tables "*decoding table*" which are required to decompress the whole program to its original format (see Fig. 1.2). For that, the size of the *compressed instructions* can be formulated like:

$$size(compressed\ instructions) = size(encoded\ instructions) + size(decoding\ table) \quad (1.2)$$

It is unclear in many of the previous work surveyed in this thesis, whether or not compression ratios reported include the decompression tables or the dictionary size. As these tables may occupy a large space in memory and impact on the final compression ratio, their size should be considered when the compression ratio is computed.

In this thesis, the term *decoding table* in Eq. 1.2 is assumed to include all kind of tables required to decompress the *compressed instructions* such as the dictionary, Look-up Tables (LUT), Line Address Table (LAT) or Address Table Translation (ATT).

From Equations 1.1 and 1.2, the total compression ratio CR can be formulated like:

$$CR = \frac{size(encoded\ instructions) + size(decoding\ table)}{size(original\ code)} \ \texttt{X}\ 100 \quad (\%) \quad (1.3)$$

In all our code compression techniques, the hardware decoder is located between the processor and the memory (e.g. Fig 1.1). Therefore, the hardware decoder will slow down the execution of the program and impact on the performance. To measure the impact of the hardware decoder on the performance, the Performance Ratio (PR) is used which is the time required to execute the *compressed instructions* divided by the execution time of the *original code*:

$$PR = \frac{execution\_time(compressed\ instructions)}{execution\_time(original\ code)} \quad (1.4)$$

As the hardware decoder degrades the performance, the performance ratio is always more than 1. Better performance ratio is obtained, when performance ratio of relatively close to 1 is achieved.

The performance index of processor is the average CPI or *Cycles Per Instruction* for a given program:

$$average\_CPI = \frac{\#cycles}{\#instructions} \quad (1.5)$$

By using pipelining, most machines are able to achieve a CPI of relatively close to 1. In the case of multi-issue (VLIW) processor architecture, the average CPI is improved to be less than 1.

1.2.2 Types of Compression

To reduce the costs associated with the large memory requirements, three commonly used methods of compression are well known and reported in literature:

- Compiler-based compression: in this method, a reduced code size is achieved at compile time through a set of operations that leave the behavior of the application unaltered, but reduce the code size [102, 103, 119, 120].
- Instruction set compaction: here, the existing instruction set is (partially) re-written in a more compact form (e.g. ARM Thumb and MIPS16 [55, 56]). The new compact form of instruction format requires less number of bits than the original one and consequently, it reduces the code size.
- Data compression techniques: in this method, a lossless data compression technique (e.g. Huffman Coding [26], or Lempel-Ziv [44], etc.) is applied to the program code at compile time, resulting in a smaller compressed program that is decompressed at run-time by additional decompression hardware.

This thesis focuses on the last two methods.

The code compression techniques can be used either when the ISA (Instruction Set Architecture) is specified (i.e. ISA-Dependent) or not (i.e. ISA-Independent). In the ISA-Dependent compression technique, the technique is specified just for that processor architecture and can not be applied to other one. This is because information about the opcodes or instruction format are utilized to build the hardware decoder. This kind of techniques achieves high compression ratio, since the number and the type of operands in the instruction format can be reduced according to the operation defined by the opcode. In the ISA-Independent compression technique, the compression technique is orthogonal to ISA and can be applied to any processor architecture. This is because it follows the traditional data compression techniques, which depend only on the statistics of instructions or part of them. The decoder in this case is simpler than the former one because it does not take the instruction format into account. The only disadvantage is that the compressed code is not so efficient in size as compared to the compression technique that specifies the ISA.
In this thesis both types of ISA dependability are investigated.

1.3 Scope of Our Research Work in this Thesis

The scope of the research conducted in this thesis (Fig. 1.3) falls into two main areas, code compression for single-issue (usually RISC) processors and code compression for multi-issue (VLIW) processors.

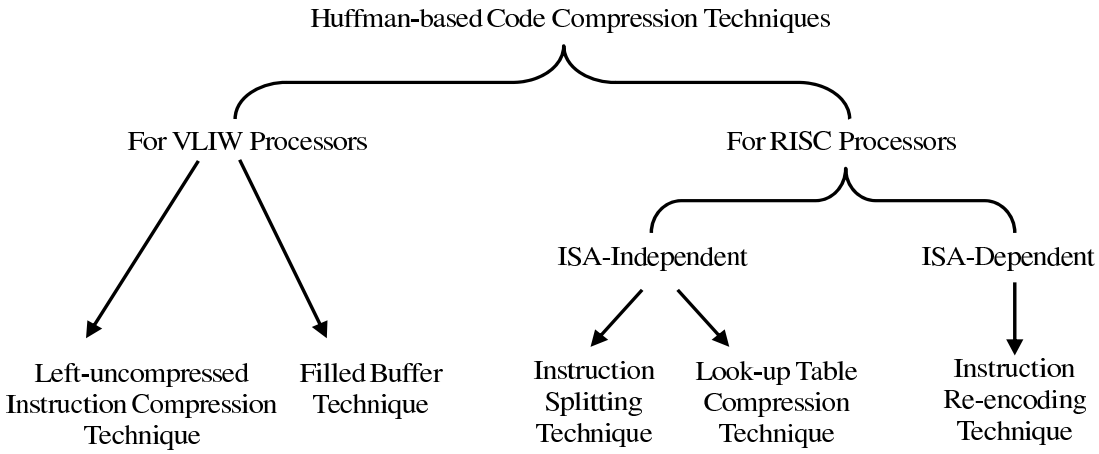


FIGURE 1.3: Scope of our research work in this thesis

In the case of single-issue processors, three Huffman-based code compression techniques are investigated, two of them are ISA-Independent (Look-up Table Compression Technique and Instruction Splitting Technique) and one is ISA-Dependent (Instruction Re-encoding Technique).
In the case of multi-issue processors, Huffman-based code compression technique is investigated in the Filled Buffer Technique) and the Left-uncompressed Instructions Compression Technique.

All five code compression techniques are described briefly by the points below:

1. Look-up Table Compression Technique [1, 2, 7, 8]

In code compression, the Look-up Tables are deployed for decoding. These tables may take a space in the memory and significantly impact the total compression ratio. The main contribution is to reduce the size of the Look-up Table using a novel compression technique that is ISA-Independent. Our compression technique is used along with two compression schemes *Dictionary-based* and *Statistical* compression schemes (detail will be shown in Section 2.3). The evaluations are conducted using a representative set of applications and

applied to three major embedded RISC processors, namely ARM, MIPS and PowerPC.

2. Instruction Splitting Technique [3]

The contribution within this technique is to enhance the compression ratio results obtained by using Huffman Coding. This is done by splitting the instructions into varying size of patterns before Huffman Coding compression is applied. This technique is also orthogonal to ISA-Independent techniques. The evaluations are conducted using a representative set of applications and applied to ARM and MIPS processors.

3. Instruction Re-encoding Technique [4]

When the ISA is specified, the code compression technique utilizes the information in opcodes or instruction format to build the hardware decoder. In this case, the compression ratio will be improved, since the number and the type of operands in the instruction format can be reduced according to the operation defined by the opcode. The contribution within this technique is to find out the *re-encodable bits* in instruction format which are suitable for re-encoding and to investigate the benefits of re-encoding these bits for a specific processor to improve the compression ratio. The evaluations are conducted using a representative set of applications and applied to ARM and MIPS processors.

4. Filled Buffer Technique [5]

VLIW processors provide higher performance and better efficiency than RISC processors in specific domains like multimedia applications etc. A disadvantage is the bloated code size of the compiled application code. The contribution within this technique is to improve the results of reducing the code size that is obtained using the Deflate Algorithm (which has been used before for data compression) by using a new technique called Filled Buffer Technique. This technique can be applied to any Lempel-Ziv family algorithm to improve the compression ratio. The evaluations are conducted using a representative set of benchmarks (from MediaBench and MiBench) and applied to two VLIW processors, namely TMS320C62x and TMS320C64x.

5. Left-uncompressed Instructions Compression Technique [6]

The main disadvantage of any code compression technique is the system performance penalty because of the extra time required to decode the compressed

instructions during run time. In this paper we improve the performance of decoding compressed instructions by using our novel compression technique (*LICT: Left-uncompressed Instruction Technique*) which can be used in conjunction with any compression algorithm. Applying LICT on the Burrows-Wheeler (*BW*) [107] code compression algorithm improves the performance explicitly (2.5x) with a little impact on the compression ratio (only 3% compression ratio loss). The evaluations are conducted using a representative set of benchmarks (from Mediabench and Mibench) and applied to the TMS320C62x VLIW processor.

1.4 Outline of Thesis

The remainder of this thesis is organized as follows:

Chapter two:
Background Theory

Chapter Two presents comprehensive overview of the theory of code and data compression (Section 2.1). The compression algorithms used in this thesis (such that Huffman Coding, Lempel-Ziv etc.) are described in Section 2.3. The different types of the hardware decoder are described in Section 2.2. The end of this chapter (Section 2.4) presents an overview of the embedded RISC processor (including the MIPS, ARM, and PowerPC processors) and the embedded VLIW Processor (including the TMS320C62x and TMS320C64x processors).

Chapter Three:
Related Work

Chapter Three presents the commercial compression techniques which have been implemented in ARM, MIPS, and PowerPC. It presents also a survey of the related literature in the field of code compression. The survey is classified on the basis of compression technique whether it is *Statistical* or *Dictionary-based* compression techniques (Section 3.2.1), and on the basis of Instruction Set Architecture (ISA)-Dependability whether it is ISA-Dependent or ISA-Independent compression techniques (Section 3.2.2).

Chapter Four:
Code Compression for RISC Processors

Chapter Four investigates the ISA-Dependability of a code compression schemes for the RISC processors. Section 4.1 describes the ISA-Independent compression techniques. Two different Huffman-based compression techniques are presented (Look-up Table Compression and Instruction Splitting techniques). Section 4.2 describes the ISA-Dependent compression techniques. Huffman-based compression technique (Instruction Re-encoding) is presented. In each of these techniques, the main compression algorithm, the hardware decoder implementation and the experimental results (compression ratio and performance) are explained in detail.
The benchmarks for the experiments are selected from MiBench [20] benchmark suite and are built for three RISC Processors, namely ARM, MIPS and PowerPC.
Section 4.3 includes a discussion about the ISA-Dependability of the code compression schemes. At the end of this chapter (Section 4.4), a comparison of our work to the previous work targeting RISC processors is presented.

Chapter Five:
Code Compression for VLIW Processors

Chapter Five presents the Filled Buffer code compression technique for the VLIW processor which is used to improve the compression efficiency for any Lempel-Ziv family algorithm. Section 5.1.2 describes the Filled Buffer Technique (FBT) and how it is applied to the Deflate Algorithm. To show the orthogonality of this technique, we applied it to another algorithm from Lempel-Ziv family which is called LZMA Algorithm.
Section 5.2 describes the implementation of the hardware decoder and Section 5.3 presents the experimental results for both Deflate and LZMA algorithms. The compression results are compared with the results of state-of-the-art previous work like"V2F" [77] (as this technique achieves very high decoding throughput).
The benchmarks for the experiments are selected form MiBench [20] and MediaBench [21] benchmark suites and are built for two VLIW Processors, namely TMS320C62x and TMS320C64x.
At the end of this chapter (Section 5.5), a comparison of our work to the previous work targeting VLIW processors is presented.

Chapter Six:
Code Compression to Improve the Performance

Chapter Six presents the left-uncompressed instruction code compression technique for the VLIW processor although it may be used for any processor architecture. It is used to improve the performance of the hardware decoder independent from the compression technique. Section 6.1 presents the basics of compressing and decompressing the code and the data. The compression technique *LICT* and the compression algorithm *BW* are

presented in Section 6.2 and Section 6.3, respectively. In Section 6.4, we use *LICT* in conjunction with the previous work [5] whereas experimental results are presented in Section 6.5.

Chapter Seven:
Conclusion

Chapter Seven presents summary for the thesis and possible further extension.

Chapter 2

Background Theory

This chapter presents background theory of data compression and code compression, and introduces the main differences between them. The compression algorithms that are used in this thesis are explained in this chapter in details. At the end of this chapter, an overview of the RISC and VLIW processors are presented.

2.1 Code Compression and Data Compression

Code compression differs from data compression in many points

First, the size of the data/code that is required to be compressed:
In data compression, it is assumed that the compression must be done in a single sequential pass over the data (i.e. compressing the whole data as one block). This is because typical data may be too large to be stored in the memory (or disk) at one time, for instance video stream. For that and during compression, history information is used by the compressor to utilize repetition in the data and to improve the results of compression.
In code compression, the compression is not applied to the whole program, because it will not be decompressed completely and executed as a burst. Instead, small segments or blocks of code (called compression blocks or codewords) are compressed individually. This is to ensure random access to important points in the code such as branch targets and function entry points. For example, since code execution can be redirected at branch instructions it would be ideal for the decompression to begin at any branch target. This effectively splits the program into blocks of instructions defined by branch targets.

Second, the way of decoding the compressed data/code:
In data compression, as the data was compressed (as whole in one block) depending upon the history information of the data stream, the decoder can only start decoding at the beginning of the data stream and continue till the end. It cannot begin decompressing

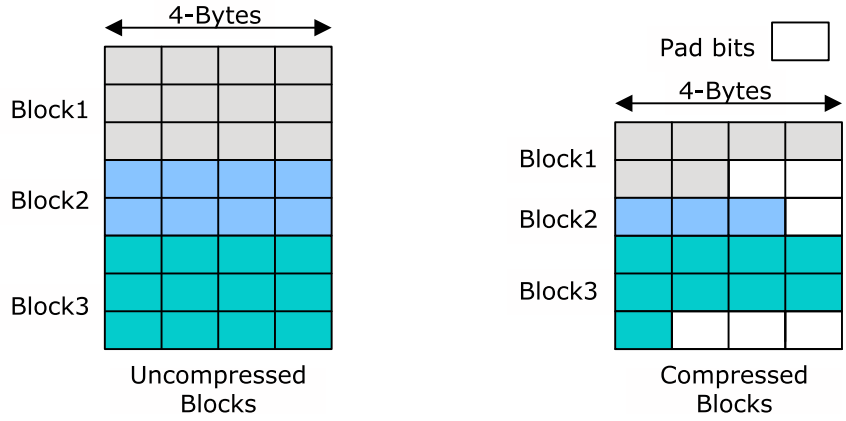


FIGURE 2.1: Alignment of compressed blocks in memory boundary

at an arbitrary point in the data stream because it will not have the history information that the decompression algorithm depends upon.
In code compression, as the program is split into different compression blocks and each block is compressed individually, the decoder will decode each compressed block completely and execute it separately.

Third, the alignment of the compressed data/code in the memory:
In data compression, most compression techniques use bit-aligned output in the memory to obtain the smallest possible representations.
In code compression, there are alignment restrictions which impose a minimum size on instructions. For example, the compression algorithm may restrict the compressed blocks to begin on byte boundaries of the memory so that the decoder can quickly access the compressed blocks. This would require the use of pad bits to lengthen the minimum size of compressed blocks as illustrated in Fig. 2.1.

Fourth, the compression results:
Data compression typically results in higher compression ratio than code compression. This is because the size of the block in data compression which is required to be compressed is larger than the size of the code compression block and usually higher redundant information is included which is used by the compression algorithm.

In code compression, when the code is compressed, the addresses of compressed instructions where they are stored in the memory will differ from the addresses of original instructions. This issue becomes important by the control transfer instructions (i.e. branch or jump instructions) because the target of the branch instructions in uncompressed code is not the same as in compressed code. To handle this issue, two ways are used in this thesis depending on the compression algorithm used.

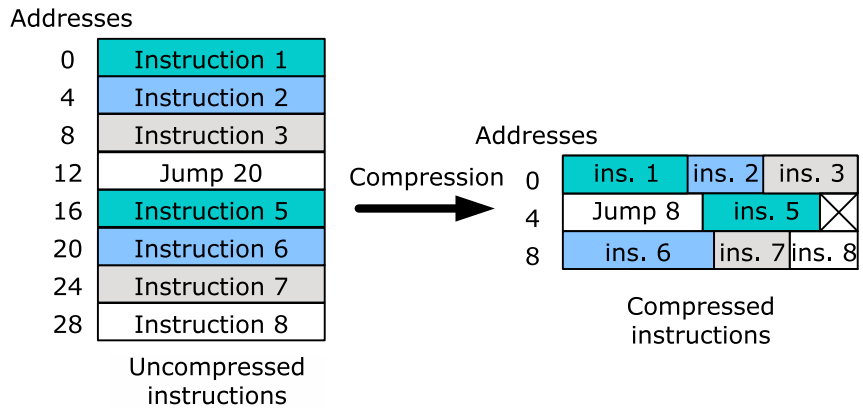


FIGURE 2.2: Solving the branch target problem for variable encoded instructions

First, when the compression algorithm generates compressed code with variable length encoded instructions:
The branch target instruction is aligned at an addressable boundary in the memory and the addresses of the branch target in uncompressed code are patched with the compressed ones as adopted from [28, 66]. The succeeding instructions are stored consecutively in memory. Fig. 2.2 shows an example for solving the branch target problem. In the uncompressed block (on the left side), the "Instruction 6" which is at address 20 is the branch target of the "Jump 20" which is at address 12. In the compressed block (on the right side), the "Instruction 6" is aligned at the boundary of address 8 and the target address of the "Jump 20" is patched with the new address 8, i.e. "Jump 8". In this case, some bits may be left unused after "Instruction 5".

Second, when the compression algorithm generates compressed code with fixed length encoded instructions:
The addresses of branch target instruction in uncompressed code are patched with the compressed ones. As the encoded instructions have a fixed length, branch target addresses are not required to be aligned at an addressable boundary in the memory. This is because the hardware decoder which is designed in this thesis can compute the address of encoded instruction and access it even if it is not aligned to memory border, because the instructions have fixed length.

Instead of batching the branch target addresses of uncompressed code with the compressed ones, Address Translation Table (ATT) may be used that contains both kind of addresses to map the addresses of compressed code to the original ones (as adopted in [19, 59]).

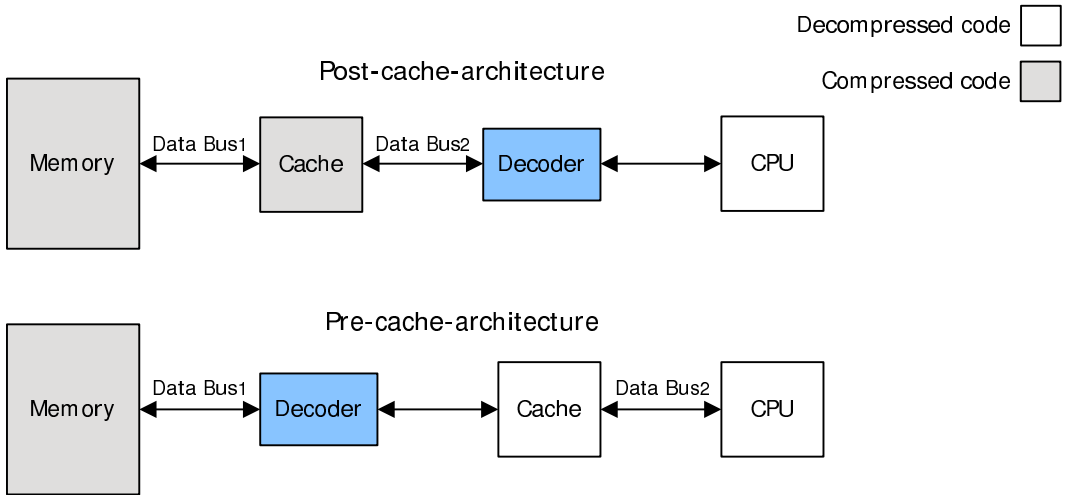


FIGURE 2.3: Placement possibilities of the hardware decoder

2.2 Placement of Hardware Decoder

The most challenging task while designing a code compression scheme is designing the decompression hardware. The decompression technique must be cost effective and efficient. Two possibilities are exist for the placement of the hardware decoder (See Fig. 2.3). Either between the instruction cache and the CPU (Post-cache-architecture) or between the main memory and the instruction cache (Pre-cache-architecture) [22, 25].

In the case of Post-cache-architecture, the code is compressed in the main memory and in the I-cache. Therefore, more area saving is achieved. Furthermore, the data bus between the main memory and the I-cache (Data Bus1) and between the I-cache and the CPU (Data Bus2) profit from the compressed code since the instructions are only decompressed before they are fed into the CPU. Therefore, less number of bit toggles is required and consequently more energy reduction can be achieved. [17] is an example of using this type of architecture.

In the case of Pre-cache-architecture, the code is compressed in the main memory but it is not compressed in the I-cache. Therefore, less area saving is achieved than the Post-cache-architecture. Furthermore, only the data bus between the I-cache and the CPU (Data Bus2) profits from the compressed code and consequently less energy reduction can be achieved than the Post-cache-architecture. On the other hand, the timing overhead for decompression could be hidden behind cache miss penalty. Therefore, less performance loss is occurred in this architecture than in the Post-cache-architecture. [19] is an example of using this type of architecture.

2.3 Basis Classification of Code Compression Techniques

Proposed compression techniques may be classified into two general groups: *Statistical compression technique* and *Dictionary-based* compression technique [26]. The *Dictionary-based* compression technique [15, 27, 28, 61, 84, 85, 96, 109] selects the frequently occurring pieces of information and places them in a Look-up Table (or dictionary). These pieces of information may be instructions, sequences of instructions, trees, tree-patterns, operand sequences, etc. In the original code, the frequently occurring pieces of information are replaced by a single new codeword which is shorter in size than the original ones. The codeword is then used as an index to the dictionary that contains the original sequence of instructions. All the codewords have fixed length.
In the *Statistical* compression techniques [19, 59, 75, 77, 80, 83, 92, 95], the frequency of instructions (or sequences of instructions) is used to choose the size of the codewords that replace the original ones. Thereby, shorter codewords are used for the most frequent instructions, whereas longer codewords are replaced by less frequent ones.
As the codewords in the *Dictionary-based* techniques have fixed lengths, its decompression time is faster than it in the *Statistical* techniques which their codewords have variable lengths (as explained in Section 4).
In the next section, details of the compression algorithms that are used in this thesis and their classifications are presented.

2.3.1 Dictionary-based Compression Techniques

2.3.1.1 Instructions Sequence Dictionary-based Technique

In fact, many sequences of instructions are repeated in a single program. For example, FOR loop will always have the same structure and many such short series of instructions will appear in the code repeatedly. The *Dictionary-based* techniques capture this information and use it to reduce the compression ratio further. Fig. 2.4 shows an example of the *Dictionary-based* technique. The compression technique finds sequences of instruction that are frequently repeated throughout a single program and then it replaces the entire sequence with a single codeword.
In Fig. 2.4, the sequence “Instruction A, Instruction B, Instruction C, and Instruction D” is replaced with “CODEWORD #1”. And the sequence “Instruction H, and Instruction I” is replaced with “CODEWORD #2”. The remaining instructions are left without compression.
All rewritten (or encoded) sequences of instructions are kept in a dictionary which is used at program execution time to expand the singleton codewords in the instruction stream back into the original sequence of instructions. All codewords assigned by the compression algorithm are merely indices into the instruction in the dictionary. The index has a

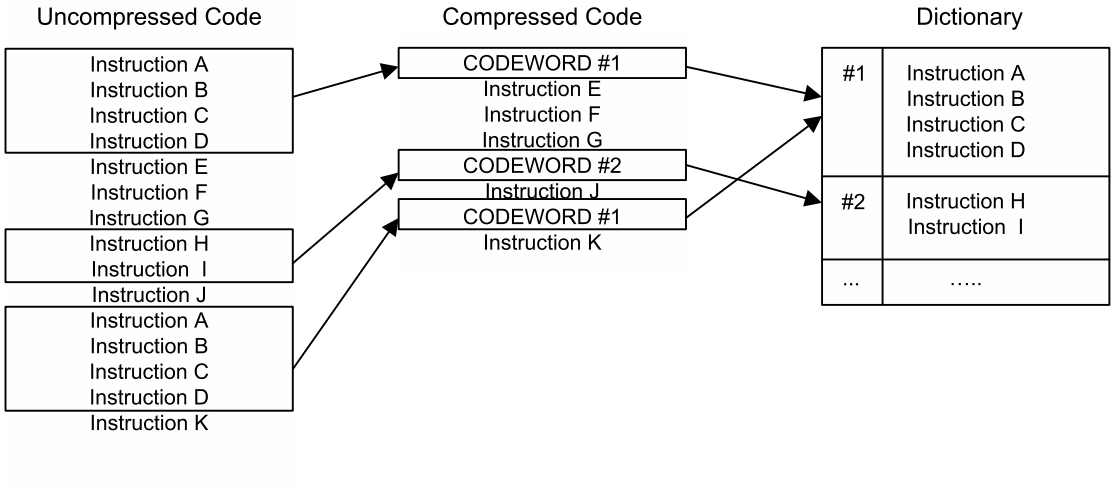


FIGURE 2.4: Dictionary-based compression technique

fixed length equal to log2 number of sequences. The final compressed program consists of codewords interspersed with uncompressed instructions and dictionary.

The compression ratio is computed as follows:

$$CR = \frac{size(Compressed\ instructions)}{size(Original\ Code)}$$

The term *Compressed instructions* includes the *Compressed Code* and the *Dictionary*. The size of each term is computed as follows:

$size(Original\_Code) = W \times N$
$size(Compressed\_Code) = W \times U + n \times \log_2(n)$
$size(Dictionary) = W \times \sum_{i=1}^{n} C_i$

By substituting these terms in compression ratio equation, we get the final compression ratio for the Instructions Sequence Dictionary-based Technique.

$$CR = \frac{W \times U + n \times \log_2(n) + W \sum_{i=1}^{n} C_i}{W \times N}$$

W: Instruction word length
N: Number of original instructions
n: Number of sequences
U: Number of remaining uncompressed instructions
C_i: Number of instructions in sequence i

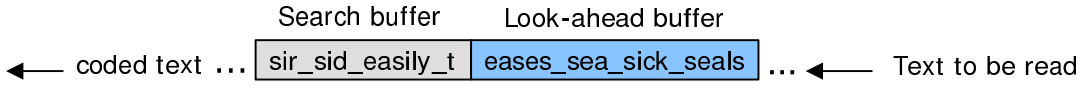


FIGURE 2.5: LZ77 sliding window

2.3.1.2 Lempel and Ziv

The Lempel Ziv (LZ) family of compression algorithms started with LZ77 Algorithm by Abraham Lempel and Jacob Ziv [44]. This algorithm has been modified later by Terry Welch [45] and called LZW algorithm. LZW is a general compression algorithm capable of working on almost any type of data. It is generally fast in both compressing and decompressing data and does not require the use of floating-point operations.

LZW is referred to as a substitutional or dictionary-based encoding algorithm. The algorithm builds a data dictionary (also called a translation table or string table) of data occurring in an uncompressed data stream. Patterns of data (substrings) are identified in the data stream and are matched to entries in the dictionary. If the substring is not present in the dictionary, a code phrase is created based on the data content of the substring, and it is stored in the dictionary. The phrase is then written to the compressed output stream. LZW generates the translation table for decompressing on the fly. There is no need to store the table in an extra file/block. This feature enables high compression ratios due to the unnecessary transmission and saving of the translation table.

The LZ77 Algorithm is based on sliding window. This window is divided into two parts (see Fig. 2.5). The part on the left is the *search buffer*. This is the current dictionary, and it includes symbols that have recently been input and encoded. The part on the right is the *look-ahead buffer*, containing text yet to be encoded.
In the Fig. 2.5, we assume that the text "sir_sid_easily_t" has already been compressed, while the text "eases_sea_sick_seals" still needs to be compressed.

The encoder scans the *search buffer* from right to left looking for a match for the first symbol "e" in the *look-ahead buffer*. It finds one at the "e" of the word "easily". This "e" is at offset (distance) of 8 from the end of the *search buffer*. The encoder then matches as many symbols following the "e" as possible. Three symbols "eas" match in this case, so the length of the match is 3. The symbols "eas" are encoded using the token (16,3,e). In general, an LZ77 token has three parts: offset, length, and next symbol in the *look-ahead buffer*. This token is written on the output stream and the window is shifted to the right four positions: three positions for the matched string and one position for the next symbol. The encoder continues encoding the symbol until all the symbols are shifted bit-wise (or byte-wise) from the *look-ahead buffer* to the *search buffer*.

Algorithm 1 LZ77 Encoding Algorithm

```
1: while look-ahead buffer is not empty do
2:    go backwards in search buffer to find longest match of the look-ahead buffer
3:    if match found then
4:       print: (offset from window boundary;)
5:       print: (length of match;)
6:       print: (next symbol in lookahead buffer;)
7:       shift window by length + 1;
8:    else
9:       print: (0, 0, first symbol in look-ahead buffer);
10:      shift window by 1;
11:   end if
12: end while
```

The pseudo code of LZ77 encoding and decoding algorithms are shown in Algorithm 1 and Algorithm 2, respectively.

Algorithm 2 LZ77 Decoding Algorithm

```
1: for all token (offset, length, symbol) do
2:    if offset = 0 then
3:       print: symbol;
4:    else
5:       go reverse in previous output by offset characters);
6:       copy character wise for length symbols;
7:       print: symbol;
8:    end if
9: end for
```

2.3.1.3 LZSS Algorithm

LZSS Algorithm is an efficient variant of LZ77 Algorithm developed by Storer and Szymanski [46]. It improves LZ77 Algorithm in three directions:
(1) It holds the *look-ahead buffer* in a circular queue. The circular queue is a bounded queue which can effectively utilize the memory space by inserting or deleting elements from any side of the queue (the front or rear side) [47].
(2) It holds the *search buffer* in a binary search tree. The binary search tree is a binary tree where the left subtree of every node A contains nodes smaller than A, and the right subtree contains nodes greater than A. The maximum number of steps needed to locate a node in a tree equals the height of the tree which is equal to $\lceil log_2 n \rceil$. Such that n is the number of nodes in the tree. Using the binary search tree, will speed up the search in the dictionary of the LZSS Algorithm in comparison to the LZ77 Algorithm.
(3) It creates tokens with two fields instead of three. An LZSS token contains just an offset and length. If no match is found, the encoder emits the uncompressed code of the

Algorithm 3 LZSS Encoding Algorithm

{P := pointer to this match}
{L := length of the match}

```
1: Place the coding position to the beginning of the input stream;
2: find the longest match in the window for the lookahead buffer
3: if L >= MIN_LENGTH then
4:    output P;
5:    move the coding position L characters forward;
6: else
7:    output the first character of the lookahead buffer;
8:    move the coding position one character forward;
9: end if
10: if there are more characters in the input stream then
11:    go back to 2;
12: end if
```

next symbol instead of the wasteful three-filed token. To distinguish between tokens and uncompressed codes, each of them is preceded by a single bit (a flag). Algorithm 3 shows the pseudo code of the LZSS encoding algorithm.

For the decoding, The window is slid over the output stream in the same manner the encoding algorithm slides over the input stream. Explicit characters are output directly, and when a pointer is encountered, the string in the window points to the output.
This algorithm generally yields a better compression ratio than LZ77 with practically the same processor and memory requirements. The decoding is still extremely simple and quick. That is why it has become the basis for practically all the later algorithms of this type.

LZSS can also be combined with the entropy coding methods: for example, Huffman Coding. The new resulted compression algorithm is called "Deflate".

Deflate Algorithm is a popular compression technique that was originally used in the well-known Zip and Gzip software to compress text files (data). It is based on the LZSS Algorithm combined with Huffman Coding (Huffman Coding is explained in Section 2.3.2.1). The LZSS Algorithm writes a pair (offset, length) on the compressed stream. When no match is found, the unmatched character is written on the compressed stream instead of the token. Thus, the compressed stream consists of three types of entities: literals (unmatched characters), offsets (distances) and lengths. Deflate Algorithm actually writes Huffman codes on the compressed stream for these entities. For that, it uses two previously prepared code tables, one for literals and lengths and the other for distances.
When a pair (offset, length) is determined, the encoder searches the table of literal/length codes to find the code for the length. This code is then replaced by a Huffman code that's

written on the compressed stream. The encoder then searches the table of offset codes for the code of the offset and writes that code on the compressed stream. The decoder knows when to expect a length code, because it always follows an offset code.

2.3.2 Statistical Compression Techniques

2.3.2.1 Huffman Coding

Huffman Coding is an entropy encoding algorithm for compression found by David A. Huffman in 1952. It is a statistical technique which reduces the amount of bits required to represent a string of symbols. In order to reduce the amount of bits required to represent a string of symbols, symbols are allowed to be of varying lengths. Shorter codes are assigned to the most frequently used symbols, and longer codes to the symbols which appear less frequently in the string (that's where the statistical part comes in).
Huffman Coding produces prefix free codes, which means that no code symbol is a prefix of a different code symbol, thus enabling decoding without prior knowledge of the code symbol's length and preventing ambiguities.
Huffman Coding algorithm basically builds a binary tree, the leaves of which represent the symbols of the source alphabet. The code length for these symbols equals their depth in the tree (that is, their distance to the root node).

The Huffman tree is constructed as follows:

- Create a forest of unconnected nodes, each node represents a symbol of the source alphabet. These nodes can be seen as trees consisting solely of the root.
- Find the two trees in the forest with the lowest probability of occurrence and combine them into a new tree by adding a new root node. The probability for this new tree is the sum of probabilities of the two old trees.
- Repeat until the forest consists of only one tree.

The actual encoding can then be done by labeling the edges that go out from each node with 0 and 1. The code for a symbol from the source alphabet is found by concatenating the labels of all edges on the path between the source node and the leaf node which is representing that symbol.

On each loop in the process of creating Huffman Tree it is possible to decide what child will be the left and what will be right. In advance if some symbols or sum of symbols have equal weights, it is possible to select each of them when minimal-weight node is searched. Therefore it is possible to create dozen different Huffman Trees. Each tree will be valid Huffman Tree and therefore can be used for compression.

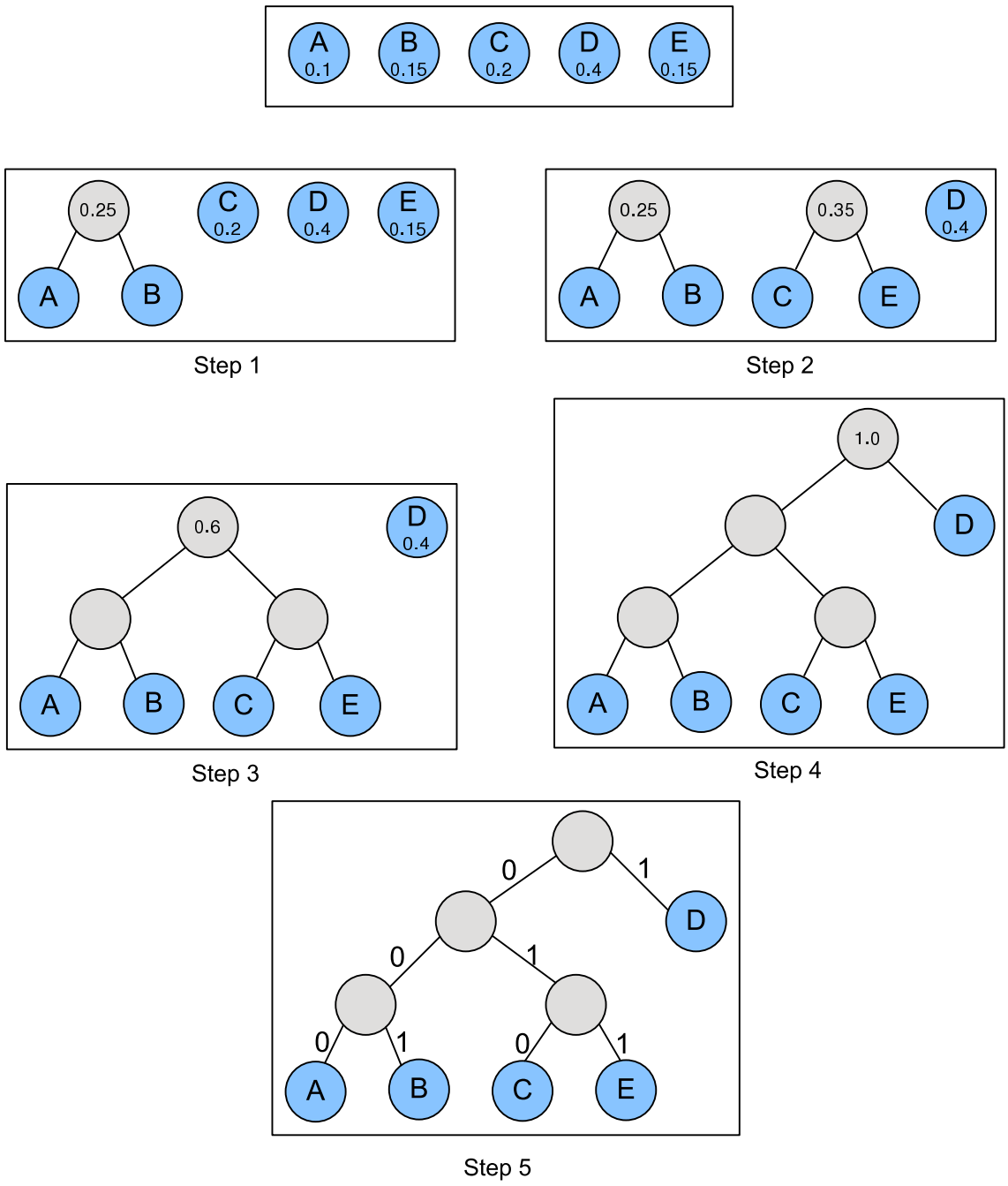


FIGURE 2.6: Huffman Code Example

Once a Huffman code has been generated, data may be encoded simply by replacing each symbol with it's code.

Figure 2.6 shows an example of encoding five alphabet symbols (A through E) with different probabilities using Huffman Coding. The probabilities of the symbols A, B, C, D, and E are 0.1, 0.15, 0.2, 0.4, and 0.15, respectively.
First, a forest consisting of one node for each symbol is created. Each node is marked

with the probability of occurrence for the symbol it represents.
To build the binary tree, each two nodes with the lowest probabilities are combined.

In the first step, nodes A and B are combined together. A new node is created with the sum of their probabilities, i.e. 0.25 (Fig. 2.6, Step1).

In the second step, nodes C and E are combined together. A new node is created with the sum of their probabilities, i.e. 0.35 (Fig. 2.6, Step2). Node D is left without combination at this step.

In the third step, another new tree with the lowest two probabilities is created. The new tree will have the nodes A, B, C, and E. The probability of occurrence for this whole tree is the sum of probabilities of these nodes, i.e. 0.60 (Fig. 2.6, Step3).

In the fourth step, the tree created in the third step will include the node D. The probability of occurrence for this whole tree is, of course, 1 (Fig. 2.6, Step4). Each symbol is represented by a leaf node, and the length of its code equals the node's distance to the root.

In the last step, the edges in the Huffman tree are marked with 0 and 1 from the bottom to the top (Fig. 2.6, Step5).

Thus, the symbols would be encoded as follows:
A: 000, B: 001, C: 010, D: 1 and E: 011

2.4 Embedded Processors

2.4.1 Introduction to RISC Architecture

Reduced Instruction Set Computer (RISC) is a type of microprocessor that reduces chip complexity by using simple and limited set of instructions.

One advantage of reduced instruction set computers is that they can execute their instructions very fast because the instructions are so simple. Another, perhaps more important advantage, is that RISC chips require fewer transistors, which makes them cheaper to design and produce.

There has been, and continues to be, some debate as to exactly what constitutes a "RISC" processor [55]. It is generally agreed, however that they can be characterized by having a load-store architecture and a fixed instruction size, both of which substantially simplify the design. A load/store machine can only read memory with a load instruction and can only write memory with a store instruction. They cannot, for example, increment the contents of a memory location with a single instruction: one must load the value

into a register, increment the register, and store the new value to memory. This need for intermediary storage makes it highly desirable to have a large number of on-chip registers. A fixed instruction size means that all discrete operation that can be performed by the CPU must be expressed in the same number of bits. Most RISC architectures are designed around a 32-bit instruction word. They were developed at a time when memory technology made 32-bit addresses (thus 32-bit registers, and thus 32-bit memory words) desirable. 32 bits is sufficient to encode a reasonably rich set of operations, and to give them the ability to access a reasonably large register set.

Certain design features have been characteristic of most RISC processors:

1. One cycle execution time: most of RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU.
2. Pipelining: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions, and to enhance the throughput of the processor.
3. Large number of registers: RISC design philosophy generally incorporates a larger number of registers to reduce memory traffic.

Well known RISC families include Alpha, ARC, ARM, AVR, MIPS, PowerPC, ...

In this thesis, to evaluate the compression techniques in Chapter 4, three major embedded processor architectures are used, namely MIPS [41], ARM [42] and PowerPC [50]. Introduction to each processor is given in the following sections.

2.4.1.1 MIPS Processor

The MIPS processor was developed as part of a VLSI research program at Stanford University in the early 80s. It implements a smaller and simpler instruction set. Each of the instructions included in the chip design runs in a single clock cycle. The MIPS processor contains 32 general purpose registers, all of which are 32-bit wide. The processor uses a technique called pipelining to more efficiently process instructions.

MIPS R3000 processor has 5 pipelining stages (see Fig. 2.7):

1. Fetch instructions from memory (IF)
2. Read registers and decode the instruction (ID)
3. Execute the instruction or calculate an address (EX)

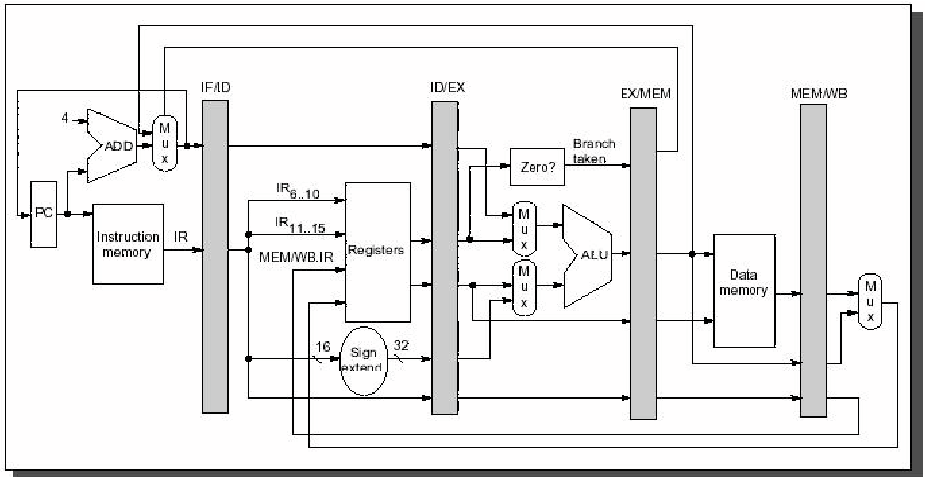


FIGURE 2.7: General Pipeline stages of a RISC processor

opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)

FIGURE 2.8: MIPS instruction format of R-Type group

opcode (6)	rs (5)	rt (5)	immediate (16)

FIGURE 2.9: MIPS instruction format of I-Type group

4. Access an operand in data memory (MEM)
5. Write the result into a register (WB)

MIPS instruction set consists of about 111 total instructions, each represented in 32-bit. The instructions are classified into four different groups according to their coding formats [41]:

R-Type: This group contains all instructions that do not require an immediate value, target offset, memory address displacement or memory address to specify an operand. This includes arithmetic and logic with all operands in registers, shift instructions and register direct jump instructions ('jalr' and 'jr'). This format has fields for specifying of up to three registers and a shift amount, as shown in Fig. 2.8.

I-Type: This group includes instructions with an immediate operand, branch instructions and load and store instructions. This format has fields for specifying of up to two registers and a 16-bit immediate field that codes an immediate operand, a branch target offset or a displacement for a memory operand, as shown in Fig. 2.9.

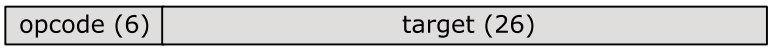


FIGURE 2.10: MIPS instruction format of J-Type group

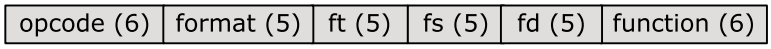


FIGURE 2.11: MIPS instruction format of Coprocessor-Type group

J-Type: This group consists of the two direct jump instructions ('j' and 'jal'). These instructions require a 26-bit coded address field to specify the target of the jump, as shown in Fig. 2.10.

Coprocessor-Type: MIPS processors all have two standard coprocessors, CP0 and CP1. CP0 processes various kinds of program exceptions. CP1 is a floating point processor. The MIPS architecture allows future inclusion of two additional coprocessors, CP2 and CP3. The instruction in this group is broken up into fields of the same sizes as in the R-type instruction format. However, the fields are used in different ways. Most floating point instructions use the format field to specify a numerical coding format like single precision, double precision or fixed point, as shown in Fig. 2.11.

For more details about the MIPS instructions, see the Appendix A.

2.4.1.2 ARM Processor

The ARM architecture is a 32-bit RISC processor developed by ARM Limited in the early 80s [54]. Because of their power saving features, ARM CPUs are dominant in the mobile electronics market. In 2007, about 98 percent of all mobile phones used at least one ARM-designed core on their motherboards [48, 49]. In 2010 the first 32nm ARM processor will be released, enabling users to benefit from an increased battery life and more attractive features.
The ARM processor has 37 registers in total, all of which are 32-bit wide. The registers are arranged into several banks, with the accessible bank being governed by the processor mode.
Each ARM instruction is encoded into 32-bit word. All ARM processor instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags

The basic encoding format of the ARM instructions, such as Load, Store, Move, Arithmetic, and Logic instructions, is shown in Fig. 2.12

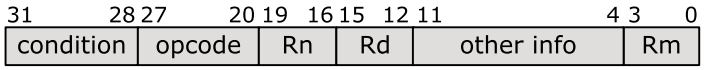


FIGURE 2.12: Basic encoding format of ARM instructions

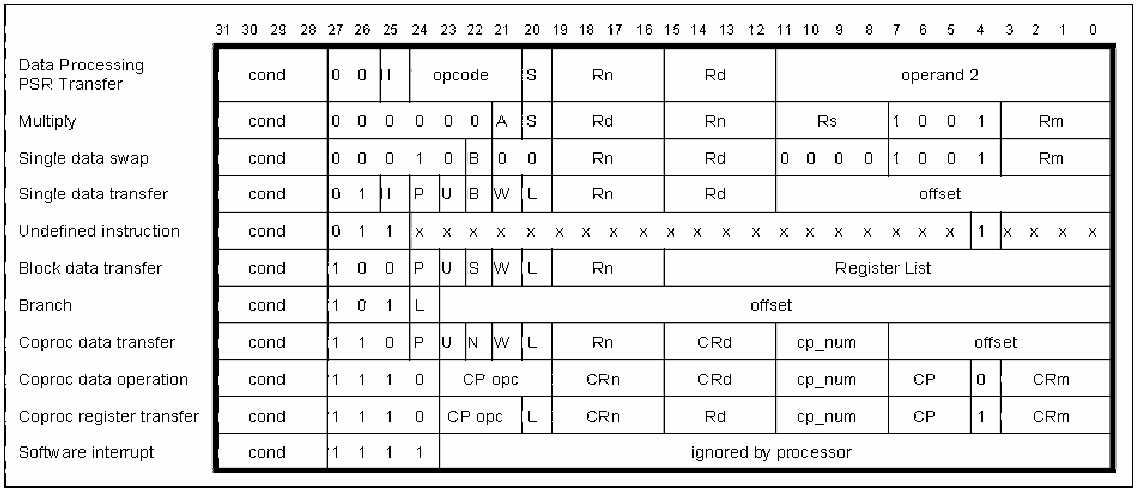


FIGURE 2.13: ARM instruction set summary

In this format, the instruction contains a conditional execution field (condition), the opcode field (opcode), two or three register fields (Rn, Rd, and Rm), and field for some required information (other info).

Fig 2.13 shows a summary of the ARM processor instruction set.

2.4.1.3 PowerPC Processor

PowerPC is a RISC instruction set architecture created in 1991 by the Apple-IBM-Motorola alliance. It was originally defined as a 32-bit architecture and was later extended to 64-bits. It is intended for a wide range of systems, including battery-powered personal computers; embedded controllers; high-end scientific and graphics workstations; and multiprocessing, microprocessor-based mainframes [50].

The PowerPC incorporates conventional RISC features, such as fixed length, consistently encoded, and relatively simple instructions. But it is optimized for single-chip implementations and has several attributes that set it apart from existing RISC processors.

First, PowerPC is superscalar, which means it can handle more than one instruction per clock cycle. Instructions can be sent simultaneously to three types of independent execution units (branch units, fixed-point units, and floating-point units), where they can execute concurrently, but finish out of order. PowerPC uses pipelined designs to process

instructions in stages for increased performance. This simplifies programming and ensures software compatibility across several implementations.
Another distinguishing feature of PowerPC is that it includes several "compound" instructions to reduce instruction-path length. This speeds up the execution time. PowerPC can also have a 64-bit data bus, while supporting both 32-bit and 64-bit versions. All PowerPC chips will run 32-bit applications as a minimum. Finally, PowerPC differs from other RISC processors by virtue of its unique branching technique and its direct support for floating point as a data type. Support for floating point data in the instruction set greatly enhances performance of computation-intensive applications.

2.4.2 Introduction to VLIW Computer Architecture

Superscalar processors decide on the fly how many instructions to issue [122]. A statically scheduled superscalar must check for any dependences between instructions in the issue packet as well as between any issue candidate and any instruction already in the pipeline. A statically scheduled superscalar requires significant compiler assistance to achieve good performance. In contrast, a dynamically-scheduled superscalar requires less compiler assistance, but has significant hardware costs.

An alternative to the superscalar approach is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences. The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present. Such an approach offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization.

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences used wide instructions with multiple operations per instruction. For this reason, this architectural approach was named VLIW, standing for Very Long Instruction Word, and denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more). The basic architectural concepts and compiler technology are the same whether multiple operations are organized into a single instruction, or whether a set of instructions in an issue packet is pre-configured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware.

To reduce this inflexibility and enhance performance of the approach, several innovations have been incorporated into more recent architectures of this type, while still requiring the compiler to do most of the work of finding and scheduling instructions for parallel execution. This second generation of VLIW architectures is the approach being pursued for desktop and server markets. In the next section, we look at the basic concepts in a VLIW architecture.

2.4.2.1 The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. Since there is not fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.
The advantage of a VLIW increases as the maximum issue rate grows. Indeed, for simple two issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four issue processor has manageable overhead, this overhead grows with issue width. Figure 2.14 shows a generic VLIW implementation with four functional units.

VLIW approaches make sense for wider processors. For example, a VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straight line code, then local scheduling techniques, which operate on a single basic block can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex global scheduling algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they must deal with significantly more complicated tradeoffs in optimization, since moving code across branches is expensive.

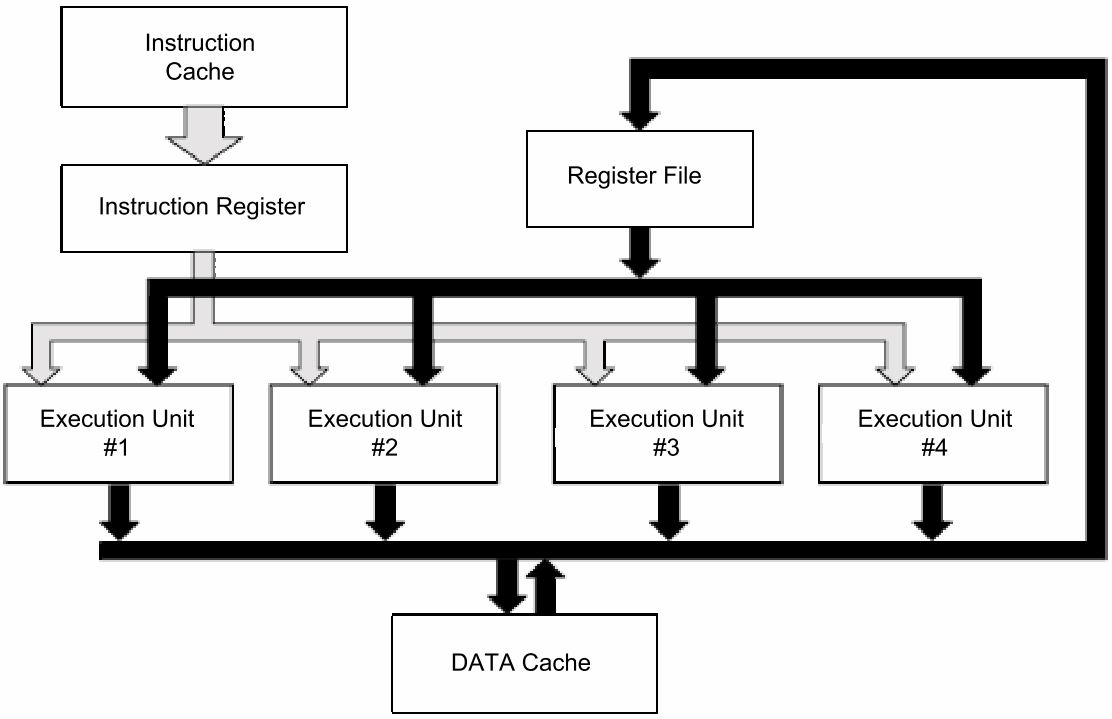


FIGURE 2.14: Block diagram of a generic VLIW Processor.

For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of lock-step operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled. The simplest VLIW instruction format encodes an operation for every execution unit in the machine. This makes sense under the assumption that every instruction will always have something useful for every execution unit to do. Unfortunately, despite the best efforts of the best compiler algorithms, it is typically not possible to pack every instruction with work for all execution units. Also, in a VLIW machine that has both integer and floating-point execution units, the best compiler would not be able to keep the floatingpoint units busy during the execution of an integer-only application.
The problem with instructions that do not make full use of all execution units is that they waste precious processor resources: instruction memory space, instruction cache space, and bus bandwidth.

There are different solutions to reducing the waste of resources due to sparse instructions [87] . For example, instructions can be compressed with a more highly-encoded representation. Any number of techniques, such as Huffman encoding to allocate the fewest bits

to the most frequently used operations, can be used. The compressed instructions are then expanded when they are read into the cache or are decoded.
Another solution is to define an instruction word that encodes fewer operations than the number of available execution units. Imagine a VLIW machine with ten execution units but an instruction word that can describe only five operations. In this scheme, a unit number is encoded along with the operation; the unit number specifies to which execution unit the operation should be sent. The benefit is better utilization of resources. A potential problem is that the shorter instruction prohibits the machine from issuing the maximum possible number of operations at any one time. To prevent this problem from limiting performance, the size of the instruction word can be tuned based on analysis of simulations of program behavior.

Early VLIWs operated in lock-step; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause all the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

In this thesis, to evaluate the Filled Buffer Compression Technique in Chapter 5, two VLIW processor architectures are used, namely TMS320C62x [51] and TMS320C64x [52]. Introduction to each processor is given in the following sections.

2.4.2.2 TMS320C62x VLIW Processor

The TMS320C62x (briefly C62x) execute up to eight 32-bit instructions per cycle. The C62x CPU consists of 32 general-purpose 32-bit registers and eight functional units. These eight functional units contain:

- Two multipliers
- Six ALUs

Features of the C62x devices include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units
  - Executes up to eight instructions per cycle for up to ten times the performance of typical DSPs
  - Allows designers to develop highly effective RISC-like code for fast development time
- Instruction packing
  - Gives code size equivalence for eight instructions executed serially or in parallel
  - Reduces code size, program fetches, and power consumption
- Conditional execution of all instructions
  - Reduces costly branching
  - Increases parallelism for higher sustained performance
- Efficient code execution on independent functional units
  - Industrys most efficient C compiler on DSP benchmark suite
  - Industrys first assembly optimizer for fast development and improved parallelization
- 8/16/32-bit data support, providing efficient memory support for a variety of applications
- 40-bit arithmetic options add extra precision for vocoders and other computationally intensive applications
- Saturation and normalization provide support for key arithmetic operations
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

The TMS320C62x uses the VelociTI architecture which is a high performance advanced VLIW architecture. The VelociTI architecture of the C62 platform of devices make them the first off-the-shelf DSPs to use advanced VLIW to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple instructions during a single clock cycle. Parallelism is the key to extremely high performance, taking these DSPs well beyond the performance capabilities of traditional superscalar designs. VelociTI is a highly deterministic architecture, having few restrictions on how or when instructions are fetched, executed, or stored. It is this architectural flexibility that is key to the

breakthrough efficiency levels of the TMS320C6000 Optimizing C compiler. VelociTIs advanced features include:

- Instruction packing: reduced code size
- All instructions can operate conditionally: flexibility of code
- Variable-width instructions: flexibility of data types
- Fully pipelined branches: zero-overhead branching

The C62x CPU,(Fig. 2.15) contains:

- Program fetch unit
- Instruction dispatch unit
- Instruction decode unit
- Two data paths, each with four functional units
- 32 32-bit registers
- Control registers
- Control logic
- Test, emulation, and interrupt logic

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 16 32-bit general-purpose registers. A control register file provides the means to configure and control various processor operations. The C62x DSP has a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces.

One instruction word comprises eight 32-bit instructions and is called a Fetch Packet. The instructions in the same Fetch Packet may be executed in one or more clock cycles (due to data dependencies). The basic format of a fetch packet is shown in Fig. 2.16. Fetch packets are aligned on 256-bit (8-word) boundaries.

The execution of the individual instructions is partially controlled by a bit in each instruction, the p-bit. The p-bit (bit 0) determines whether the instruction executes in parallel with another instruction. The p-bits are scanned from left to right (lower to

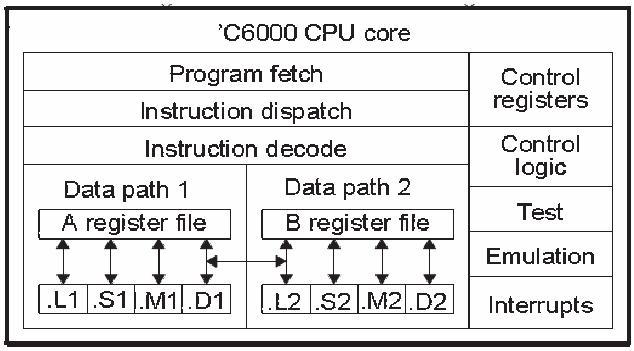


FIGURE 2.15: Block diagram of the TMS320C62x CPU.

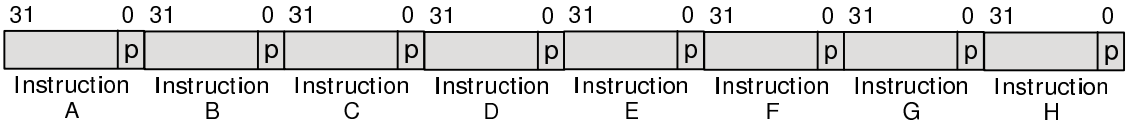


FIGURE 2.16: Basic Format of Fetch Packet.

higher address). If the p-bit of instruction i is 1, then instruction i + 1 is to be executed in parallel with (in the the same cycle as) instruction i. If the p-bit of instruction i is 0, then instruction i + 1 is executed in the cycle after instruction i. All instructions executing in parallel (in one clock cycle) constitute an execute packet. An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit. Hence, each Fetch Packet may be made up of several Execute Packets. An execute packet cannot cross an 8-word boundary. Therefore, the last p-bit in a fetch packet is always cleared to 0, and each fetch packet starts a new execute packet.

There are three types of p-bit patterns for Fetch Packets. These three p-bit patterns result in the following execution sequences for the eight instructions:

- Fully Serial
- Fully Parallel
- Partially Serial

Fig. 2.17 shows an example of the execution sequence of the "Fully Serial" p-Bit Pattern. The number of cycles required to execute the Fetch Packet is 8 cycles.

Fig. 2.18 shows an example of the execution sequence of the "Fully Parallel" p-Bit Pattern. All eight instructions are executed in parallel (in one clock cycle).

Fig. 2.19 shows an example of the execution sequence of the "Partially Serial" p-Bit Pattern. 4 clock cycles are required to execute the Fetch Packet. Instructions C, D, and

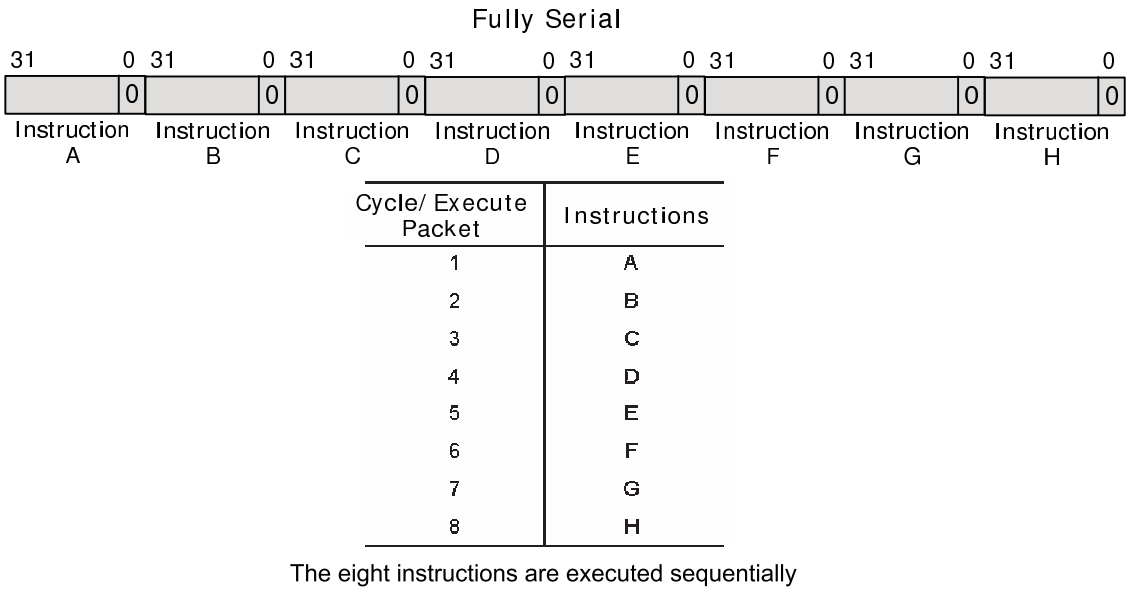


FIGURE 2.17: Fully Serial p-Bit Pattern in a Fetch Packet

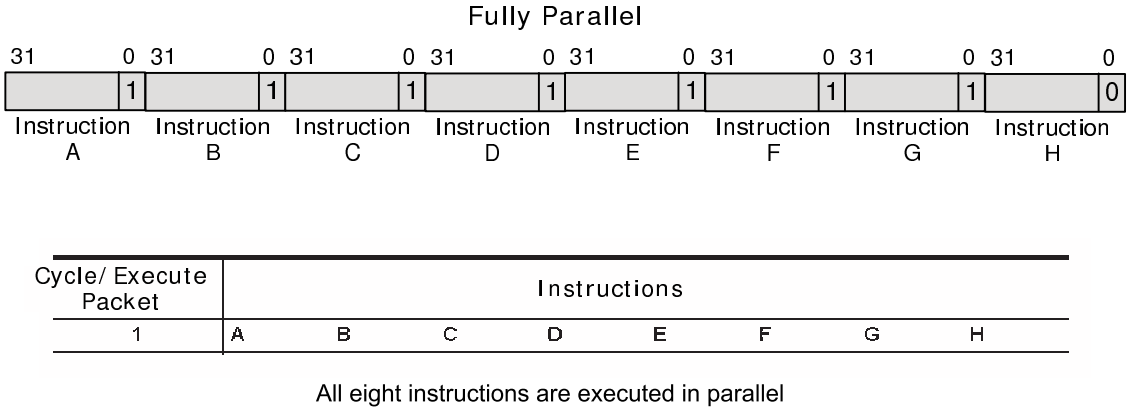


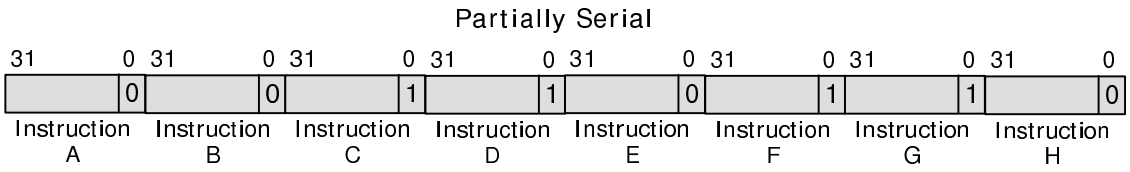
FIGURE 2.18: Fully Parallel p-Bit Pattern in a Fetch Packet

E do not use any of the same functional units. This is also true for instructions F, G, and H.

If a branch in the middle of an Execute Packet occurs, all instructions at lower addresses will be ignored. In Fig. 2.19, if a branch to the address containing instruction D occurs, then only D and E Instructions will be executed. Even though instruction C is in the same Execute Packet, it will be ignored. Instructions A and B will be also ignored because they are in earlier Execute Packets.

2.4.2.3 TMS320C64x VLIW Processor

TMS320C64x (briefly C64x) is a developed version of the C62x and is a part of the DaVinci multimedia processor by Texas instruments. It consists of 64 general-purpose



Cycle/ Execute Packet	Instructions		
1	A		
2	B		
3	C	D	E
4	F	G	H

FIGURE 2.19: Partially Serial p-Bit Pattern in a Fetch Packet

32-bit registers and eight functional units. C64x has the same features of the C62x. It has also some additional feature:

- Each multiplier can perform two 16 X 16 bit or four 8 X 8 bit multiplies every clock cycle.
- Quad 8-bit and dual 16-bit instruction set extensions with data flow support
- Support for non-aligned 32-bit (word) and 64-bit (double word) memory accesses

These features decrease the number of NOPs and consequently reduces the size of compiled program code in comparison to the C62x.

2.4.3 Comparison between RISC and VLIW processors

The differences between RISC, and VLIW are in the formats and semantics of the instructions [87].
RISC instructions specify simple operations, are fixed in size, and are easy (quick) to decode. RISC architectures have a relatively large number of general-purpose registers. Instructions can reference main memory only through simple load-register-from-memory and store-register-to-memory operations. RISC instruction sets do not need microcode and are designed to simplify pipelining.

VLIW instructions are like RISC instructions except that they are longer to allow them to specify multiple, independent simple operations. A VLIW instruction can be thought of as several RISC instructions joined together. VLIW architectures tend to be RISC-like in most attributes.

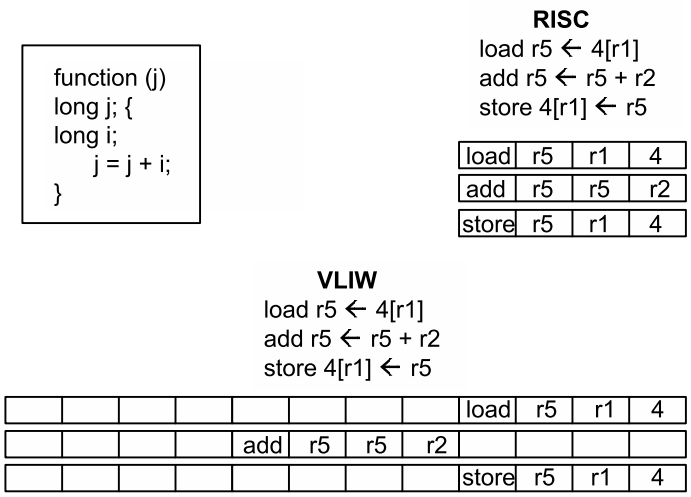


FIGURE 2.20: Example for implementing a function in RISC and VLIW processors.

Figure 2.20 shows a C-language code fragment containing small function definition. This function adds a local variable to a parameter passed from the caller of the function. The implementation of this function in RISC and VLIW code is also shown. This example illustrates the differences between the architectures.

The RISC code is artificially inefficient. Normally, a good compiler would pass the parameter in a register, which would make the RISC code consist of only a single register-to-register add instruction. For the sake of illustration, however, the code will consist of three instructions as shown. These three instructions load the parameter to a register, add it to the local variable already in a register, and then store the result back to memory. Each RISC instruction requires four bytes.

The VLIW code is similarly hampered by poor register allocation. The example VLIW architecture shown has the ability to simultaneously issue three operations. The first slot (group of four bytes) is for branch instructions, the middle slot is for ALU instructions, and the last slot is for the load/store unit. Since the three RISC operations needed to implement the code fragment are dependent, it is not possible to pack the load and add in the same VLIW instruction. Thus, three separate VLIW instructions are necessary.

With the code fragment as shown, the VLIW instruction is depressingly inefficient from the point of view of code destiny. In a real program situation, the compiler for the VLIW would use several program optimization techniques to fill all three slots in all three instructions. It is instructive to contemplate the performance each machine might achieve for this code. We need to assume that each machine has an efficient, pipelined implementation.

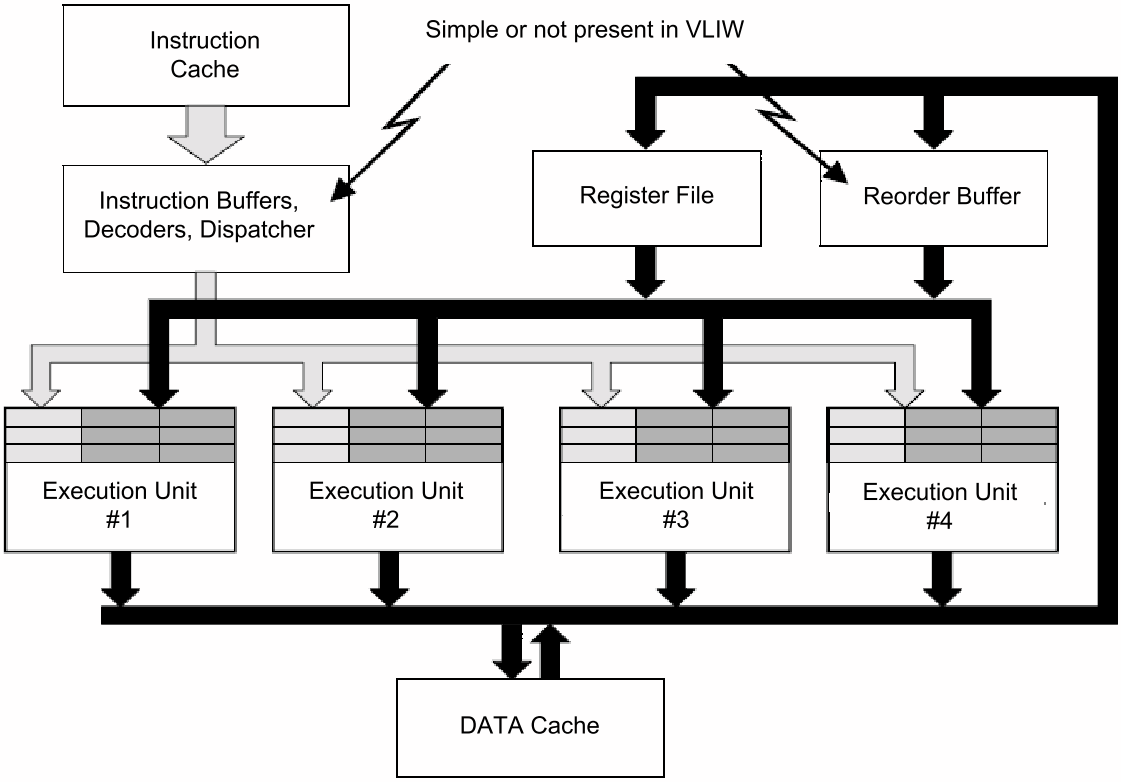


FIGURE 2.21: Block diagram of a superscalar RISC Processor.

A RISC machine would be able to execute the fragment in three cycles.
The VLIW machine, assuming three fully-packed instructions, would effectively execute the code for this fragment in one cycle. To see this, observe that the fragment requires three out of nine slots, for one-third use of resources. One-third of three cycles is one cycle.

High-performance RISC designs are called superscalar implementations. Superscalar in this context simply means beyond scalar where scalar means one operations at a time. Thus, superscalar means more than one operation at a time.
Some more recent RISC architectures have been designed with superscalar implementations in mind. The most notable examples are the DEC Alpha and IBMPOWER (from which PowerPC is derived).

Figure 2.21 shows high-level block diagram of a superscalar RISC processor implementation. The implementation consists of a collection of execution units (integer ALUs, floating-point ALUs, load/store units, branch units, etc.) that are fed operations from an instruction dispatcher and operands from a register file.
The execution units have reservation stations to buffer waiting operations that have been issued but are not yet executed. The operations may be waiting on operands that are not yet available.
The instruction dispatcher examines a window of instructions contained in a buffer. The

dispatcher looks at the instructions in the window and decides which ones can be dispatched to execution units. It tries to dispatch as many instructions at once as is possible, i.e. it attempts to discover maximal amounts of instruction-level parallelism. Higher degrees of superscalar execution, i.e., more execution units, require wider windows and a more sophisticated dispatcher.
The reorder buffer is used to undo the effects of speculatively executed instructions in the case of a mispredicted branch.

A VLIW implementation achieves the same effect as a superscalar RISC implementation, but the VLIW design does so without the two most complex parts of a high-performance superscalar design.
Because VLIW instructions explicitly specify several independent operationsthat is, they explicitly, specify parallelismit is not necessary to have decoding and dispatching hardware that tries to reconstruct parallelism from a serial instruction stream. Instead of having hardware attempt to discover parallelism, VLIW processors rely on the compiler that generates the VLIW code to explicitly specify parallelism. Figure 2.14 shows a generic VLIW implementation, without the complex reorder buffer and decoding and dispatching logic.

Chapter 3

Related Work

In this chapter, we first introduce the commercial compression techniques which have been implemented in ARM, MIPS, and PowerPC. Then, we present a survey of the related literature in the field of code compression.

3.1 Commercial Implementations of Code Compression Techniques

Among 32bit embedded processors, ARM, MIPS and PowerPC were the first to develop code-density tricks in real system to reduce their memory footprints.
Information about these techniques is presented in the following sections.

3.1.1 ARM Thumb

In the most speed-critical of embedded devices, the cost of memory is much more critical than the execution speed of the processor [57]. To reduce memory requirements and cost, Advanced RISC Machines (ARM) created the Thumb instruction set as an option for their RISC processor cores.

Like the ARM architecture, the Thumb processor is an advanced RISC load/store machine. The Thumb shares many properties with the ARM as it operates as a subset of the ARM architecture. These two processors are fundamentally the same; they run on the same silicon chip and operate in much the same way, indeed, one can switch between the two modes in the same program. What makes the Thumb different from the ARM is the register set, register size, and instruction size.

The Thumb register set is a subset of the ARM register set. Instead of 16 GPRs, only 8 general purpose registers, R0-R7, are available. The register set also differs in size, 16-bits

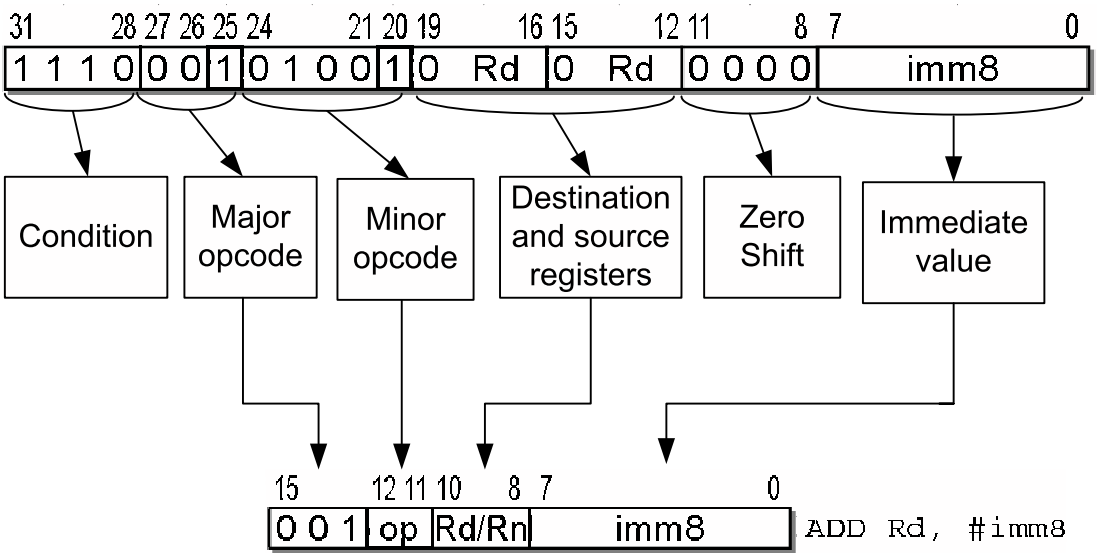


FIGURE 3.1: Mapping of ARM instruction format of ADD to Thumb.

for Thumb and 32-bits for ARM. In addition to these GPRS, as in the ARM state, are the PC, SP, SPSR, and Link registers. The Thumb registers are directly mapped from the ARM state registers, facilitating a convenient transfer of data when switching between the two modes. The Thumb instruction set, like the register set, is taken directly from the ARM architecture. Thumb implements a condensed subset of the ARM instructions and reduces them in size from 32-bits to 16-bits. By doing so, the number and strength of instructions are similarly reduced. The Thumb ISA consists of a base of 19 Opcode formats2 representing instructions ranging from data processing LDR, STR to the most complex arithmetic, MUL. The instructions only work on the limited register set of the Thumb and are no longer conditionally executed by default. Fig. 3.1 shows an example of mapping ADD instruction in ARM instruction format to Thumb.

A disadvantage related to these compacted ISAs is that they increase the instruction number of the user program by 40% in comparing to ARM instruction number of the same program. This results in slower timing performance. It was reported in [28] that Thumb programs run 15%-20% slower than ARM programs.

Using Thumb, a code saving of 30% may be achived. This is referring to 70% compression ratios.

3.1.2 MIPS16e

MIPS16 does not propose a code compression technique which is integrated with the processor in a system-on-a-chip design. Instead, it develops code-density tricks in real

system to reduce their memory footprints.

It is classified by MIPS as an architecture extension, meaning that, while MIPS16 support is not mandatory for all future MIPS implementations, it is the standard mechanism for code compression across all suppliers of MIPS RISC CPUs. It has been designed to be compatible with the existing 32-bit (MIPS-I/II) and 64-bit (MIPS-III) architecture and programming model.

MIPS16 defines a 16-bit fixed-length instruction set architecture (ISA) that is a subset of MIPS-III. To shrunk the length of the instruction from 32-bit to 16-bit, the three components of the instruction word, opcodes, register numbers, and immediate values should be modified [55].

The first step that was taken was to perform statistical analysis on a number of MIPS binaries from a variety of applications from embedded, real-time, and workstation environments. This was to determine the frequency distributions of opcode use, of the number of registers simultaneously in use, and of the number of significant bits in immediate values. The results showed that, while the opcode and function code fields could be reduced, and some instructions "thrown away", the MIPS instruction set was already very lean. While the base MIPS instruction set has 6 bits of major opcode field, sometimes modified by a 6 bit function code, MIPS16 reduces the major opcode field and the function modifier to 5 bits each, and defines a total of 79 instructions, of which 24 are only required for MIPS-III implementations supporting 64-bit data words.

More leverage was to be had in reducing the size and number of register specifier fields in the instructions. The analysis showed that, most of the time, compiler-generated code was using 8 or fewer registers. Restricting MIPS16 to 8 registers allows register specifiers of 3 bits instead of 5. The R-Format standard MIPS instructions support 3 operands, two inputs and an output. In many cases, MIPS16 only permits two register specifiers per arithmetic instruction. One of the input registers must also be used as the result register, overwriting that input.

Perhaps the biggest saving comes from restrictions on the size of immediate values expressible. In the place of the 16-bit immediate field of the MIPS I-Format instructions, most MIPS16 immediate fields are restricted to 5 bits. Some are restricted to 3 or 4. An example of the resulting mapping is given in Fig. 3.2.

The MIPS16 architecture provides for the efficient run-time switching between compressed and 32-bit modes of operation through the JALX (or Jump And Link with eXchange) instruction. This is like the MIPS-I JAL instruction in that it transfers control to the specified address while saving the address of the instruction logically following the jump,

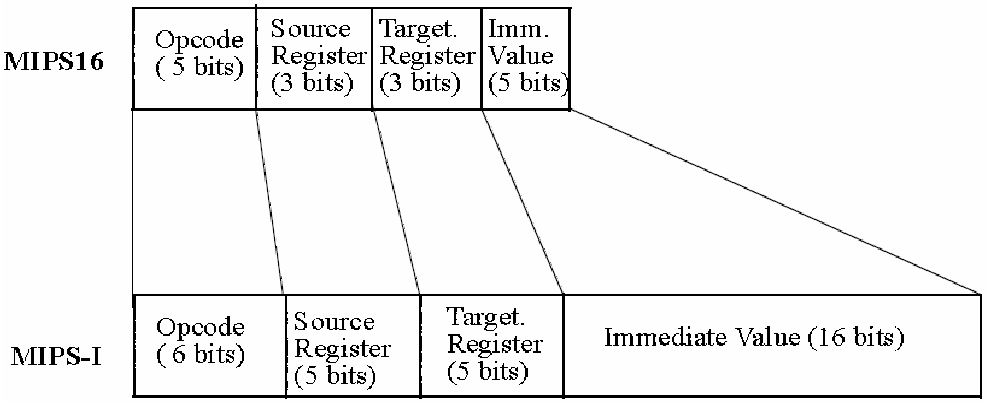


FIGURE 3.2: Mapping of MIPS-I instruction format to the compressed form of MIPS16.

the return address, in a link register. The JAL is the MIPS mechanism for subroutine calling. The JALX extends the semantics by also toggling the state of the instruction decode logic between 32-bit MIPS mode and MIPS16 mode. A 32-bit subroutine can call a 16-bit subroutine, and vice versa. The previous state is merged with the return address, and restored automatically on return from the subroutine.

MIPS16 instructions are half the size of their standard MIPS counterparts, but also somewhat less expressive. The careful design of the compressed instruction set has minimized the impact of this loss of expression. More instructions are required to perform some operations, but with compilers optimized for MIPS16, a net code saving of 40% has been achieved across a range of embedded and desktop codes. This is referring to 60% compression ratios.

The main shortcomings of MIPS16 are:

- performance penalty caused by the lack of several instructions in the dense instruction set
- Time required to switch between 32-bit and 16-bit modes

This results in that the programs which are using MIPS16, are running 15%-20% slower than programs which are using standard RISC instruction set [28].

3.1.3 IBM CodePack for PowerPC

CodePack, introduced by IBM [72] in 1998, is a prefix coding method used to store complete PowerPC instructions in memory in a compressed format. Unlike Thumb and

MIPS16e, IBMs CodePack system really does compress executable code. CodePack represents the state of the art in hardware-assisted decompression and is the only architecture currently implemented on silicon [16]. It is like running WinZip on your PowerPC software. CodePack analyzes and compresses entire programs, producing a compressed version that has to be decompressed and executed on-the-fly. For all its complexity, CodePack delivers about the same 20-30 percent space savings as the others [73].

The main goal of the CodePack compression scheme was to develop a method to efficiently compress PowerPC application code which at runtime could quickly be decoded with a small amount of logic. Another key design point was to maintain full instruction set capability.

3.1.3.1 Compression Technique

To compress PowerPC code, the first thought was to substitute variable-length bit patterns for full 32-bit instructions. This method did not produce appreciable compression results because the distribution of unique instruction patterns in most application programs is too uniform. For that, CodePack splits the PowerPC instructions into two 16-bit halves, with each half compressed separately. Two decode tables of 512 entries each are generated and the size of the compressed instructions is between 7 bits in the best case and 38 bits in the worst case.

In CodePack, the following steps are conducted for compression:

- The frequency of the unique high 16-bit and low 16-bit instruction patterns that exist in the text sections is counted.
- The two lists are sorted according to their frequency
- The first 512 entries of each list are extracted and compressed using Huffman Coding (the patterns that appear frequently in the text section are replaced with short bit code words and the less frequently patterns are replaced with longer code words). One decode table with 512 entry for each list is created. These tables will be later used by the decompression core to decompress the compressed instructions.
- The high and low 16-bit patterns in the text sections are replaced with the appropriate entry from the decode tables. Since the tables have a fixed number of entries, a special tag is used to mark those entries that do not appear in the table.

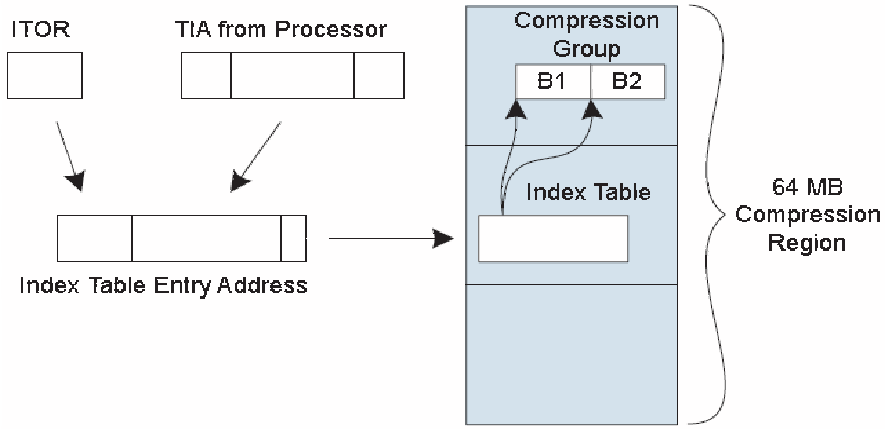


FIGURE 3.3: Locating compression blocks in CodePack.

Compressed instructions are stored consecutively in memory making sequential execution easy. However, since compressed instructions are no longer fixed in length and all application programs have execution discontinuities in them (such as branches and exceptions), a method is needed to map the original instruction addresses to their corresponding location in compressed memory. If an index table was created in memory that mapped each possible instruction address to its equivalent compressed address, the benefits of the compression would quickly be lost due to the size of the index table. However, if instructions are assembled into groups, an index table entry is only required for each group and the index table shrinks to a reasonable size with only a small loss in efficiency.

CodePack assembles an aligned 64-byte piece of decompressed memory (i.e. 16 instructions) into a compression block. Each two compression blocks constitute one compression group. Hence, a compression group is made up of an aligned 128-byte piece of decompressed memory (see Fig. 3.3).

An index table is a table in memory made up of a series of 32-bit entries that map Target Instruction Addresses (TIAs) to their respective addresses in compressed memory. There is one entry for each compression group. The information in an index table entry allows the decompression core to determine the starting address of both compression blocks within the compression group (see Fig. 3.4). Since there is one entry per compression group, index table overhead is 4 bytes per 128 bytes of uncompressed instructions (just slightly over 3%). An index table can be up to 2 MB in size, which would cover an entire 64-MB compression region.

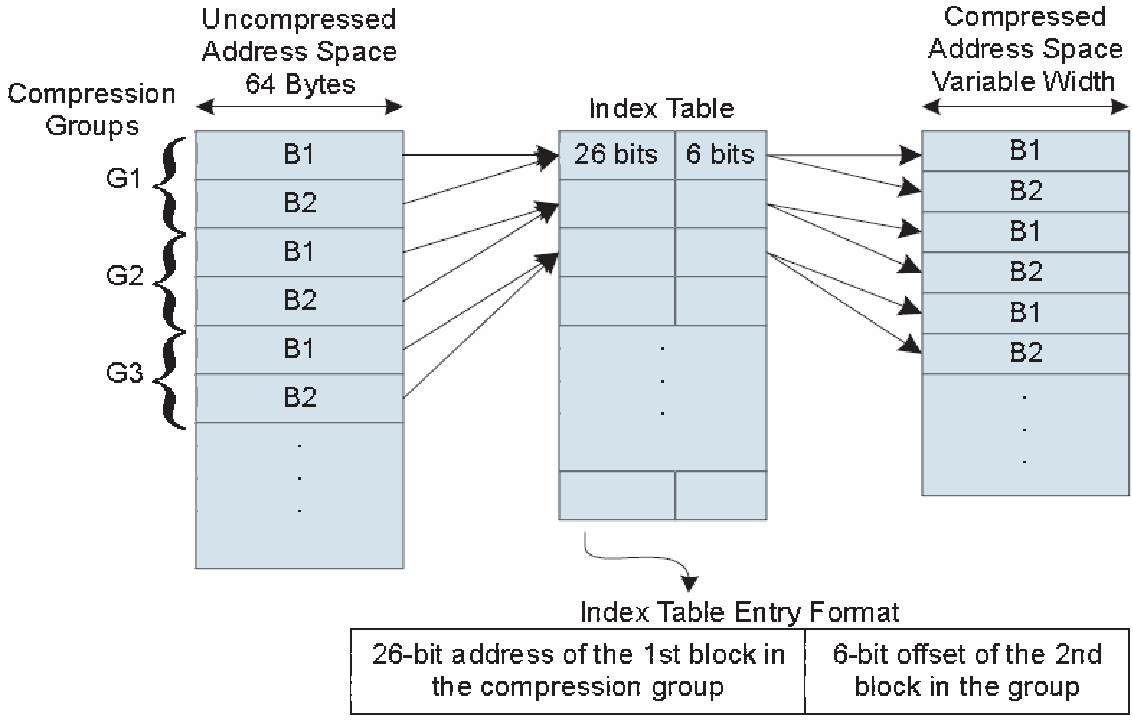


FIGURE 3.4: Index Table mapping of Target Instruction Address (TIA) to compressed memory.

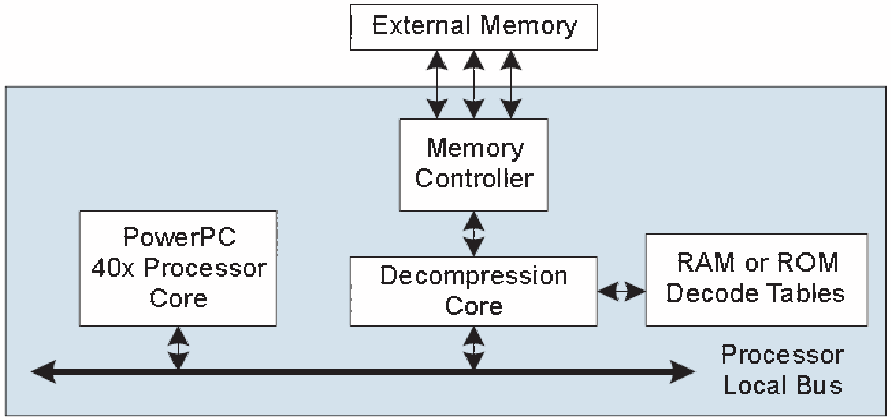


FIGURE 3.5: An integrated design including the CodePack decompression core.

3.1.3.2 Decompression

The hardware decoder is a silicon-efficient ASIC core that is placed between the processor and the memory controller in an integrated system-on-a-chip design (see Fig. 3.5). The core decompresses instructions on-the-fly only as needed by the processor.

The following steps are conducted for decompression:

- The processor fetches an instruction at a TIA in memory configured as compressed.
- The decompression core calculates the address of the index table entry of the compression group containing.
- The decompression core fetches the index table entry from external memory.
- The decompression core calculates the starting address of the compression block containing the target instruction using the information in the index table entry (see Fig. 3.3).
- The decompression core reads (burst mode) the compression block containing the target instruction from memory. As the compression block is read in, the decompression core uses the contents of the decode lookup tables to decompress the block.

3.1.3.3 Results

CodePack has reported performance of an overall program size reduction of 35-40% [72] (i.e. a compression ratio of 60-65%). This does not appear to take into account the size of the decompression unit which must be included for the compression to become effective. CodePack uses variable length encoding and requires the use of a mapping table to calculate the given address of a given instruction.
Fig. 3.6 shows the compression ratios of some benchmarks obtained when CodePack was used [16].

The decompression adds a latency to the processors pipeline. This latency appears clearly by the branch instructions. This is because the decoder can only decompress full compression blocks (16 instructions) as whole. For that, if a branch target instruction requested by the processor is located at the end of the compression block, this causes the decompression core to read and decode the block serially, by starting with the first instruction of the block and continuing until all the instructions before the branch target have been read and decompressed.

3.2 Classification of Related Work

Previous work in code compression can be categorized by means of different criteria, for example, software-based compression techniques [31, 62, 63, 88, 90, 91], hardware-based compression techniques [14, 89], or compiler-based techniques [119, 120], etc.
Two classifications for the previous code compression work are used in this thesis:

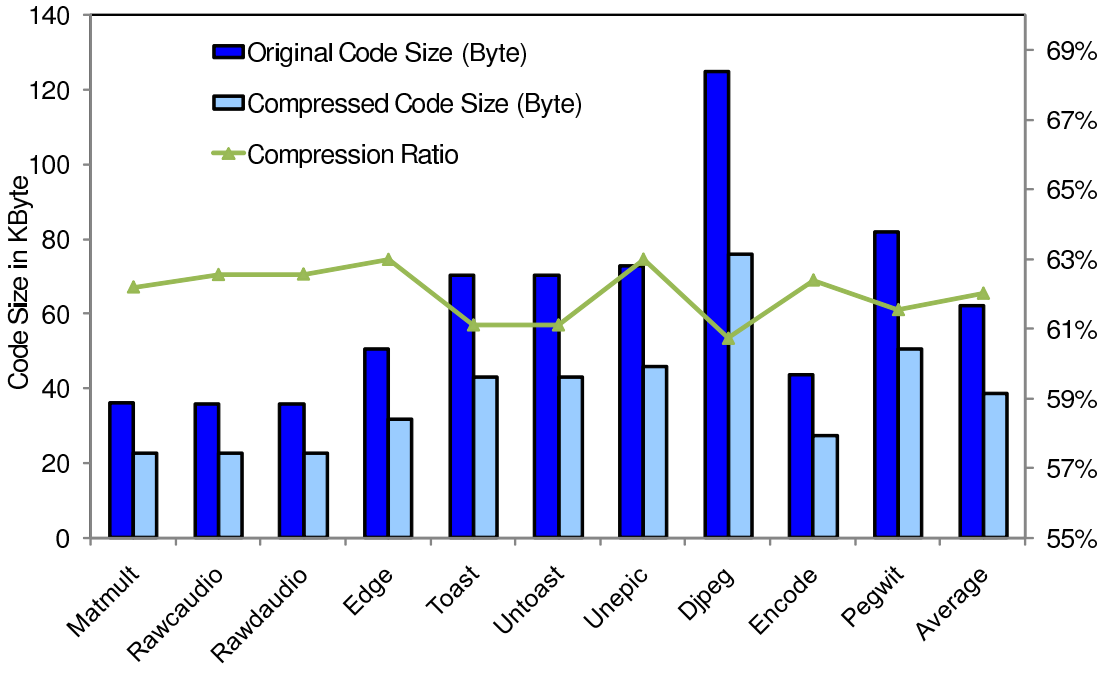


FIGURE 3.6: CodePack Compression Ratio

(1) Classification by method of compression. The previous work in this classification is organized by Dictionary-based such as CodePack in [71, 72, 98] or SADC in [65], and *Statistical compression techniques* such as Arithmetic Coding [74, 77, 93] or Markov models [17].
(2) Classification by ISA-Dependability. The previous work in this classification is organized depending on whether the compression technique is specified for one Instruction Set Architecture (ISA) or it is orthogonal to any architecture.

3.2.1 Classification by Method of Compression

There are several related approaches that use *Dictionary-based compression method*.

Yoshida, Song, Okuhate, Onoye, and Shirakawa [27]
Yoshida et al. [27] noted that compilers tend to generate many duplicate instructions. They developed a compression algorithm to unify the duplicated instructions existing in the embedded program and assign a compressed object code to such an instruction. Compression can be achieved because the compressed instructions are shorter than the original ones. The original instructions were stored in the dictionary in the memory. The compression technique operates by searching through the program of N instructions, each having m bits, to find a complete list of N distinct instructions. Then assigns a number i to each distinct instruction as a log n-bit code. After that, a transform table has to be constructed to transform each pseudo code to m-bit-wide instruction and to implement it in the instruction decoder (see Figure 3.7).

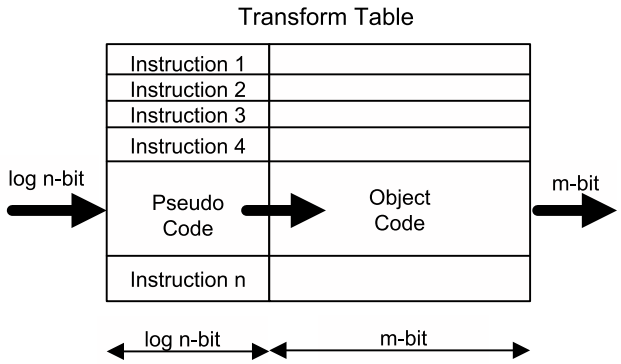


FIGURE 3.7: Transform table of the compression technique in [27].

An optional subcode compression extension was to further reduce the size of instructions by separately compressing codes for registers and flags.

Power dissipation of memory units depends on their physical size. Power reduction was quantified by the equation:

$$P = \frac{N \times \log_2(n) + k \times n \times m}{N \times m}$$

Because of each compressed instruction has its own mapping to the original one, there will be no control transfer instructions problem. While all the compressed instructions in the memory has the same instruction width (log n-bit), there will be no instruction alignment problem in the memory.
Using the ARM610 core, Yoshida et al. ran their compression system on the Dhrystone benchmark. Opcode size of compressed programs typically decreased from 32 bits to 12 bits. A program compression ratios between 22.7% and 54.0% were achieved and power reduction of memory between 19.57% and 42.33% were obtained. These results do not take into account the large external ROM size overhead.

Benini, Macii, and Nannarelli [15]
Benini et al [15] proposed a new DF (Decompress on Fetch) architecture that focuses on reducing decoding overhead on energy and performance. Their technique guarantees that the storage requirements for the compressed program is reduced. The compression algorithm has been designed specifically for fast and low-energy decoding during cache lookup. The code is initially profiled and the subset Sn of the n most frequently executed instructions is obtained. $Log_2 n$ long compressed codes are assigned to the n most frequent instructions.
Instructions are compressed in groups with the size of one cache line. instruction i is compressed only if it belongs to a group of instructions that can be stored in a compressed line. The latter is a group of more than four adjacent instructions which, after compression, will be stored in four consecutive words. The size (four words) and the

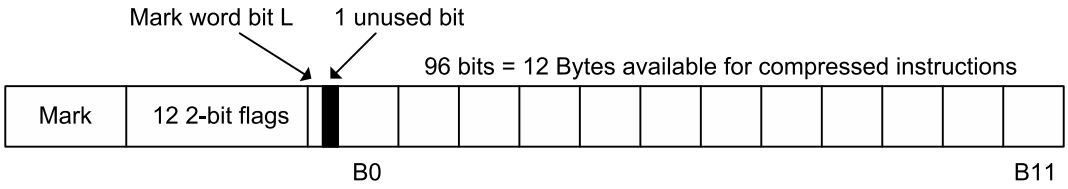


FIGURE 3.8: compressed Line Structure in [15].

memory alignment of a compressed line is such that it fits in a single cache line, when it is cached. on every cache miss, a new cache line is fetched from memory. The line may either contain four uncompressed instructions, or a first word containing mark and flags, followed by three words containing five or more compressed instructions.

The mark is an unused instruction opcode, while the flag bits are divided in 12 groups of 2 bits each, one group for each of the bytes of the remaining three words of the line. One additional flag bit L is reserved at the end of the flags (see Fig. 3.8).

The flags values are assigned as follows: 00 if the corresponding byte contains a compressed instruction; 01 if the corresponding byte contains 8 bits of an uncompressed instruction ; 11 if the corresponding byte is left empty for alignment reasons; 10 is used to signal the last compressed instruction in the line. The last flag bit L marks if the last instruction in the line is compressed or not. Compressed line stores between a minimum of 5 instructions, and a maximum of 12 instructions.

Decompression is performed by addressing the decompression table (a fast RAM containing 256 32-bit words) with the 8-bit compressed instruction code. Benini et al tested their compression technique using Super DLX Core and some of the C benchmarks distributed in the Ptolemy package. Average code size reduction was around 28% and energy saving of 30% has been achieved.

Nam, Park, and Kyung [109]

Nam et al [109] achieved average compression ratios of 63-71% on SPEC95 benchmarks for varying VLIW architectures using a Dictionary compression method and compared the difference in performance of the identical whole instructions and the isomorphic instructions (i.e. the split instructions into opcode/operand fields).
The main idea here is that common "instruction word" are stored in a dictionary, and are replaced in memory with an index that points to the correct dictionary entry. When finding common instructions, two techniques are used, locating identical instruction words and locating isomorphic instruction words.

Isomorphic instruction words are words that contain the same set of opcodes, but the operands used are different; or words that containthe same set of operands even though the opcodes are different. The sort of partitioning is reminiscent of the opcode/operand

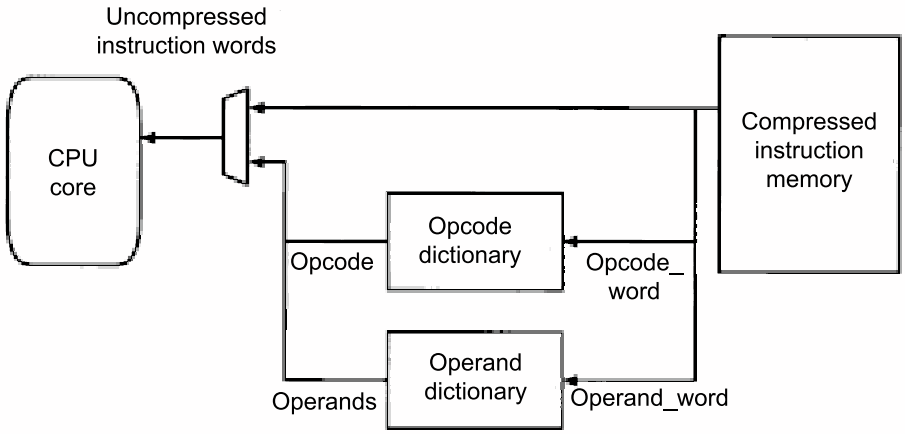


FIGURE 3.9: Instruction fetch path in [109] for VLIW processor-based systems

expression tree partitioning methods for RISC processors except that expression trees are replaced with execution packets.

The authors used the separation into opcodes and operands across the entire fetch-packet. Hence, for an x-issue processor, there will be x opcodes and x operand streams. Two dictionaries were required, one to hold the opcode entries and the other to hold operand entries as shown in Fig. 3.9. The decoder checks the incoming instruction words to determine whether they are compressed or not. For an uncompressed instruction word, it proceeds in a conventional fashion through the upper path. When the decoder encounters a compressed instruction word, it is recovered to the uncompressed one through the lower path by retrieving the original opcodes and operands concurrently from the corresponding entries of the dictionaries pointed to by the opcode_word and the operand_word. Then the uncompressed instruction word is issued to the functional units.

Two methods of investigation common instruction words were compared (identical whole instruction words; and isomorphic split into opcode/operand fields) in varying VLIW architectures. The results showed that using the isomorphic instruction words method out-performed the identical instruction words method by a compression ratio difference of at least 17%.

Liao, Devadas, and Keutzer.

In [61], Liao et al extracted the common sequences of the program and placed them in a dictionary. Instances of these sequences are replaced by mini-subroutine calls to the dictionary. Mini-subroutine call uses a simple CALL instruction but without passing the parameters.

Finding the subsequences was the heart of their compression algorithm. Their system worked on blocks in the control flow graph, building basic blocks with unique successors,

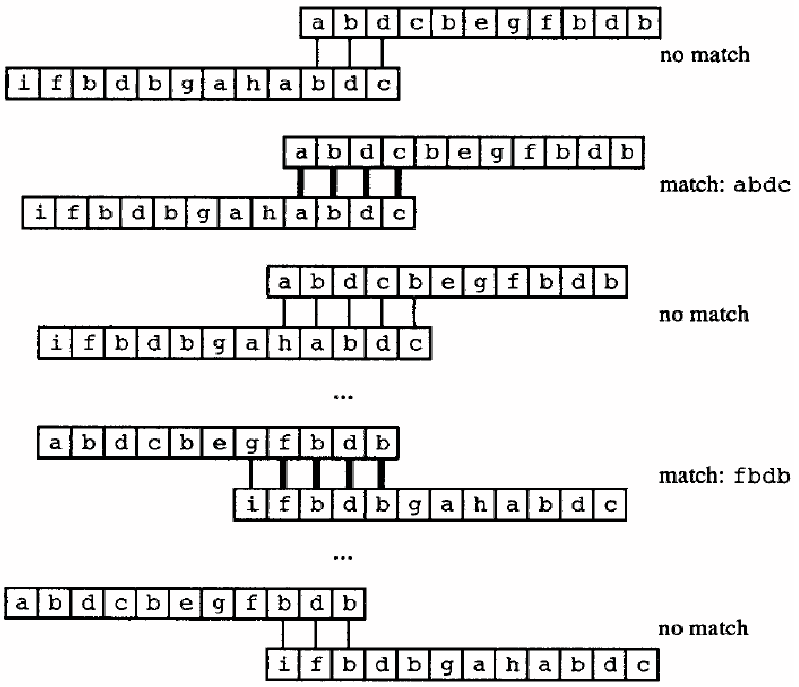


FIGURE 3.10: Identifying common substrings in [61]

called extended blocks. Because this blocks have unique successors, they have unique exit points, which may be made into return statements in the extracted mini-subroutines. The sequences to be extracted are not restricted to basic blocks. Conditional branches are allowed between the start of the block and the inserted return statement. Because subsequences vary significantly between programs, the dictionary was static.

The process of code compression consists of three phases: dictionary entry generation, substitution, and dictionary generation:

- Dictionary entry generation: The instruction stream is first divided into basic blocks, and then each block is compared with every other block, as well as itself, for common substrings, which has $O(n^2)$ worst case running time. A threshold on the minimum length T (for example, 3) of substrings is prescribed, so that only potentially beneficial substrings are extracted. A simple algorithm is used to find common substrings. The operation of the algorithm is illustrated in Fig. 3.10. The two blocks are placed against each other with every possible region of overlap, beginning with the first T instructions of the first block and the last T instructions of the second. The matching substring or substrings in this overlapping region are identified and stored in a table. The second block is then shifted to the right by one instruction, and the process is repeated until the last T instructions of the first block are reached.

- Substitution: After dictionary entries are generated, they have to be sorted according to their length then the occurrence of each entry in the instruction stream has

to be replaced by a symbolic pointer, beginning with the longest entry. The exact location and usage of the dictionary entries are not known until after the dictionary generation phase.

- Dictionary generation: A dictionary entry i can be subsumed by another entry j if it is a suffix of entry j. In other words, entry i has to be removed from the dictionary because it is effectively available through entry j and this can make the dictionary smaller. In the instruction stream, symbolic pointers that point to j entries in dictionary are replaced by the appropriate instructions (CALL or CALD with the correct arguments).

This compression technique has been applied on some benchmarks compiled for the TMS320C25 VLIW processor. Reductions averaging 12% were obtained including the dictionary overhead.

Lefurgy, Bird, Chen, and Mudge [28, 29]

Lefurgy et al [28, 29] investigated *Dictionary compression schemes* with fixed and variable length codewords. They analyzed a program and replaced common sequences of instructions with a single instruction codeword. A microprocessor executes the compressed instruction sequences by fetching codewords from the instruction memory, expanding them back to the original sequence of instructions in the decode stage, and issuing them to the execution stages.

Their algorithm is divided into 3 steps:

- Building the dictionary: By using greedy algorithm to quickly determine the dictionary entries and assign codeword to each one. Obviously, codewords with more bits can index a larger range of dictionary entries. The dictionary entries have been limited to sequences of instructions within a basic block and allowed branch instructions to branch to the codewords, but they may not branch within encoded sequences. Branches with offset fields have been left without compression and their offset fields are patched to point to compressed instructions.
- Replacing instructions with codewords: The greedy algorithm replaces them automatically.
- Encoding: Fixed-length codewords of size 16-bit have been used to enable fast decoding and also Variable-length codewords of size 4-bit, 8-bit, 12-bit, and 16-bit were tried.

General design for a compressed program processor is given in Fig. 3.11. Since the compressed program may contain both compressed and uncompressed instructions, there

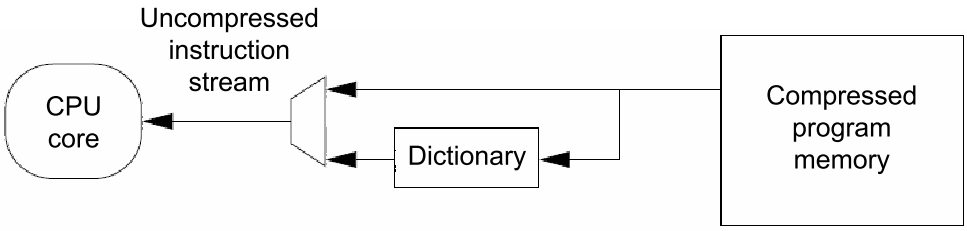


FIGURE 3.11: Overview of compressed program processor in [28]

are two paths from the program memory to the processor core. Uncompressed instructions proceed directly to the normal instruction decoder. Compressed instructions must first be translated using the dictionary before being decoded and executed in the usual manner.

Experiments were run on the PowerPC instruction set, compiled with gcc2.7.2. When variable-length codewords were used with more frequent encodings assigned shorter codewords, code reduction of 30% to 50% was achieved. This is non-selective approach that can result in code expansion for those instructions only occurring once. The same authors also presented a selective version in [30, 32].

Corliss, Lewis, and Roth [96, 97]

In [96, 97], instruction operand parameters have been deployed to catch a larger number of identical instruction sequences and replace them with codewords in the dictionary. Dynamic Instruction Stream Editing (DISE) generalizes the notion of an identical instruction sequence by extending the codeword with operand parameters. As a result, code sequences that only differ in the set of operands used can use the same codeword but with different parameters. In this way, DISE manages to use the same codeword and dictionary entry to similar, but not identical, sections of instructions. In particular, two instruction sequences that only differ in e.g. the choice of source operands, can use the same codeword with a set of parameters designating the source operands. Best compression ratio achieved was 65%.

Fraser [100]

Fraser [100] introduced an instruction which allows the direct interpretation and execution of programs compressed in a LZ77-like fashion. This instruction, called echo, repeats a sequence of instructions at a given offset and length from the current execution point. A difference from traditional *Dictionary-based compression* schemes is that no separate dictionary is used since echo instructions identify sequences inside the program. For examole, The assembler instruction:

echo .-5,3

commands the (hardware or software) interpreter to fetch and execute three instructions starting five bytes back from the echo instruction.

The semantics for an echo instruction is presented in the following pseudo-code:

1. Save the PC.
2. Subtract the contents of the echo instructions displacement field from the PC.
3. Set N to the contents of the echo instructions length field.
4. Fetch and execute the instruction at the address in the PC.
5. Decrement N and go back to Step 4 if the result exceeds zero.
6. Restore the PC and bump it past the echo instruction.

As a follow-on study, Lau et al. [99] considered a hardware implementation of the echo instruction, and also extended it by allowing bitmasks instead of length to allow the merger of similar, though not identical sections of code. The bitmask is used to generate all instruction sequences that are subset of the coded instruction sequence. By using Bitmask Echo, the authors achieved on average a compression ratio of about 85% for applications from the MediaBench suit compiled for the Alpha ISA.

Thuresson and Stenstrom [101]

Thuresson and Stenstrom [101] evaluated how much extended *Dictionary-based code compression techniques* can reduce the static code size. They evaluated two previously proposed schemes, DISE [96] and bitmask [99]. They also proposed a new scheme which is combination of these two previously proposed schemes. Their new scheme achieves additional compression by extending the baseline scheme with parameters allowing sequences of similar instructions to be represented using one codeword. It is an extension to DISE which designates one of its parameters as a bitmask instead of an operand parameter, creating a flexible framework with highly parameterizable dictionary entries. By adding information in the dictionary about how the operands should be interpreted, all three operands can be used as operand parameters when preferred. In their proposed codeword format, shown in Fig. 3.12, the OP-code triggers the replacement; P1, P2, and P3 are parameters that can be used as operands in the codewords, and ID identifies the dictionary entry in the replacement table. BM is the bitmask used to cancel out instructions in the dictionary entry. The cost of the dictionary entry is the number of instructions it holds.

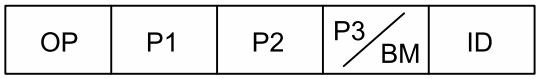


FIGURE 3.12: Proposed codeword format for the extended dictionary-based compression scheme in [101]

Thuresson and Stenstrom used programs from the MediaBench suite for evaluation and found that by comparing to a baseline *Dictionary-based code compression scheme*, bitmask-enabled codewords results in 5% better compression ratio while operand-enabled codewords achieve about 20% improvement. Their proposed framework and algorithm for combining both schemes achieved at least the same reduction in code size as operand parameters, but enabled more efficient coding of the dictionary allowing for a more efficient implementation.

Lekatsas and Wolf [64]

Lekatsas and Wolf [64, 65] proposed a new compression approach called Semi-Adaptive Dictionary Compression (SADC). In this encoding scheme, frequency tables are drawn up of all opcodes, as well as all groups of 2 and 3 consecutive opcodes. The compression then continues by taking combinations with the largest frequencies, and replacing them with new augmented instructions. For example:

LD $R1, 10($R2)
ADD $R3, $R4, $R5
becomes: LDADD $R1, 10($R2), $R3, $R4, $R5

This way, combinations of frequent consecutive opcodes or frequent opcode-register or immediate combinations are allocated one dictionary codeword. This process is a repetitive one, that encodes the program once the most frequent combination is found, and then gathers statistics again for the next pass, using the new encoded program. This is repeated until the dictionary is full.

SAMC is targeted for instruction sets with fixed-sized instructions and can work for any such architecture. Decompressed engine has been placed between the cache and CPU. CPU core has been left intact and all necessary decoding takes place in an add-on module. The compressor goes through the program once and compresses all instructions using arithmetic coding except for branches. Branches are packed into 2, 3 or 4 bytes depending on the offset size, and the offsets are patched to point to compressed addresses. Jump with registers used for return from subroutines need not to be changed and can be compressed using arithmetic coding.

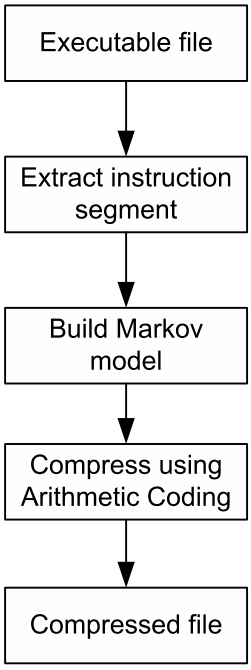


FIGURE 3.13: Encoding process of SAMC [101]

Fig. 3.13 shows a flow diagram for the encoding process of SAMC. Encoding table will be generated using arithmetic coding in combination with Markov model which is adapted to the instruction set and the application. The compression works on a bit-by-bit basis and the generated table has to be stored in main memory.

Compression averages for SADC came in at 50% across MIPS benchmarks tested and 65% across x86 benchmarks.

Clausen, Schultz, Consel, and Muller [67]

Clausen et al [67] used a similar macro-building compression technique. Their area of research was on embedded version of Java for Systems where RAM memory was a small as 6K. Here Java classes had been stripped of much extraneous information not necessary for the embedded market. The authors extended the instruction set of the JVM. First, each instruction was formed into group of length 1. Then groups were expanded by splitting. Bytecode switch, jump subroutine and return statements were considered unfactorizeable. Macros were formed by greedily selecting the group that provides the greatest code size savings, until no more unused instruction codes or groups exist. The average compression ratio for bytecode size was 79%, and 84% when the dictionary size was counted.

Lin, Xie, and Wolf [81, 82]

Lin et al. [81, 82] proposed a variable-size code compression method based on LZW

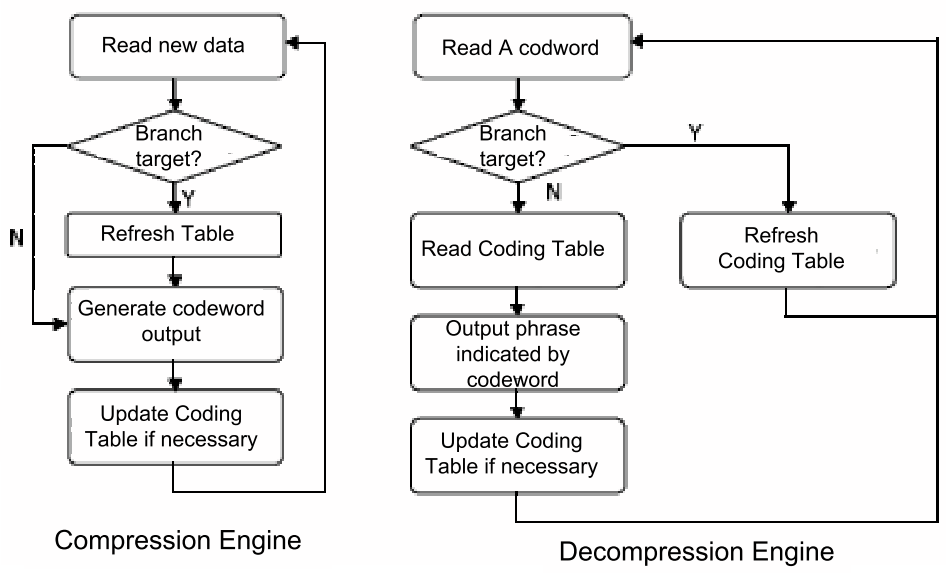


FIGURE 3.14: Flowchart of compression and decompression methods [81]

for VLIW processor. Their method generates adaptive coding table on-the-fly during compression and decompression to avoid storing it in the memory.

During compression phase, the compressor finds the longest phrase in the table, sends the codeword to the output, and adds the phrase with the next byte as a new entry. Once the table is full, the compressor keeps on using the existing table to compress upcoming data. When a codeword is read from the memory, they check if it is a branch target. If yes, the engine shifts out the padding from buffer, resets coding table, and restarts decompression at a byte-aligned position. Otherwise, the decompression core gets a codeword, looks it up, outputs the content, and adds the old phrase with the first element of next phrase as a new entry. Fig. 3.14 shows the flow chart of compression and de-compression methods. In both compression and decompression, the coding table is reset if the incoming address is a branch target; otherwise, the table will be updated when necessary. Execution flow might change and the target address for branch or jump is computed during runtime; however, locations of possible targets are determined once the code compiled.

Lin et al. achieved an average compression ratio of 75,2% targeting TMS320C6x VLIW processor.

Das, Kumar, and Chakrabarti [104, 105]

In [104, 105] the authors developed a *Dictionary-based algorithm* that utilizes unused encoding space in the ISA of RISC processors to encode codewords and address issues arising from variable-length instructions. This dictionary based method decompresses code at decode time. The frequently occurring bunches of instructions are encoded by a trap instruction with a modified unused field.

A compression reduction of 10%-30% is achieved.

There are several related approaches that use *Statistical compression method.*

Wolfe and Chanin [19]
Wolfe and Chanin [19] designed new RISC architecture called CCRP (Code Compressed RISC Processor) which has an instruction cache that is modified to run with compressed programs. At compile-time the cache line bytes are Huffman encoded in the memory. At run-time cache lines are fetched from main memory, uncompressed, and put in the instruction cache. Instructions fetched from the cache have the same addresses as in the uncompressed program. Therefore, the core of the processor does not need modification to support compression.

The mapping created by the compressor for translating branch addresses was called LAT (Line Address Table). It is only mapped to the start of blocks. Therefore, several instructions may have to be decompressed from the block before the target instruction was finally found. The most recently used LAT entries stored in the special cache called CLB (Cached Lookaside Buffer). Each LAT entry has a 24-bit base address followed by 8 entries indicating the length of the next compressed block. The LAT added 3.125% to the size programs.

The authors compressed the programs byte-wise by assigning shorter variable-length codewords to the most frequent bytes. Two Huffman[58]-based encoding schemes were used including bounded Huffman and preselected Huffman. They targeted MIPS2000 architecture and achieved compression ratios between 65% and 75%.

A follow up study has been done by Kozuch and Wolfe [59]. They compared the compression of 15 programs from the SPEC benchmark suite targeting the VAX, MIPS R4000, SUN 68020, SPARC, IBM RS6000, and Motorola MPC603 architectures. They found that statically compiled programs varied in size considerably from 2.7x to 4.9x for the MPC603 programs. In order to determine whether less dense code was more compressible, the zeroth and the first order entropy of the programs on each architecture was calculated. The results showed that the MIPS instruction set is much more compressible than other instruction sets using zeroth order compression. Furthermore, first order compression may achieve substantial compression improvement (almost 10% average compression ratio improvement).

Witten, Neal, and Cleary [95]

One of the most widely used forms of statistical encoding is a technique known as Arithmetic Coding. First introduced in [94] although formalized for application to data compression in [95].

Lekatsas et al [23, 24] combined arithmetic coding with Markov models. They separated types of instructions into four groups, each group had a short prefix to identify it. The prefix are "0" for the instructions with immediate, "11" for branches, "100" for fast dictionary instructions, and "101" for uncompressed instructions. Group 1 instructions were compressed using the Markov model and arithmetic coding. Group2 were compressed by rewriting them in a form without unnecessary bits. Group 3 instructions were looked up in a 256 entry table. The phases of compression were as following: Phase 1 of compression made a pass to build the Markov model. Phase 2 compressed group 1 instructions. Phase 3 compressed branches only. Phase 4 patched the branch offsets that have been marked in the two previous phases. Overall compression ratios between 52% and 56% were achieved, considering code only.

Xie, Wolf, and Lekatsas [75]

Xie et al [75] used the reduced-precision Arithmetic Coding technique by dividing each fetch packet and dividing it into sub-blocks. These sub-blocks are compressed by applying reduced-precision Arithmetic Coding with individual (vertical compression) and common (horizontal compression) statistical models. This breaking down into sub-blocks is done to improve the speed up due to the parallel nature of being able to decode more than one sub-block at once. This is seen as a trade-off between decompression speed and compression ratio [76], seeing as too many sub-blocks will reduce the ability to compress well, but too few sub-blocks means the decoding process will take longer. They analyzed this trade-off by comparing similar system with 4-byte, 8-byte and 16-byte sub-blocks. Increasing the block size decreases the compression ratio, but also increases the time taken to decompression. The 16-byte sub-block scheme yields the best compression ratios at 67% - 69% but processing 11.2 - 11.5 bits per clock cycle; whilst the 4-byte sub-block scheme although processing 47.01 - 47.42 bits per clock cycle has a compression ratio of 76% - 80%.

Xie et al [74, 78, 80] also presented a class of code compression techniques called variable-to-fixed code compression (V2FCC), which uses variable-to-fixed coding schemes based on either Tunstall coding or arithmetic coding. The use of variable-to-fixed encoding means that codewords are arbitrarily assigned and this assignment can be used to an advantage to reduce the number of bit toggles on the instruction bus. Experimental results for a VLIW embedded processor TMS320C6x showed that the compression ratios using memoryless V2FCC and Markov V2FCC were around 82.5% and 70%, respectively.

In [77, 79], the same authors used the static model in which the compression for all applications is independent from distribution of bits in the instruction word. They reported

an average compression ratio of 82% with one "Fetch Packet" decoding in each clock cycle.

3.2.2 Classification by ISA-Dependability

Several related approaches belong to the ISA dependent category.
Two commercial proposals for such methods are ARM Thumb [56] and MIPS16 [55]. Both of which feature 16-bit instructions based on 32-bit instruction set. In each system, the processor code expands the 16-bit instruction to 32-bit just after they are fetched from instruction memory. Instruction chosen for the Thumb set were selected due to their frequency of use, importance for generating small code or lack of need for full 32 bits. Instructions for the MIPS16 set were selected by analyzing a number of applications to determine the most frequently generated instructions. In order to reduce the number of instruction bits to 16, the number of registers that can be referenced was decreased to 8, and the size of immediate fields was shrunk. The shortened instruction set is not capable of generating complete programs; special instructions are used to switch between 16-bit and 32-bit instruction modes. ARM programs compiled with Thumb support have code sizes about 30% less than when they are compiled for 32-bit instructions alone. Similarly, code sizes in programs produced with MIPS16 support are about 40% smaller than for programs using only 32-bit MIPS instruction set [28]. This is referring to 70% and 60% compression ratios for ARM Thumb and MIPS16, respectively, but running 15%-20% slower than programs using standard RISC instruction set.

Larin and Conte [60]
Larin and Conte [60] conducted a comparison between code compression methods and a tailored encoding of the Instruction Set Architecture. They used TEPIC architecture which is 40-bit VLIW processor. Huffman coding was compared for 1 byte symbols, 4-byte stream and the whole 40-bit words. Streams were formed by breaking each instruction up into 4 separate parts with the first 9-bit, second 12-bit, third 14-bit and the last 5-bit. Each stream was compressed separately and decompressed in parallel. In the tailored ISA method, instructions were compacted into the smallest number of bits required to still represent the same information. This was done by removing the unnecessary bits from the instructions. Code compression ratios of stream and byte encoding were achieved between 70% and 78%. Tailored encoding achieved better compression ratios which were between 60% and 67%. These compression ratios did not include the Address Translation Table required to maintain branch target information.

Okuma, Tomiyama, Inoue, Fajar, and Yasuura [106]
An instruction encoding technique has been proposed in [106]. Okuma et al. noted that

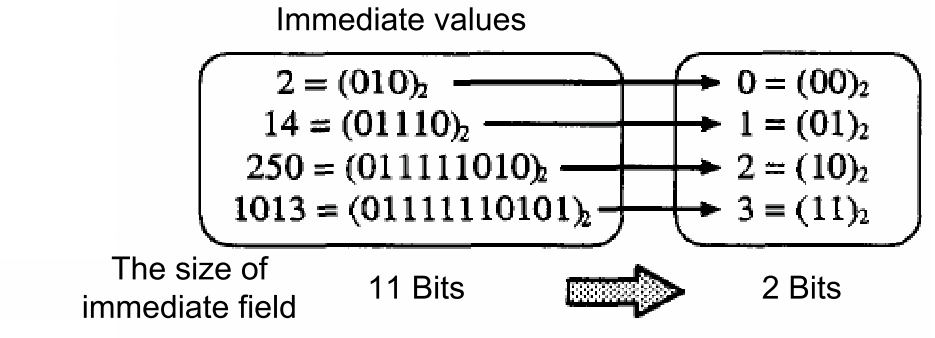


FIGURE 3.15: Encoding of immediate values in [106]

there is no application that all values which can be represented by the immediate field are used, therefore they proposed their instruction encoding techniques, that re-encodes the immediate field of an instruction with special regard to a minimum instruction-word length using a look-up table based immediate encoding approach.

The set of immediate values which appear in the program is fixed in the stage of the system design. Thus these immediate values can be encoded and the length of the field of immediate values can be reduced dramatically. Fig. 3.15 shows an example of the encoding techniques. In this example, there are four immediate values. The maximum value of the bit width in the immediate values determines the size of an immediate field, which is 11 bits here. When these values are encoded into codes from 0 to 3, the size of the immediate field is 2 bits. Therefore, the instruction word length can be reduced by at most 9 bits. Okuma et al. proposed several encoding technique to reduce the instruction word length, and used ROMs to implement the decoders. Their technique requires complex decoder logic and furthermore restricts the maximum number of immediate values in the application.
The authors applied their encoding techniques to three embedded applications, ghostscript, mpeg2 decoder, and mpeg2 encoder. They used gcc-dlx as compiler which based GNU CC for DLX architecture. They reported an average overall memory reduction of 12.4% for the DLX processor including the cost of the ROM decoders.

Ishiura and Yamaguchi [110]
Ishiura and Yamaguchi [110] investigated new code compression based on *Statistical method* for VLIW processors. Their compression method is called Automatic Field Partitioning. They reduced the problem of compressing code to the problem of finding the field partitioning that yields the smallest compression ratio. A field partitioning is a way of dividing up the bits of an instruction such that if the instruction is defined as the bit stream B = $b_1 b_2 ... b_w$, then a field partitioning can be described as F = $\{f_1, f_2, ..., f_n\}$ where each f_i is a non-empty field that contains one or more of the bits in B, and each b_i must appear in one and only one of the fields.

Compression ratio of 60% was achieved by taking into account not only the program size, but also the sizes of the ROMs that would be required as part of decompression engine to decompress the fields. No information is included on the design of such engines, although it is clear that the decompression unit would not be able to be designed until the software was known, hence applicable only to processors for specific (known) applications.

Menon and Shankar [116]

In [116], the authors divided instructions into 11 groups based on the class of the instruction set architecture. Within a group, they partitioned the instruction into two 2-byte segments and created tables for each group. They achieved an average compression ratio of 70% with 3 bytes decompression each cycle.

Lin and Chung [117]

In [117], a new *Dictionary-based compression approach* is used which divides the instructions into opcodes and operands to reduce the redundancy and then extracts the sequences and store them in a dictionary. The key idea is to explore the relations between the current operand to be compressed with those already compressed. On an average, a 46% compression ratio was achieved for the *ARM* processor. The decoder in this approach is complex which can reduce the performance.

Several related approaches belong to the ISA independent category.

Aho, Centoducatte, Azevedo, and Pannain [68, 69]

Aho et al [68, 69] proposed a new compression technique based on expression trees. They introduced three encoding schemes based on symbols having varying degrees of connection to expression trees or parts thereof. An expression tree is defined as a set of instructions found in a given program such that certain criteria are met, which includes rules for blocks to be started at intuitive intervals, for example at branch locations and function entry points. This means that a program is divided up into many of these expression trees and then compressed depending on what codeword is assigned to that tree. They showed that for the SPEC standard programs [70], on average, 24% of all such expression trees in a given program were distinct and almost 76% of trees were repeats. In each of the three algorithms presented, the Huffman encoding scheme was used to make use of exponential distribution of expression trees. The three algorithms are Tree-Based Compression (TBC), Pattern-Based Compression (PBC), and Instruction-Based Compression (IBC). The compression ratios achieved for TBC, PBC and IBC were 60%, 61% and 53%, respectively.

Prakash and Sandeep [111, 112]

Prakash et al [111, 112] presented an ISA independent code compression scheme that divides instructions into two 16-bit halves. For each half, a dictionary is constructed that contains a choice set of vectors such that a majority of the vectors used throughout the program in that half of the instruction differ from one of the dictionary vectors by a Hamming distance of at most 1 (the Hamming distance between two vectors is the number of bits that are different). Each compressed instruction is then replaced by two codewords representing each half-instruction. These codewords are combination of the indexes into the relevant dictionaries as well as information about which bits are toggled. They reported compression ratios between 68,9% and 76% with 21 cycles to decode a "Fetch Packet".

Ros and Sutton [114, 115]

Ros and Sutton ([114] and [115]) developed ISA independent dictionary compression scheme based on victor Hamming distances for the complete 32-bit instructions. The encoding scheme is based on the appropriate selection of dictionary vectors such that all program vectors are at most a specified Hamming distance from a dictionary vector. Bit toggling information is used to accurately restore original code. The algorithm is divided into the four steps:

1. File Input and Dictionary Construction: The benchmark to be compressed is read in, one 32-bit vector at a time, and a frequency distribution of all the used vector space is constructed.

2. Reduced Dictionary Selection: The purpose of this pass is to select from the dictionary, a subset of vectors such that all original dictionary vectors are at most a set Hamming distance from any one of the reduced dictionary vectors.

3. Reduced Dictionary Fill and Codeword Assignment: The reduced dictionary is analyzed and filled with further vectors such that the bits required for the indexing of the reduced dictionary is unchanged.

4. Compression Application: The compression scheme is applied by converting each 32-bit vector into compressed code. The compressed code comprises a codeword, a set number of bits to denote the number of toggles and up to 7 sets of 5-bit toggle locations. An example of this is shown in Fig. 3.16.

In [114], Ros and Sutton could achieve compression ratios of 72.1% to 80.3% targeting TMS320C6x VLIW processor.

In [115], They improved their Hamming distance based code compression by using a post-compilation technique for the greedy reassignment of a general purpose scratch registers.

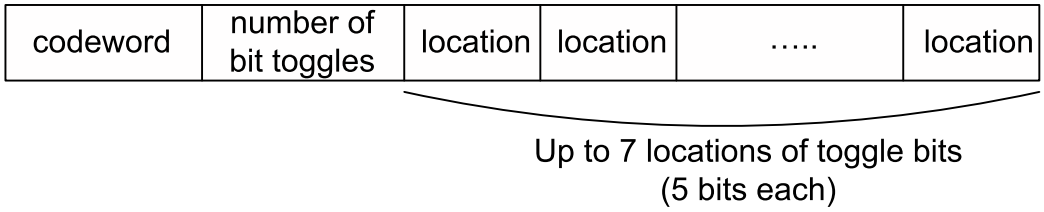


FIGURE 3.16: Format of Compressed Program Code in [114]

This technique improved the compression ratios by 3% to 4%.

Chapter 4

Code Compression for RISC Processors

By the late 1980s, every major workstation vendor had converted to one of the RISC CPU families. Largely due to the performance advantage, and in part due to the intrinsically lower cost of a simpler CPU, RISC also made rapid headway in the real-time sector.

As in most engineering decisions, RISC architecture involves certain trade-offs. RISC designs are easier to pipeline, and generally support a higher clock rate and higher performance, but they also generally require more instructions to do the same amount of useful work. This translates to a high instruction bandwidth requirement, which is usually satisfied by an instruction cache. Moreover, since all instructions are the same size, a certain amount of both program memory and instruction bandwidth is wasted for those simple instructions that could in theory be expressed in fewer bits. Indeed, in CISC achitectures, which have a variable instruction size, simple operations have a compact expression. Thus RISC processors have historically, and by and large correctly, had a reputation for having relatively poor code density.
For the manufacturers of workstations who were the first adopters of RISC CPUs, this code bloat was a small price to pay for the advantage in performance to be gained [55].

Code Compression is a solution to the code density and bandwidth issues for RISC design.

In all our compression techniques, Huffman Coding is used as compression algorithm to compress the program instruction code.
Huffman Coding [26] is a well known method based on probabilistic distribution. Applied to instruction compression, most frequent occurring instructions are encoded with the shortest code words. The code words are then used as indices to a decoding table which contains the original instructions.

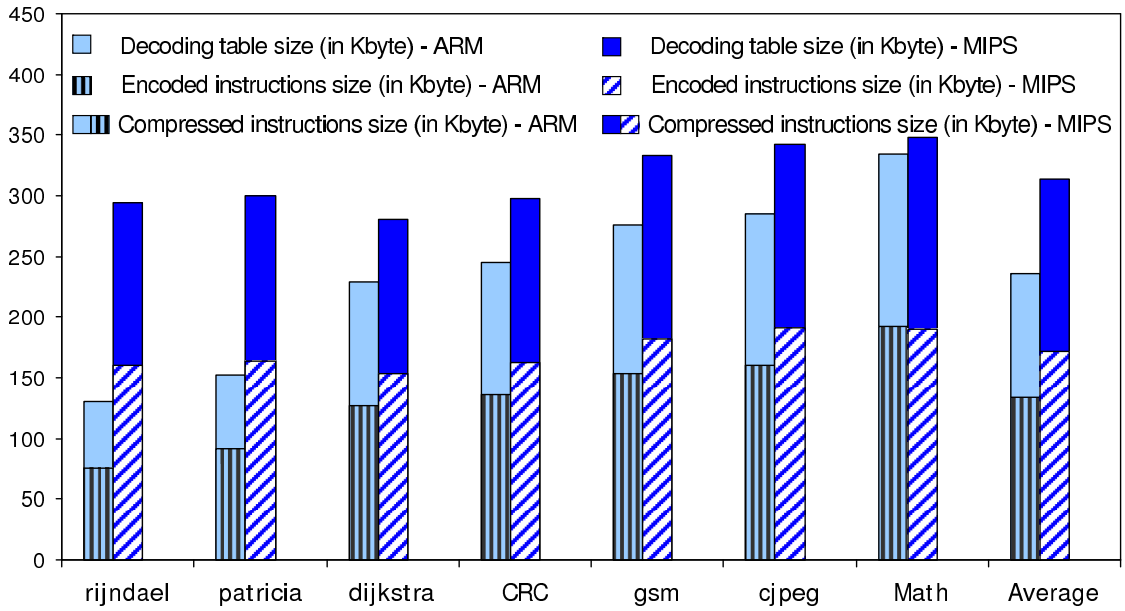


FIGURE 4.1: Comparing the size of the Huffman decoding table to the size of the Huffman compressed instructions for different benchmark programs and processor architectures.

Among all *Statistical* compression algorithms, Huffman Coding offers the best compression since it provably provides the shortest average codeword length [36]. Another advantageous property of a Huffman code is that it is prefix free; i.e. no codeword is the prefix of another one. This makes the decoding process simple and easy to implement (more details about Huffman Coding are given in Section 2.3.2.1). For these reasons, we selected Huffman Coding as a compression technique to encode the instruction object code. However, the size of the decoding table generated for decompression may be large and may negatively affect the final compression ratio (Eq. 1.3). Thus, it diminishes the advantages that have been obtained by compressing the instructions. Fig. 4.1 shows the size of the decoding table in comparison to the size of compressed instruction code (when Huffman Coding is used for compression) for different applications from MiBench [20]. The benchmarks are compiled for ARM(SA-110) and MIPS(4KC) processors. The bar labeled "Average" (rightmost bar) shows the average across all benchmarks. Figure 4.1 shows that the average size of the decoding table is more than 40% of the size of the compressed instructions (which includes the sizes of the decoding table and the encoded instructions), when Huffman Coding is used for compression. Hence, an efficient compression ratio can not be accomplished by only minimizing the encoded instructions but also the decoding table must be considered (Eq. 1.3).

In this chapter, we present three different Huffman-based code compression techniques for RISC processors. In all techniques, the sizes of encoded instructions and decoding table are decreased explicitly.

The first two techniques "Look-up Table Compression Technique" and "Instruction Splitting Technique" are ISA-Independent (Instruction Set Architecture Independent). They can be applied to any processor architecture independent from the instruction set format. In this thesis, the compression techniques are applied to three different RISC Processors, namely ARM, MIPS and PowerPC. The "Look-up Table Compression Technique" is applied to two compression schemes: the *Dictionary-based* compression scheme and the *Statistical* compression Scheme. Both schemes are presented in the Sections 4.1.1 and 4.1.2, respectively. While the "Instruction Splitting Technique" is presented in Section 4.1.3.

The third compression technique "Instruction Re-encoding Technique" is ISA-Dependent. It results in better compression ratios in comparison to the ISA-Independent techniques as it is applied to a specific processor architectures. In this thesis, the ISA-Dependent compression technique is applied to two processor architectures, namely MIPS and ARM, and is presented in Section 4.2.

This chapter also discusses the dependability of the compression technique on the instruction set architectures and its impact on the compression ratio and the performance of the hardware decoder (Section 4.3).
A comparison of our work to the previous work on code compression for RISC processors is presented at the end of this chapter (Section 4.4).

The three code compression techniques are published in [1, 2, 3, 4, 7, 8].

4.1 ISA-Independent Compression Techniques

The ISA-Independent code compression techniques are entirely orthogonal to approaches that take particularities of a certain instruction set architecture into account. That means the achieved total compression ratio could be further improved if the ISA-specific knowledge were used on top of our technique (as explained in Section 4.2).
The ISA-Independent code compression techniques follow the traditional data compression schemes, which depends only on the statistics of instructions or part of them.

Two ISA-Independent Huffman-based compression techniques "Look-up Table Compression Technique" and "Instruction Splitting Technique" are used to to reduce the size of the compressed instructions which includes the size of encoded instructions and the size of decoding table (see Fig. 1.2). To show the efficiency of the "Look-up Table Compression Technique", we apply it to two compression schemes *Dictionary-based* compression scheme and *Statistical* compression scheme. Both schemes are presented in the Sections

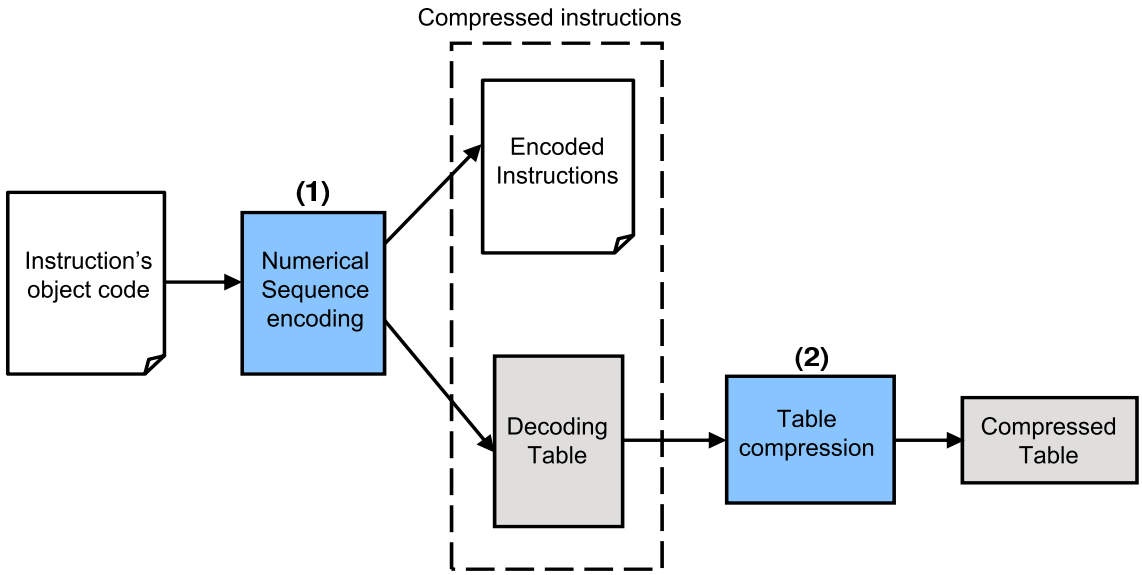


FIGURE 4.2: Compression steps for the Dictionary-based Scheme

4.1.1 and 4.1.2, respectively. The "Instruction Splitting Technique" is presented in Section 4.1.3.

The concept of compressing a Look-up Table in *Statistical* schemes first appeared in our previous work [1]. Afterwards, the work wa generalized by using it along with *Dictionary-based* and *Statistical* compression schemes, and published in [2] and [7]. This work was also published in a book chapter [8].

4.1.1 Look-up Table Compression Technique for Dictionary-based Compression Schemes

In this section a novel, hardware-supported code compression technique is introduced. Besides the encoded instructions, also the Look-up Tables are compressed, that can become significant in size if the application is large and/or high compression is desired. The evaluations are conducted using a representative set of applications from MiBench [20] and are built for three major embedded processor architectures, namely ARM, MIPS and PowerPC.

In order to demonstrate the usefulness of our Look-up Table compression technique, we deploy it in conjunction with *Yoshida's Technique* [27] that uses a *Dictionary-based* scheme to generate the Look-up Table.
In our scheme we conduct the following steps for compression (see Fig. 4.2):
(1) The object code of instructions is encoded (sequentially as a whole) with a fixed-length

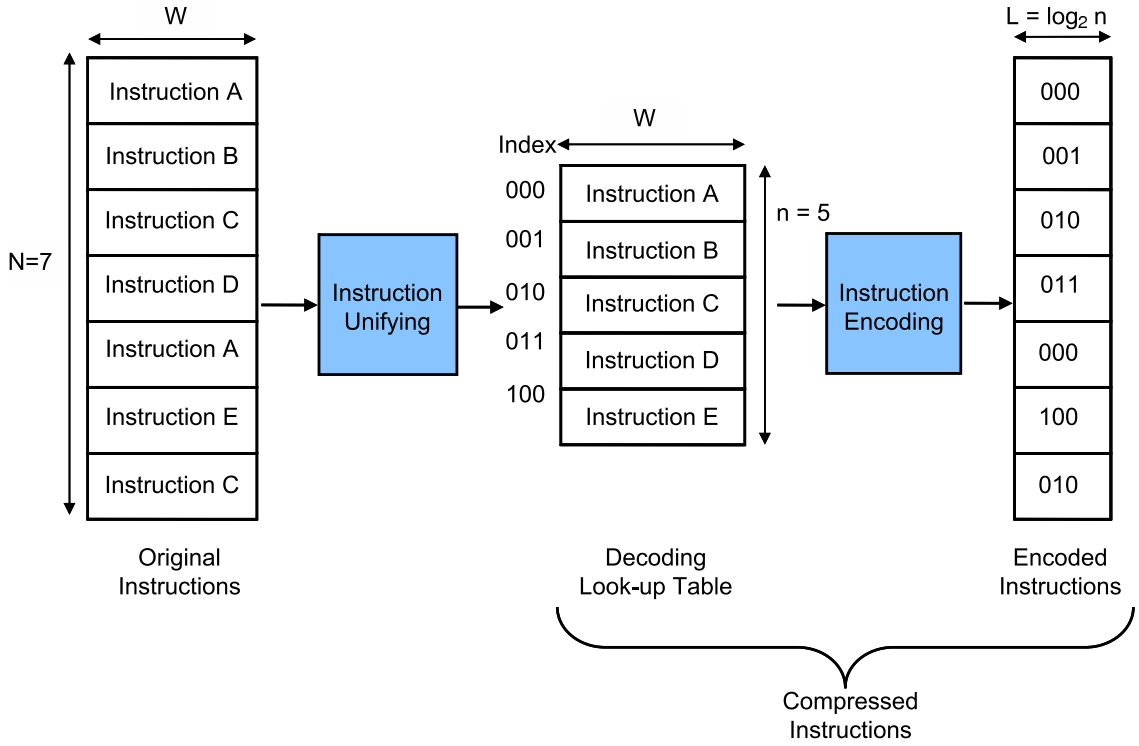


FIGURE 4.3: Example for generating compressed instructions in the Dictionary-based Scheme

code using the numerical sequence encoding technique (explained in Section Fig. 4.1.1.1). The unique original instructions[1] are stored in a Look-up Table.
(2) The Look-up Table is then compressed using our table compression technique.

To solve the problem of locating the branch target addresses in memory, we patch these addresses to the compressed ones as adopted from [28] (details are presented in Section 2.1). As the encoded instructions have a fixed length, we do not need to align the first instruction of the branch target address at an addressable boundary. This is because our hardware decoder can compute the address of an encoded instruction and access it even if it is not aligned to memory border. Consequently, there is no branch-penalty caused by the code compression in this scheme.
The following subsections explain the compression steps in detail.

4.1.1.1 Generating the Compressed Instructions (Encoded Instructions and Decoding Table)

To generate the encoded instructions and the Look-up Table, we first unify the original (i.e. uncompressed) instruction words. For that, we extract the unique instruction words

[1] A unique instruction is a 32-bit word including operator(s), operands etc. that is unique in its bit pattern.

and store them in one Look-up Table. In the original code, we encode all unique instruction words as a whole by using the numerical sequence encoding technique (as used in [27]). In this encoding technique, every unique instruction word is replaced with a binary index to the Look-up Table in ascending order. There, the index has a fixed length equal to $\log_2$ number of unique instruction words.
Fig. 4.3 illustrates, by means of a simple example, how the Look-up Table and the encoded instructions are generated. There, the number of original instructions is N = 7, the number of unique instructions is n = 5, and $index\_length = encoded\_instruction\_length = \log_2(5) = 3$ bits. The compression ratio is computed as follows:

$size(original\_instructions) = W \times N$
$size(encoded\_instructions) = N \times \log_2(n)$
$size(decoding\_table) = size(table's\ columns) = \sum_{i=1}^{W} C_i$

By substituting these terms in Eq. 1.3, we get the compression ratio for the *Dictionary-based* schemes.

$$CR_{dictionary} = \frac{N \times \log_2(n) + \sum_{i=1}^{W} C_i}{W \times N} \tag{4.1}$$

W: Number of table columns (Instruction word length)
N: Number of original instructions
n: Number of unique instructions (number of table entries)
C_i: Size of table column i (in bit)

Obviously, in order to improve the compression ratio by decreasing the table size, either the number of table columns (W) or the size of the table columns (C_i) need to be decreased. Note that (W) is fixed. That leaves (C_i) as an option. Decreasing the size of the table columns (C_i) to compress the decoding table is explained in the next section.

4.1.1.2 Compressing the Decoding Table

As explained in the Section 4.1.1.1, the original (i.e. uncompressed) unique instruction words are stored in a Look-up Table. The number of table columns is equal to the instruction word length in bits (W). If the instruction word length, for instance, is 32 bits, the number of table columns is 32. Minimizing the decoding table cost can be achieved by reducing the size (in bits) of the table columns as shown in Eq. 4.1. The principle of compressing the table is to minimize the number of bit transitions per column and then saving only the indices where a bit toggle occurs instead of saving the complete

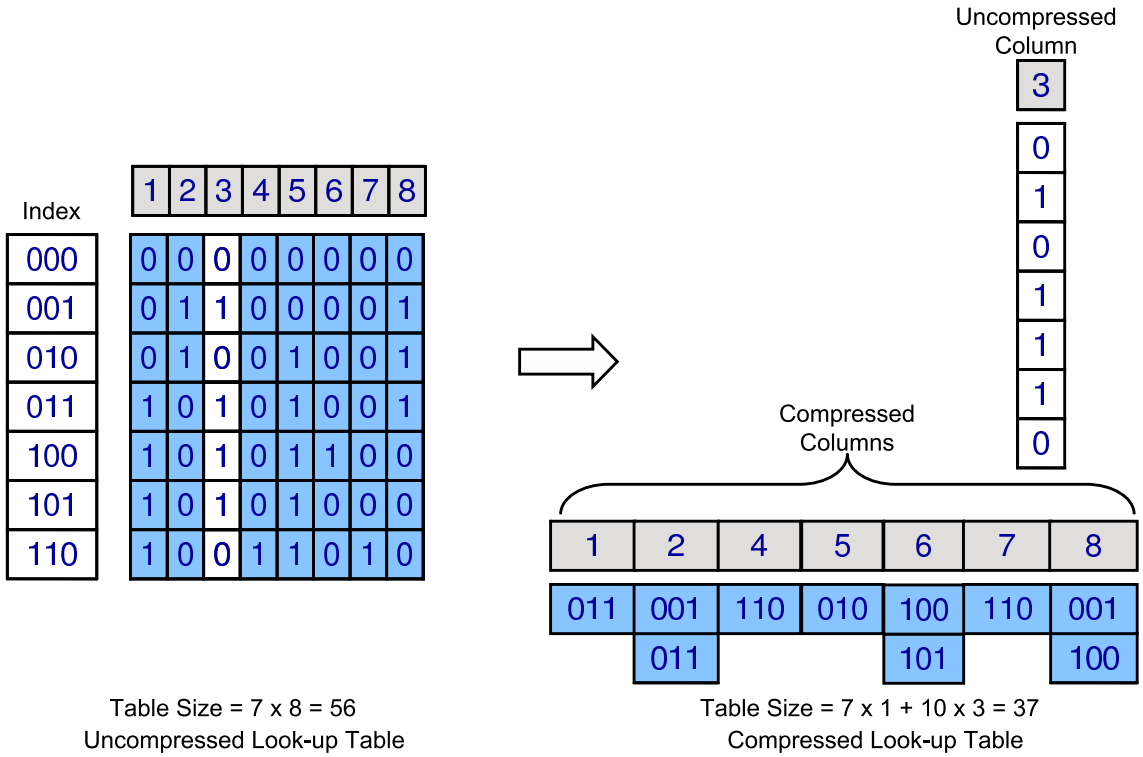


FIGURE 4.4: Simple example for compressing table with 8 bits symbols

column. Fig. 4.4 shows an example of compressing a Look-up Table with 7 symbols and 8 bits each. The size of the table (before compression) is *56* bits. The number of unique instructions in the table is $8 \Rightarrow$ index length $= 3$ bits. Hence, the column can be compressed if it has a maximum of *2* transitions (because more than *2* transitions will need more than *7* bits which was the cost of the uncompressed column). In that case, *7* columns will be compressed and one column will be left without compression. The size of the table after compression is minimized to *37* bits (from *56* bits before).

To compress the Look-up Table and compute its cost, we use the function "*Decoding Table Compression*" (DTC) in Algorithm 4: Starting from the first column, the function counts the number of bit toggles (T) that occur in this column (lines 10-13). Then it computes the cost (C) of compressing this column (line 14). The column cost is the sum of table indices where bit toggles occur in this column. If the sum is less than the original column cost (n), then the column can be compressed (lines 15-17). Otherwise, the column is left without compression. These steps are repeated for all table columns (W). Finally, the function returns the cost of the compressed table (line 23).
The algorithm DTC assumes that the toggle starts from 0 or 1 depending on the first entry of all columns. If the number of zeros in the first entry of all columns is more than the number of ones, the algorithm assumes that the toggle starts from 0 (i.e. if the first entry of a column is 1, it is considered as a toggle). Otherwise, the toggle starts from 1.

Algorithm 4 DTC: Decoding Table Compression

```
{ n: Number of table entries}
{n0: Number of zeros in first table entry}
{n1: Number of ones in first table entry}
{W: Width of Table (Number of table columns}
{ L: Length of table index}

Function DTC (n, entry, L, cost) {
/* Initialization */
Number of Toggles T = 0, Column cost C = 0
if n0 > n1 then
   toggle starts from 0
else
   toggle starts from 1
end if
/* Algorithm Start */
for all columns i of W do
   for all entries j of n do
      if Toggle(j) [ 0 →1 or 1→0 ] is true then
         T(i) = T(i) + 1 {Count the toggles}
      end if
   end for
   cost of column i is C(i) = T(i) × L
   /* check if the column is compressible */
   if n > C(i) then {Compress the column}
      i. save the index at every toggle
      ii. cost = cost + C(i)
   else
      i. keep column(i) without compression
      ii. cost = cost + n
   end if
end for
return(cost)
}
```

Achieving higher table compression ratio depends on the way of sorting its entries. Finding the optimum solution of sorting the entries is NP complete. Testing every possibility for sorting the (n) entries would require $n!$ comparisons. Therefore, we sort the entries in two phases; In the first phase, we generate *Gray Code* for (W) bits (i.e. the table width), then we locate each table entry (after converting it to decimal) in its corresponding position in the generated *Gray Code*. In this case, the number of transitions between each table entry is minimized and more table columns can be compressed. If a table (with W bit width) contains 2^W different entries (i.e. all the possible combinations of W bit), then *Gray Code* can give the optimal solution of sorting table entries (i.e. number of transition between any two successive instructions is 1). Since the number of table entries is much smaller than 2^W, this sorting phase does not provide an optimal solution. We therefore

Algorithm 5 TES: Table Entries Sorting

{n: Number of table entries}
{W: Length of table entry}
{$S_1, S_2, \ldots, S_n$: Entries of table}

1: Distance = 0
2: New_Distance = 100000 {Big number as initial value}
3: Min_Distance = 1000 {Minimum distance defined}
/* Algorithm Start */
4: **call** sort($S_1, S_2, \ldots, S_n$) {Sort entries using Lin-Kernighan sorting}
5: **for all** entries i of n **do** {Compute distance}
6: A = S_i XOR S_{i+1}
7: $D_i = \sum_{j=1}^{W} A_j$ {j is index for bit position in A}
8: $Distance = Distance + D_i$
9: **end for**
10: **if** $Distance < New\_Distance$ **then**
11: $New\_Distance = Distance$
12: **if** $New\_Distance <= Min\_Distance$ **then**
13: goto 20
14: **else**
15: goto 4 {Repeat for better solution}
16: **end if**
17: **else** {Could not find better solution}
18: goto 4 {Repeat to find better solution}
19: **end if**
20: **return**($S'_1, S'_2, \ldots, S'_n$) {Return sorted entries}

use the second sorting phase "*TES: Table Entries Sorting*" (Algorithm 5). It uses *Lin-Kernighan* heuristic sorting (LK) [33] which generates an optimal or near-optimal solution for the symmetric traveling salesman problem (*TSP*). The *TES* algorithm is used to sort the table entries such that the sum of the distances between each two successive entries from the first one (the top of the table) to the last one (the bottom) is minimal. In our case, the distance between two entries is the *Hamming Distance* which is the sum of bit toggles between these entries (lines 5-9). This may be computed using an XOR gate. At the end, the algorithm returns the sorted entries. The complexity of sorting the table entries in the *TES* algorithm is O(n·log n) where n is the number of the unique original instructions in the Look-up Table (i.e. entries of the Table) [34].

Fig. 4.5 illustrates an example for a Look-up Table (before and after sorting its entries using the *TES* algorithm). This figure (on the left) shows unsorted symbols and the distance between every 2 consecutive symbols. The sum of distances from the top of table to the bottom is 25. Using the *TES* algorithm, (Fig. 4.5, right) the sum of the distances is decreased to 14. Hence, if we compress the unsorted and the sorted tables using our *DTC* Function (in Algorithm 4), we find that their costs are 53 bits and 36 bits, respectively. Sorting the entries of the Look-up Table does not have an impact on the size of the

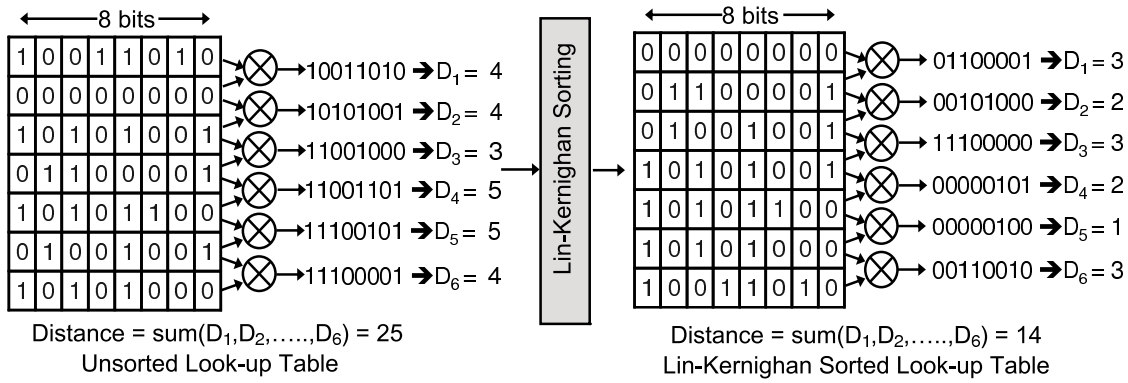


FIGURE 4.5: Example for sorting table with 8-bits symbols using Table Entries Sorting algorithm (TES)

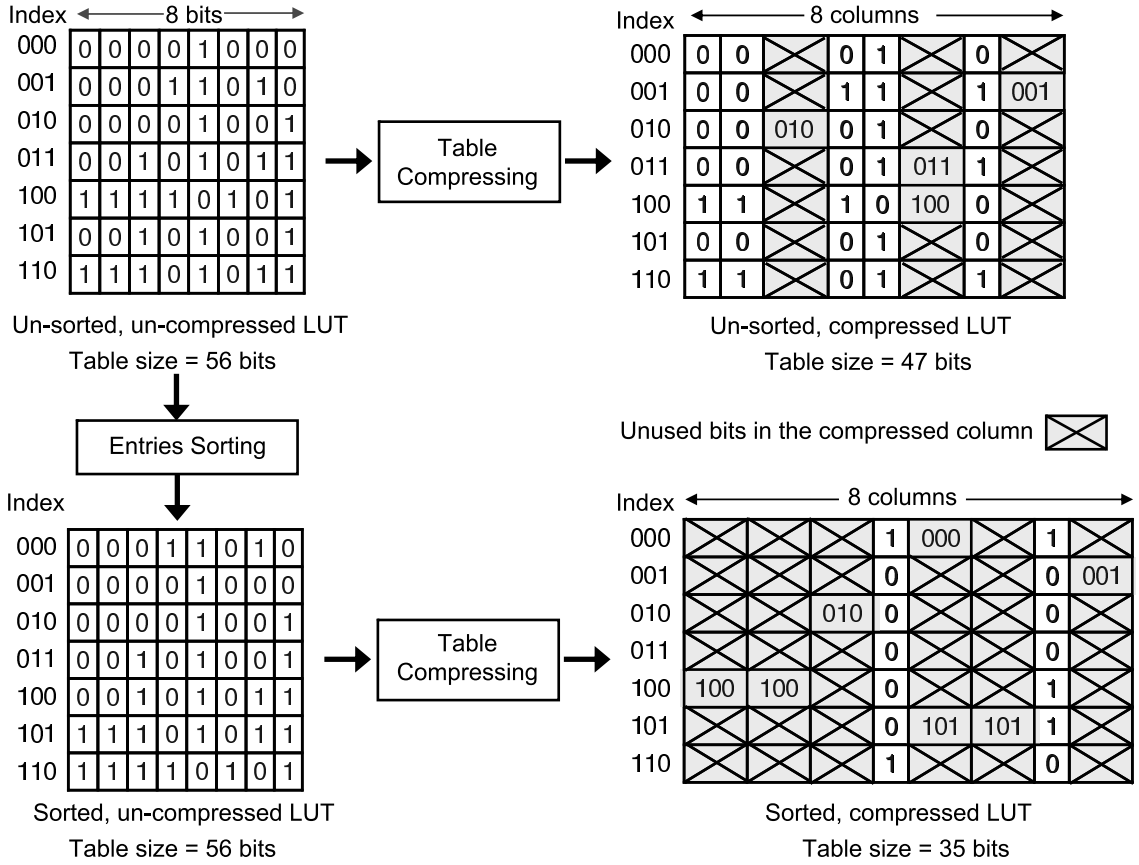


FIGURE 4.6: Sorted and Un-sorted Look-up Table (LUT) compression

encoded instructions because all instructions have the same code length (log_2 n). It will just decrease the size of the Look-up Table and consequently improve the compression ratio (Eq. 4.1).

To show the importance of sorting the table entries and how it improves the cost of the compressed table, we illustrate a demonstration for compressing a Look-up Table (LUT) with a number of entries $n = 7$ and instruction word length $W = 8$ (Fig. 4.6). The top

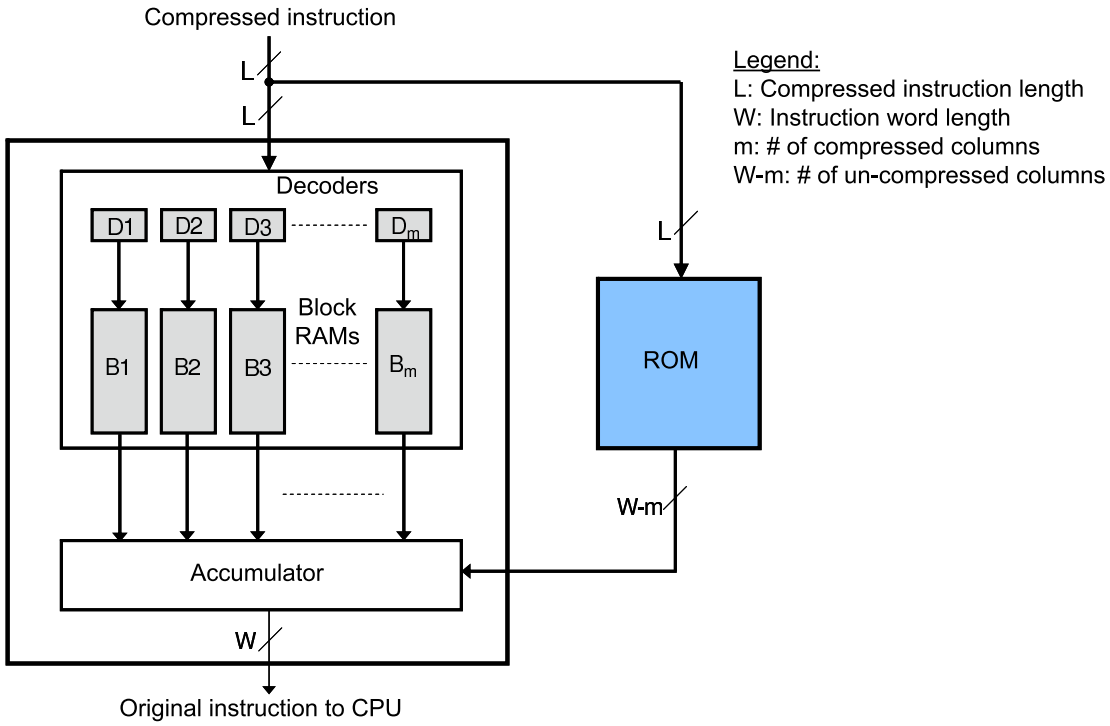


FIGURE 4.7: Look-up Table decoder

part of this figure shows that the size (cost) of the original table (without sorting its entries) is 56 bits. By compressing it using the Algorithm 4, its size becomes 47 bits. The bottom part of this figure shows the sorted table (using *Gray Code* followed by the Algorithm 5). After compressing this table, its size becomes 35 bits.

4.1.1.3 Hardware Implementation

The general decompression hardware consists mainly of two parts: Look-up Table decoder and Canonical Huffman decoder (See Section 4.1.2.4). As the Huffman Coding algorithm is not used in the *Dictionary-based* compression scheme, the decompression hardware in this scheme contains only the Look-up Table decoder part.

Fig. 4.7 shows our Look-up Table decoder. In this decoder, the compressed columns are stored in FPGA *Block RAMs*, one column in each *Block RAM*, while the uncompressed columns are stored in external ROM. Every compressed column has its own column decoder and can operate with the others in parallel. If the number of compressed columns is m, then the number of uncompressed columns in the external ROM is W-m. Each column decoder has information about the column position in the original Look-up Table and the number of toggles it has. The column decoder contains a comparator to scan the Block RAM entries until the compressed instruction is found. All of the column decoders operate asynchronous and receive the same encoded instruction (which has the length L) simultaneously. When the decoder receives the encoded instruction, it finds out its

position in each *Block RAM*. If it is in an even position, the decoder generates '0' for that position, otherwise it generates '1'. The original bits of uncompressed table columns are retrieved directly from the ROM. The accumulator concatenates the bits in the correct positions and generates the W bits decompressed instruction.
Returning to the example in Fig. 4.4: if the compressed instruction is located in the table at the entry "101", the ROM outputs the bit '1' to the accumulator for the bit at the position number 3. For the bits at the positions 1, 2, 4, 5, 6, 7 and 8, the controller finds that "101" is at the 1st. position, 2nd. position, 0th. position, 1st. position, 2nd. position, 0th. position, and 2nd. position of the block RAMs respectively. So the generated bits will be 1, 0, 0, 1, 0, 0 and 0, respectively (considering that '0' is generated fro the even position and '1' is generated for odd position). Finally, the decompressed instruction obtained is "10101000".
In the *Dictionary-based* scheme, the Look-up Table decoder has an additional task (not shown in the figure). It receives the uncompressed address from the CPU and then computes the compressed one of the encoded instruction in the memory. This can be done because the encoded instructions in the memory have a known fixed length. Hence, aligning the first instruction of the branch target address at an addressable boundary is not required. this eliminates the branch-penalty caused by code compression. The compressed address in the memory is computed as following:

$$\textit{Compressed address in memory} = \frac{\textit{uncompressed address from CPU}}{\textit{encoded instruction length in memory}} \tag{4.2}$$

We implemented the decoder in VHDL and synthesized it with Xilinx ISE8.1 for an FPGA prototyping board with VirtexII family devices "ChipIt Platinum Edition" [37]. The maximum frequency achieved for our Look-up Table decoder was 330 MHz (access time of 3 ns). The number of slices [38] needed for the decoder was 430.

The experimental results of the *Dictionary-based* compression scheme are presented with the results of the *Statistical* compression scheme in Section 4.1.2.5.

4.1.2 Look-up Table Compression Technique for Statistical Compression Schemes

To show the efficiency of our "Look-up Table Compression Technique", we also apply it to the *Statistical* compression schemes. In this type of scheme, we optimize the number and size of generated Look-up Tables to improve the compression ratio. The evaluations are conducted using a representative set of applications from MiBench [20] and are built for three major embedded processor architectures, namely ARM, MIPS and PowerPC.

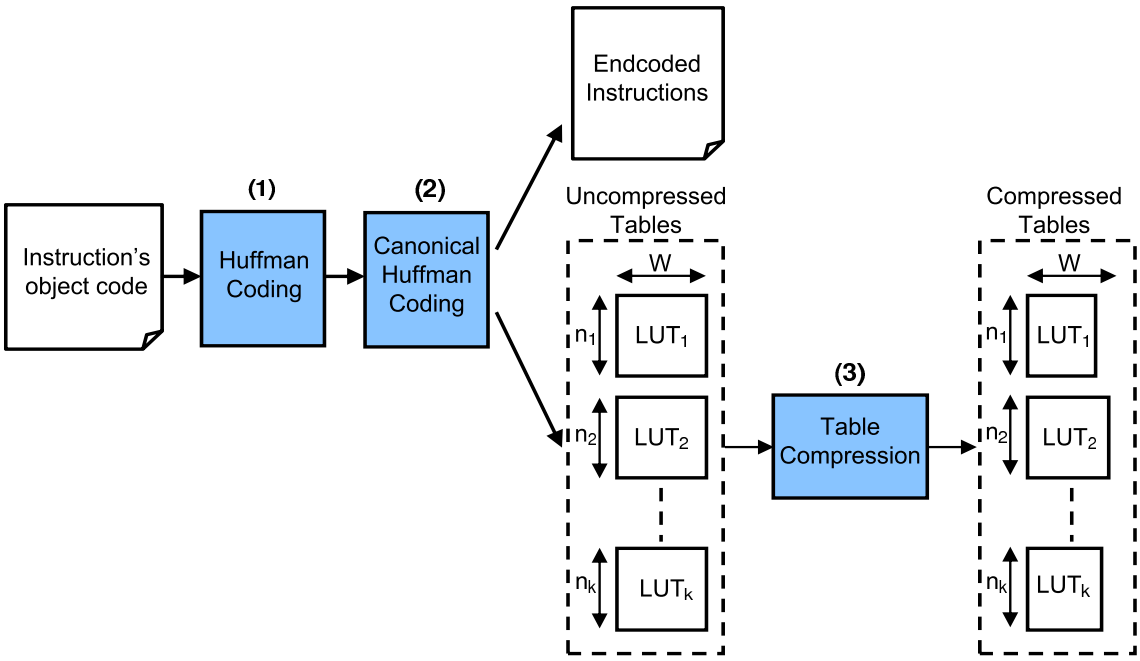


FIGURE 4.8: Compression steps for the Statistical compression scheme

The efficiency of this scheme (in terms of code size) depends on the frequency of occurrences of all unique instructions or sequences thereof.

In this scheme we conduct the following steps (see Fig. 4.8):

1. The object code of instructions is encoded (sequentially as a whole) with a variable-length code using Huffman Coding.
2. The Huffman-encoded instructions are re-encoded again using Canonical Huffman Coding to save space needed for the decoding tables. The unique original instructions are stored in different Look-up Tables (LUTs) depending on their encoded instruction length (one Look-up Table for each instruction length).
3. The Look-up Tables are compressed by sorting its entries using the *TES* (Algorithm 5), as used in the *Dictionary-based* scheme.

To solve the problem of locating the branch target addresses in memory, We align the first instruction of the branch target address at an addressable boundary and patch these addresses to the compressed ones as adopted from [28] (see Fig. 2.2). We store the succeeding instructions consecutively in memory. Alignment of branch target addresses on memory borders causes an overhead in the compressed code size of around 3% (see Fig. 4.20). The incurred overhead is already factored into the compression ratios we

report.
The following subsections explain in detail the compression steps.

4.1.2.1 Huffman Coding

In this step we encode the object code of the instructions using Huffman Coding.
Huffman Coding [26] is an entropy encoding algorithm based on the estimated probability of occurrence for a block of code (which can be one or a sequence of instructions). The most frequently occurring blocks are encoded with short codewords, whereas the less frequently occurring ones are encoded with large codewords. In this way, the average codeword length is minimized (more details about Huffman Coding are given in Section 2.3.2.1). It is obvious however that, if all distinct blocks in a code appear with the same (or nearly the same) frequency, then no compression can be achieved. Among all statistical codes, Huffman Coding is one of the best compression technique since it provably provides the shortest average codeword length [36]. However, the problem with Huffman Coding is the variable-length codes. This is a major problem when it comes to hardware implementation. In addition to that, the instructions are stored non-contiguously in Look-up Tables. This will take space in memory and will diminish the benefits which may be achieved using code compression. To overcome this problem we use Canonical Huffman Coding [35] (as explained in Section 4.1.2.2).

4.1.2.2 Canonical Huffman Coding

Canonical Huffman Coding is a subclass of Huffman Coding that has a *numerical sequence property*, i.e. codewords with the same length are binary representations of consecutive integers. Using Canonical Huffman Coding therefore allows for a space- and time-efficient decoding [39]. To encode the instructions using Canonical Huffman Coding, we first encode the instructions using Huffman Coding to find out the code length for every instruction and the frequency of every length. The instructions with the same Huffman code length are stored in one Look-up Table contiguously. We obtain as many Look-up Tables as we have different code lengths. Canonical Huffman codeword for any instruction in a Look-up Table is its index in that table. To find out the codewords, we just need to compute the first codeword in each table, i.e. for each code length (because the remaining codewords are computed by increasing the first codeword by one sequentially). The first codeword in the Look-up Table of longest code length is '0'. The first codeword in the Look-up Tables of other length are computed as follows:
The first codeword in a LUT of 'L' code length =
The last codeword in the LUT of 'L-1' code length + 1

Symbols	A	B	C	D	E	F	G	H
Distribution	3/28	1/28	2/28	5/28	5/28	1/28	1/28	10/28
Huffman Code	001	0000	1001	101	01	0001	1000	11
Canonical Code	010	0000	0001	011	10	0010	0011	11

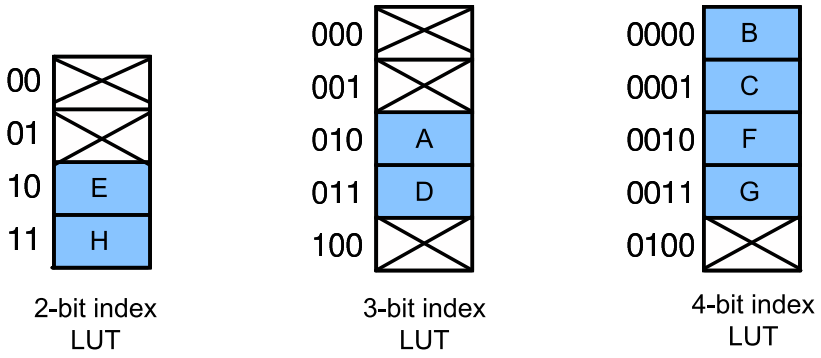


FIGURE 4.9: Look-up Tables generated from Canonical Huffman Coding

Fig. 4.9 illustrates an example for constructing Canonical Huffman codes from Huffman codes for given symbol probabilities (assuming that every symbol denotes a unique object instruction code).
Encoded instructions using Canonical Huffman Coding consume less space in memory because they are contiguously stored in the Look-up Tables.
We may compute the compression ratio in this scheme as follows:
$size(encoded\_instructions) = \sum_{i=1}^{L} N_i \times CL_i$

$size(decoding\_tables) = \sum_{i=1}^{L} \sum_{j=1}^{W} C_{ji}$

By substituting these terms in Eq. 1.3, we get the compression ratio for the *Statistical* schemes.

$$CR_{statistical} = \frac{\sum_{i=1}^{L} N_i \times CL_i + \sum_{i=1}^{L} \sum_{j=1}^{W} C_{ji}}{W \times N} \tag{4.3}$$

L: Number of different code lengths (number of different Look-up Tables)
W: Instruction word length (table width)
N_i: Number of instructions which have the code length i (frequency of code length i)
CL_i: The code length in table i
C_{ji}: The size of column j in table i
N: Number of all original instructions

If *L=1* (there is only one Look-up Table), then we will obtain the same compression ratio as in *Dictionary-based* scheme (Eq. 4.1).

4.1.2.3 Minimizing the Cost of the Look-up Tables

To minimize the cost (size in bit) of the Look-up Tables, we use two methods:

(1) Minimizing each Look-up Table size separately.
In this case, the instructions that belong to a given Look-up Table are sorted within this table and compressed using the scheme presented in Section 4.1.1.2. This will minimize the cost of the Look-up Tables and will have no impact on the encoded instruction size because the number of instructions which have the code length 'i' (i.e. N_i) will not be changed after the sorting. The cost of the Look-up Table is computed using the function *DTC* in Algorithm 4.

(2) Minimizing the cost of Look-up Tables all together.
That means, the instructions that belong to any Look-up Table may be transferred to a new Look-up Table if that will improve the final compression ratio. The instruction can only be transferred to a new Look-up Table if the index size of the new Look-up Table is longer than the index of its original Look-up Table (to maintain the Huffman prefix free property). Note that this process will decrease the number of Look-up Tables by deleting the instructions from some Tables and inserting them in another ones. This will give a better chance of compressing more columns in each table and consequently minimize the total compressed tables cost. On the other hand, this process is counterproductive for the encoded instructions size as they are generated using the (non-resorted) Look-up Table. For example, transferring the instruction 'H' in Fig. 4.9 from LUT2 (i.e. Look-up Table with 2-bits index) to the free entry in LUT4 (i.e. the index '0100') will increase the encoded instructions size by 2 times the frequency of this instruction (i.e. 2 x 10 = 20 bits).

If some instructions are transferred from one Look-up Table to another one, the Efficiency (E) is computed as follows:

$$E = \textit{compressed tables gain - encoded instructions loss} \tag{4.4}$$

such that, the *compressed tables gain* is the difference between the size of the compressed tables before and after transferring instructions between them. The *encoded instructions loss* is the difference between the size of the encoded instructions before and after transferring instructions. We found that transferring instructions from one Look-up Table to another one, in a way that the difference in the size of their indices is more than '1' bit, will increase the encoded instructions loss more than the compressed tables gain and consequently will negatively impact the compression ratio. Therefore, we only transfer

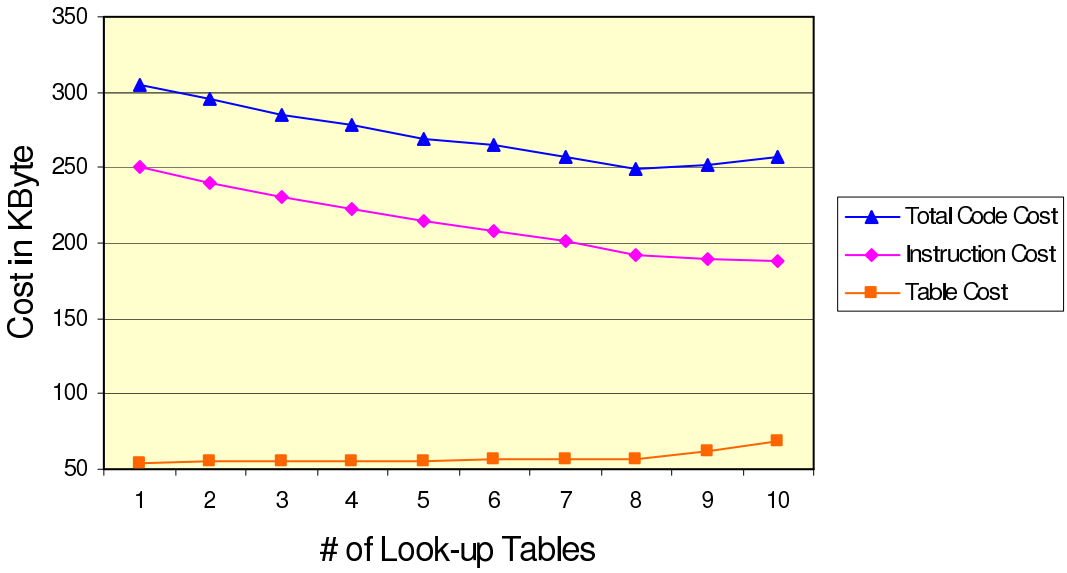


FIGURE 4.10: Optimizing the number of Look-up Tables

the instructions between two successive Look-up Tables (i.e. the difference in the size of their indices is just '1' bit).

Fig. 4.10 illustrates the effects of decreasing the number of Look-up Tables on the encoded instructions size, the compressed Look-up Tables size, and the total compressed code size for "Math" benchmark (compiled for ARM). Decreasing the number of Look-up Tables to be '1' can achieve the best table compression because this will give a better chance to compress more columns in each table through re-occurring patterns. On the other hand, this will increase the encoded instructions cost to its maximum value because all instructions (most frequent and less frequent sequences) will have the longest codeword. Consequently, the code cost will be increased, too. The optimum solution in this example is 8 Look-up Tables. This will increase the tables cost slightly but will reduce the instructions cost significantly and consequently the total code cost will be reduced.

Algorithm 6 shows how to minimize the Look-up Tables cost by transferring the instructions between them: After initializing the parameters, we start from the first two consecutive tables and compute the cost of these tables before transferring instructions using the function *DTC* (in Algorithm 4) (lines 5-7 of Algorithm 6). The number of transferred instructions is given by the range [k = 1 to k = # of all instructions in Table 1]. The number of repetition steps is given by "repeat = 10". The algorithm chooses (k) random instructions from the first table and transfers them to the second table (line 12), and computes the loss in the size of the encoded instructions (line 13). The algorithm then computes the cost of the tables after transferring the instructions (lines 15-17), and

Algorithm 6 TCM: Tables Cost Minimization

{L: Number of Look-up Tables}
{F: Frequency of instruction}
{N1, N2: Number of all instructions in Table 1 and Table 2}
{ch[min,max]: Minimum and maximum number of transferred instructions}
{index1, index2: Index length of Table 1 and Table 2}
{old_t_1, old_t_2: Table 1 and 2 before transferring instructions}
{new_t_1, new_t_2: Table 1 and 2 after transferring instructions}

1: default_efficiency = 0
2: k = ch.min = 1
3: ch.max = N1
4: repeat = 10 {# of repetition steps}
/* Algorithm Start */
/* Compute the tables cost before the transferring */
5: cost1 = DTC(N1, old_t_1, index1, cost1)
6: cost2 = DTC(N2, old_t_2, index2, cost2)
7: cost_before = cost1 + cost2
8: $temp\_t_1 = old\_t_1, temp\_t_2 = old\_t_2$
9: **while** k < ch.max **do**
10: **for all** steps s of repeat **do**
11: **for all** instructions i of k **do**
12: Transfer k random instructions from $temp\_t_1$ to $temp\_t_2$
13: Loss = Loss + F(i) {Compute the loss}
14: **end for**/* Compute the tables cost after the transferring */
15: cost1 = DTC(N1-k, $temp\_t_1$, index1, cost1)
16: cost2 = DTC(N2+k, $temp\_t_2$, index2, cost2)
17: cost_after = cost1 + cost2
18: Gain = cost_before - cost_after
19: efficiency = Gain - Loss
20: delta = efficiency - default_efficiency
/* Check if the transferring is good */
21: **if** delta > 0 **then**
22: default_efficiency = efficiency
23: $new\_t_1 = temp\_t_1, new\_t_2 = temp\_t_2$
24: **else**
25: Return the transferred instruction to $temp\_t_1$
26: **end if**
27: **end for**
28: k++
29: **end while**
30: **return**(new_t_1, new_t_2)

the gain in the size of the tables (line 18). Afterwards, the algorithm computes the efficiency (line 19) as in Eq. 4.4. If the result is improved, the algorithm keeps the new tables, otherwise it returns the transferred instructions in this step back to the Table 1. Finally, the algorithm returns the two new tables which achieve the best efficiency (line

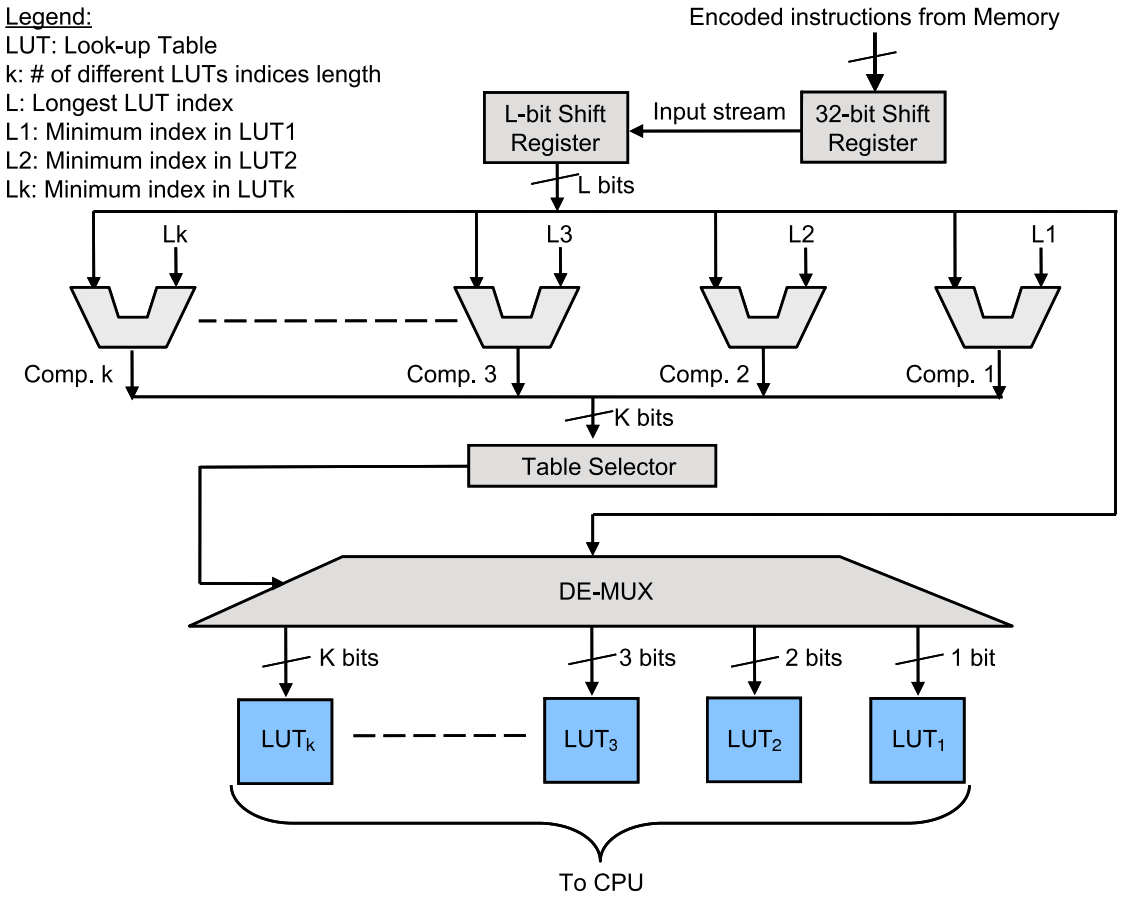


FIGURE 4.11: Canonical Huffman Decoder

30). We repeat the algorithm to minimize the cost of each two consecutive tables. The complexity of the *TCM* algorithm is $O(\frac{n(n+1)}{2}.repeat)$. This complexity depends on the number of the unique original instructions 'n' in the Look-up Table (i.e. entries of the Table) and the number of repetition steps "repeat". Increasing the parameter "repeat" can improve the results but it also increases the algorithm time. The complexity of this algorithm to obtain the optimal results would be $O(2^n - 1)$.

4.1.2.4 Hardware Implementation

The decompression hardware of the *Statistical* compression scheme consists of two parts: a Look-up Table decoder (as it is used in *Dictionary-based* compression scheme, Section 4.1.1.3) and a Canonical Huffman decoder.

Fig. 4.11 illustrates the Canonical Huffman decoder, which is designed to decode Canonical Huffman encoded instructions on the fly.
The decoder contains two shift registers: 32-bit and L-bit shift registers (L is the longest Canonical Huffman encoded instruction). The main task of the 32-bit shift register is

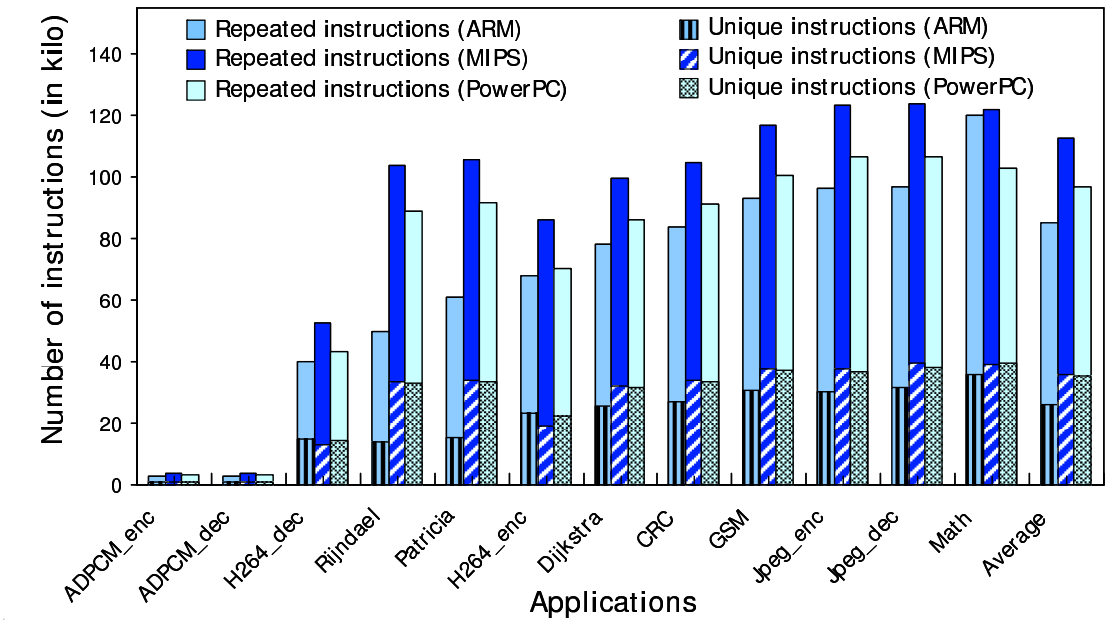


FIGURE 4.12: Number of original (repeated and unique) instructions for ARM, MIPS and PowerPC

to receive the encoded instructions and to keep the L-bit shift register filled each time its content is reduced by shifting the encoded instructions serially into it. The L-bit shift register transfers the L-bit encoded instructions to the comparators unit which has a number of comparators equal to the number of different encoded instructions length (k). The task of these comparators is to decode the length of the encoded instructions from the incoming (L) bits. The comparators operate simultaneously. Each comparator compares the incoming (L) bits with the minimum index of the corresponding table. If the incoming (L) bits are bigger or equal to the minimum index of that table, the corresponding comparator outputs a '1', otherwise '0'. The table selector selects the smallest comparator which outputs '1'. This comparator number refers to the length of an encoded instruction (i.e. the corresponding compressed Look-up Table). The compressed Look-up Tables (LUTs) are decoded using the Look-up Table decoder explained in the previous section. The decoder has been implemented in VHDL and synthesized with Xilinx ISE8.1 for an FPGA prototyping board [37]. The maximum frequency achieved for our Canonical Huffman decoder was 280 MHz (access time of 3.5 ns) and 600 slices were used.

4.1.2.5 Experimental Results

In this section we present the performance results of both compression schemes: *Dictionary-based* and *Statistical* compression schemes. In order to show the efficiency of our schemes, we conducted the results for three major embedded processor architectures, ARM (SA-110), MIPS (4KC) and PowerPC (MPC85) (details of these processors are explained in

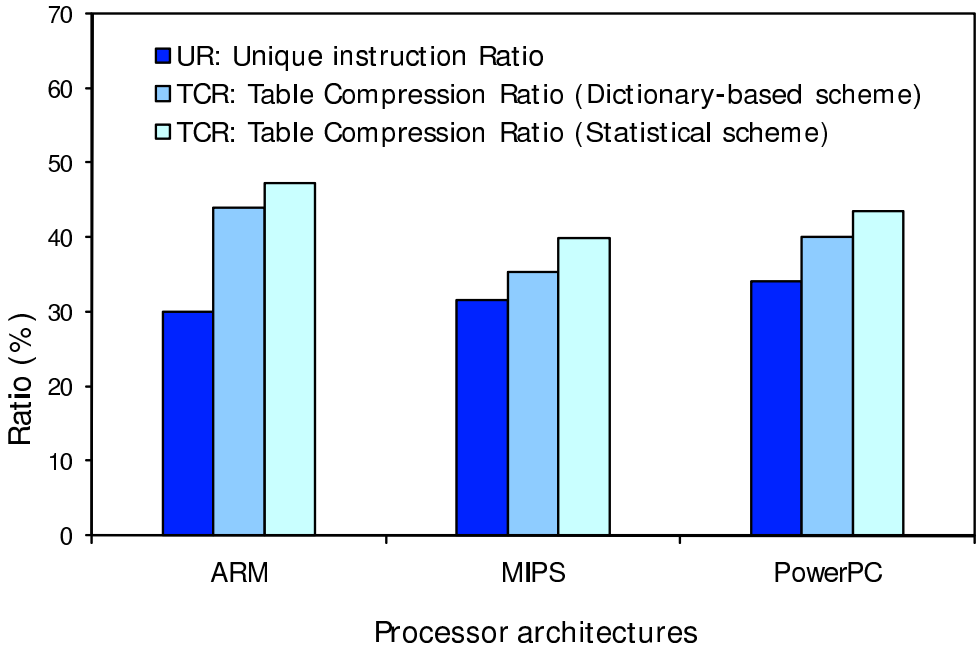


FIGURE 4.13: The average unique instruction and Table Compression Ratios

Section 2.4). It is also a goal to demonstrate the orthogonality as far as specific ISAs are concerned. For all architectures and all schemes the MiBench [20] benchmark suite is used as a representative (in terms of application domains and size) set of applications. We compiled the benchmarks using three cross-platform compilers [53], each for one target architecture. The final results are presented in Figures 4.12-4.20. They do account for the overhead stemming from the Look-up Tables. In each diagram, the bar labeled "Average" shows the average across all benchmarks. Fig. 4.12 presents the number of original (repeated and unique) instructions for different benchmarks and across the three architectures. This figure shows that the number of instructions generated by compiling a benchmark for the ARM architecture is always less than compiling the same application for MIPS or the PowerPC since the ARM instruction set is the most dense among the other RISC processors. This will result in the fact that the number of unique instructions will also be lowest for the ARM. The ratio of the number of unique instructions to the number of original ones, (denoted as *UR*) is presented in Fig. 4.13. This ratio gives an idea of how important the Look-up Table compression can be: in fact, we have found that the amount of unique instructions can account for 31.5%, 29.5% and 34.3% of all instructions for ARM, MIPS and PowerPC, respectively. Hence, the Look-up Table has a significant effect on the final compression ratio. In other words: neglecting Look-up Table compression may result in unacceptable allover (i.e. accounting for all induced overhead) compression ratios (Eq. 1.1). Furthermore, Fig. 4.13 shows also the average of the table compression ratio *TCR*, across all benchmarks, for both schemes (the middle and the right bars) in comparison to the *UR* (the left bar). *TCR* in the first scheme means the ratio of

the table size after the compression to its original size before the compression, but in the second scheme, it means the ratio of the sum of the tables cost after the compression and after minimizing their costs to the sum of their original cost before compression. Thus, a short bar means good compression performance. From the experimental results we can observe the following:

1. *TCR* is better for the applications with more unique instructions. In Fig. 4.12, the number of unique instructions on average is more for MIPS compared to the other architectures and hence, the *TCR*, in Fig. 4.13, is the best for MIPS. This has been expected since a large number of unique instructions results in large Look-up Table and this gives more chances to re-order its entries and to achieve better table compression. Hence, Look-up Table compression is especially useful in cases where the code is less compressible. It is therefore a powerful technique in cases where traditional code compression schemes do not perform well.

2. *TCR* in the first scheme is better than in the second (i.e. in the *Dictionary* and *Statistical* compression schemes) because the Look-up Tables in the second scheme are separated into a few smaller Look-up Tables, each of which needs to be compressed separately. Minimizing the table cost in the second scheme improves the *TCR* but it is still better in the first scheme.

Figures 4.14, 4.16 and 4.18 show the compression results for both schemes for the architectures ARM, MIPS and PowerPC, respectively. In each chart, the first bar stands for the original code size that includes the size of the unique instructions (the original table size) and the size of repeated instructions. The second and the third bars stand for the compressed code size of the first and the second schemes, respectively. The compressed code size includes the size of the compressed table(s) plus the size of the encoded instructions. The second scheme achieves a better compression ratio *CR* than the first one although the *TCR* is better for the first scheme, because of using the Canonical Huffman Coding properties that go along well with our Look-up Table compression. The average compression ratios achieved using the first compression scheme are 59%, 60% and 62% and using the second scheme are 53%, 51% and 55% for ARM, MIPS and PowerPC, respectively. Note that no ISA-specific knowledge has been used to obtain these ratios. The best compression ratio in our second compression scheme was obtained for the MIPS architecture because it has a large number of unique instructions. On the other hand, if the number of unique instructions is increased resulting in a larger *UR* (example: the *UR* for PowerPC in Fig. 4.13), this will result in a lower compression ratio because the instruction repetition (re-occuring patterns) is decreased.

Figures 4.15, 4.17 and 4.19 show the compression results but without applying our table compression technique. The results are shown for both schemes and for the architectures

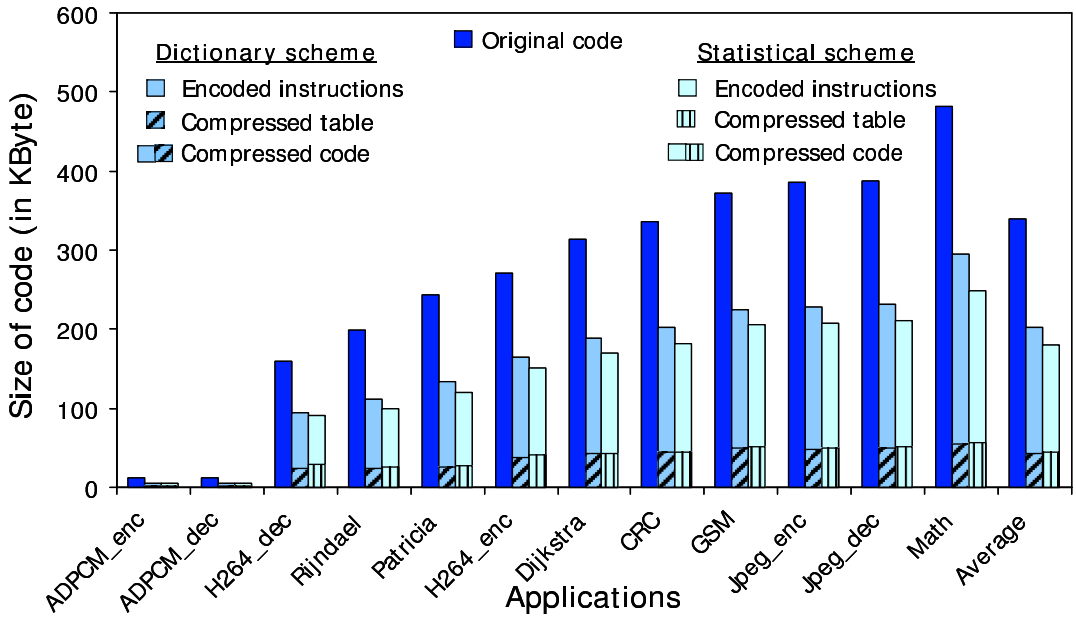


FIGURE 4.14: Compression results for ARM (the decoding table is compressed)

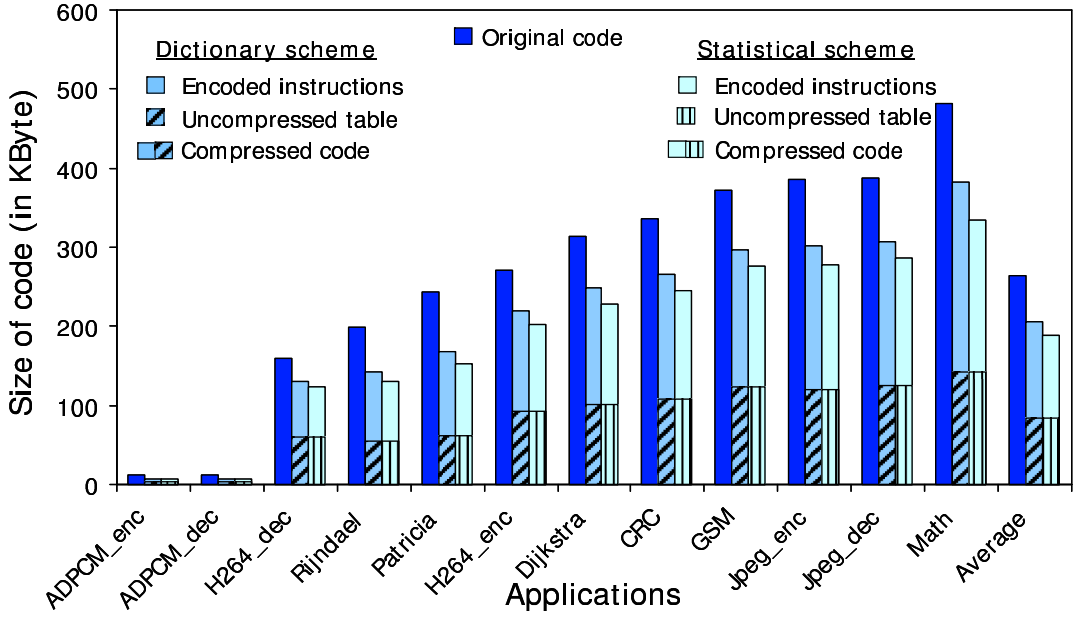


FIGURE 4.15: Compression results for ARM (the decoding table is not compressed)

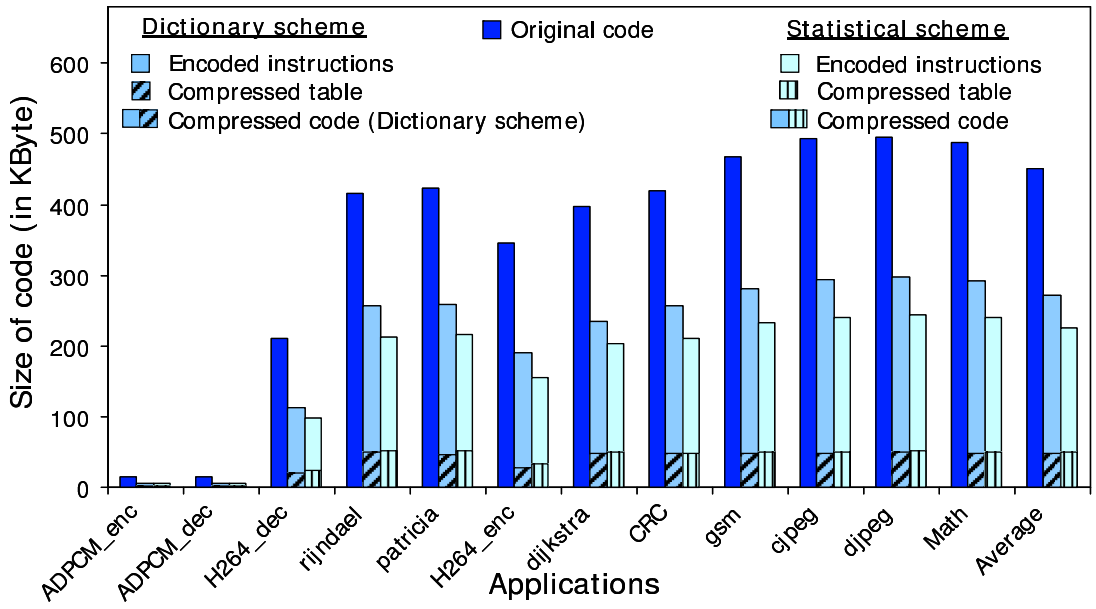


FIGURE 4.16: Compression results for MIPS (the decoding table is compressed)

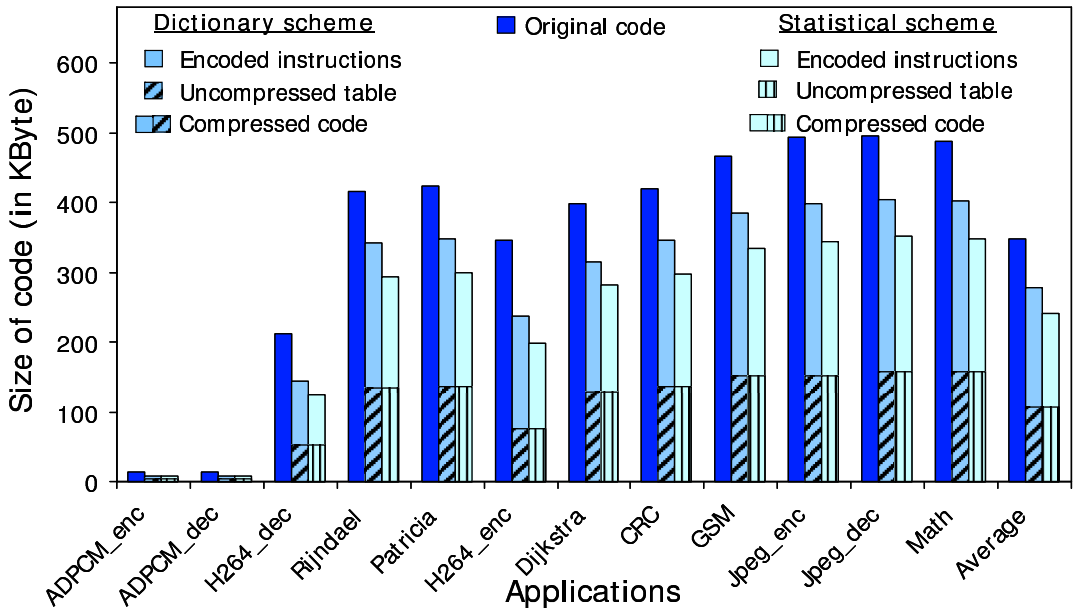


FIGURE 4.17: Compression results for MIPS (the decoding table is not compressed)

ARM, MIPS and PowerPC, respectively. Comparing these figures with figures 4.14, 4.16 and 4.18 shows the efficiency for compressing the Look-up Table. The figures 4.15, 4.17 and 4.19 show that the second scheme achieves higher compression than the first one. This is because in the first scheme, all instructions (most and less frequent occurring) are encoded with the same codeword length. But in the second scheme, the most frequent occurring instructions are encoded with small codeword length and vice versa. The average compression ratios achieved (without compressing the decoding table) using the

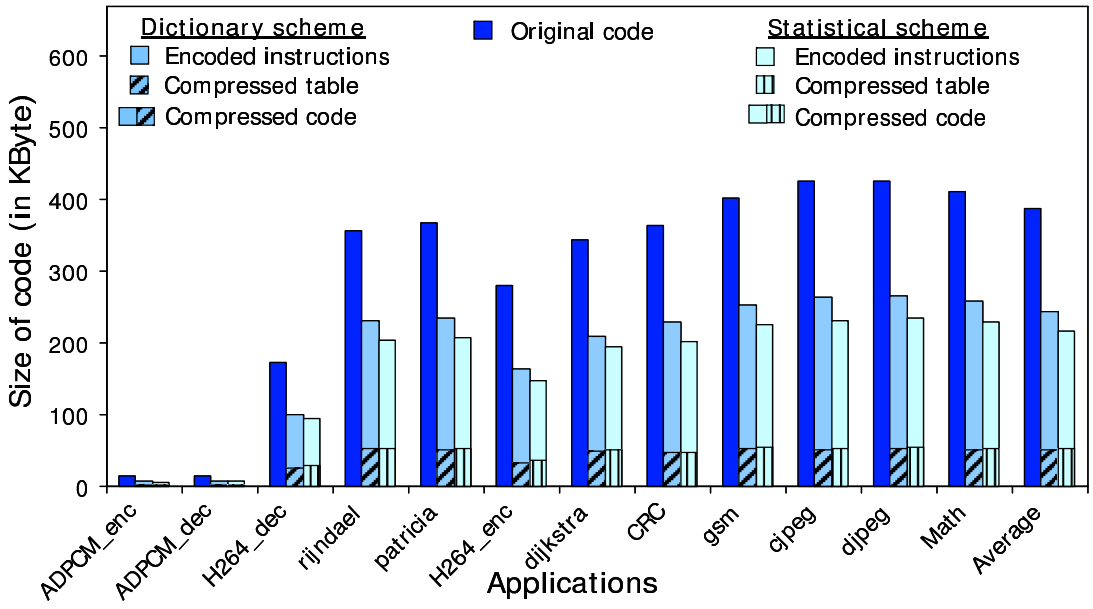


FIGURE 4.18: Compression results for PowerPC (the decoding table is compressed)

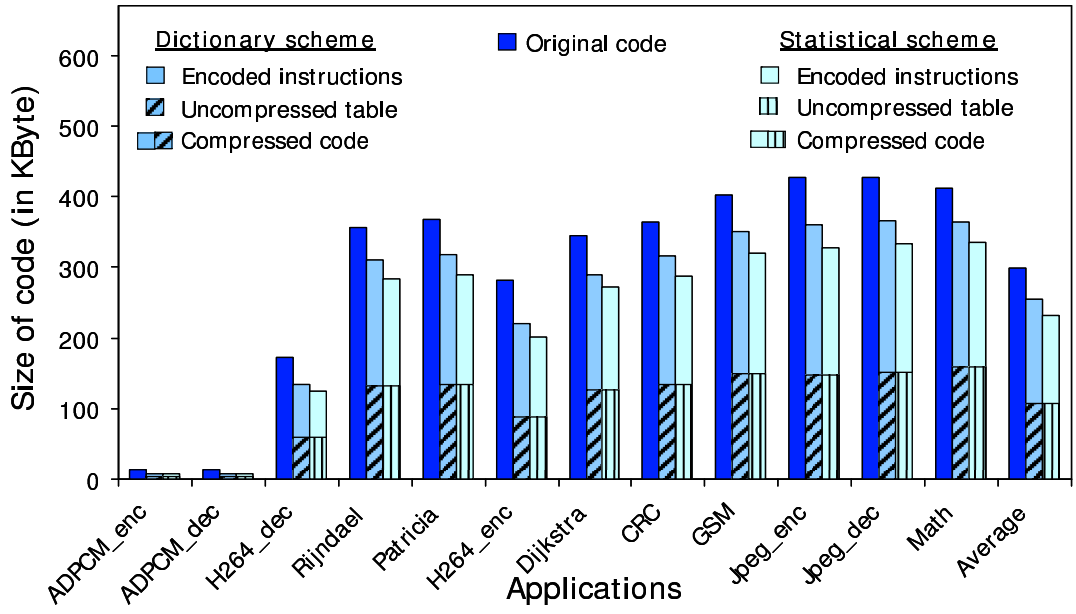


FIGURE 4.19: Compression results for PowerPC (the decoding table is not compressed)

first compression scheme are 78%, 79% and 85% and using the second scheme are 71%, 69% and 77% for ARM, MIPS and PowerPC, respectively.

Fig. 4.20 shows the overhead in the compressed code size (for the ARM Processor) caused by the aligning of the branch target addresses on memory borders. It is occurring in *Statistical* scheme only (because this case does not exist in *Dictionary* scheme). The incurred overhead is already factored into the compression ratios we report.

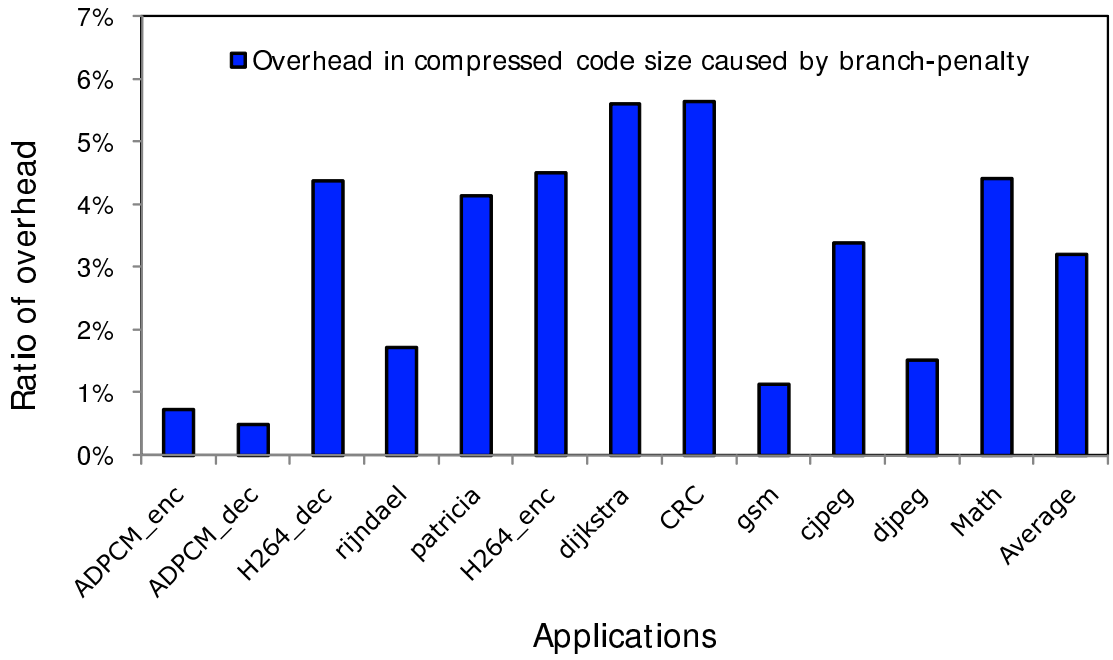


FIGURE 4.20: Overhead in compressed code size caused by branch-penalty

4.1.3 Instruction Splitting Technique

Using Huffman Coding as a compression technique for instruction code generates a large size decoding table in comparison to the size of encoded instructions (as shown in Fig. 4.1). This table may negatively impact the final compression ratio (Eq. 1.3) as discussed in Section 4.1.2. The problem of the large size of decoding table has been solved (in Section 4.1.2) by using our Look-up Table compression technique (in Algorithm 4).
If the Look-up Table is optimized (i.e. reduced in size) before the Look-up Table compression technique is applied to it, this may result in better compression ratio than what has been achieved in Section 4.1.2.

In this section, we present our second ISA-Independent code compression technique which is called "Instruction Splitting Technique". This technique analyzes the decoding table generated by using the Huffman Coding compression algorithm, and minimizes its size before Look-up Table compression is applied to it.
This technique is published in [3].

To reduce the size of the decoding table, we first need to analyze the instructions within this table.
We may divide the instructions of an application into two types:

- The instructions which are not repeated within the application ($frequency = 1$). We call them *"unique instructions"*.

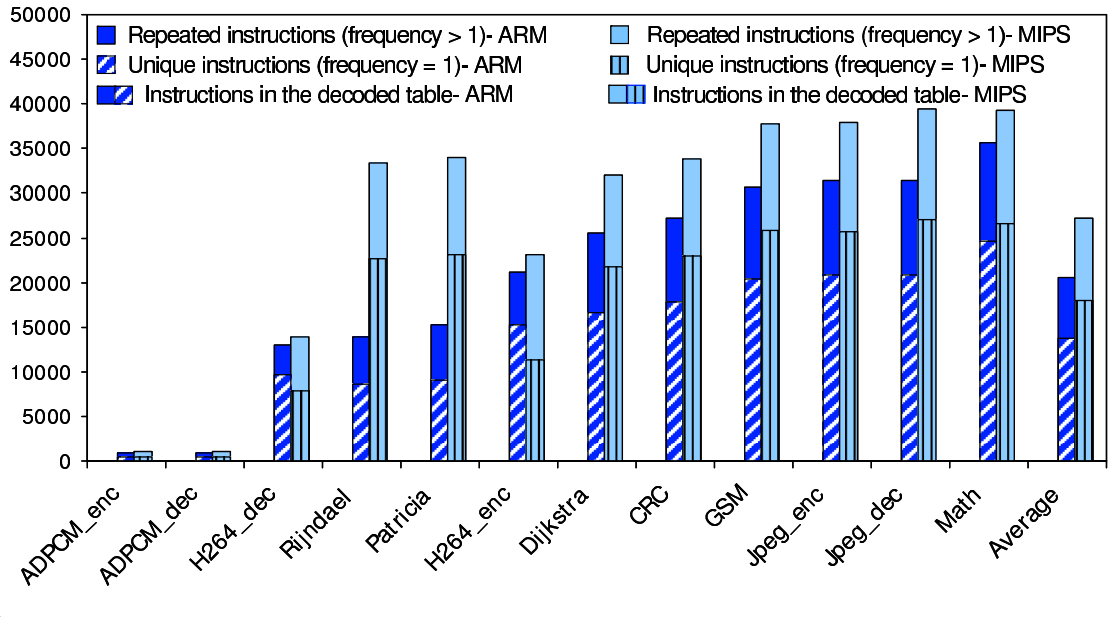


FIGURE 4.21: Number of *"unique instructions"* and *"repeated instructions"* within the decoding table.

- The instructions which are repeated within the application ($frequency > 1$). We call them *"repeated instructions"*.

When we apply Huffman Coding afterwards for code compression, the *repeated instructions* will be encoded with shorter code words than the *unique instructions*. Though this is beneficial for the size of the (compressed) instruction code, unfortunately an instruction of either type takes the same space in the decoding table — regardless of its repetition frequency!
By analyzing these two types of instructions (using a variety of different benchmarks) compiled for ARM and MIPS, we found that the *unique instructions* in fact dominate with more than 65% of the decoding table size (see Fig. 4.21, and remember that we discussed before in Fig. 4.1 that the decoding table itself occupies more than 40% of the space that is needed to comprise the whole code i.e. encoded instructions plus decoding table). Hence, a key challenge in our technique is to replace the *unique instructions* in the decoding table with shorter parts of instructions (called patterns).
In our code compression technique "Instruction Splitting", we conduct the following steps:

1. The object code of the *unique instructions* ($frequency = 1$) is extracted and split into an arbitrary number of variable-length patterns. The patterns are chosen in such a fashion that each pattern will have a high repetition frequency.

2. A fixed-length code word is assigned to each pattern of the *unique instruction*.

3. Fixed-length code words are also assigned to the object code of the *repeated instructions* (frequency > 1).

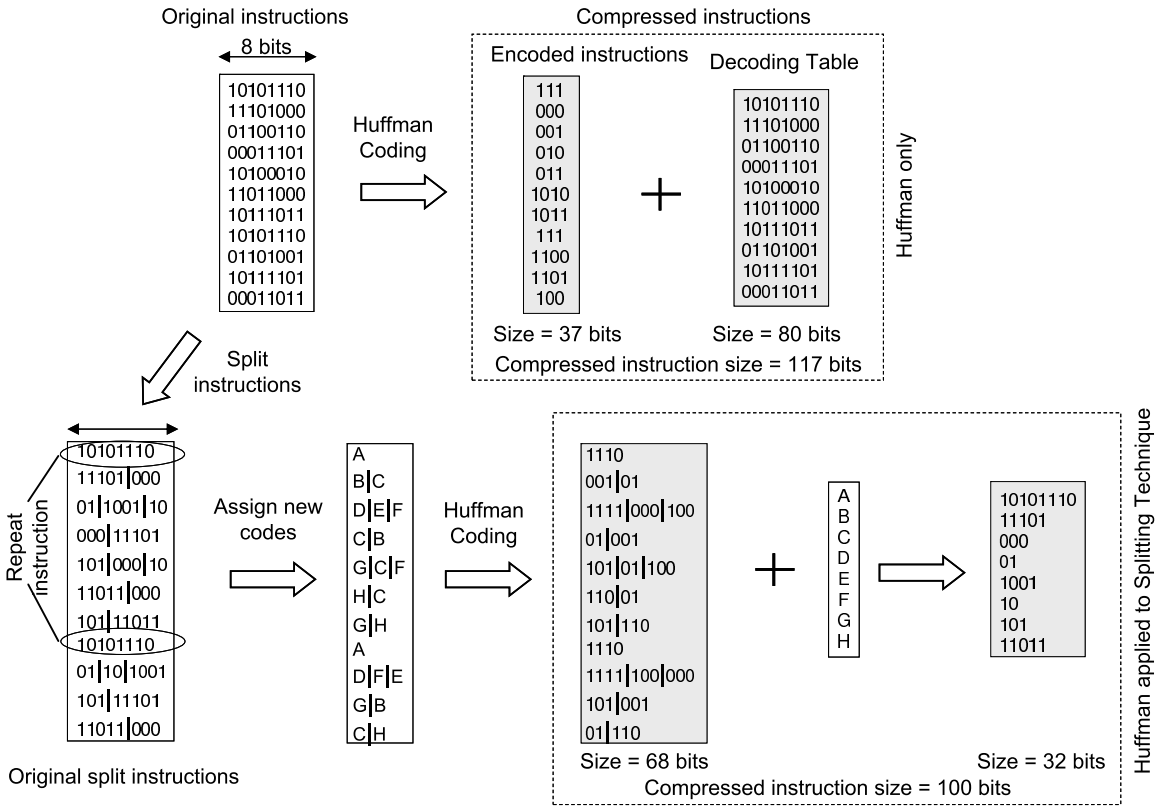


FIGURE 4.22: Example: Code compression using Huffman Coding only (upper part) and our *"Instruction Splitting Technique"* (lower part)

4. Finally, Huffman Coding is applied to *all* code words and a decoding table is generated.

These steps are explained in detail in the next sections.

4.1.3.1 Splitting Algorithm

As discussed before, the *unique instructions* dominate a large part of the decoding table. We therefore split the *unique instructions* into different variable-length patterns depending on their frequency using our *Instruction Splitting Algorithm*.

An overview in form of an example is given in Fig. 4.22. The upper part of this Figure shows a simple example for 11 instructions of 8-bits width each. First, the original instructions are compressed using traditional Huffman Coding. The sizes of the encoded instructions, the decoding table, and the compressed instructions are 37, 80, and 117 bits, respectively. The large size of the decoding table generated for decompression may negatively affect the final compression ratio (Eq. 1.1).
In the lower part of Fig. 4.22, the first instruction does not belong to the *unique instructions* type because it indeed is repeated in the example. Therefore, it is left without

splitting. The second instruction is a *unique instruction*. Our splitting algorithm found that the best splitting form is (5,3), i.e. a 5-bit pattern followed by a 3-bit pattern (because the first pattern is repeated 3 times and the second one is repeated 5 times within all the patterns in the example). The third instruction is split in the form (2,4,2) also for reasons of advantageous repetition frequencies throughout all patterns and so on.

Algorithm 7 shows the pseudo code for splitting the *unique instructions* into different variable-length patterns and assigning fixed-length code to each pattern: The algorithm starts by searching for an initial pattern 'pat' with full-word length ($pat\_len = 32$) in the program. It starts with the instruction 'i' of the program (line 7). In line 8, the algorithm chooses all the possible patterns which have the length 'pat_len' from the current instruction 'i' by calling the function 'choose_pattern'. The first possible pattern is the MSB 'pat_len' bits of the current instruction 'i'. The other possible patterns are created by shifting the instruction 'i' by the counter value 'cnt' to the right and then choosing the MSB 'pat_len' bits from the shifted value. The number of possible patterns is $(32 - pat\_len + 1)$. The frequency of the pattern 'pat' in the whole program is counted and stored in the counter 'pat_freq' (lines 10-14). This frequency is compared with the frequency of all the other patterns in the instruction 'i'. The highest frequency is registered in the variable 'pat_freq_new' (lines 15 and 16). Considering new instruction as a current one, the highest pattern frequency is computed among all the instructions and the highest result is registered in the variable 'another_pat_freq' (lines 20 and 21). Finally, if the pattern frequency over all the program instructions is more than the minimum accepted frequency (which is equal to 2), the pattern is replaced with an unique fixed-length code, and the algorithm searches for another pattern with the same length 'pat_len' in the program. Otherwise, it reduces the length of the searched pattern by 1 (line 28), and repeats the whole process until 'pat_len' is equal to 2. After splitting the *unique instructions* into different variable-length patterns and assigning new unique fixed-length codes to these patterns as shown in Fig. 4.22 (lower part), we compress these codes along with the non-split instruction (which are also assigned a new code) using Huffman Coding, as will be explained in Section 4.1.3.2.

4.1.3.2 Applying Compression on Split Instructions

In this step, we use the Huffman Coding algorithm as a compression technique for the fixed-length codes assigned in the previous step. It will generate the encoded instructions and the decoding table. The size of the encoded instructions (in bits) generated by the "Instruction Splitting Technique" is larger compared to their size in the case they they generated using sole Huffman Coding, (see Fig. 4.22). At a first glance that might seem disadvantageous, but it has been expected since any *unique instruction* is split at least into 2 parts and hence, it has more number of bits than the instruction which consist of

Algorithm 7 Instruction Splitting

```
{n: # of all instructions}
{pat: The searched pattern in the program}
{pat_len: Length of the pattern}

pat_len = 32 {Maximum pattern length}
min_accepted_freq = 2 {Minimum accepted pattern frequency}
pat_freq = 0 {Initial frequency of the pattern pat}
pat_freq_new = 0
another_pat_freq = 0
while pat_len> 1) do
  for all instructions i of n do
    for all counters cnt of (32 - pat_len + 1) do {Test all possible patterns in i which have length pat_len}
      Choose_Pattern(cnt, i, pat_len)
      for all instructions x of n do {Search for the pattern pat in all instructions}
        if pat is exist in x then
          pat_freq = pat_freq + 1 {Count the repetition}
        end if
      end for
      if pat_freq > pat_freq_new then {Store the highest repetition for this pattern}
        pat_freq_new = pat_freq
        pat_freq = 0
      end if
    end for
    if pat_freq_new > another_pat_freq then {Store the highest repetition among all patterns}
      another_pat_freq = pat_freq_new
      pat_freq_new = 0
    end if
  end for
  if another_pat_freq > min_accepted_freq then
    Replace the pattern pat with unique_constant
  else
    pat_len = pat_len - 1
  end if
end while


Function Choose_Pattern cnt, i, pat_len {
i = cnt << i {Shift the instruction i by the value cnt to left}
pat = the MSB pat_len bits of i
return(pat)
}
```

one single part, i.e. the *repeated instruction*.

But: now the size of the decoding table is noticeably smaller since it has variable-length instruction patterns which are frequently repeated in the program. Fortunately, the total

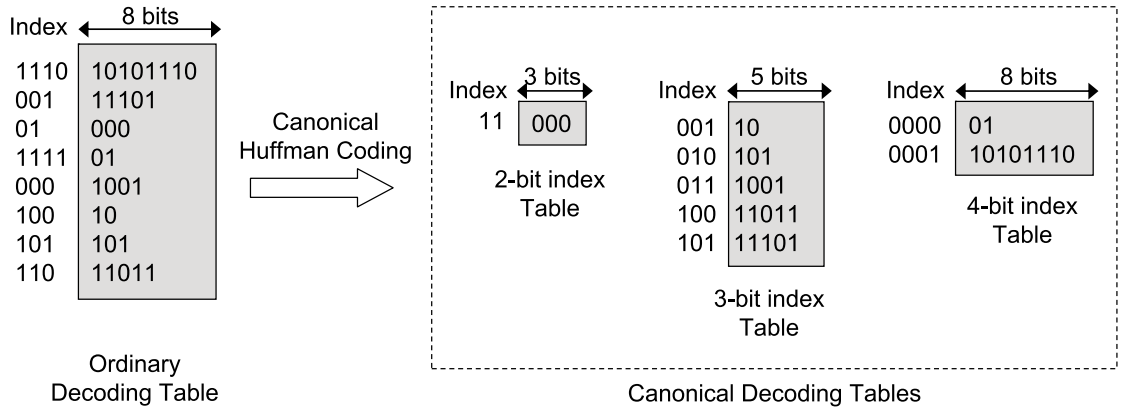


FIGURE 4.23: Example: Applying Canonical Huffman Coding to improve sparseness of decoding table as opposed to traditional Huffman Coding

size of the compressed instructions in our technique (encoded instructions + decoding table) is now smaller than compared to the case where Huffman Coding is applied solely. In fact, the total compression ratio (as to Eq. 1.1) has improved. In Fig. 4.22, the size of the compressed instructions is now reduced to 100 bits (117 bits before).

The encoded instructions are aligned in the memory and the problem of locating the branch addresses in memory is solved by patching these addresses to the compressed ones as adopted from [28].

In the"Instruction Splitting Technique", the entries of the decoding table have variable-length codes and the indices (i.e. the encoded instructions) to these entries are also variable in length. This makes the decoding difficult when it comes to hardware implementation. In addition to that, the size needed to store the decoding table in the memory is large (because of the sparseness of the table). To solve this problem, we use the Canonical Huffman Coding (as explained in Section 4.1.2.2).
This coding technique re-encodes the encoded instructions such that the instructions with the same length are binary representations of consecutive integers [35].

Fig. 4.23 shows an example for applying the Canonical Huffman Coding to the decoding table which is generated in Fig. 4.22. In this figure, the indices (the encoded instructions) for the decoding table have 3 different lengths (2, 3 and 4 bits). Therefore, we split the table into three tables, one for each different index length by using Canonical Huffman Coding [35], which re-encodes these indices such that they become contiguous in each table. Re-encoding the encoded instructions will change these instructions themselves but will not change their sizes determined by using Huffman Coding, because Canonical Huffman Coding maintains the size of the encoded instructions generated from Huffman Coding. To make the length of the entries in each table uniform, the table is split into other tables, one for each different entry length. In Fig. 4.23, the 3-bit index table for example is split into four different tables since it has four different entry lengths.

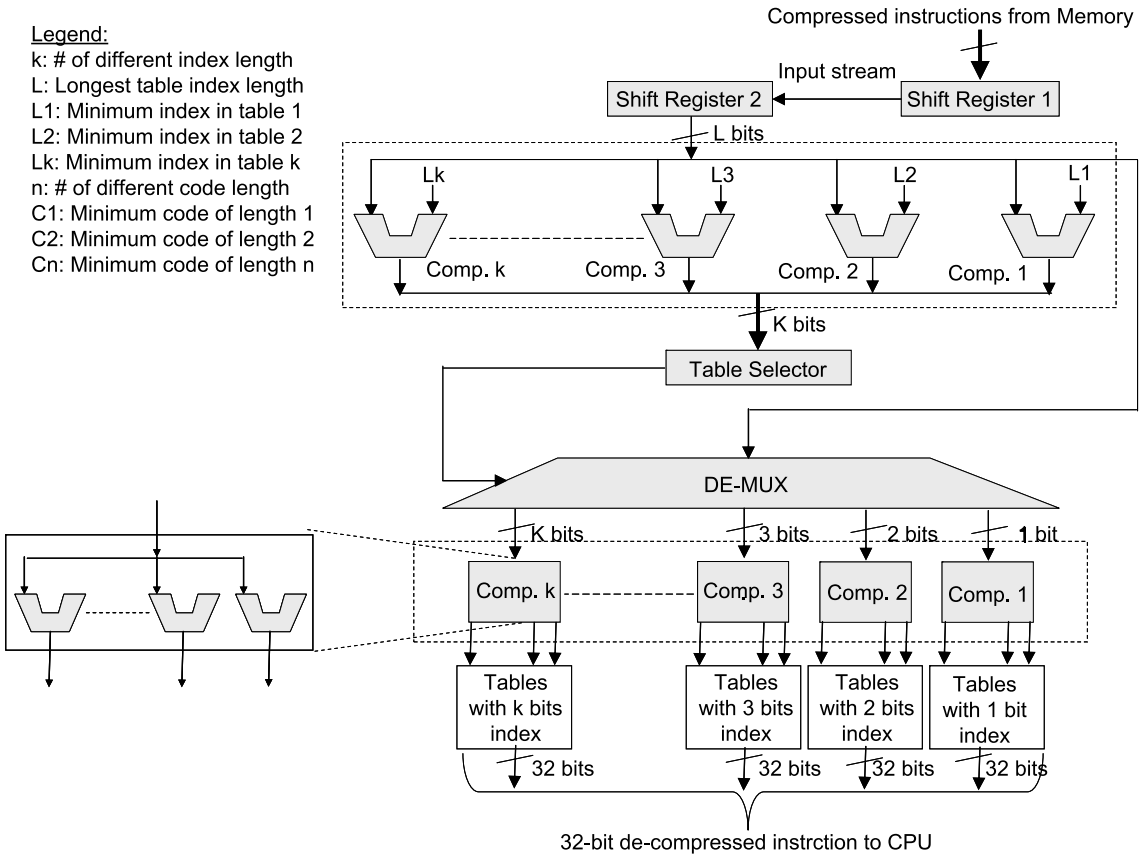


FIGURE 4.24: Decoding i.e. decompression hardware

This way, we have efficiently compressed the instructions. So far, all steps where applied off-line i.e. during design time. At run-time, the challenge is to decode (decompress) the instructions in order to feed the processor with regular (uncompressed) instructions. This is explained in the Section 4.1.3.3.

4.1.3.3 Decompression via Hardware

The hardware decoder architecture is illustrated in Fig. 4.24. It consists of two shift registers, de-multiplexer and two groups of comparators. The first shift register receives the 32-bit compressed instruction word from the memory. This word may contain one or more compressed instruction words for either a complete original instruction or parts of it. The main task of this register is to keep the second shift register filled each time its content is reduced by shifting the compressed instruction word serially into it (as explained in the Canonical Huffman Decoder, Fig. 4.11). The second shift register has a length equal to the longest encoded instruction L, i.e. the longest index, (in Fig. 4.23, L = 4). It transfers the L bit table index to the first group of comparators. The task of these comparators is to decode the length of the encoded instructions from the incoming L bits. The number of the comparators in this group is equal to the number of different

index lengths k, (in Fig. 4.23, $k = 3$). Each comparator compares the incoming L bits with the minimum index of the corresponding table. If the incoming L bits are bigger or equal to the minimum index of that table, the corresponding comparator outputs a '1', otherwise '0'. In Fig. 4.23, the minimum indices for the 2-bit, 3-bit and 4-bit index tables are '11', '001' and '0000', respectively. Since the number of bits of the minimum indices are less or equal to L, these indices need to be filled up with zeros from the right side to become equal in length to L as for the comparison. The table selector observes the comparator outputs to find the smallest comparator which outputs a '1'. The number of this comparator refers to the corresponding table that contains the original instruction (or part of it). This number is also the value by which the first shift register shifts its contents to the second shift register to allow a new code word to be decoded. The second group of the comparators specifies the length of the original instruction (or part of it) inside the table which has been selected from the first group of the comparators. The number of the comparators in this group is equal to the different lengths of instruction parts in each table. In Fig. 4.23, the 3-bit index table needs 4 comparators.

The whole decoder has been implemented in VHDL. It has then been synthesized using Xilinx ISE8.1 for VirtexII and has been implemented on a scalable FPGA platform *"Platinum"* from *Pro-Design*. On this platform it has extensively been tested along with the minimum necessary environment of a memory and ARM and MIPS cores. It also allowed us to measure the performance (presented later). The maximum frequency of 250 MHz (average access time of 4 ns) was achieved and just around 1200 slices were used.

4.1.3.4 Experiments and Results

In this section, we present the results of our compression scheme, and compare it to the results of sole Huffman Coding.

In order to show the efficiency of our compression technique, we conducted experiments for two major embedded processor architectures, namely ARM(SA-110) and MIPS(4KC). For both architectures the MiBench [20] benchmark suite served as a representative set of applications.[2] We compiled the applications using two cross-platform compilers, each for one target architecture. Figures 4.25 and 4.26 present the compressed and the original instructions of the benchmarks compiled for ARM and MIPS, respectively. The vertical axis denotes the size of the code in bytes. The first bar of each benchmark shows its original size before the compression. The second bar shows the size of the compressed instructions (encoded instructions + decoding table) using sole Huffman Coding. The

[2]Note that our scheme is most beneficial for large applications since the likelihood of repetitive patterns increases. Also, the additional hardware in form of the decompression hardware and Look-up Tables stays nearly constant from a certain benchmark size on.

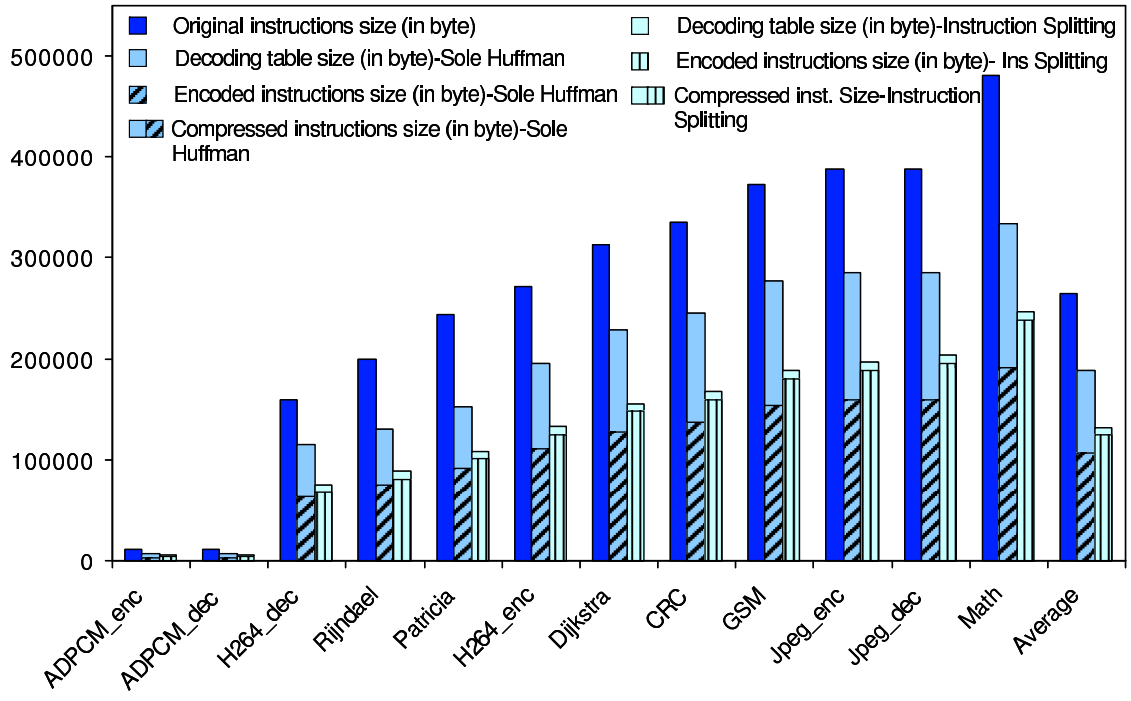


FIGURE 4.25: The compressed and the original instructions compiled for ARM.

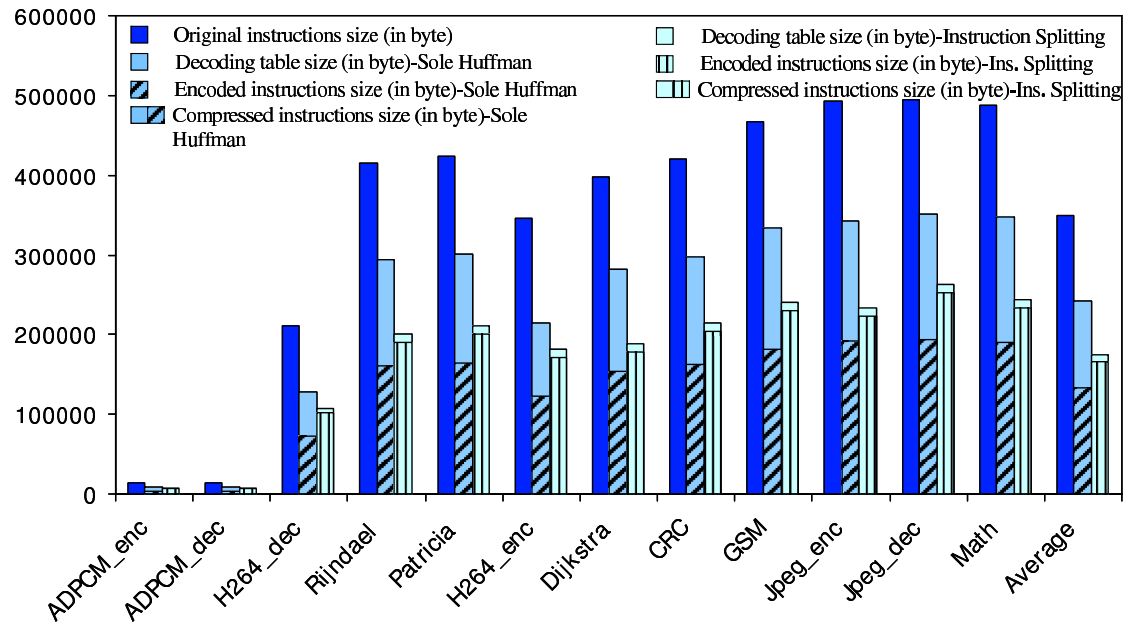


FIGURE 4.26: The compressed and the original instructions compiled for MIPS.

third bar shows the size of the compressed instructions using our "Instruction Splitting Technique". In each diagram, the bar labeled *"Average"* shows the average across all benchmarks in that diagram. Fig. 4.27 presents the final compression ratios in percentage according to Eq. 1.1 for both compression schemes and all applications (for ARM and MIPS).

From the experimental results we can observe and conclude the following:

1. The lowest (i.e. the best) compression ratios achieved using our compression scheme are 44% and 47% for ARM and MIPS, respectively. On average, compression ratios of 47% and 49% are obtained. As is shown, these are the *total* compression ratios i.e. also accounting for the table size besides the pure encoded instructions size. These numbers compare very favorably to compression ratios achieved by others (see Introduction). Even the relative improvement over sole Huffman Coding is 23% (see Fig. 4.27). This shows the effectiveness of conducting the instruction splitting scheme before applying Canonical Huffman Coding. A further advantage of our scheme is that it does not use ISA-specific knowledge.

2. For any of the architectures, the ratio of the decoding table size to the compressed instructions can be as large as around 40% when applying sole Huffman Coding. But that ratio is only 8% in our scheme. That means, our scheme saved about 32% off the size of the decoding table. That is actually where the improvements of the "total size" comes from.
   Previous research had mostly focused on improving (minimizing) the size of the encoded instructions themselves but not on improving the overhead that comes with it. It seemed that after several years of research in the field, in most recent years research in this area almost diminished since improvements were marginal because of sophisticated code compression schemes that had been proposed. However, with shifting the focus to the associated overhead we have achieved a further remarkable improvement.

3. Finally, code compression does not entirely come free. On the plus side it does reduce the code size and therefore minimizes memory requirements and even when factoring in the hardware for decompression and decoding tables: a large net gain remains. However, a bit of performance loss is the price we have to pay. Fig. 4.28 shows the number of execution cycles needed to execute the original and the compressed programs (in the *Statistical* compression scheme) using the SimpleScaler/ARM [40] performance simulator. In our case the performance loss is due to the time needed to fill the *shift register 2* (see Fig.4.24) with the incoming compressed instructions from the *shift register 1* every time a branch instruction occurs.

As a side note, it can be seen that ARM code compiles denser than MIPS code. Still, the compression ratio of ARM is 2% better on average (47% compared to %49 on MIPS; note that these numbers are relative i.e. refer to the initial uncompressed code that is obviously different in size in both cases). The 4ns latency of the decoding hardware we reported earlier could be further reduced if the decoding hardware would be built into the CPU itself as part of the instruction decode phase.

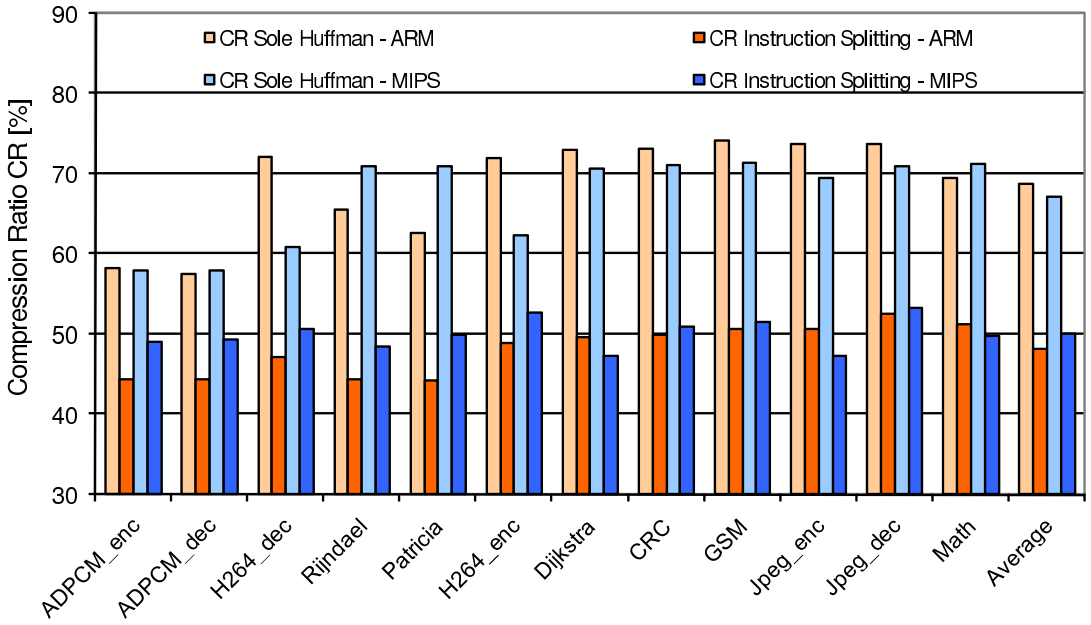


FIGURE 4.27: Compression ratios using sole Huffman Coding and our scheme for ARM and MIPS.

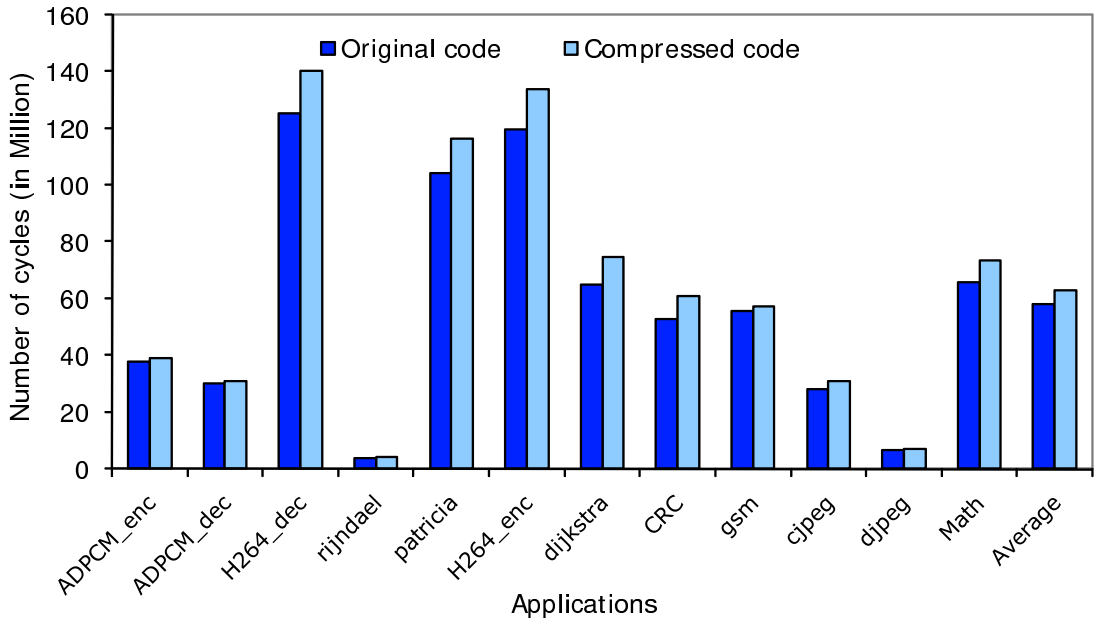


FIGURE 4.28: Performance of the "Instruction Splitting Technique"

4.1.4 Discussion

In this section, we discuss the compression results of the our Look-up Table Compression Techniques for *Dictionary-based and Statistical* schemes. Both techniques are ISA-Independent. They can be applied to any processor architecture independent from the instruction set format. These techniques can be also applied by any previous work to improve the final compression ratio.

In the Look-up Table Compression Technique for Statistical compression scheme, the most frequent occurring instructions are encoded with small codeword lengths and vice versa. For that, the average length of the instruction is less than the length of compressed instruction in the Dictionary-based compression scheme as the instructions in this scheme have a fixed length. This will result in better compression ratio in the case of the Statistical compression scheme (53%, 51%, and 55%) in comparison to the Dictionary-based scheme (59%, 60%, and 62% for ARM, MIPS and PowerPC, respectively).

The hardware decoder in the Dictionary-based scheme consists of one part which is the Look-up Table decoder, but in the *Statistical* scheme it consists of two parts: the Look-up Table decoder and the Canonical Huffman decoder. This will also result in slower decoding frequency and more number of slices used by the hardware decoder in the Statistical scheme in comparison to the Dictionary-based.

Our compression techniques archive better compression results in comparing with the stat-of-the-art compression technique used in PowerPC from IBM which is called CodePack (not better than 60% compression ratio has been achieved using CodePack), althoug our compreesion technique is not located on the same chip with the processor as it is in the CodePack.

Better compression ratio and performance may be achieved when our compression technique is integrated with the processor on the same chip in ASIC design.

4.2 ISA-Dependent Compression Technique

The ISA-Dependent compression technique results in better compression ratios in comparison to the ISA-Independent techniques as it is applied to a specific processor architectures.
When the ISA is specified, the code compression technique utilizes the information in the opcodes or the instruction format to build the hardware decoder. In this case, the compression ratio will be improved, since the number and the type of operands in the instruction format can be reduced according to the operation defined by the opcode.

In this chapter, we introduce new ISA-Dependent compression technique (called "Instruction Re-encoding") to improve the compression ratio that has been achieved using the previous ISA-Independent compression techniques ("Look-up Table Compression Technique" and "Instruction Splitting Technique"). On the other hand, This technique is not general for any Processor architecture (as in the ISA-Independent compression techniques), but it is based on the instruction format and the application itself, i.e. it is specified just for one specific processor architecture.

The crux of the "Instruction Re-encoding Compression Technique" is to find the position of bits in the instruction format which is suitable for re-encoding. We call those bits *re-encodable bits* [3]. Re-encoding those bits must have no affect on the functionality of instructions. we re-encode those bits to decrease the number of toggles in each table column (details are given in Section 4.2.1) and consequently to decrease the size of the decoding table.
Reducing the size of the decoding table will improve the final compression ratio *CR* according to the equation Eq. 1.3.

By analyzing a large set of benchmarks (MiBench), we found that the average size of the *re-encodable bits* can reach up to 26% of the original code size (as shown in Section 4.2.3). Those bits may be discarded from the instruction words or re-encoded depending on the compression algorithm used to achieve a better compression ratio.
Our "Instruction Re-encoding Technique" can be generally used in any ISA specification, if the *re-encodable bits* in the instruction format are known and extracted. Therefore, we apply our technique to two embedded processors, namely MIPS and ARM.

4.2.1 Steps of Instruction Re-encoding Compression Technique

In this technique, we conduct the following steps (See Fig. 4.29):

1. The instruction format of the original code is analyzed for a specific application to detect the *re-encodable bits*. These bits can be re-encoded without effecting on the instruction functionality. We use different techniques to find those bits and to increase their size. The *re-encodable bits* are then replaced with *don't care* symbols 'X'. We call the code in this case "modified code".

2. The modified code is compressed using the Huffman Coding algorithm. The encoded instructions and the decoding table are generated.

[3] Re-encodable bits are bits in the instruction format that may be re-encoded because they are not used for decoding the instruction but just to maintain the word alignment in the memory

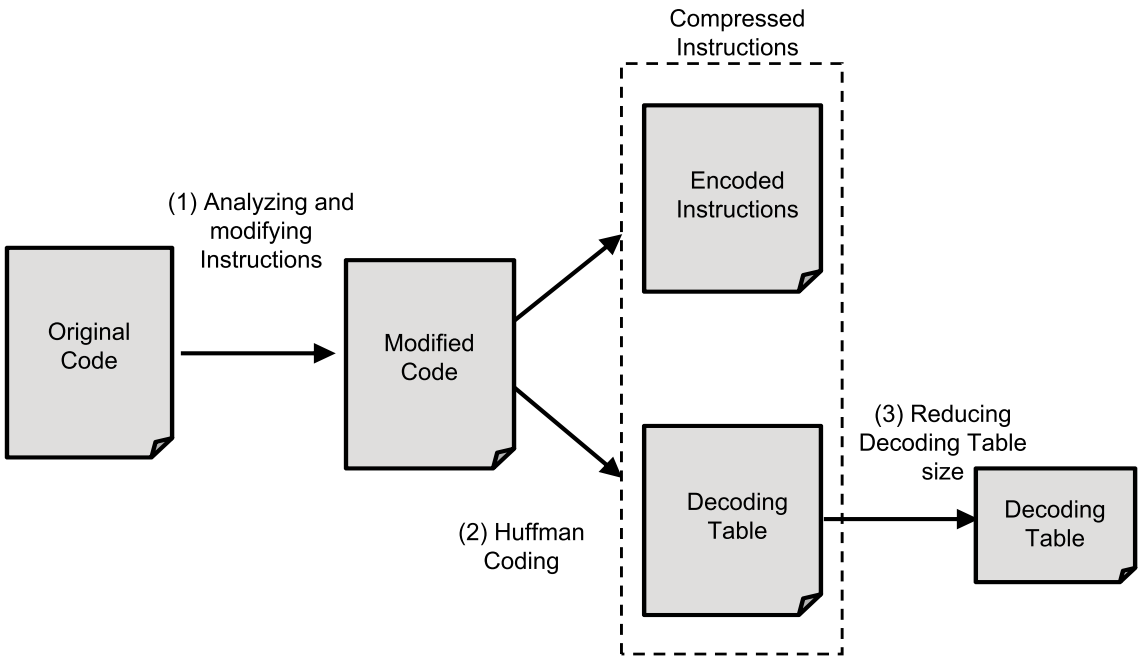


FIGURE 4.29: Steps of "Instruction Re-encoding Technique"

3. The size of decoding table is reduced by encoding the *don't care* symbols 'X' in each instruction to be identical to the preceding one.

These steps will be explained in detail in the following sections.

4.2.1.1 Analyzing the Instruction Format

The instruction set of any architecture is classified into different groups according to their coding formats. One group may contain instructions that have three register fields. Another one may have instructions with two register fields and immediate operand. Some instructions which have only one register and one target address fields may belong to another group, etc.
The first step in our code compression technique is to analyze the instruction format for a specific processor architecture and for a specific application. The purpose of that is to detect the *re-encodable bits* in the instructions of that application and then replace them with *don't care* symbols 'X'.

We use three different techniques to detect and increase the number of the *re-encodable bits* for a specific application:

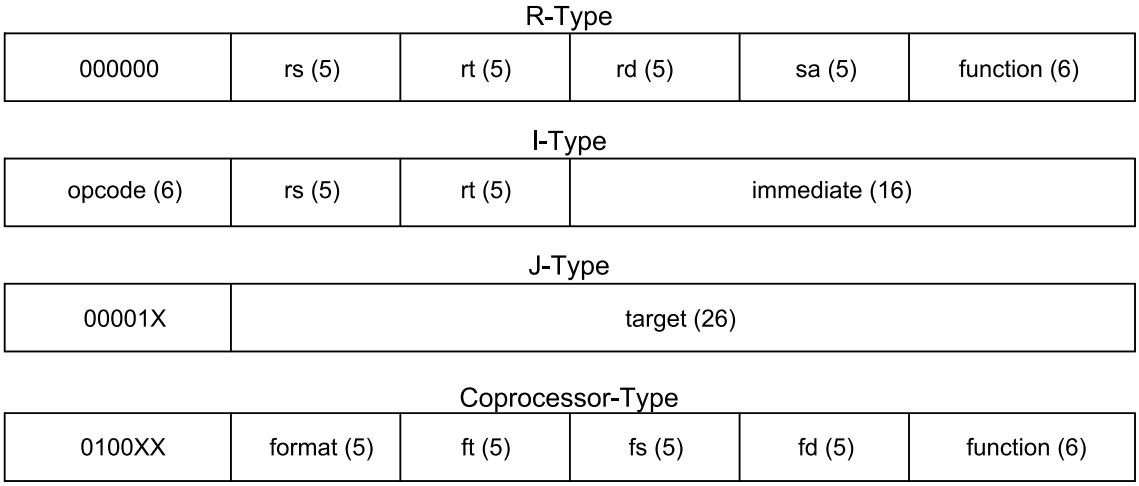


FIGURE 4.30: MIPS instruction groups

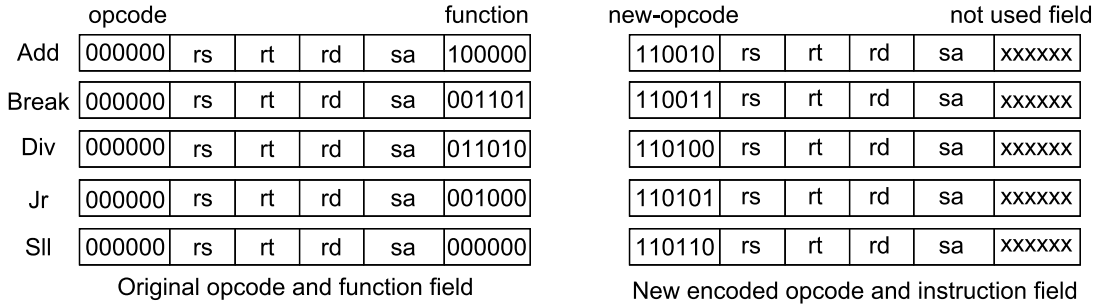


FIGURE 4.31: Example for re-encoding the opcode in the "R-Type" group.

1- Optimizing the opcode size:

In this technique, we explore the opcode of each instruction group for a specific application. Most of applications do not utilize all the available opcodes of an instruction group. Therefore, we re-encode the opcodes in this group to have less number of bits than the original ones. The new opcodes cover only the opcodes needed for that application. We replace the remaining unused bits of the opcodes with *don't care* symbols 'X'.
An evaluation has been conducted for two processor architectures, MIPS and ARM, on a large set of benchmarks (MiBench).

MIPS instructions are classified into different groups according to their coding formats [41]. The opcode can differ from one group to another (see Fig. 4.30). For example, the opcode of the instruction in "R-Type" group is "000000" and the instruction is specified by a function field which needs also 6 bits. This will reserve 12 bits in the instruction format to decode the instruction. Instead, we can re-encode the opcode field with a new code (which is not used by the instructions) and replace the 6-function-field bits with *don't care* symbols 'X'. If the application use only 32 different "R-Type" instructions (or less), then only 5 bits can be used for re-encoding the opcode field and one bit can be replaced with 'X'. Fig. 4.31 shows an example for different instructions in the "R-Type"

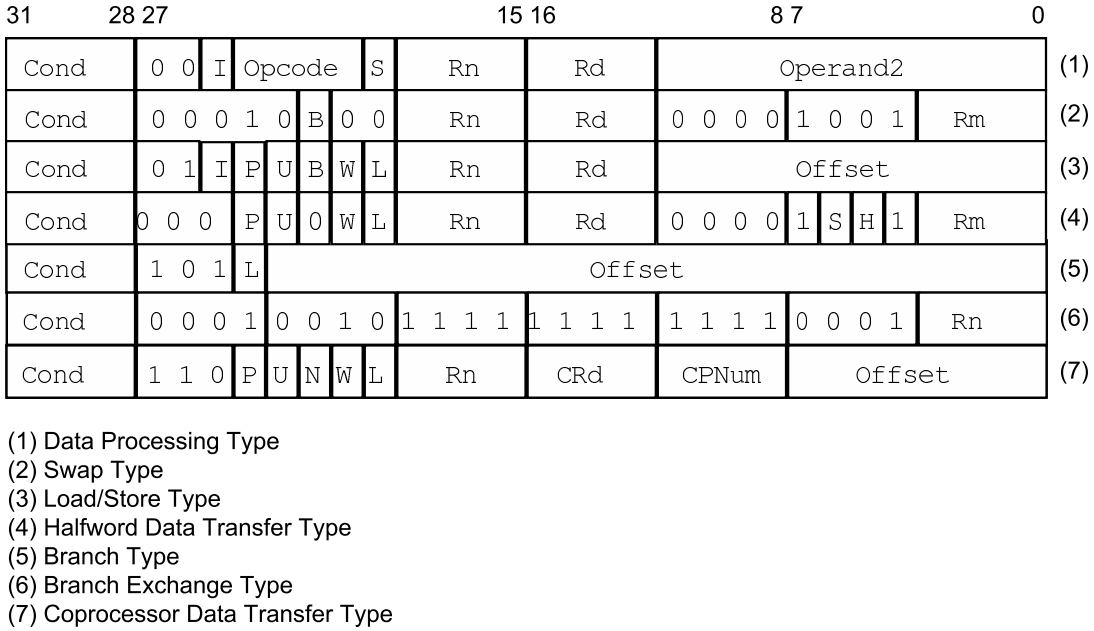


FIGURE 4.32: Example for ARM instruction groups

group before and after re-encoding the opcode. On the left side of this figure, the opcode for all instructions is the same but the function field is different. For that, we assign new different opcode for each instruction and *don't care* symbols 'X' for the function field (On the right side). The new opcode is selected to be unique and not used by the other MIPS instruction groups.

In "J-Type" group, all instructions use the opcodes "00001X" (see Fig. 4.30).

In "Coprocessor-Type" group, all instructions use the opcodes "0100XX". The floating point instructions use the opcode "010001". The format field (5 bits) and the function field (6 bits) in the instruction format are used to specify the instructions in this group (see Fig. 4.30). Actually, we do not need all these bits for specifying the instruction. Therefore, we can use the function field to specify the opcode (which can accept up to 64 different floating point instructions) and replace the bits in the format field with *don't care* symbols 'X'.

ARM instructions are also classified into different groups according to their coding formats [42]. All instructions are conditionally executed depending on the instruction's condition field (see Fig. 4.32). The condition field (bits 31:28) determines the circumstances under which an instruction is to be executed. ARM instructions contain primary opcode and secondary opcodes. For example, the instruction in the "SWAP" group has primary opcode "00010" (bits 27:23) and three secondary opcodes "00" (bits 21:20), "0000" (bits 11:8) and "1001" (bits 7:4). We investigated the opcodes in all groups and found that the secondary opcode in "SWAP" group "0000" (bits 11:8) may be replaced with symbols "XXXX" without causing any collisions. Another example is given by the

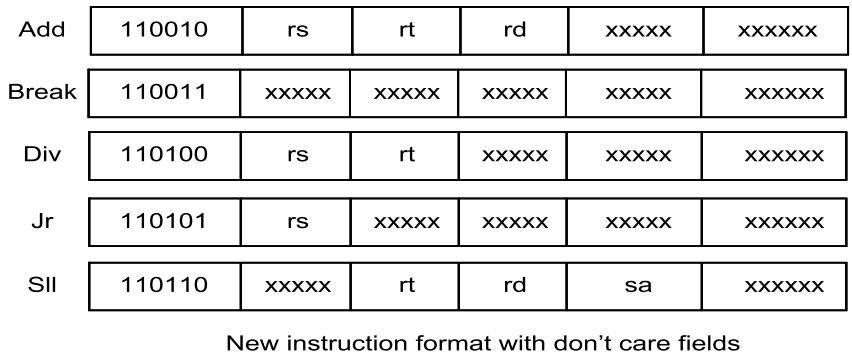


FIGURE 4.33: Replacing unused fields in the "R-Type" group with don't care symbols 'X' for different instructions

instructions in the "Halfword Data Transfer" group. Their secondary opcode "0000" (bits 11:8) may be also replaced with symbols "XXXX" without collisions. The same scenario can be applied to the "Branch Exchange" group. Its opcode has '24' bits which can be shortened with a new one. The new opcode must be unique for a specific application and the remaining bits may be replaced with symbols 'X'.

2- Finding the unused register fields:
In this technique, we explore the unused register fields of each instruction in the group and then replace them with *don't care* symbols 'X'.

In **MIPS** architecture, for example, some instructions in "R-Type" group utilize the 'rs', 'rt' and 'rd' register fields and leave the 'sa' field unused. Other instructions use two registers, one register or even do not use any register field, like "Break". All the unused register fields may be replaced with 'X'. The same thing can be applied to the other MIPS groups. Fig. 4.33 shows an example for some instructions with unused register fields after replacing them with symbols 'X'.

In **ARM** architecture, the instructions in "Data Processing" group can reach more than 50% compared to instructions in all groups for a specific application. "MOV" and "MVN" are two instructions in the "Data Processing" group whose frequencies can reach more than 50% compared to other instructions in the same group. These two instructions utilize the 'Rd' register field (bits 15:12) and the "Operand2" field (bits 11:0), but they leave the 'Rn' register field (bits 19:16) unused (see Fig. 4.32). This register field may be replaced with symbols 'X'. The same scenario can be applied to other instructions in all groups.

Algorithm 8 Re-encoding immediate field

{n: # of instructions which has immediate value}
{imm_len: length of immediate value in bits}
{j: length of selected pattern in bits}

```
1: for all instructions i of n do
2:    j = 1
3:    while j < imm_len do
4:       Freq = frequency of j for all instructions
5:       Gain = Freq x j
6:       j = j + 1
7:    end while
8:    Find the highest Gain for pattern j in instruction i
9: end for
10: Find the highest Gain for pattern j in all instructions
11: Replace the pattern j with symbols 'X'
```

3- Reducing the size of immediate or target offset fields:
This technique can detect a high number of unused bits compared to the other previous techniques. Algorithm 8 shows the pseudo code of re-encoding the immediate or target offset field. Normally, the immediate or target offset values occupy the least significant bits in their fields, leaving the most significant bits in the field either unused or less frequently used. Therefore, we search in these fields the most frequent sequence of bits through all instructions starting from the most significant bits side toward the least significant side and for a specific application. The most frequent sequence of bits (we call them patterns) may be replaced with symbols 'X'. Of course, we leave some bits to distinguish between those instructions which have the symbols 'X' and the instructions who have not.

In **MIPS** architecture, this technique can be applied to all instructions in "I-Type" and "J-Type" groups. In addition to that, in "J-Type" group we can replace the last two least significant bits (bits 1:0) with symbols 'X' because we know that they are always '0' since the instructions are word-aligned. When we decode these instructions, we have to replace these bits again with zeros.

In **ARM** architecture, this technique can be applied to all instructions in "Load/Store", "Branch" and "Coprocessor Data Transfer" groups and most of the instructions (which need immediate or target offset fields) in "Data Processing" group (Fig. 4.32).

4.2.1.2 Huffman Coding Algorithm

The second step in this compression technique is to use the Huffman Coding algorithm (see Fig. 4.29). The most frequently occurring blocks of code (which can be one or a sequence of instructions) are encoded with short codewords, whereas the less frequently occurring ones are encoded with large codewords. We call these codewords "encoded instructions". In this way, the average codeword length is minimized (more details about Huffman Coding are given in Section 2.3.2.1).

The size of the decoding table (which is generated using Huffman coding) is large and may impact on the final compression ratio.
To reduce the size of the decoding table, we re-encode the *re-encodable bits* in the instruction format (which we extracted and created in Section 4.2.1.1) as a primary step to further achieve reduction in the decoding table.

4.2.1.3 Reducing the Size of Decoding Table

As explained in the previous section, the Huffman Coding algorithm generates variable length encoded instructions and a decoding table which is used to retrieve the original instructions. The decoding table contains the original unique instructions. The encoded instructions are used as indices to the decoding table. While the encoded instructions have variable length, they can not be used as indices to only one decoding table. For that, we divide the decoding table into different decoding tables as many as we have different compressed instruction lengths (as we did in the "Instruction Splitting Technique"). In this case, the encoded instructions which have the same code length are used as indices to the same decoding table.
As the encoded instructions are stored non-contiguously in each decoding table, we re-encode them using Canonical Huffman Coding (as we did in the previous technique).
Reducing the size of any decoding table may be achieved by reducing the size of its columns. To accomplish this, we use our Look-up Table compression technique which we used in 4.1.1.2. In this technique we compress each decoding table column by storing the address where the transition (0—>1 or 1—>0) in each column happens in the decoding table instead of storing the complete column. If the size needed to store these addresses is less than the size of the complete column, the column can be compressed. Otherwise, we leave the column without compressing.

The crux of our compression technique is to reduce the number of transitions happening in each decoding table column. This can be achieved by:

1. Selecting a good sorting algorithm to sort the decoding table entries.

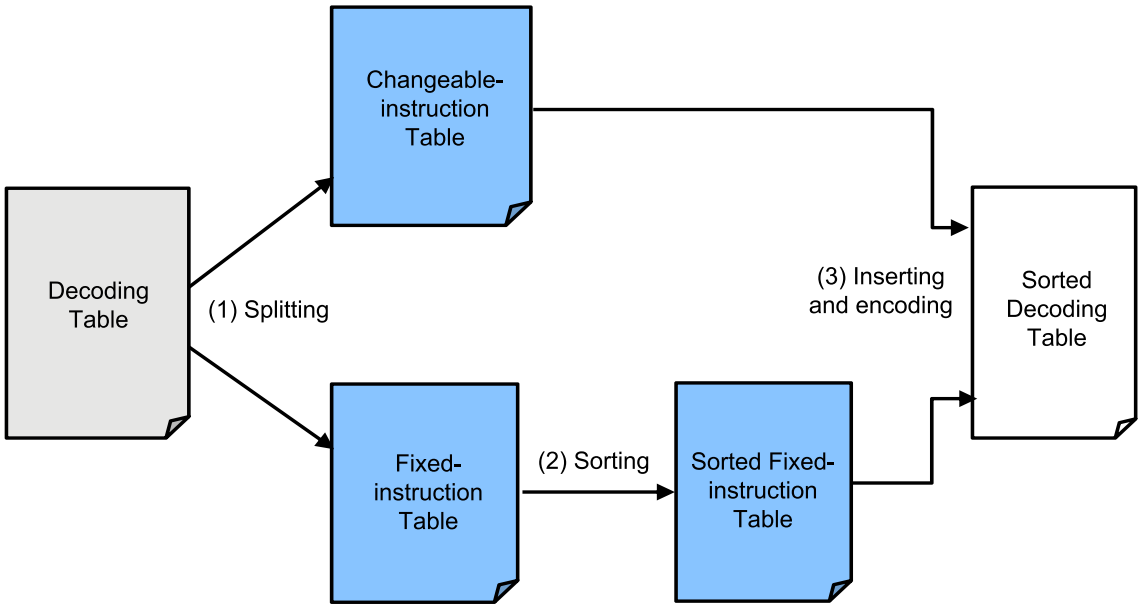


FIGURE 4.34: Steps for reducing the size of the decoding table

2. Encoding the *don't care* symbols in the instruction format to be identical to the preceding instruction in the decoding table.

Fig. 4.34 shows the three steps we use to reduce the size of the decoding table. These steps need to be applied to each decoding table. They are performed off-line (in the design time). Hence, it does not matter how much time these steps will take.

In the first step, we separate the decoding table into two tables: the fixed-instruction table and the changeable-instruction table. The fixed-instruction table contains the instructions which can not be changed, i.e. the instructions which do not have any *don't care* symbol 'X'. The changeable-instruction table contains the instructions which have at least one 'X' symbol.

In the second step, we sort the entries of the fixed-instruction table to minimize the number of transitions in each column. This will compress more table columns and achieve better table compression. We sort the table entries through two phases (as we did in our previous work [2]). In the first phase, we generate *Gray Code* for 32 bits (i.e. the instruction word length). The generated code has a property that each code differs from the former one in one bit. Therefore, we locate each instruction of the fixed-instruction table in its corresponding *Gray Code* position. This technique does not give the optimal solution because the instructions do not cover all the generated *Gray Codes*. For that, we use the *Lin-Kernighan* sorting algorithm (as a second phase for sorting). This algorithm sorts the table entries in a way that the sum of the distances between two successive

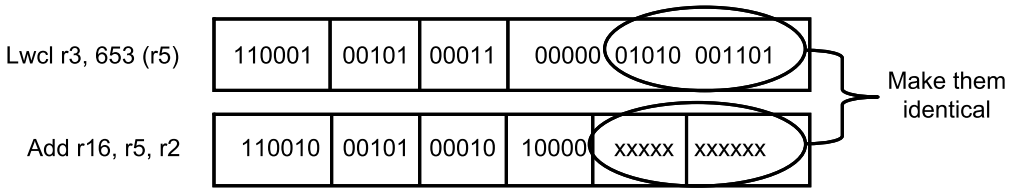


FIGURE 4.35: Example for re-encoding the 'X' fields: make them identical to the preceding instruction

entries from the first entry to the last one is minimal (or close to it) [1]. In our case, the distance between two entries is the number of positions in which the corresponding bits are different. Sorting the instructions using the *Lin-Kernighan* algorithm after ordering them using *Gray Code* gives better compression than using only the *Lin-Kernighan* algorithm [1].

In the third step of our technique for reducing the size of the decoding table, we insert the instructions of the changeable-instruction table in the sorted fixed-instruction table in the position where each inserted instruction must be identical in the most bits to the preceding one. The 'X' field of the inserted instruction must be encoded to be the same as that field of its former instruction. Fig. 4.35 shows an example for selecting the instruction "Add" to be located after the "Lwcl" instruction in the decoding table. This is because most bits of both instructions are identical. The 'X' fields in the "Add" are then encoded to be identical to the same fields in the "Lwcl" instruction.

Another technique may be used to reduce the number of bit transitions in the decoding table columns. This can be done by swapping the position of register fields in instruction format to be identical to the fields of the successive instruction (if this will not change the instruction functionality). For example, in MIPS architecture, some successive instructions use the register fields 'rs', 'rt' and 'rd'. The register fields 'rs' and 'rt' in the successive instructions seem to be identical if the positions of these register fields are reversed. Swapping their positions will have no effect on the instruction functionality, but on the other hand, will reduce the number of bit transitions in the decoding table columns and consequently will reduce the size of the compressed instructions as well. The following ADD and XOR instructions, for example, have different 'rs' and 'rt' register fields:

ADD r5, r3, r4

XOR r6, r4, r3

Swapping the registers 'r3' and 'r4' in any of the previous instructions will make the fields 'rs' and 'rt' in both instructions identical.

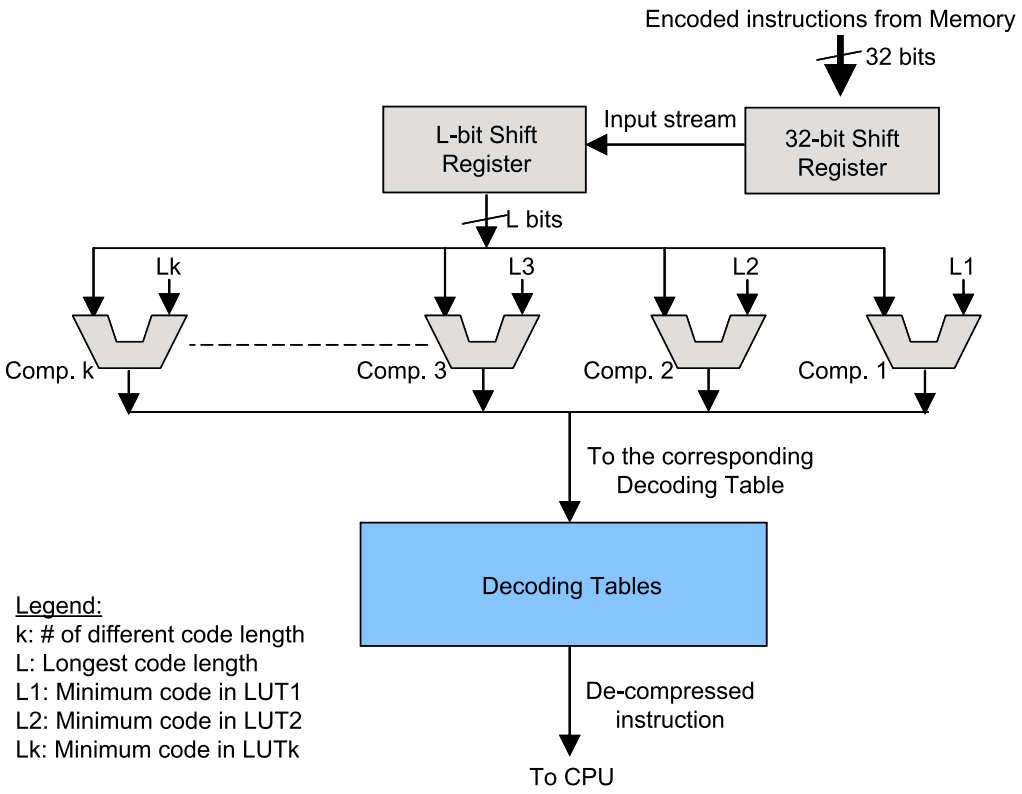


FIGURE 4.36: Hardware decoder of the "Instruction Re-encoding Technique"

The previous steps reduce the number of bit transitions in each column. Therefore, we can compress more columns of the decoding table by storing in each column only the address where the bit transition happens instead of storing the complete column. This will consequently reduce the size of the decoding table.

4.2.2 Hardware Decoder

The hardware decoder in this technique is based on the decoder of the "Instruction Splitting Technique" (which is presented in Section4.1.3.3). It decodes the compressed instructions in two stages. In the first one, the length of the compressed code is computed (Fig. 4.36). This can be done by using as much comparators as there are different compressed code lengths, i.e. one comparator for each length (as explained in the "Instruction Splitting Technique" decoder). The second stage in the decoding is to retrieve the original instruction from the specified decoding table. This can be done by finding out the number of bit transitions in each compressed column for the incoming compressed code. If the number of bit transitions is even, the bit in the corresponding column is '0'. Otherwise, it is '1'.
When the instruction is decoded, it needs a slight modification to match the original instruction format. For example, in "R-Type" group of MIPS architecture, the "000000" is assigned to the opcode field and the correct instruction function is assigned to the

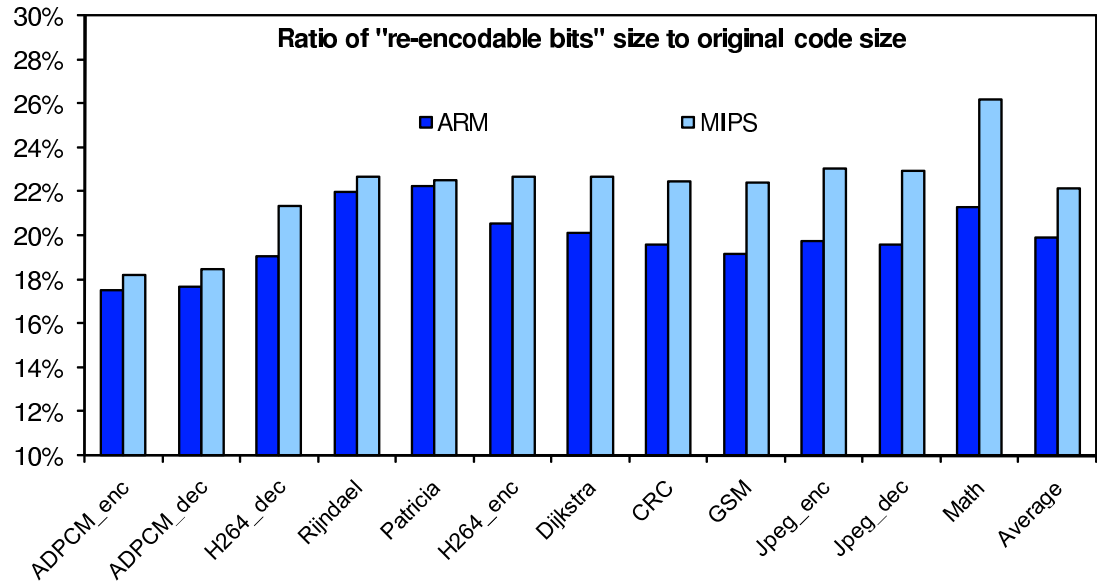


FIGURE 4.37: Ratio of *re-encodable bits* size compared to original code size for ARM and MIPS processors

function field.

The hardware decoder has been implemented in VHDL. It has then been synthesized using Xilinx ISE8.2 for VirtexII and implemented on a scalable FPGA platform *"Platinum"* from *Pro-Design* [37]. On this platform it has extensively been tested along with the minimum necessary environment of a memory and ARM and MIPS cores. It also allowed us to measure the performance (presented later).
An average access time of 5 ns (200 MHz) was achieved and less than 1500 slices were used for the hardware decoder.

4.2.3 Experimental Results

From the experimental results we can observe and conclude the following:

1. The "Instruction Re-encoding Technique" analyzes the instructions to find out the *re-encodable bits* and to increase the number of these bits. Fig. 4.37 shows how large the number of these bits can be for different applications compiled for ARM and MIPS processors. The size of the *re-encodable bits* in the instructions can reach up to 22% and 26% of the size of the whole instructions, for ARM and MIPS processors, respectively. This ratio differs depending on the instruction format and the number of instructions in the application.

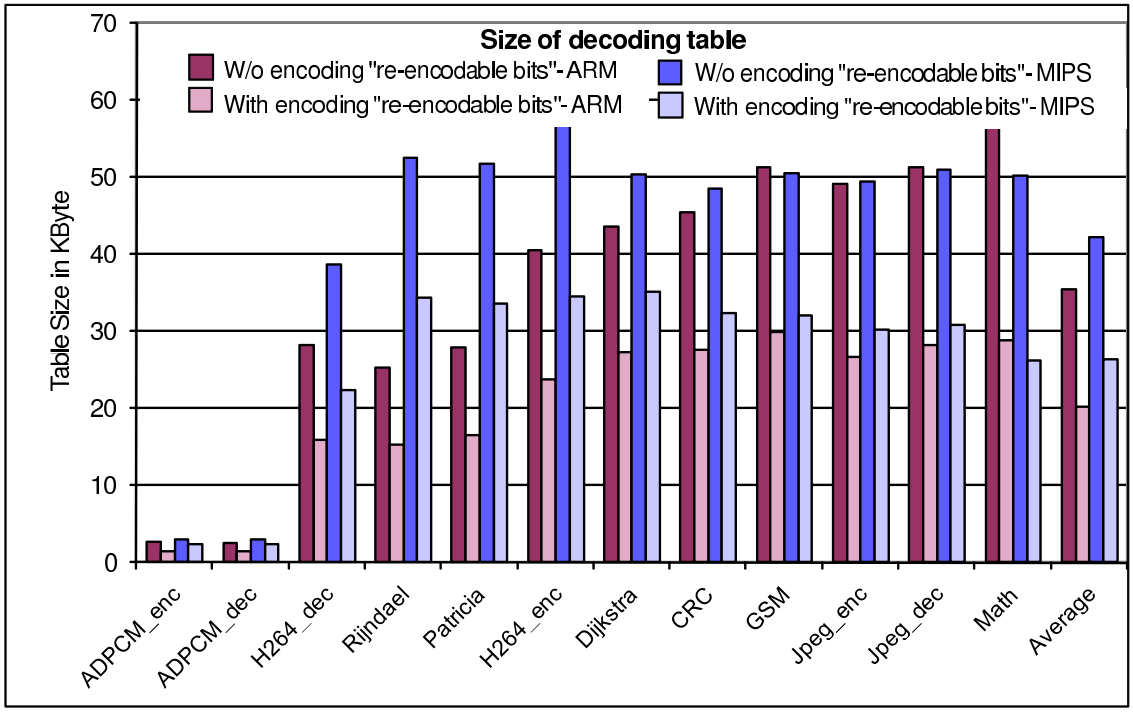


FIGURE 4.38: Size of decoding table for ARM and MIPS processors

2. Encoding the *re-encodable bits* in the instruction format to be identical to the preceding instruction reduces the size of decoding table (as declared in Section 4.2.1.3). To show the efficiency of this technique, we compare the size of the decoding table before and after encoding the *re-encodable bits* in the instruction format. The results are presented in Fig. 4.38. This figure shows that encoding those bits may reduce the size of the decoding table on average by more than 35% for both processors.

3. The compressed instructions include the encoded instructions and the compressed decoding tables. Hence, encoding the *re-encodable bits* reduces the size of compressed instructions because the size of the decoding table is reduced. Consequently, the compression ratio is improved when the *re-encodable bits* are re-encoded (Fig. 4.39). The compression ratios achieved differ between 44% and 49% for the ARM processor (on average 46%) and between 43% and 47% for the MIPS processor (on average 45%), depending on the size of the application and the instruction format of the processor. For large applications, our compression technique gives better results. This has been expected since a large number of instructions result in more *don't care* fields and this gives more reduction in the compressed instruction size. In addition to that, the large number of instructions result in large decoding tables and this gives more chances to re-order their entries and to achieve better table compression.

4. Figure 4.40 shows the time taken by the original and the compressed code (in Million of cycles) for the ARM processor. A performance loss of 13% is the price to pay

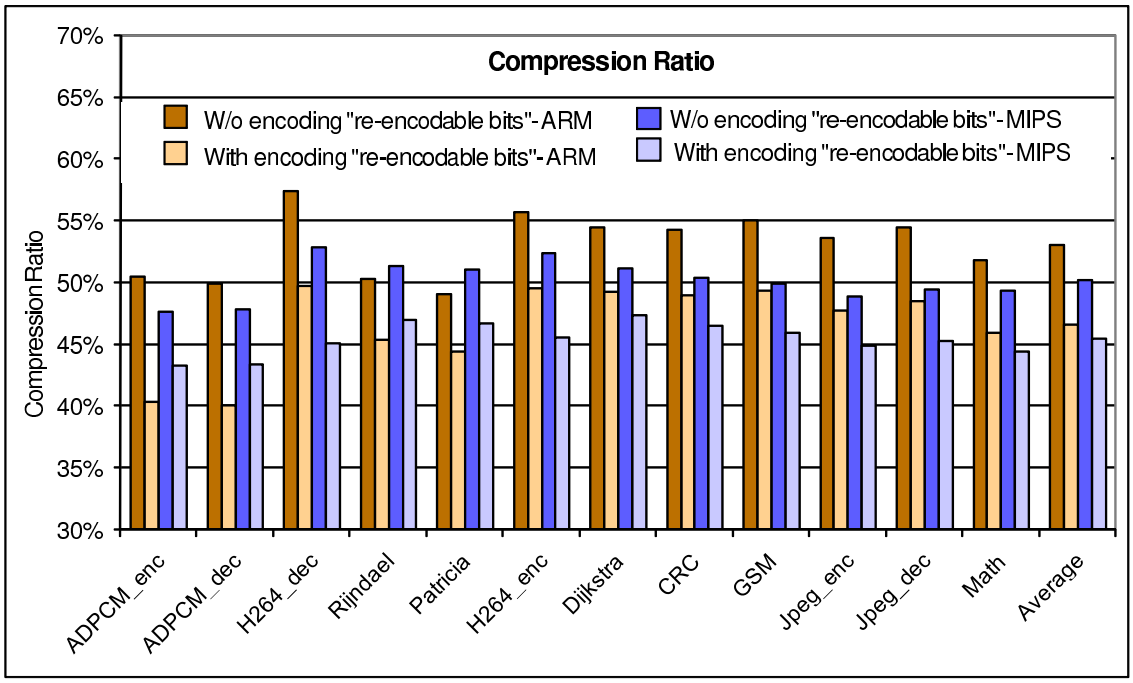


FIGURE 4.39: Compression ratios for ARM and MIPS processors

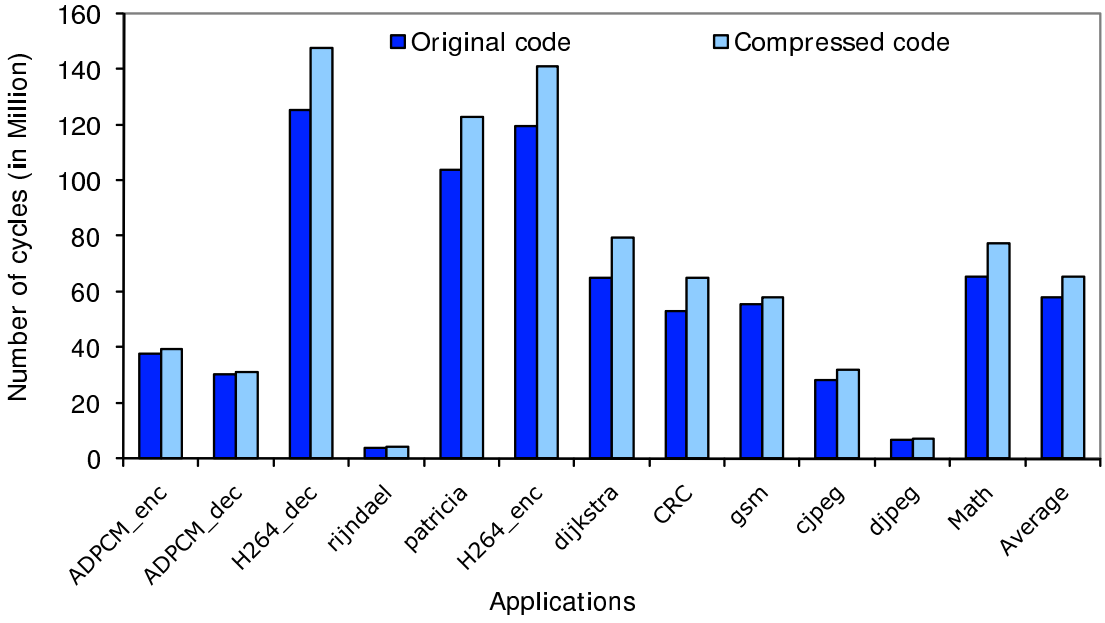


FIGURE 4.40: Time taken by the original and the compressed code for ARM processor (in Million of cycles)

for compressing the instructions. This is because of the hardware decoder which consists of different stages (see Fig. 4.36).

4.3 Discussion

In this section, we discuss the compression results of our code compression techniques; "Look-up Table Compression Technique for *Dictionary-based* schemes" (Section 4.1.1), "Look-up Table Compression Technique for *Statistical* Compression Schemes" (Section 4.1.2), "Instruction Splitting Technique" (Section 4.1.3), and "Instruction Re-encoding Technique" (Section 4.2).

The "Look-up Table Compression Technique" and the "Instruction Splitting Technique" are ISA-Independent techniques. They can be applied to any processor architecture independent from the instruction set format.
The "Instruction Re-encoding Technique" is ISA-Dependent. It is applied to a specific processor architectures as it is based on the instruction format of that processor.

The ISA-Dependent code compression techniques result in better compression ratios in comparison to the ISA-Independent techniques, but on the other hand, the performance of the hardware decoder is better for the ISA-Independent techniques as the decoder passes through less decoding stages than the decoder of the ISA-Dependent techniques.

"Look-up Table Compression Technique" is applied to three processors ARM, MIPS, and PowerPC. It results in better compression results for the applications with more unique instructions. As the number of unique instructions on average is more for MIPS compared to other architectures (Fig. 4.12), this results in that the table compression ratio *TCR* for the MIPS processor being the best (Fig. 4.13. This has been expected since a large number of unique instructions results in a large Look-up Table and this gives more chances to re-order its entries and to achieve better table compression.

As the table compression ratio is the best for MIPS processor, the average compression ratio through all benchmarks is the best for MIPS processor compared to other processor architectures, regardless of the compression technique (i.e. *Dictionary* or *Statistical* compression techniques).

Applying "Look-up Table Compression Technique" to the Statistical compression scheme, achieved better compression ratio than applying it to the *Dictionary-based* scheme. This is because in the Statistical compression scheme, the most frequent occurring instructions are encoded with small codeword lengths and vice versa, but in the *Dictionary-based* scheme, all instructions (most and less frequent occurring) are encoded with the same codeword length.

The average compression ratios achieved using the "Look-up Table Compression Technique for Dictionary-based Compression Scheme" were 59%, 60% and 62% and using the "Look-up Table Compression Technique for Statistical Compression Scheme" were 53%, 51% and 55% for ARM, MIPS and PowerPC, respectively (Section 4.1.2.5).

On the other hand, the maximum frequency of the hardware decoder achieved in the *Dictionary-based* scheme was better than it was in the *Statistical* scheme (330 MHz in the *Dictionary-based* scheme and 280 MHz in the *Statistical* scheme). This is because the hardware decoder in the *Dictionary-based* scheme consists of one part which is the Look-up Table decoder (Section 4.1.1.3), but in the *Statistical* scheme it consists of two parts: the Look-up Table decoder and the Canonical Huffman decoder (Section 4.1.2.4). This will also result in more number of slices used by the hardware decoder in the *Statistical* scheme in comparison to the *Dictionary-based* scheme (600 slices in the *Dictionary-based* scheme and 430 slices in the *Statistical* scheme).

"Instruction Splitting Technique" improved the average compression ratio by reducing the size of decoding table before the Look-up Table compression technique was applying to it. The average compression ratios were achieved for ARM and MIPS processors were 47% and 49%, respectively (Section 4.1.3.4).

The maximum frequency of the hardware decoder were slower than the "Look-up Table Compression Techniques for *Dictionary-based* and *Statistical* schemes" and more number of slices were required. A maximum frequency of 250 MHz and number of slices of 1200 were achieved in addition to 5% loss of the hardware perfromance (Section 4.1.3.3).

The "Instruction Re-encoding Technique" achieved the best compression ratio among all our compression techniques as it is an ISA-Dependent technique, i.e. it is applied for a specific processor. The average compression ratios achieved for ARM and MIPS were 46% and 45%, respectively.

On the other hand, a slower hardware frequency (200 MHz) and more number of slices (1500) were required by this technique in comparison to the "Instruction Splitting Technique". The hardware performance loss was increased to 13% as more time is required for decoding and re-arranging the positions of the opcodes, registers, etc.

For Experiments, we selected ARM, MIPS, and PowerPC processors to evaluate our compression techniques. The reason for that is, these three processors have a commercial implemented code compression techniques. The processors who are using the compression techniques are ARM Thumb, MIPS16, and CodePack for PowePC.
Comparing the results obtained from our code compression techniques with the results

obtained from using ARM Thumb, MIPS16, or CodePack, we find that our compression techniques achieve better compression ratios and performance.

4.4 Comparing to Previous Work

In Table 4.1, we show the average compression ratio through different benchmarks for our RISC code compression techniques and some previous work (presented in Section 3) targeting different processors. Compared to previous work, our compression technique "Instruction Splitting" achieves high compression ratios among other ISA-Independent techniques (on average 47% and 49% for ARM and MIPS processors, respectively). It is independent of the instruction set architecture and can be applied to any RISC processor. Our ISA-Dependent compression Technique "Instruction Re-encoding" also achieves high compression ratios compared to other ISA-Dependent techniques (on average 46% and 45% for ARM and MIPS processors, respectively). It can be generally used with any ISA specification if the *re-encodable bits* in the instruction format are known and extracted.

TABLE 4.1: Comparing our compression techniques to previous art targeting RISC processors

Reference	Compression scheme	Architecture	CR	ISA dependability
Game(IBM) [72]	CodePack	PowerPC	60%-65%	Independent
Lefurgy [28]	Dictionary	PowerPc, ARM	61%, 66%	Independent
Yoshida [27]	Dictionary	ARM	37.5% w/o LUT	Independent
DAS [104, 105]	Dictionary	CR16C	70%	Independent
Lau [99]	Bitmask	Alpha	85%	Independent
Lekatsas [64, 65]	SADC	x86	65%	Independent
Lekatsas [24]	Markov Model	SPARC	56%	Independent
Our LUT Compression Technique, Sec. 4.1.1	**Dictionary**	**e.g. ARM, MIPS, PowerPC**	**59%, 60%, 62%**	**Independent**
Our LUT Compression Technique, Sec. 4.1.2	**Huffman**	**e.g. ARM, MIPS, PowerPC**	**53%, 51%, 55%**	**Independent**
Our Instruction Splitting Technique, Sec. 4.1.3	**Huffman**	**e.g. ARM, MIPS**	**47%, 49%**	**Independent**
Thumb [56] and MIPS16 [55]	Reduced Instructions	ARM, MIPS	70%, 60%	Dependent
Wolfe [19]	CCRP	MIPS	75%	Dependent
Chung [117]	Operand Field Remapping	ARM	46%	Dependent
Corliss [96]	Instruction Operand	Alpha	65%	Dependent
Okuma [106]	Instruction Encoding	DLX	85%	Dependent
Benini [15]	Dictionary	DLX	70%	Dependent
Our Instruction Re-encoding Technique, Sec. 4.2	**Huffman**	**e.g. ARM, MIPS, ...**	**46%, 45%, ...**	**Dependent**

Chapter 5

Code Compression for VLIW Processors

Embedded software has grown in size exponentially in recent years [9]. Two important issues have to be considered in embedded system design, (1) The program memory of an embedded application which may occupy about 3-7 times the silicon area of an embedded processor [98]. Hence, reducing the size of an embedded application becomes important in embedded system design. (2) The performance of the processor becomes very important issue in embedded system design. Hence, selecting a high performance processor may fulfill the requirements of the market.

VLIW processors provide higher performance than RISC processors for a broad range of applications because of its ability to exploit fine-grain instruction-level parallelism. It has been described as a natural successor to RISC, as it moves complexity from the hardware to the compiler, allowing simpler and faster processors.
The drawback of the VLIW processors is the bloating code size of their compiled applications in comparison to the size of the same applications compiled for RISC processors. Fig. 5.1 shows the percentage of the average code size through different benchmarks of MiBench on different processor architectures. The average size of applications compiled for VLIW processor is 60% larger than the size of the same applications compiled for ARM processor.

To host the vast amount of software in an efficient way, code compression may be used which, beside memory size reduction, it can also reduce the power consumption [43, 87]. Therefore, code compression is an important issue for VLIW processors.

Code compression differs from data compression (as explained in Section 2.1) in the size of information that needs to be compressed and decompressed. In data compression, the

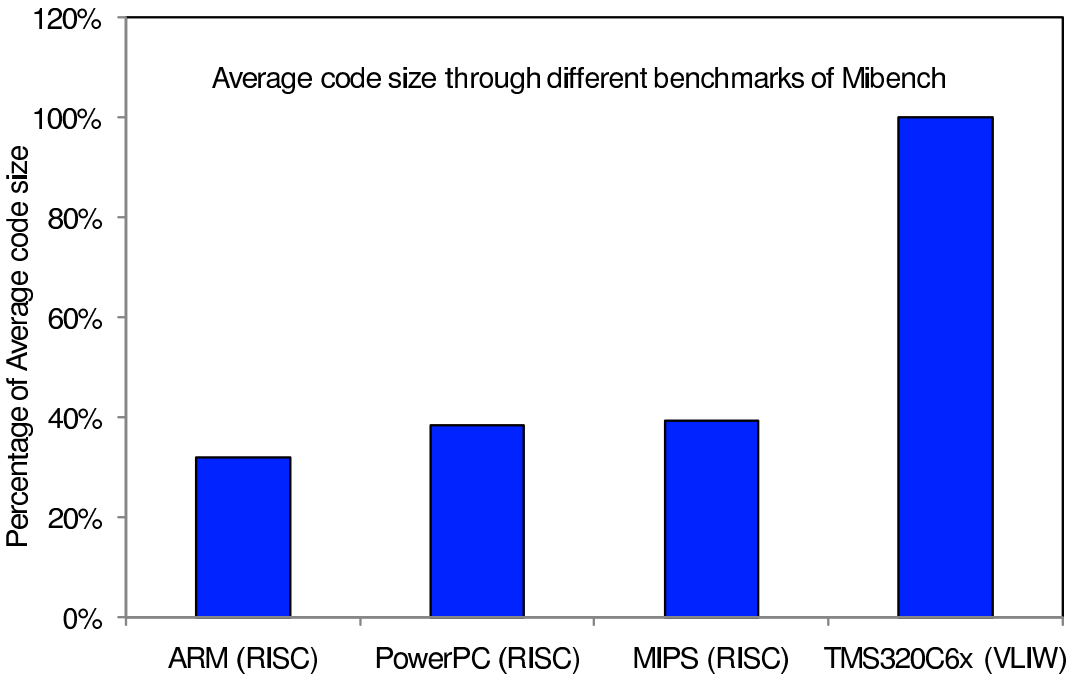


FIGURE 5.1: Percentage of the average code size through different benchmarks of MiBench on different processor architectures

data is compressed and decompressed as a whole (i.e. as one block), while in code compression, small segments or blocks of code (called "compression blocks") are compressed and decompressed individually to ensure *Random access* [1] in decompression. For that, data compression typically results in higher compression ratio than code compression.

The data compression Lempel-Ziv family algorithms [2] use "sliding window" technique (see Fig. 2.5) to match series of bits in the *look-ahead buffer* to string already in the *search buffer* (as explained in Section 2.3.1.2).
At the beginning of compressing each compression block, the *search buffer* is empty. Hence, the first series of bits in the *look-ahead buffer* will find no match in the *search buffer* and remain without compression. This will impact negatively on the final compression ratio.

In this chapter, we *explicitly* reduce the size of compressed instructions by using our novel technique (we call it Filled Buffer Technique) which can be applied to any compression algorithm of Lempel-Ziv family, targeting any VLIW processor architecture.
The Filled Buffer Technique fills in the *search buffer* with series of bits (patterns) at the beginning of compressing each compression block. Therefore, the bits in the *look-ahead buffer* may be compressed because they may match the bits in the *search buffer*. This

[1] For example, branching and function entry points must be able to be decompressed on demand
[2] LZ77, LZR, LZSS, LZB, LZH, LZ78, LZW, LZC, LZT, and LZJ

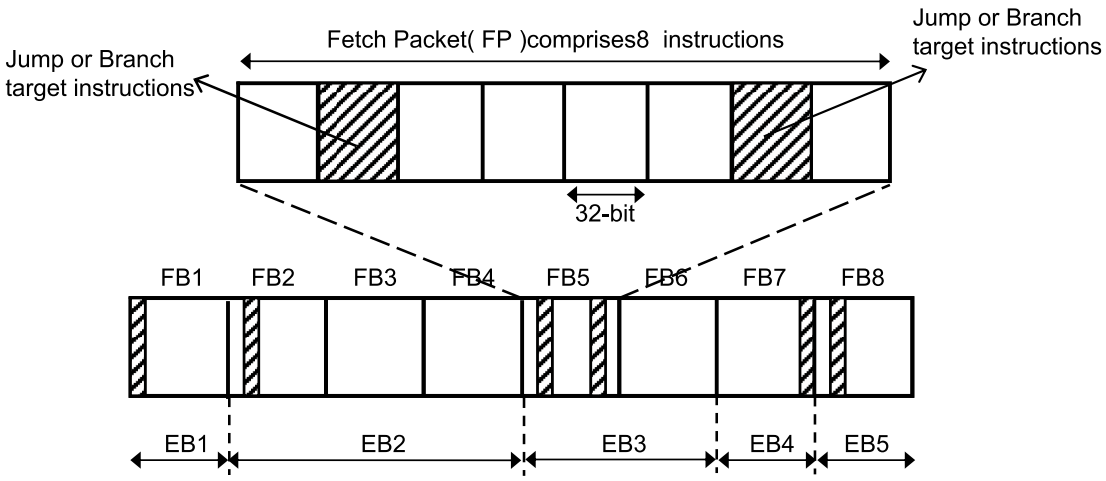


FIGURE 5.2: Example for exploring the Extended Blocks (EB) form "Fetch Packets" (FP) for compression

technique improves the compression ratio (on average) by more than 13% in comparison to the previous techniques (see Section 5.3).

We select from the LZ (Lempel-Ziv) family algorithms an efficient data compression algorithm called Deflate Algorithm [107] to apply our Filled Buffer Technique to it (more details about the Deflate Algorithm are presented in Chapter 2.3.1.3). It is based on the LZSS Algorithm (Chapter 2.3.1.3) combined with Huffman Coding (Chapter 2.3.2.1). We select the Deflate Algorithm as it is common compression technique which is originally used in Zip and Gzip software to compress files (data).

In this chapter, we introduce the Deflate Compression Algorithm and our Filled Buffer Technique.
To show the orthogonality of our technique, we apply it to another algorithm from the Lempel-Ziv family which is called LZMA Algorithm [86] and show the efficiency of using our technique. We also compare our compression results with the results of a previous work "V2F" [77] (as this technique achieves very high decoding throughput), and show the improvement in compression ratio because of using our Filled Buffer Technique. We conduct experiments for two VLIW embedded processors, namely TMS320C62x and TMS320C64x (more details about these processors are presented in Section 2.4.2). For both architectures the MediaBench [21] and MiBench [20] benchmark suites are served as a representative set of applications.

5.1 Our Code Compression Technique

Deflate Algorithm gives high compression ratio with the requirement of long texts (i.e. data compression). But when it is used to compress program code, it is applied to small compression blocks such as *Basic Blocks* (BB) individually. This will shrink the compression ratio as the probability of occurrence for instructions is reduced. Applying a compression algorithm on larger compression blocks improves the final compression ratio (as explained in Section 5.3).
In the VLIW processor, if a branch target instruction is appeared in the middle of the "Fetch Packet", only the instructions which follow the branch target will be executed. Hence, we may extend the compression blocks to have more instructions than the *Basic Blocks*. We call the new explored blocks *"Extended Blocks"* which may be explored according to the following rules:
(1) The *Extended Block* may contain one or more complete "Fetch Packets".
(2) The branch target instruction should only exist in the first "Fetch Packet" of the *Extended Block*.
(3) The first "Fetch Packet" of the *Extended Block* may contain more than one branch target instruction.

The new *"Extended Blocks"* may be used for any compression technique and are generic to any VLIW processor architecture.

Fig. 5.2 shows an example for exploring the *Extended Blocks*. In this figure there are 8 "Fetch Packets" (FP1 - FP8). The dashed area refers to a branch target instruction. The number of *Basic Blocks* (BB) is 6 (because of the 6 branch target instructions). The first *Extended Block* (EB1) contains only one "Fetch Packet" (FP1) because the next branch target instruction appears directly in the next "Fetch Packet" (FP2). The second *Extended Block* (EB2) contains 3 "Fetch Packets" (FP2 - FP4), and so on. The number of *Extended Blocks* in this example is reduced to 5.

In the next sections, we introduce the Deflate Algorithm and our Filled Buffer Technique.

5.1.1 Deflate Compression Algorithm

Deflate Algorithm [107] is a data compression algorithm that is originally used in the Zip and Gzip software to compress text files (data). It is based on an optimized version of LZ77 Algorithm (which is called LZSS Algorithm) combined with Huffman codes (See top part of Fig. 5.3).

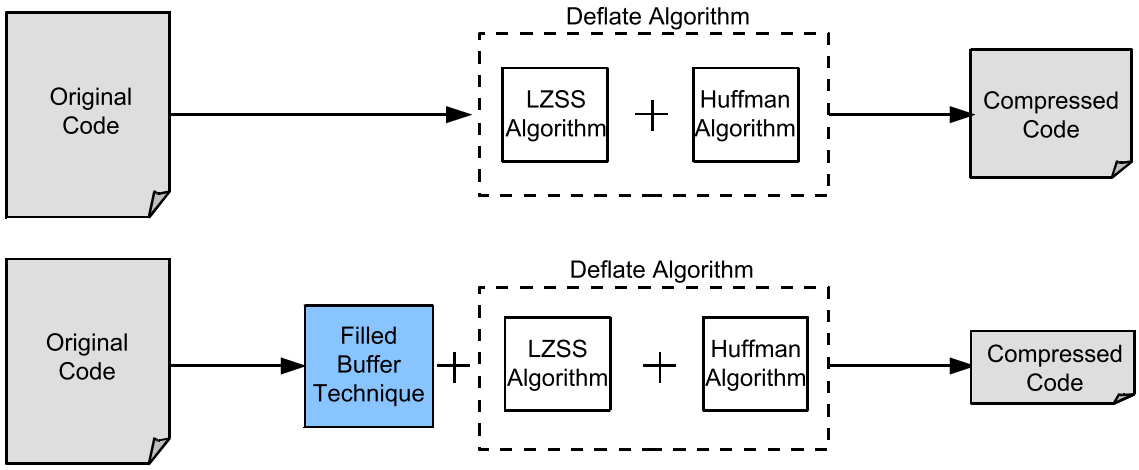


FIGURE 5.3: Steps for our code compression technique

The principle of the LZSS Algorithm (see Algorithm 9) is to use a part of the previously-seen input stream as the dictionary. The encoder shifts the input in that window from right to left as strings of bits (patterns) are being encoded (line 2). Thus, the technique is based on a *sliding window*. The window (see Fig. 5.4) is divided into two parts. The part on the left side is the *search buffer*. This is the current dictionary, and it includes patterns [3] that have recently been input and encoded (each symbol denotes 8-bit pattern of instruction). The part on the right side is the *look-ahead buffer*, containing patterns yet to be encoded.

let x$[0...N-1]$ be the input string. Assume also the prefix x$[0...i-1]$ has been compressed so far. The dictionary at this moment consists of all the substrings x$[i-j...k]$ where j$\in [1,...,M]\,, k \in [i-j,...,i-j+F-1]$ and M, F are two parameters of the algorithm. The next step is to find the longest prefix of x$[i...N]$ which matches an entry of this dictionary. If this prefix is of length r and $x\,[i-q...i-q+r-1]$ is the matching string in the dictionary (q$\in [1...M]$), then we replace the prefix x$[i...i+r-1]$ with the pointer (q,r) (it is called "offset", "length") and we proceed to the position $i+r$ of the input string. Notice that if the character x_i does not occur within the last M preceding characters, then we cannot find any matching prefix at position i. In this case, we leave the character x_i without compression (it is called "literal"). The algorithm 9 adds the flag '0' to the "length" (line 6) and the flag '1' to "literal" to distinguish between the uncompressed and compressed ones. Finally the algorithm returns the compressed stream (line 10) which contains the uncompressed patterns ("literals") and the compressed ones pair of("offsets","lengths").

Fig. 5.4 shows an example for compressing Extended Block which contains 4 instructions using the LZSS Algorithm.

[3] pattern is a string of consecutive bits of instruction

These instructions compose the string of symbols "ABEABCABCDBCAAAA" (each symbol denotes a pattern which is a 8-bit of the instruction). The encoder in this example outputs the compressed stream "ABE(3,2)C(3,3)D(6,3)(1,3)". The encoder adds 1-bit flag at the beginning of each compressed- uncompressed pattern.
We found through different sizes of benchmarks, that the pattern of 8-bit gives better compression ratio than other pattern length (see experimental results in Section 5.3). We found also that the number of instructions in any Extended Block does not exceed 256 (i.e 1024 8-bit patterns), therefore, in the LZSS Algorithm we selected the size of the *search buffer* to be 1024 Byte. The size of the *look-ahead buffer* has no impact on the compression ratio. For that, we use the default size of the *look-ahead buffer* in the LZSS Algorithm which is 256 Byte. In this case, the "offset" will be encoded with 10 bits, the "length" and "literal" will be encoded with 8 bits plus i bit flag, each.

The second step of the Deflate Algorithm is to encode the compressed stream using three different models of Huffman Coding:
Static-Static Huffman Tables:. This is the standard model of the Deflate Algorithm, in which two static code tables are prepaid for encoding, previously. One to encode the "literals" and "lengths" and the other to encode the "offsets". The encoder replaces the codes that's written on the compressed stream with the new codes of the tables.
Dynamic-Static Huffman Tables: In this model, we create Huffman table for the "literals" and "lengths" only. For the "offsets", we use the prepared static table.
Dynamic-Dynamic Huffman Tables: In this model, we create two Huffman tables, one for "literals" and "lengths" and the other for the "offsets". Depending on their frequency of repetition, shorter code words are used for the most frequent patterns, whereas longer code words are used for the less frequent ones.

Algorithm 9 LZSS Compression Algorithm

/* *pat*: pattern which is consecutive bits of instruction (pat is 8-bit long) */
/* LB: Look-ahead Buffer */
/* SB: Search Buffer */

Function LZSS (*pat*) **{**
pat >> LB {shift patterns in the Look-ahead Buffer}
if *pat* in SB match *pat* in LB **then** {pattern is compressed}
    find $offset$ and *length*
    pat ⇒ ($offset$,*length*) {pat is compressed as ($offset$,*length*)}
    length + '0' {add '0' as flag for uncompressed}
else {pattern is not compressed}
    pat + '1' ⇒ *literal* {uncompressed pat and flag is called *literal*}
end if
return (*literal*,$offset$,*length*) {return compressed stream}
}

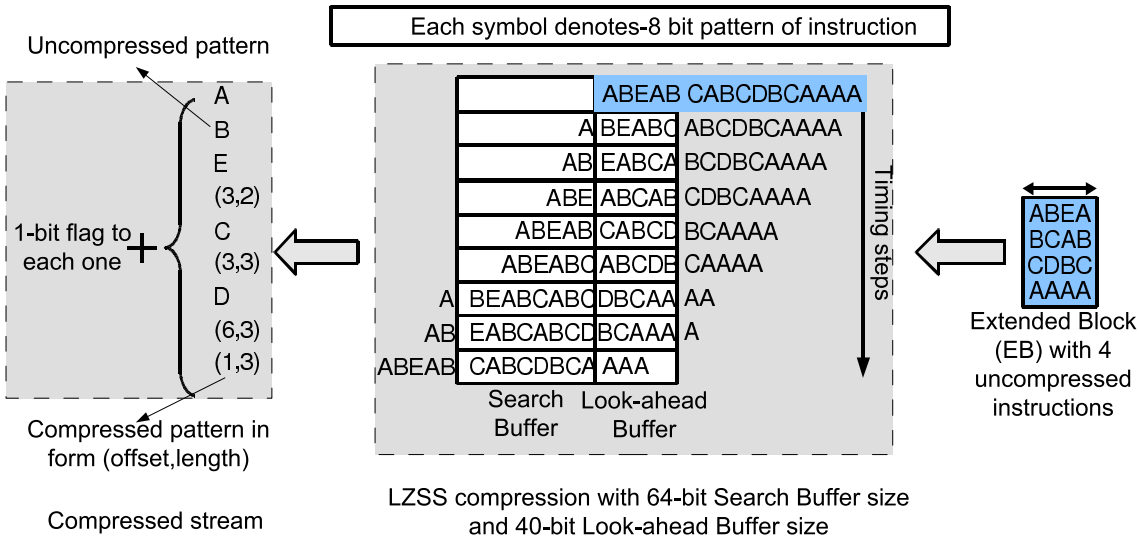


FIGURE 5.4: Example for compressing an Extended Block using the LZSS Algorithm

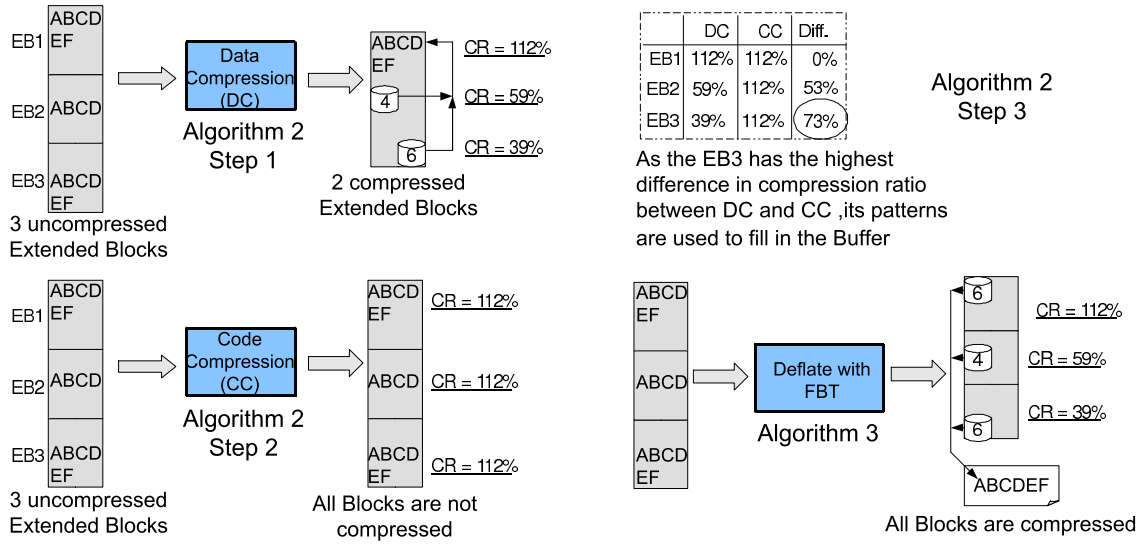


FIGURE 5.5: Example to compress 3 Extended Blocks using the Deflate Algorithm in conjunction to the Filled Buffer Technique

5.1.2 Our Filled Buffer Technique

Our main goal in this work is to decease the overhead in the code size which is arisen because of using code compression rather than data compression and to improve the final compression ratio. As the *search buffer* is empty at the beginning of encoding each *Extended Block*, the first patterns in the *look-ahead buffer* will find no match in the *search buffer* and remain without compression. This patterns unmatching occurs only one time in data compression (at the beginning of compressing the whole block) but in the code compression it occurs as many as there are *Extended Blocks*.

After compression, the uncompressed pattern ("literal") will have one bit (the flag bit)

Algorithm 10 Filled Buffer Technique (three steps)

/* i: 32-bit instruction */
/* EB: Extended Block */
/* CR_1: Compression ratio in case data compression*/
/* CR_2: Compression ratio in case code compression*/
/* diff: Difference in Compression ratio between CR_1 and CR_2 */

/* Compress whole application as one Block (data compression) */

1: **for all** instructions i in the application **do** {partitioning}
2: partition i into 4 *pat* {each pattern is 8-bit long}
3: **end for**
4: **Call** LZSS (*pat*) {call function LZSS in Algorithm 9}
5: **for all** patterns *pat* in application **do** {compute CR_1 for each *pat*}
6: CR_1 = compressed pattern size(in bits)/8
7: **end for**

/* Compress each EB in application separately (code compression)*/

1: **for all** Extended Blocks EB in the application **do**
2: **for all** instructions i in the EB **do** {partitioning}
3: partition i into 4 *pat*
4: **end for**
5: **Call** LZSS (*pat*) {call function LZSS in Algorithm 9}
6: **for all** patterns *pat* in EB **do** {compute CR_2 for each *pat*}
7: CR_2 = compressed pattern size(in bits)/8
8: **end for**
9: **end for**

/* Compute the difference in compression ratios for each pattern */

1: **for all** patterns *pat* in application **do** {compute diff}
2: diff= $CR_2 - CR_1$ {Compute the difference for each pattern}
3: **end for**
4: sort patterns *pat* by diff descendingly
5: select the first 1024 sorted patterns to fill in the SB

Algorithm 11 Deflate Algorithm with the Filled Buffer Technique

1: **for all** Extended Blocks EB in the application **do**
2: SB $\Leftarrow$ sorted pat {fill in Search Buffer with first 1024 sorted patterns}
3: **Call** LZSS (*pat*) {call function LZSS in Algorithm 9}
4: **end for**
5: Compress the compressed stream using Huffman Coding

more than the original one and this will consequently increase the overhead each time there is a new *Extended Block*. To decrease this overhead, we use a novel and efficient technique (we call it Filled Buffer Technique) before applying the compression algorithm (see the bottom part of Fig. 5.3). In this technique the encoder fills in the empty *search buffer* (off-line) with selected patterns at the beginning of encoding each *Extended Block*. For that, the patterns in the *look-ahead buffer* may match those filled patterns in the *search buffer* and then may be compressed with a pairs of ("offset", "length").

To find those patterns which may be used to fill in the *search buffer*, we use Algorithm 10 which basically consists of three steps. In the first step, we compress the whole application as one block (i.e. data compression) using the LZSS Algorithm (lines 1-4) and then we compute the compression ratio for each pattern of all instructions (lines 5-7). In the second step, we compress each *Extended Block* of the application separately (i.e. code compression) using the LZSS Algorithm (lines 1-5) and then we compute the compression ratio of each pattern in the *Extended Block* (line 7). In the third step, we compute the difference in compression ratios of patterns between the first and the second steps (lines 1-3). This difference implies the overhead due to using the LZSS Algorithm as code compression technique rather than using it as data compression technique. Then, we sort the patterns by their difference in compression ratios descendingly (line 4). As the size of the *search buffer* is 1024 Byte, we just need to select the first 1024 sorted patterns to fill in the *search buffer* at the beginning of compressing each *Extended Block*.

To compress the *Extended Blocks* using the Deflate Algorithm in conjunction to our Filled Buffer Technique, we use Algorithm 11. In this algorithm, and before compressing each *Extended Block*, we fill in the *search buffer* with the first 1024 sorted patterns (from Algorithm 10). Then, we compress the patterns using the LZSS Algorithm (Algorithm 9) in line 3 and Huffman Coding (line 5).

Fig. 5.5 shows an example for the Filled Buffer Technique. Assuming that we have three *Extended Blocks*, we want to find the patterns which are required to fill in the *search buffer* before compressing each *Extended Block*. According to Algorithm 10, we first compress the three blocks together using the LZSS Algorithm as data compression technique. As the *search buffer* is empty at the beginning of the compression, the patterns "ABCDEF" in EB1 will be left without compression but the patterns "ABCD" in EB2 and 'ABCDEF" in EB3 will find a match in the *search buffer* and will be compressed using the pair ("offset", "length"). Then, we compress each *Extended Block* using the LZSS Algorithm, separately. As the *search buffer* is empty at the beginning of compressing each *Extended Block*, the patterns in each *Extended Block* will find no match and will be left without compression. Each uncompressed pattern will have one extra flag bit to

show that it is not compressed. Considering that each pattern has 8-bit long and the "offset" is encoded in 10-bit, the compression ratio for any uncompressed pattern will be 112% but for each pattern in EB2 and in EB3 will be 59% and 39%, respectively. The differences in compression ratios between each pattern in EB1, EB2 and EB3 will be 0%, 53% and 73%, respectively. As the patterns in the EB3 achive the highest difference in compression ratios, we select its patterns to fill in the *search buffer* at the beginning of compressing each *Extended Block*. Compressing the *Extended Blocks* after applying our Filled Buffer Technique will improve the compression ratio for the *Extended Blocks* (EB2 and EB3) to be 59% and 39%, respectively.

5.2 Decompression Architecture Design

To decode the compressed instructions which are compressed using the Deflate Algorithm, we use two sequential decoders (Fig. 5.6), the Huffman decoder and the LZSS decoder

5.2.1 Huffman Hardware Decoder

An efficient way to store the Huffman Tables is to use the Canonical Huffman Tables [39]. Each table stores the codes of the same length contiguously. To decode these tables we derived the hardware decoder from [2] and optimized it to improve its throughput. The hardware architecture is illustrated in Fig. 5.6. It consists of four components: shift register, comparators unit, Look-Up Table for each code length and multiplexer. We optimized the comparators unit and Look-Up Tables to be integrated in one pipeline stage. The optimized decoder decodes the Huffman codes in three phases (three pipeline stages). In the first phase, the shift register receives a 32-bit compressed Huffman code and shifts its contents by the length of previous decompressed code (in bits). The shift register outputs k-bit equal to the longest Huffman code in the Look-up Tables. In the second phase, the k-bit output of the shift register is transferred to the comparators unit and to the Look-up Tables simultaneously. The number of comparators and look-up Tables is equal to number of different code length. The incoming k-bit is compared in each comparator to the maximum code of its length and the outputs of these comparators control the multiplexer output in the third phase. In parallel to the comparators unit, the incoming k-bit is also transferred as indices to the Look-up Tables (according to the length of each table). As Huffman codes are prefix free, the output of only one Look-up Table will be considered. In the third phase, the multiplexer chooses one output of the Look-up Tables according to the control signal which is received from the comparator unit.
The Huffman decoder outputs 10-bit code which may be a "literal" (9-bit), "length" (9-bit) or "offset" (10-bit). For that, the main task for the LZSS decoder is to decode the

content of the 10-bit output of the Huffman decoder ("literal", "length" or "offset") and then to build the buffer in order to retrieve the original instructions.

5.2.2 LZSS Hardware Decoder

LZSS decoder (Fig. 5.6) consists of a special multiplexer and a shift register. The input of the multiplexer is connected to the output of the Huffman decoder. LZSS Decoder decodes the instructions in three phases. In the first phase, The multiplexer receives the 10-bit code from the hardware decoder. Its main task is to analyze the incoming code to decode its content, i.e. "literal", "length" or "offset". The "literal" and "length" are 8-bit long each (from bit0 to bit7). In each of them, the bit8 is a flag bit and the bit9 is not used. The "offset" is 10-bit long. Fig. 5.7 shows the state diagram of the LZSS decoder to decode the incoming 10-bit code form the Huffman decoder. When a new 10-bit code is arrived, the Bit number 8 is checked. If it is '0', that means the code is a "literal" and the multiplexer transfer the 8-bit directly to the shift register (in the second phase). If the bit8 is '1', then the code is "length" and the next 10-bit code will be "offset". In this case the multiplexer keeps the 8-bit "length" and waits for the next 10-bit "offset". When it receives the "offset", it transfer the pair ("offset", "length") to the shift register. In the third phase, the shift register builds the *look-ahead buffer* and generates the original instructions.

We designed both of decoders in VHDL and implemented them using Xilinx ISE9.2 for scalable FPGA platform *"Platinum"* from *Pro-Design*. An average access time of 3 ns was achieved and just around 800 slices were used.

5.3 Experiments and Results

In this section, we present the experimental results of our Filled Buffer Compression Technique. We conducted experiments for two VLIW processors (Texas Instruments), namely TMS320C62x and TMS320C64x. For both architectures, different sizes of benchmarks from MediaBench [21] and MiBench [20] are served as a representative set of applications. We compiled and linked the applications using the Code-Composer-Studio (CCS) from Texas Instruments [118], and we used the simulator "c6xsim" [108] to get the performance results. The experimental results are shown in figures 5.8 - 5.15 and explained in the following sections. The bar labeled *"Average"* shows the average across all benchmarks in that diagram.

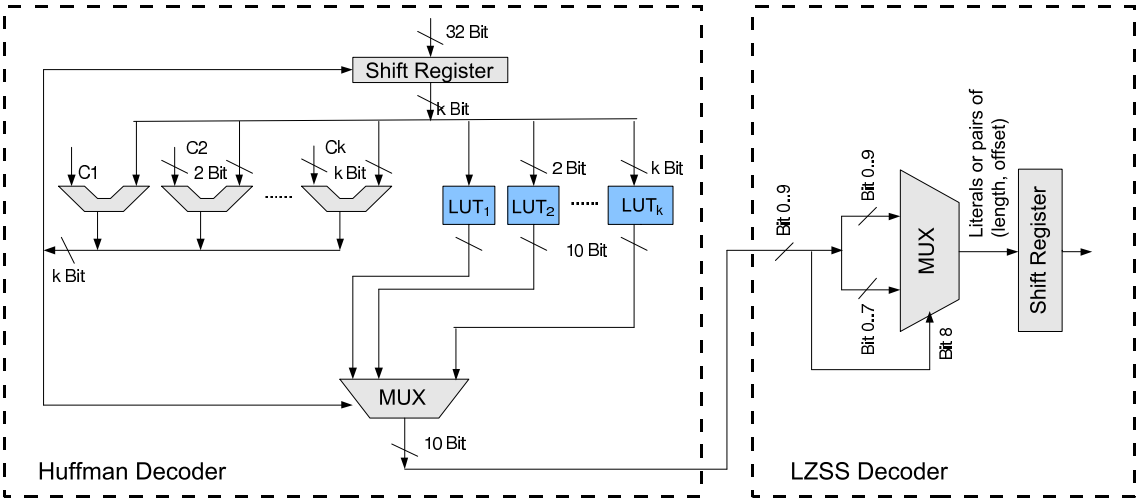


FIGURE 5.6: Hardware Decoder

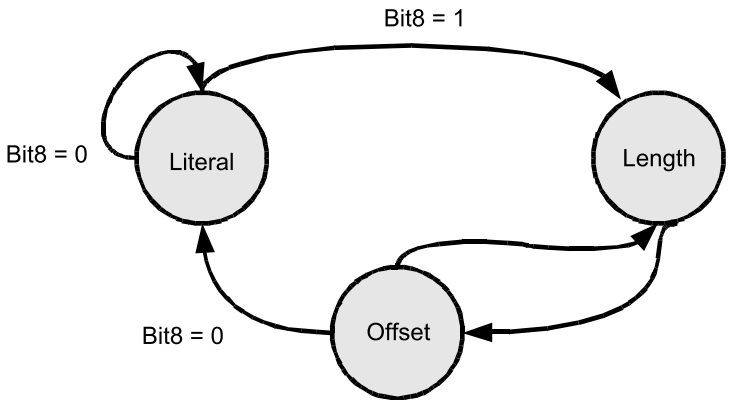


FIGURE 5.7: State diagram of the LZSS decoder

5.3.1 Statistics of the Benchmarks

Fig. 5.8 shows the number of total instructions, Basic Blocks, and Extended Blocks for different benchmarks compiled for TMS320C62x and TMS320C64x VLIW processors. This figure shows that the difference in number of instructions for both processors is between 12% and 18%. On average, the number of instructions for the benchmarks compiled for the C64x processor is 16% less than the C62x processor. This is because the C64x processor is a developed version of the C62x one (as explained in Sec. 2.4.2). Fig. 5.8 shows also that the number of the *Extended Blocks* (on average) are 43% and 65% less than the number of the *Basic Blocks* for C62x and C64x processors, respectively. This shows the importance of applying the compression technique on the *Extended Blocks* other than the *Basic Blocks*.

Benchmark	C62x			C64x		
	# total instructions	# Basic Blocks	# Extended Blocks	# total instructions	# Basic Blocks	# Extended Blocks
fft	11208	935	447	9296	679	284
basicmath	13184	1107	467	10752	828	299
unepic	27728	2293	1260	23600	1336	527
mpeg2enc	47168	3828	1952	39936	2064	606
rdjpgcom	68528	4613	2640	59912	1731	599
wrjpgcom	82632	5445	3154	72464	1915	654
djpeg	83920	5564	3223	73608	1968	663
jpegtran	90832	5982	3491	79792	2063	686
ansi2kr	91776	6074	3563	80560	2079	688
Average	57442	3982	2244	49991	1629	556

FIGURE 5.8: Number of original instructions, Basic and Extended Blocks for the benchmarks compiled for C62x and C64x processors

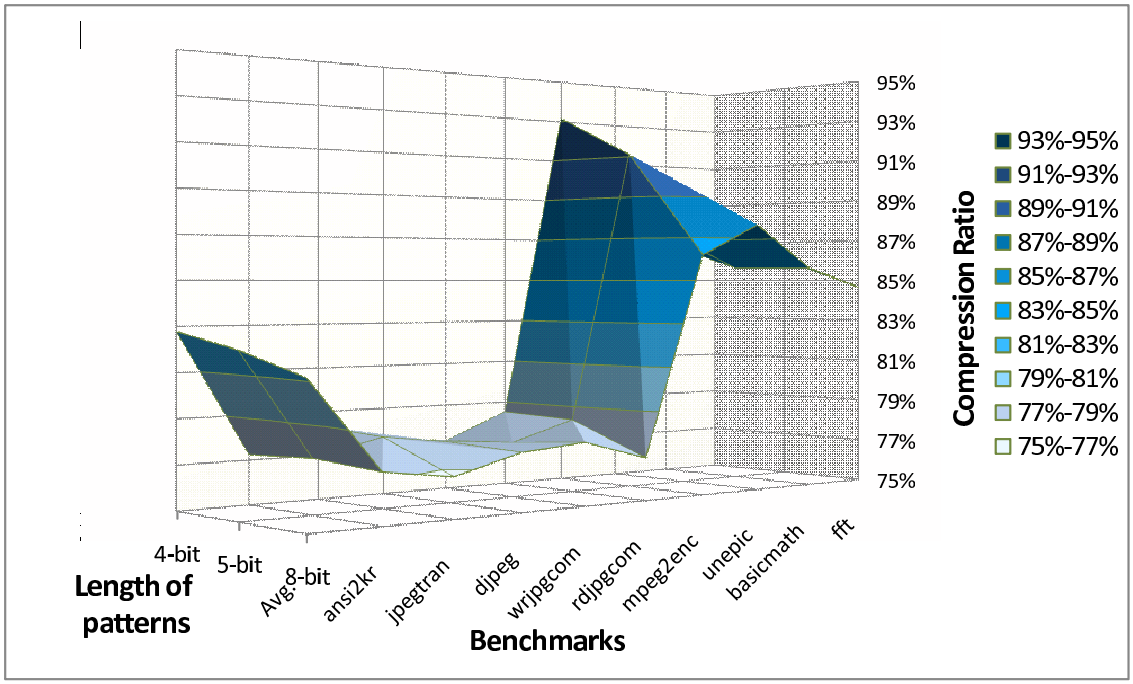


FIGURE 5.9: Compression Ratios for different Benchmarks compressed using the LZSS Algorithm for 4-bit, 5-bit and 8-bit pattern length

5.3.2 Compression Ratios Using the Deflate Algorithm

Figures 5.9 shows the compression ratio when the LZSS Algorithm is used for 4-bit, 5-bit, and 8-bit pattern lengths. The pattern of 8-bit gives (on average) better compression ratio than other pattern lengths (82.78%, 82.11%, and 81.61% for 4-bit, 5-bit, and 8-bit, respectively). For that, we selected the 8-bit length pattern to be compressed as one symbol in the Deflate Algorithm. Patterns with more than 8-bit long improves the results slightly, but on the other hand, the encoder requires more time for compression.

Figures 5.10 and 5.11 show the compression ratios for different benchmarks using the three models of the Deflate Algorithm for the C62x and C64x processors respectively. The first bar of each benchmark in these figures shows the compression ratio when the first model of the Deflate Algorithm (Static-Static Huffman Tables) is used as a data

compression technique, i.e. by compressing the whole application as one compression block. The second, third, and fourth bars show the compression ratios for the first model (Static-Static Huffman Tables), the second model (Dynamic-Static Huffman Tables), and the third model (Dynamic-Dynamic Huffman Tables) of the Deflate Algorithm, respectively.
In each of these bars, the bottom part shows the results by using our Filled Buffer Technique in that model. The whole bar presents the results without using our technique. From these figures, we can observe that the data compression ratio is improved when the size of the benchmark (i.e. the number of instructions) is increased. This is because the patterns in the *look-ahead buffer* (for the large size benchmarks) may have higher chance to find a match in the *search buffer* than the smaller size benchmarks. The matched patterns then may be compressed using the pair ("offset", "length"), and better compression ratio may be achieved.
The average data compression ratios for both processors are 56% and 58%.
The difference between the first and second bars of each benchmark in figures 5.10 and 5.11 shows the overhead which occurs because of using the code compression technique (i.e. applying the compression technique to each compression block separately) in comparison to data compression (i.e. applying the compression technique to whole application as one block). The overhead (on average) is 27% for the C62x and 13% for the C64x.

Using our Filled Buffer Technique in the top of the Deflate Algorithm improved the compression ratios for the processors C62x and C64x (on average) by 12% and 9% in the first model, and by 13% for both processors in the second model, and by 14% and 11% in the third model, respectively.
The final compression ratios (using our Filled Buffer Technique) for both processors are on Average 71% and 62% in the first model, 62% and 56% in the second model, and 61% and 56% in the third model, respectively.

We can conclude from the figures 5.10 and 5.11 that the "Dynamic-Static Huffman Tables" model gives better compression ratio results for small size benchmarks such that the fft and the Basicmath (which have less than 20000 instructions). For the largest benchmarks (which have more than 20000 instruction word), the "Dynamic-Dynamic Huffman Tables" model is the best choice for compression from the perspective of compression ratio.

5.3.3 Compression Ratios Using the LZMA Algorithm

To show the orthogonality of our Filled Buffer Technique, we applied it to another data compression algorithm of the (Lempel-Ziv) family Which is called the LZMA Algorithm. LZMA Algorithm (Lempel-Ziv-Markov chain-Algorithm) [107] is a data compression algorithm which is based on the LZSS Algorithm combined with Range Coding. The *Extended*

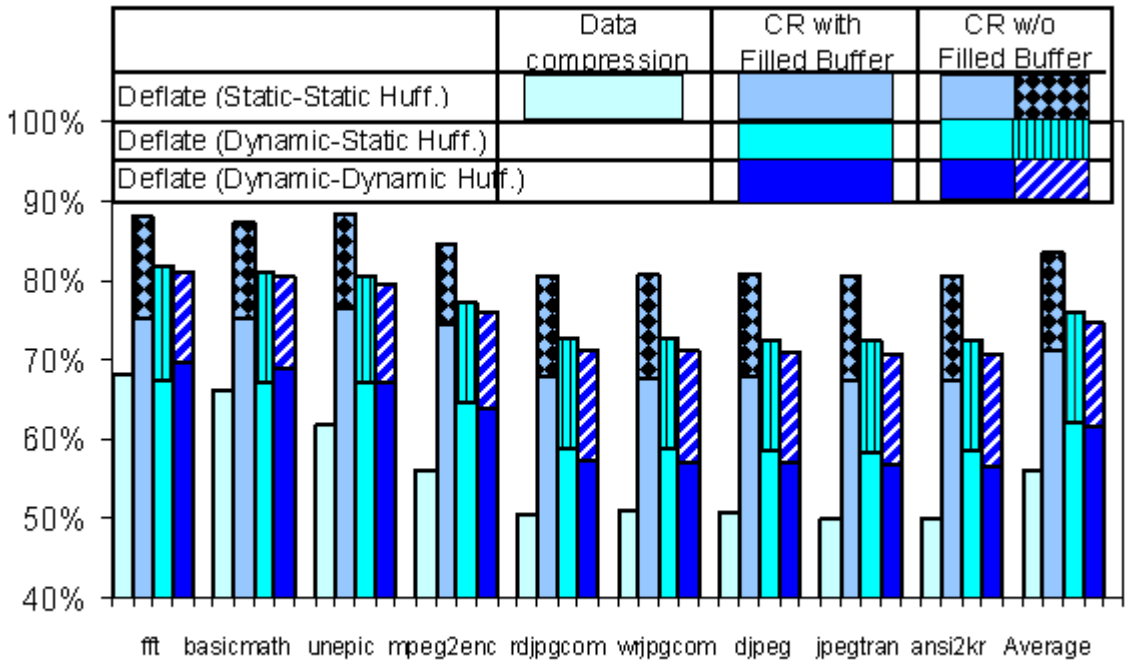


FIGURE 5.10: Compression ratios using three different models of the Deflate Algorithm for C62x VLIW processor

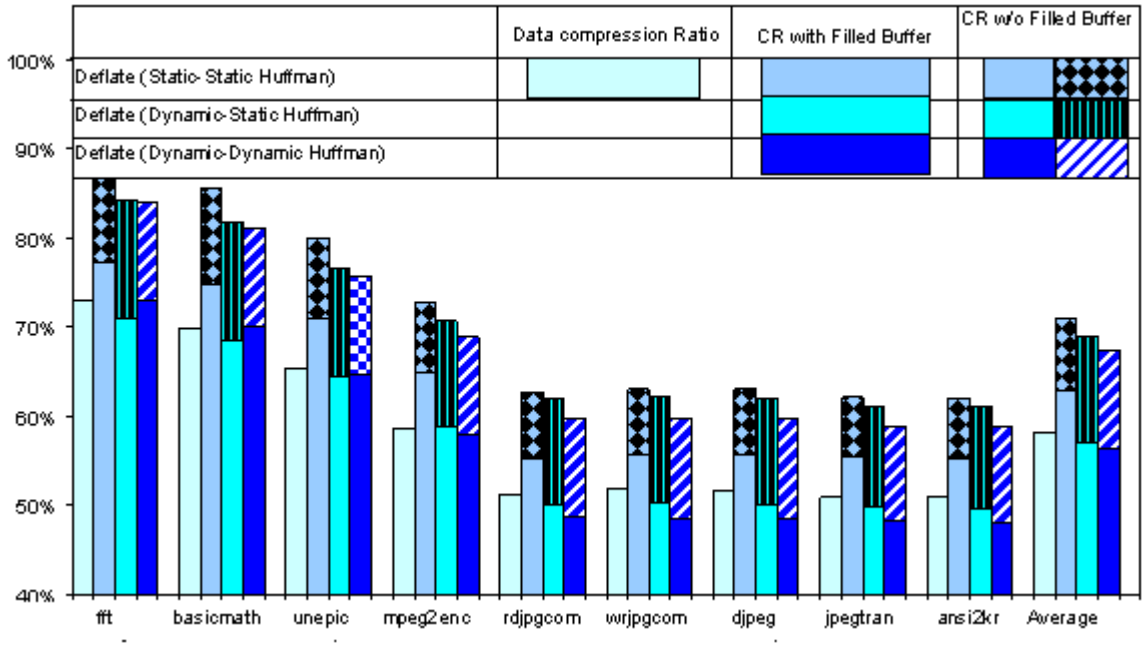


FIGURE 5.11: Compression ratios using three different models of the Deflate Algorithm for C64x VLIW processor

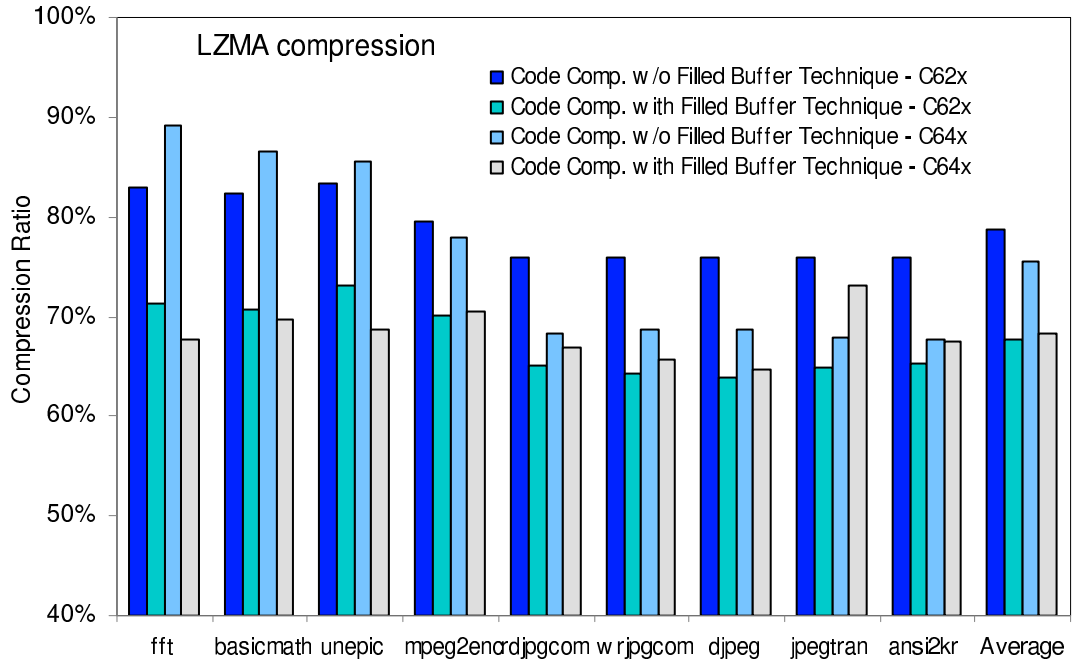


FIGURE 5.12: Compression ratios using the LZMA Algorithm for C62x and C64x VLIW processors

Blocks are compressed using the LZSS Algorithm (as explained in Sec. 5.1.1), and the obtained compressed stream is then encoded using Range Coding.

Fig. 5.12 shows the compression ratio results for both processors (C62x and C64x). Using the Filled Buffer Technique at the top of the LZMA Algorithm improved the compression ratios on average by 13% and 7% for both processors. The final compression ratio of 67% and 68% were achieved for the C62x processor and the C64x processor, respectively.

Comparing the results of the Deflate and the LZMA algorithms, we find that the LZMA Algorithm achieves better data compression ratios, but the Deflate Algorithm achieves better compression ratios.

5.3.4 Performance

The hardware decoder (Huffman and LZSS) decodes the compressed instructions in six phases which are executed in three clock cycles. Fig. 5.13 Shows the summary for these phases. Each "literal", "length" and "offset" are compressed using Huffman codes. Hence, the performance of the decoder depends on the number of the Huffman compressed codes.

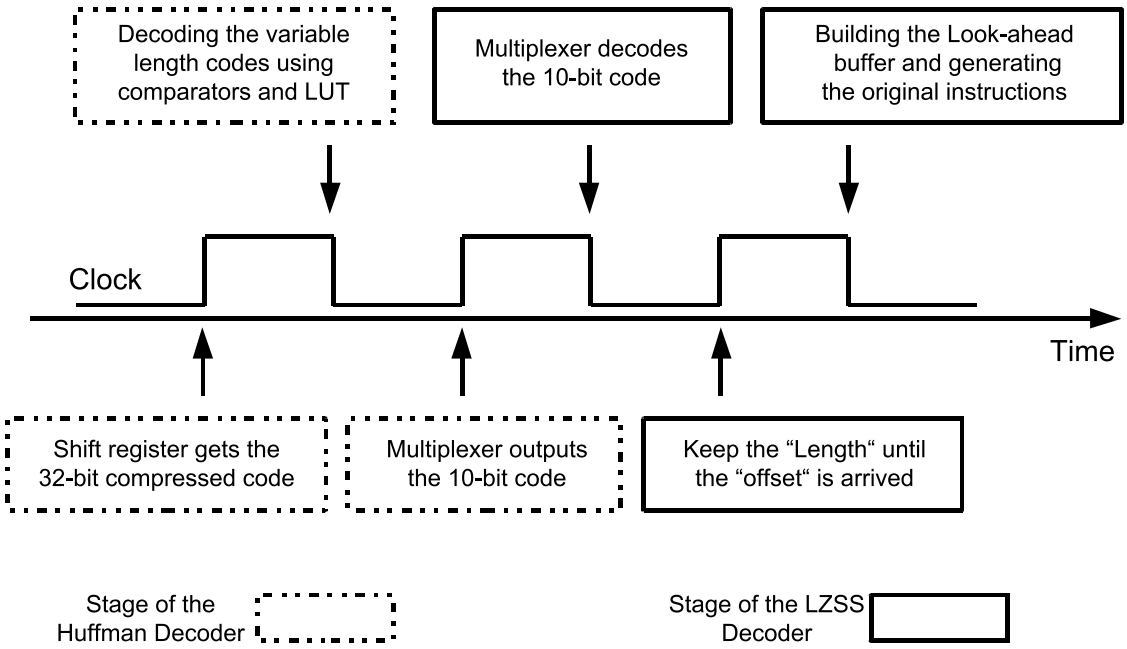


FIGURE 5.13: Pipeline stages of the Deflate hardware decoder

Fig. 5.14 shows the performance of the hardware decoder for the C62x and the C64x processors, respectively. In this figure, we can observe that the performance is better for the largest size of benchmarks such that the jpegtran and the ansi2kr (4.27 Byte/Cycle for C62x and 5.34 Byte/Cycle for C64x). This is because the *Extended Blocks* in the large benchmarks contain more number of instructions than the small ones and consequently, they have more compressed "Fetch Packets" which are decompressed with less number of clock cycles. On average, the performance of 4 and 4.8 Byte/Cycle was achieved for C62x and C64x, respectively.
The performance may be improved by improving the hardware decoder to decode the compressed instruction in parallel and not in serial. This is a point for future work.

5.3.5 Improving the Results of Previous Work Using our Filled Buffer Technique

As the hardware decoder of the work of Xie et al. [77] decodes one "Fetch Packet" in each clock cycle (parallel decoding), we selected their compression scheme (V2F) to apply our compression technique to it and to improve their final compression ratio (which was 82%). For that, we compressed the *Extended Blocks* using the LZSS Algorithm and then encoded the compressed stream with the V2F scheme (which was used in [77]).

Fig. 5.15 shows the compression ratio results without using the Filled Buffer Technique (first bar) and after using it (second bar) for the C62x processor.

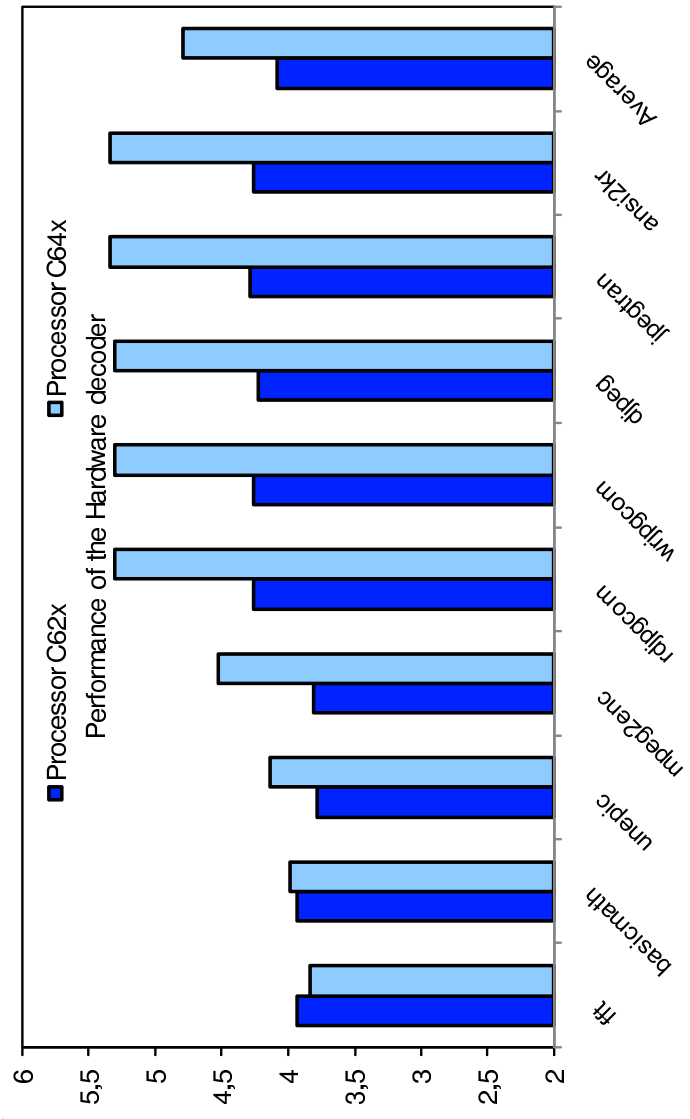


FIGURE 5.14: Performance of the Deflate hardware decoder for C62x and C64x processors

Using the Filled Buffer Technique improved the final compression ratio (on average) from 82% to 72% (i.e. 10% compression ratio improvement). with the same achieved performance results.

5.4 Discussion

Code Compression is very important for VLIW processors because of the large size of their applications in comparison to RISC processor. For Example, the average size of applications compiled for VLIW processor is 60% larger than the size of the same applications compiled forARM processor.

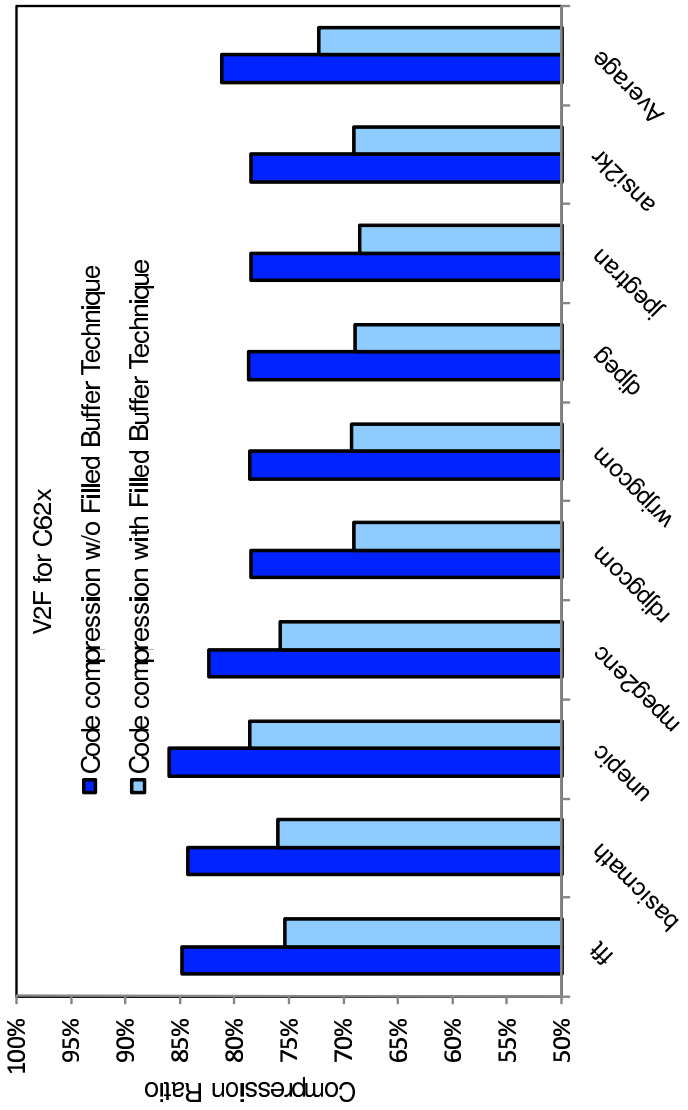


FIGURE 5.15: Compression ratios using our Filled Buffer technique combined with the V2F [77] for C62x VLIW processor

As code compression technique, we selected Deflate because it is generally fast in both compressing and decompressing data and does not require the use of floating-point operations.

We conducted experiments for two VLIW processors (Texas Instruments), namely TMS320C62x and TMS320C64x. These processors are common and widely used because of their high performance and simple implementation.

Our Filled Buffer Technique which can be applied to any compression algorithm of Lempel-Ziv family, targeting any VLIW processor architecture, to improve the compression results. When our Filled Buffer Technique has been used in the top of the Deflate Algorithm, the compression ratios have been improved by 14% and 11% for the processors C62x and C64x, respectively

Comparing to previous work, our technique achieves high compression ratios among other state-of-the-art techniques. In addition to that, the results of the previous work may be achieved, when our Filled Buffer Technique is used in combination to their techniques. For example, the LZW algorithm was used in the previous work [81] and [82]. No better compression ratio than 75% was achieved in both of them. As the LZW algorithm is one of Lempel-Ziv family algorithms, our Filled Buffer Technique may be applied to it to improve the final compression ratio without any impact on their performance (for real example, see Section 5.5).

5.5 Comparing to Previous Work

In Table 5.1 we show the average compression ratio and performance through different benchmarks for our Filled Buffer Technique and few previous work (presented in Section 3) targeting the TMS320C6x VLIW processors.

Comparing to previous work, our technique achieves high compression ratios among other techniques (on average 61% and 56% for TMS320C62x and TMS320C64x VLIW processors, respectively). It is independent from the instruction set architecture and can be applied to any VLIW processor.

TABLE 5.1: Comparing our compression technique to previous art targeting VLIW processors

Reference	Compression scheme	CR	PR(Byte/cycle)
Larin [60]	Huffman Coding	70%-78%	not specified
Sutton [102]	compiler optimized	69,5%-88,5%	not specified
Prakash [112]	Bit Flip	68,9%-76%	1,5-4,6
Xie [77]	V2FCC-static	82,5%	32
Xie [80]	V2FCC-Markov	72%	4
Liao [61]	Mini-subroutine call	88%	not specified
Ros [114]	Hamming	72,1%-80,5%	not specified
Menon [116]	classes based	70%	3
Ishiura [110]	Field Partitioning	60%	not specified
Lin [81]	LZW	75,2%	12,29
Nam [109]	Dictionary technique	63%-71%	not specified
Our Filled Buffer Technique, Sec. 5.1	**Filled Buffer+Deflate**	**56%**	**4,8**

Chapter 6

Code Compression to Improve the Performance

Code compression becomes an important issue when it comes to VLIW architectures [80]. This is because of the bloated code size of the compiled applications in comparison to RISC processors.
To measure the efficiency (in terms of code size reduction) of any compression technique, the compression ratio (CR) is used which is the size required to store the compressed instructions in memory divided by the size required to store the original instructions.
To measure the impact of the hardware decoder on the performance, the Performance Ratio (PR) is deployed which is the time required to execute the compressed instructions divided by the execution time of the original code. The time required to execute the compressed instructions includes the time of decoding the compressed instructions plus the time of executing the decompressed instructions.
As the time of decoding and executing the instructions is always longer than the time of executing the original ones, this results in a performance penalty in the decoding. This is the main disadvantage of using code compression that negatively impacts the embedded system. However, the entire research in the area has always focused on achieving better code compression ratio ignoring (or without explicitly targeting) the performance penalty problem of the hardware decoder. Hence, an efficient code compression technique can be obtained not only by reducing the code size but also with no (or slight) performance penalty.

In this chapter, we improve the performance of decoding compressed instructions by using our novel compression technique which can be used in conjunction with any compression algorithm to improve the performance.
We use our *Left-uncompressed Instruction Technique* along with the Burrows-Wheeler (BW) compression algorithm [107] (explained later) and we show that it results in a high

compression ratio (58%) without loss in performance. To show the orthogonality of our technique, we use it also in conjunction with our compression algorithm (*FBT: Filled Buffer Technique*) from [5] and we show that our technique explicitly improves the performance ratio with only a slight impact on the compression ratio.

The evaluations are conducted using a representative set of applications from *MediaBench* [21] and *MiBench* [20] and are built for two VLIW DSPs from Texas Instruments, namely the TMS320C62x [51] VLIW processor, though our technique can be applied to any other processor architecture.

6.1 Basics of code/data compression

Code compression differs from data compression in different aspects:
First, the size of the data/code that is required to be compressed:
In data compression, it is assumed that the compression must be done in a single sequential pass over the data (i.e. compressing the whole data as one block).
In code compression, the compression is not applied to the whole program, because it will not be decompressed completely and executed as a burst. Instead, small segments or blocks of code (called compression blocks or codewords) are compressed individually. This is to ensure random access to important points in the code such as branch targets and function entry points.

The second aspect of the difference between code and data compression is the way of decoding the compressed data/code:
In data compression, as the data was compressed depending upon the history information of the data stream, the decoder can only start decoding at the beginning of the data stream. It cannot begin decompressing at an arbitrary point in the data stream because it will not have the history information that the decompression algorithm depends upon.
In code compression, as the program was split into different compression blocks and each block is compressed individually, the decoder will entirely decode each compressed block and execute it separately.

The third aspect is the compression result:
Data compression typically results in higher compression ratio than code compression because of the size of data which is required to be compressed.
Hence, to compress a program code, first the compression blocks should be specified, then a good data compression algorithm should be used. To decompress a compressed program,

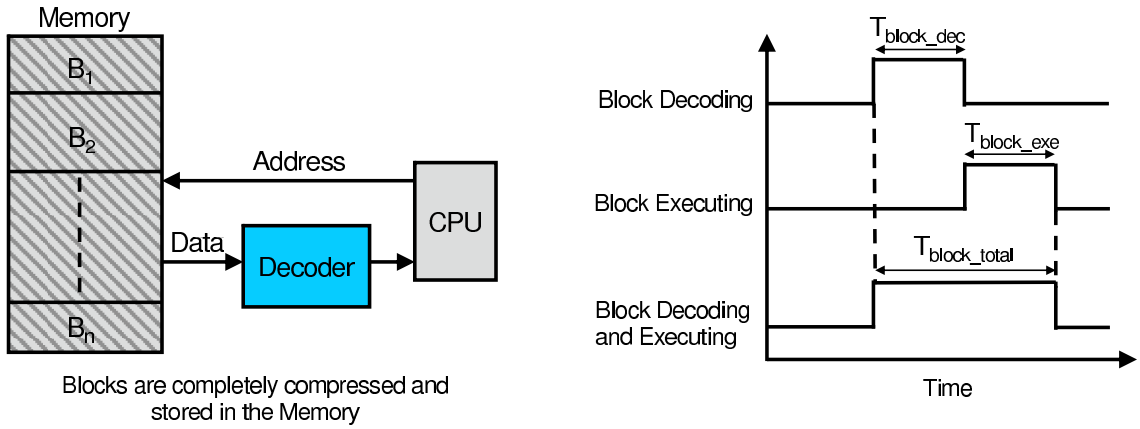


FIGURE 6.1: Traditional decoding techniques. Each block is decoded first and then executed.

a fast decoder should be implemented to keep the loss in performance of decoding as low as possible.

6.1.1 Traditional Code Compression Techniques

In most of the previous work, a data compression algorithm is selected and applied to each compression block separately (to ensure random access). The compressed blocks are then stored contiguously in memory with a constraint that the beginning of each compressed block should be aligned to the memory boundary so that the decoder can quickly access the compressed blocks.
The decompression (see Fig. 6.1), hardware decoder is implemented and placed between the memory and the CPU. Each compressed block is **decoded** first using the hardware decoder and then **executed** by the CPU. The time required to complete this process (T_block) is as follow:

$$T_{block\_total} = T_{block\_dec} + T_{block\_exe}$$

T_{block_total}: Time required to decode and execute the compressed block.
T_{block_dec}: Time required to decode the compressed block.
T_{block_exe}: Time required to execute the decompressed block.

One of the disadvantages of using code compression is the performance loss due to the extra time required to decode the compressed code before execution. Interestingly, the entire research in the area has always focused on achieving a better code compression ratio without explicitly targeting the performance loss problem of the hardware decoder.

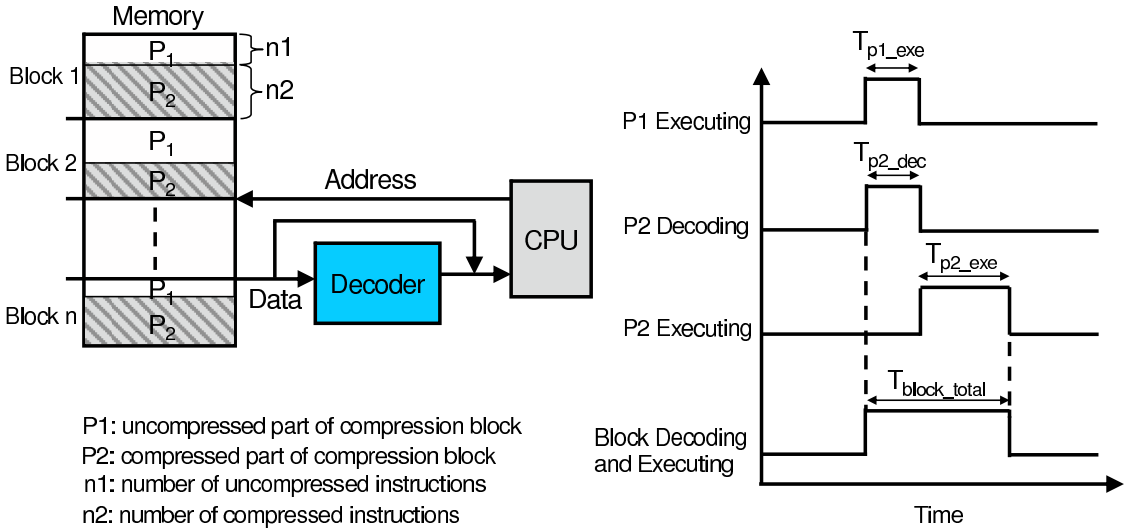


FIGURE 6.2: Our compression technique. During the execution of uncompressed instructions in each block, compressed instructions are decoded

We use a new compression technique to compress the program code with no impact on performance. Improving performance results in slight degrading the compression ratio. Hence, there is a trade-off between the compression and the performance. *LICT* may be used in conjunction with any other compression technique to improve performance.

6.2 Left-uncompressed Instruction Compression Technique (LICT)

We conduct the following steps for *LICT*:
I. The compression blocks (*Extended Blocks*) of the program code are specified and extracted.
II. Each block is split into two parts. The first part "P1" contains the first *n1* instructions of the compression block (*n1* is explained in the next step) and the second part "P2" contains the remaining instructions *n2* in the compression block.
III. The second part "P2" of each compression block is compressed using a data compression algorithm. The instructions *n1* in the first part "P1" of the compression block are left uncompressed. *n1* is selected such that the time required to execute the instructions in the first part "P1" of the compression block is equal to the time required to decode the compressed instructions in the second part "P2" (explained later).
Hence, during execution the uncompressed instructions *n1* of each compression block, the *n2* compressed instructions of that block (which exist in "P2") will be decoded. For that, the time required to decode the compressed instructions will be hidden by the time required to execute the uncompressed instructions and no performance loss will occur during the decoding.

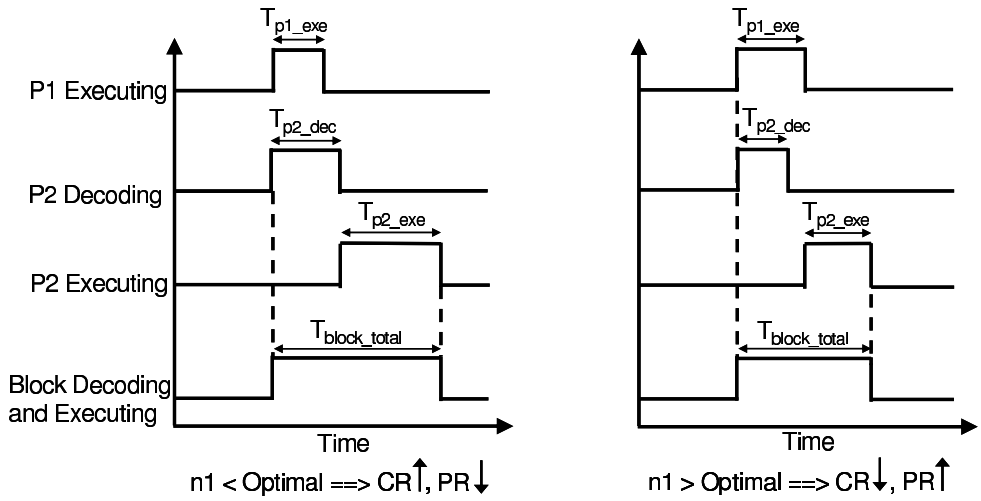


FIGURE 6.3: Trade-off between Compression Ratio (CR) and Performance Ratio (PR) according to number of uncompressed instructions (*n1*)

Fig. 6.2 shows the same example used in Fig. 6.1 but using our *LICT*. When the compression block is fetched to be executed by the CPU, the uncompressed instructions in the first part "P1" are transferred directly to the CPU. The compressed instructions in the second part "P2" are first transferred to the hardware decoder then to the CPU for execution. The time required to decode and execute a compressed block, in this case, is computed as following:

$$T_{block\_total} = T_{P1\_exe} + T_{P2\_exe} \tag{6.1}$$

OR

$$T_{block\_total} = T_{P2\_dec} + T_{P2\_exe}$$

Considering that $T_{P1\_exe} = T_{P2\_dec}$

T_{P1_exe}: Time required to execute uncompressed instructions "*n1*".
T_{P2_dec}: Time required to decode the compressed instructions "*n2*".
T_{P2_exe}: Time required to execute the decoded instructions "*n2*".

To select the number of uncompressed instructions *n1*, three different cases may be considered (see Fig. 6.3):

I. If the number of uncompressed instructions in the compression block is more than the optimal [1] case, then the time required to execute the uncompressed instructions will be longer than the time required to decode the compressed instructions. This will result in a low compression ratio but high performance ratio (Fig. 6.3, right part). The total time of decoding and executing the compressed instructions is computed as follows:

$$T_{block\_total} = T_{P1\_exe} + T_{P2\_exe}$$

II. If the number of uncompressed instructions in the compression block is less than the optimal case, then the time required to decode the compressed instructions will be longer than the time required to execute the uncompressed instructions. This will result in high compression ratio but will impact the performance ratio (Fig. 6.3, left part). The total time of decoding and executing the compressed instructions is computed as follows:

$$T_{block\_total} = T_{P2\_dec} + T_{P2\_exe}$$

III. If the number of uncompressed instructions is selected to be optimal, then the time required to decode the compressed instructions and to execute the uncompressed ones will be equal. This will result in high compression and low performance overhead. The total time of decoding and executing the compressed instructions is computed as in Eq. 6.1.

Algorithm 12 (*LICT*) shows how to determine the instructions *n1* in the first part "P1" which have to be left uncompressed. This number will keep the time of executing the uncompressed instructions equal to the time of decoding the compressed instructions. The algorithm starts by moving all instructions of the compression block to its second part "P2" and resetting "P1" (lines 2-3). The algorithm then starts moving the instructions "P2" to "P1". After moving each instruction, the algorithm compresses the instructions in the "P2" (line 5) and computes the time required to decode the compressed instructions (line 6) and to execute the instructions in the "P1" (line 7). The algorithm continues moving the instructions from "P2" to "P1" until the times of executing and decoding becomes equal. At this point, the number of moved instructions from "p2" to "p1" (which is *n1*) results in the optimal compression and performance ratios. The algorithm repeats all steps for each compression block in the application.

[1] The optimal number is the number of instructions which are left uncompressed to achieve the highest compression and performance ratios.

Algorithm 12 LICT: Left-uncompressed Instructions Compression Technique

/* "B": Compression Block */
/* "P1": First part of the compression Block "B" */
/* "P2": Second part of the compression Block "B" */
/* "n": Number of instructions in "B" */
/* *n1*: Number of instructions in "p1" */
/* *n2*: Number of instructions in "p2" */

/* Find the optimal number of instruction which have to be left uncompressed */

1: **for all** Compression Blocks "B" in the application **do**
2: "B"→"P2" {Move all instructions in "B" to "P2", i.e. n2=n}
3: 0→ "P1" {Reset "P2", i.e. n1=0}
4: Move one instruction from "P2" to "P1" {n1=n1+1, n2=n2-1}
5: Compress "P2" {Using the compression algorithm}
6: Compute the time of decoding "P2" {i.e. T_{P2_dec}}
7: Compute the time of executing "P1" {i.e. T_{P1_exe}}
8: **if** $T_{P1\_exe} < T_{P2\_dec}$ **then**
9: GOTO 4 {Time of executing "P1" is still shorter than decoding "P2"}
10: **else** {Times are equal}
11: Return(*n1*) {This is the optimal number of uncompressed instructions}
12: **end if**
13: **end for**

6.3 Burrows-Wheeler Code Compression Algorithm

To compress the compression blocks (i.e. *Extended Blocks*) we use the Burrows-Wheeler (*BW*) compression algorithm [107]. The *BW* algorithm works in a *block mode*, where the input stream is read block by block and each block is encoded separately as one string. We selected this algorithm for compression because it can achieve very high compression ratios. In addition, this algorithm has not been used in the previous work for code compression.

The main idea of the *BW* algorithm is to start with a string "S" of "n" symbols and to scramble them into another string "L" that satisfies two conditions:

I. Any region of "L" tends to have a concentration of just a few symbols, i.e. if a symbol "s" is found at a certain position in L, then other occurrences of "s" are likely to be found nearby. This property means that "L" can easily and efficiently be compressed with the Move-to-front (MTF) method.

II. It is possible to reconstruct the original string "S" from "L".

6.3.1 The *BW* Encoding Steps

The mathematical terms for scrambling symbols is *permutation*, and it is straightforward to show that a string of "n" symbols has n! permutations. Therefore the permutation used by the *BW* has to be carefully selected.
To encode the string "S" using the *BW* algorithm (Fig. 6.4, top part), the following steps are performed:

1. String "L" is created by the encoder as a permutation of the given input string "S". Some more information denoted by I is created for decoder
2. The encoder compress "L" and "I" and writes the result to the output stream. This step starts with the Move-to-front (MTF) method [107] and then applies Huffman Coding [26].

To create the string "L" from the given input string "S" of "n" symbols, the encoder constructs an n x n matrix. In this matrix, the encoder stores the string "S" in the top row, followed by "n-1" copies of S, each cyclically shifted one symbol to the left. The matrix is then sorted lexicographically by rows. The permutation "L" selected by the encoder is the last column of the sorted matrix (this column contains concentrations of identical symbols).
The other information required to reconstruct "S" from "L" is the row number of the original string in the sorted matrix. This number is stored in "I".
For example, to create the string "L" from the string S="swiss-miss" using the *BW* algorithm, first the matrix 10 x 10 is created where the string "S" is stored in the top row, followed by 9 copies of S, each cyclically shifted one symbol to the left. After sorting the matrix lexicographically by rows, the string "L" will be the last column of this matrix which is L="swm-siisss". As the original string "S" is located in row number 8 of the sorted matrix, the other information required to reconstruct "S" from "L" is I=8.

The input string S, in our case, is considered as the binaries of instructions which are located in the second part "P2" of the compression block.

After the string "L" and the variable "I" are created, they are compressed using the Move-to-front (MTF) method [107] followed by Huffman Coding [26]. Using MTF method increases the concentrations of identical symbols in the String "L" and improves the compression results when the Huffman Coding is used afterwards.

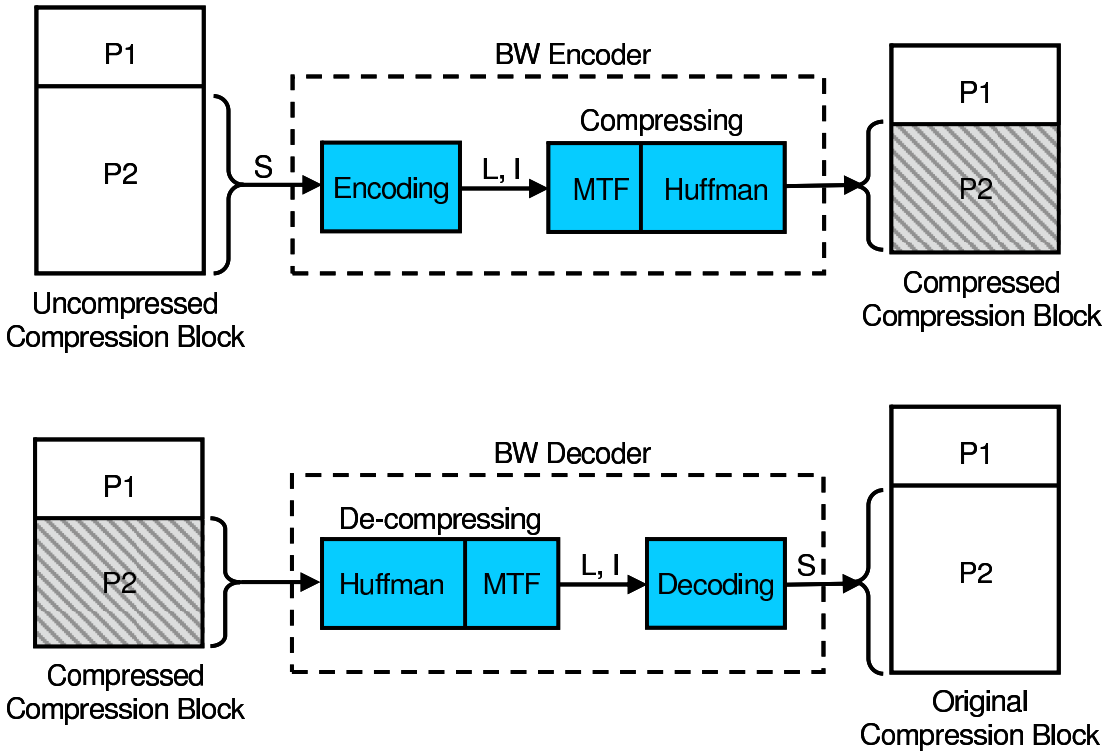


FIGURE 6.4: The encoding and decoding steps of the *BW* compression algorithm

6.3.2 The *BW* Decoding Steps

To decode the encoded string "S" using the *BW* algorithm (Fig. 6.4, bottom part), the following steps are used:

1. The decoder reads the output stream and decodes it by applying the same methods, as in step 2 of the encoding steps, but in reverse order. The results are string "L" and variable "I".
2. Both "L" and "I" are used by the decoder to reconstruct the original string S.

To construct the original string "S" from "L" and "I", the decoder constructs the first column "F" of the sorted matrix from "L". This is a straightforward process, since "F" and "L" contain the same symbols and "F" is sorted. The decoder simply sorts string "L" to obtain "F".
The relation between the elements of "L" and "F" may be prepared by an auxiliary array "T". For example, if the first element of "T" is 4, this imploys that the first symbol of "L" is located in position 4 of "F".
The decoder uses "L", "I" and "T" to reconstruct "S" according to:

$S[n-1-i] \leftarrow L[T^n[I]], \quad for \;\; i = 0, 1, ..., n-1$
$where \;\; T^0[j] = j, and \;\; T^{i+1}[j] = T[T^i[j]]$

6.4 Applying *LICT* to the FBT

To show the efficiency of our code compression technique, we apply it to state-of-the-art work "FBT" [5] to improve the performance of the hardware decoder, although our compression technique can be applied to any other previous work to improve performance. We selected the work [5] because it achieves high compression ratios using the same VLIW processor architecture as we use.

In the previous work [5], the authors used a "Deflate" code compression algorithm. "Deflate" is a data compression algorithm which is based on an optimized version of LZ77 (called LZSS) combined with Huffman codes. The authors enhanced the "Deflate" algorithm by using a new technique called "Filled Buffer Technique" which can be applied to any *Lempel-Ziv* family algorithm to improve the compression ratio. As compression blocks, they used the "*Extended Blocks*" which include in more instructions in comparison to the size of the "*Basic Blocks*" and applied their Filled Buffer Technique and the "Deflate" compression algorithm to these "*Extended Blocks*".
Using our *LICT* in conjunction with their technique improves the decoding performance explicitly with little impact on the compression ratio (as explained in the experimental results).

6.5 Experiments and Results

In this section we present the experimental results and performance of our compression technique in addition to the results obtained by applying our compression technique to previous work [5]. All experiments are conducted for the TMS320C62x VLIW processor (from Texas Instruments). The applications are compiled and linked using the Code-Composer-Studio (CCS) from Texas Instruments and the experimental results are obtained using the simulator "c6xsim" [108].
The experimental results are shown in figures 6.5 - 6.11. In each diagram, the bar labeled "Average" shows the average across all benchmarks in that diagram.
Fig. 6.5 shows the number of the *Basic Blocks* and *Extended Blocks* for different benchmarks. As shown in this figure, the number of *Extended Blocks* is much smaller than the number of *Basic Blocks* for the same benchmark (in average it may be 48% smaller). A smaller number of *Extended Blocks* means bigger size than the *Basic Blocks*, i.e. more

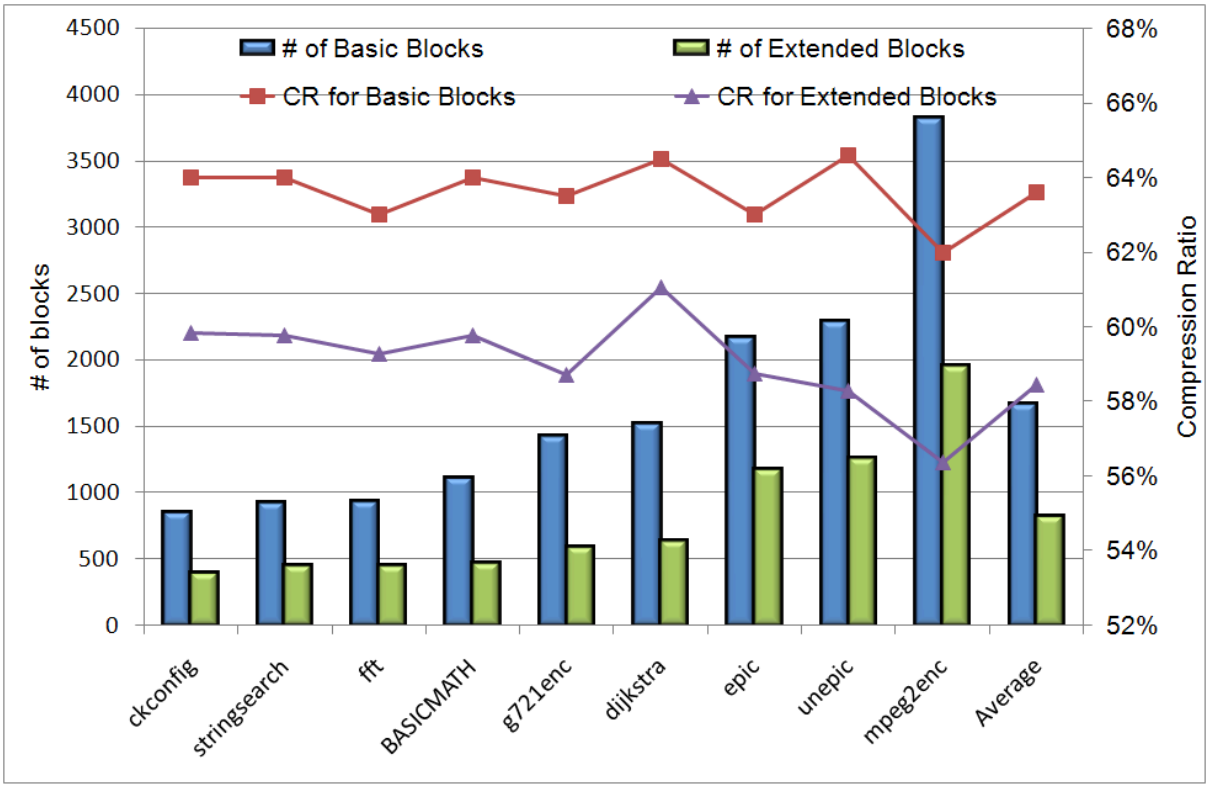


FIGURE 6.5: The *BW* compression algorithm for *Basic Blocks* and *Extended Blocks*

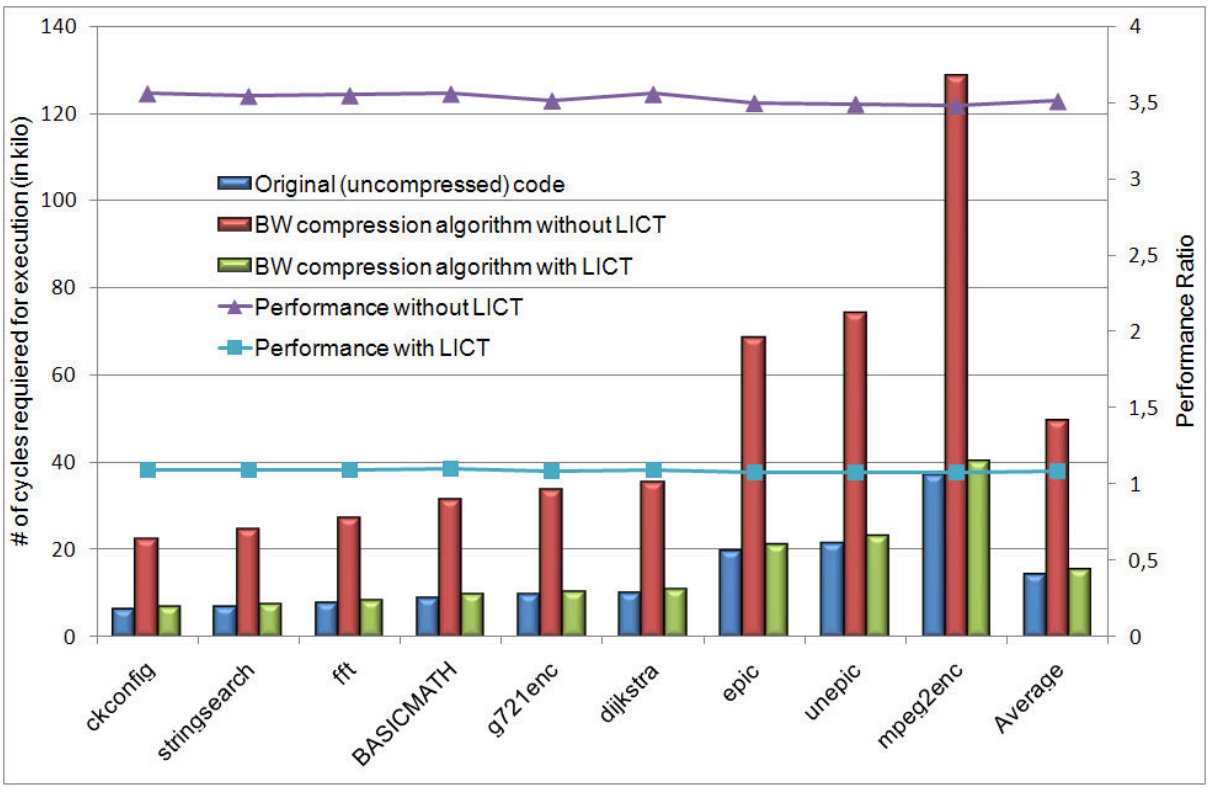


FIGURE 6.6: Performance ratio for different size of benchmarks. Using *LICT* improves the performance ratio explicitly

number of instructions are included in the *Extended Blocks* in comparision to the *Basic Blocks*. This will result in a better compression ratio when the compression technique is applied to the *Extended Blocks* other than the *Basic Blocks*. Fig. 6.5 shows also the compression ratios of different benchmarks when the *BW* compression algorithm is applied to the *Extended Blocks* and the *Basic Blocks*. The average compression ratio achieved for *Extended Blocks* is 58% and for *Basic Blocks* 63%. For that, we use the *Extended Blocks* as the compression blocks in all of our experimental results.
Fig. 6.6 shows the performance ratio and the number of static cycles required to execute different benchmarks of different size. The number of static execution cycles is the sum of all cycles required to execute the *Extended Blocks*. The first bar of each benchmark in this figure shows the number of execution cycles of the original application (i.e. without compression). The second bar shows the number of cycles required to decode the compressed instructions and to execute them. The instructions are compressed using the *BW* compression technique without using *LICT* (i.e. each compression block is compressed and decompressed completely). The average performance ratio obtained in this case is 3.5. The third bar shows the number of decoding and executing cycles when *LICT* is used. Fig. 6.6 shows that using our compression technique improves the performance in average by 2.5x. For that, there will be no performance loss in the hardware decoder.
Our compression technique (*LICT*) does not entirely come for free. On the plus side, it does improve the performance of the hardware decoder. However, a small increase of compressed code size is the price to pay (see Fig. 6.7). This figure shows the original code size (first bar) for different benchmarks. The second and third bars show the compressed code using the *BW* compression algorithm without and with using *LICT*, respectively. The average compression ratio achieved using the *BW* compression algorithm is 55%. When *LICT* is used, the average compression ratio is degraded to 58%. As is shown, this is the total compression ratio, i.e. it is also accounting for the hardware size besides the pure compressed code size. The resulting numbers compare very favorably to compression ratios achieved by others. Hence, our compression technique is the basis for not-yet-achieved compression ratios without performance loss in comparison to previous work.
When *LICT* is applied, the number of uncompressed instructions that are left in each compression block is selected to be optimal, i.e. this number results in high compression and performance ratios. As the compression blocks have a different number of instructions, the optimal number of the instructions that are left without compression differs from one compression block to the next. Therefore, information about this number is stored for each compression block to be used later in the decoding phase (the results in Figures 6.6 and 6.7 include this information).

Figure 6.8 shows the trade-off between the performance and compression ratios for the "mpeg2enc" benchmark. In this figure, the number of the instructions that are left uncompressed in each compression block is selected to be the same for each block. The range

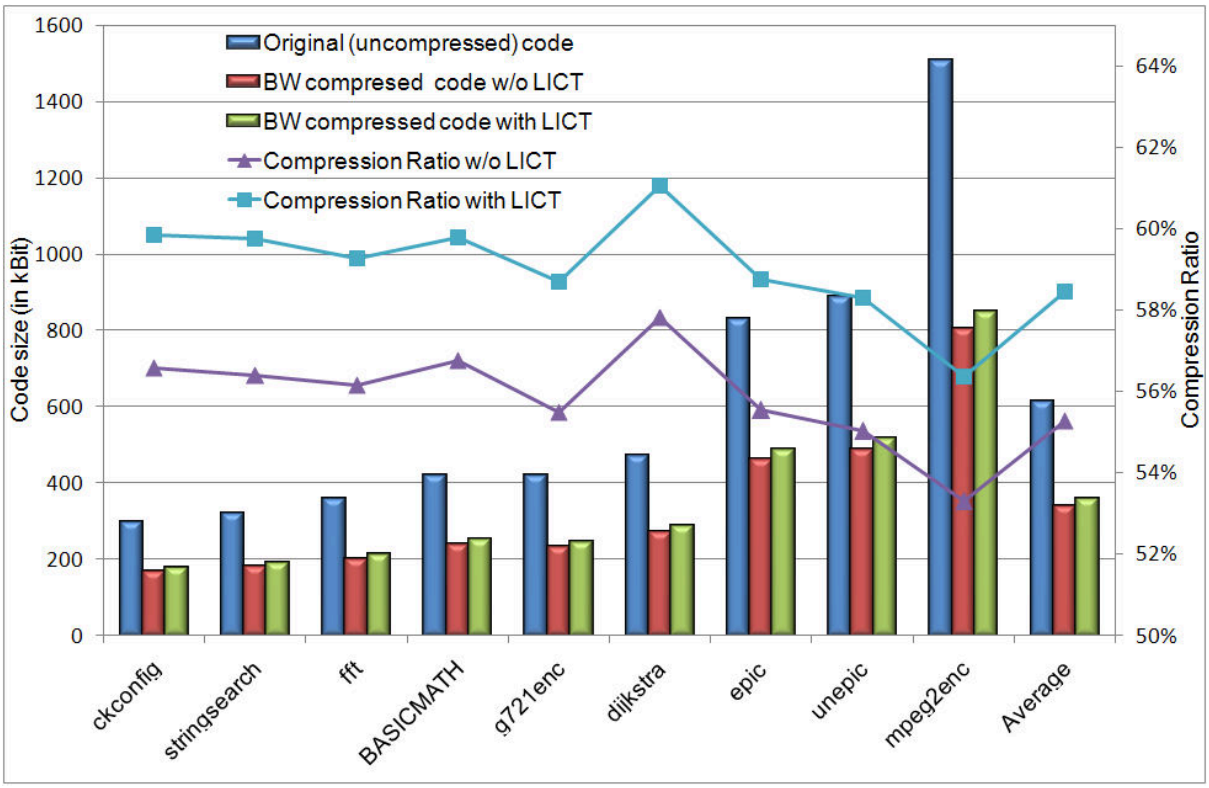


FIGURE 6.7: Compression ratio for different size of benchmarks. Using *LICT* results in a slight degrading in the compression ratio

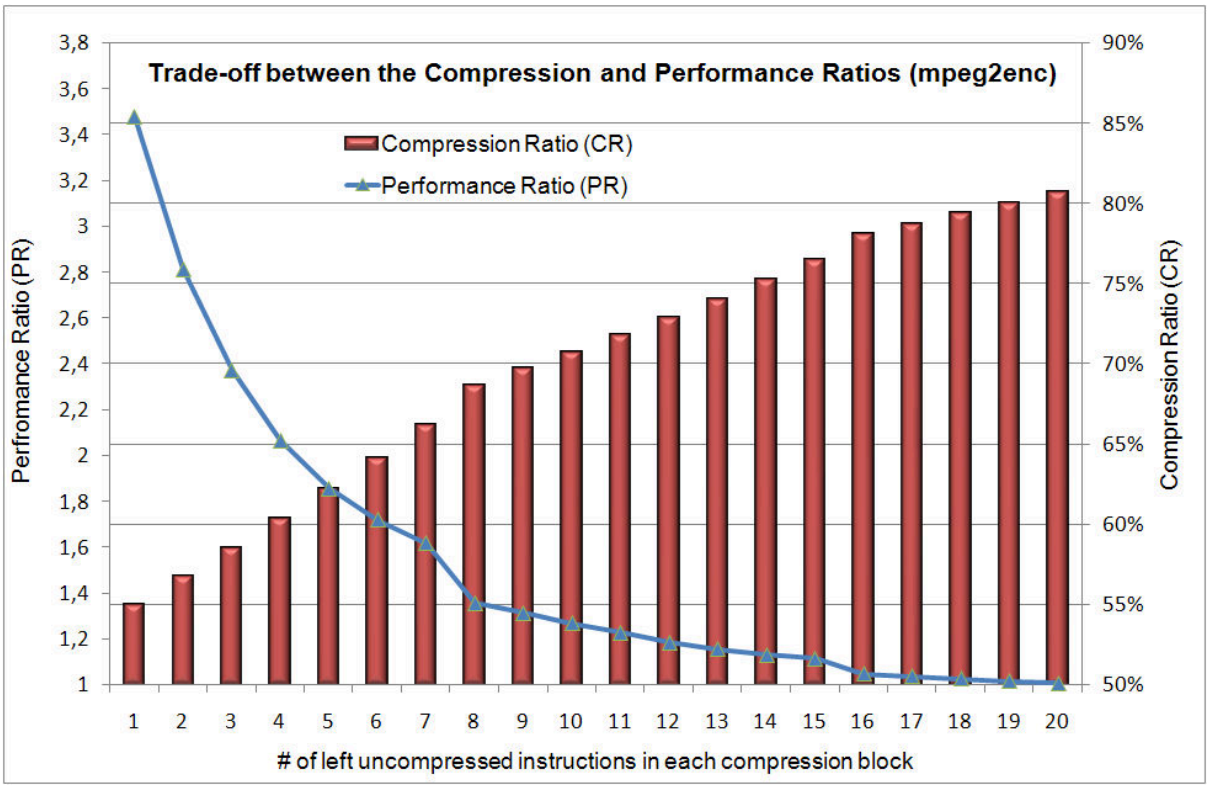


FIGURE 6.8: Trade-off between the compression ratio and performance ratio when the number of left uncompressed instructions is changed

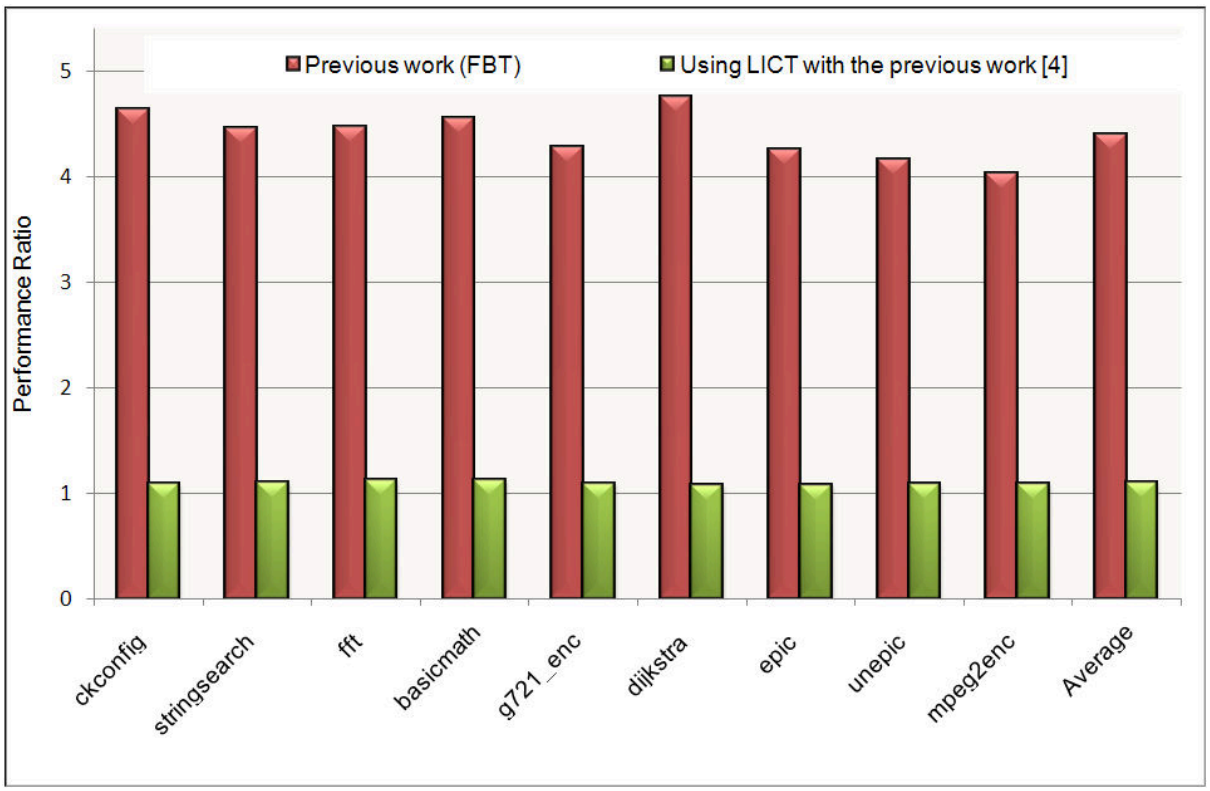


FIGURE 6.9: Performance ratio when *LICT* is applied to the previous work [5]

of the uncompressed instructions is presented in this figure between 1 and 20 instructions. If the compression block has less instructions than the instructions which are required to be left uncompressed, then the compression block is completely left uncompressed. From Fig. 6.8 we can observe and conclude that by increasing the number of uncompressed instructions in each compression block, the number of instructions that are required to be compressed will be decreased and consequently the compression ratio will be decreased, i.e. unimproved (the compression ratio is decreased from 55% by 1 uncompressed instruction to 80% by 20 uncompressed instructions). On the other hand, increasing the number of uncompressed instructions reserves more time to the decoder to decode the compressed instructions during the execution of the uncompressed instructions. This will improve the performance ratio from 3.47 (by 1 uncompressed instruction) to 1.0 (by 20 uncompressed instructions), i.e. 2.47x.

Figures 6.9, 6.10 and 6.11 show the results of applying *LICT* to the previous work (Filed Buffer Technique) [5] (although our technique can be applied to any previous compression technique to improve their performance). In Figures 6.9 and 6.11, the first bar in each benchmark shows the results of the sole Filed Buffer Technique, i.e. without applying *LICT* to it. The second bar shows the results of the Filed Buffer Technique in conjunction with *LICT*. Fig. 6.9 shows that using *LICT* improves the performance ratio of the decoder in average by 3.3x (details are presented in Fig. 6.10). That means the decoder decodes the compressed instruction without (or with a very slight) performance penalty. On the

Benchmark	Original Execution (Byte/cycle)	Compressed Execution (Byte/cycle)	Performance Ratio without LICT	Performance Ratio with LICT
ckconfig	5,97	1,29	4,63	1,09
stringsearch	5,79	1,3	4,45	1,1
fft	5,84	1,31	4,46	1,12
g721_enc	5,47	1,28	4,28	1,09
basicmath	5,95	1,31	4,54	1,12
dijkstra	5,94	1,25	4,75	1,07
epic	5,31	1,25	4,25	1,07
unepic	5,23	1,26	4,15	1,08
mpeg2enc	5,11	1,27	4,03	1,08
Average	5,62	1,2875	4,39	1,09

FIGURE 6.10: Results of applying *LICT* to the previous work [5]

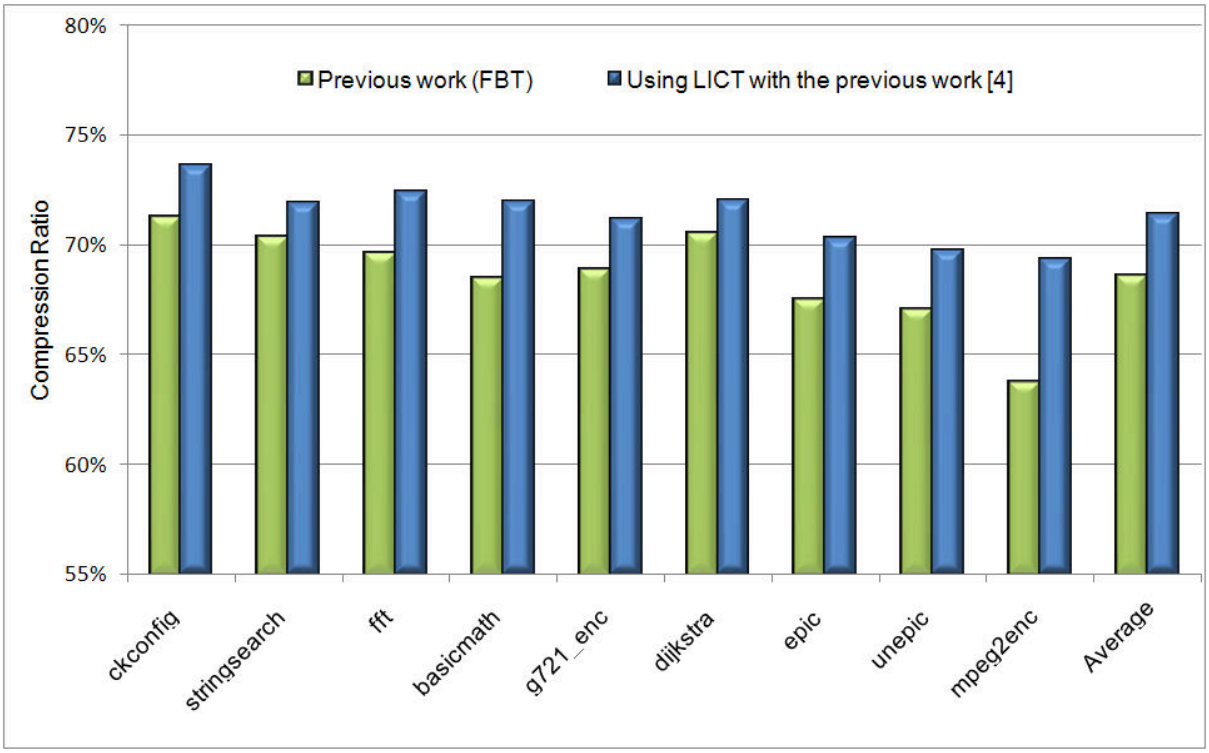


FIGURE 6.11: Compression ratio when *LICT* is applied to the previous work [5]

other hand, a slight degrading in the compression ratio is observed (from 68% to 71%) because of the left uncompressed instructions (see Fig. 6.11).

Chapter 7

Conclusion

This chapter summaries the thesis and discusses possible future work.

7.1 Thesis Summary

This thesis presented novel Huffman-based code compression techniques as a key to efficient code density.

"Look-up Table Compression Technique" was presented in conjunction with Statistical and Dictionary-based compression schemes. It focused on the overhead that comes when a compression technique used, namely the decoding table size. Previous work had mainly focused on the code density itself and paid little attention to the overhead. A fair presentation of a compression ratio, however, needs to consider in all incurred overhead.
"Look-up Table Compression Technique" is orthogonal to any ISA-specific characteristics and may be applied to any compression technique that generates large decoding table size. When this technique was applied to Statistical compression scheme, average compression ratios of 52%, 50%, and 53% were reported for ARM, MIPS, and PowerPC, respectively.

Further reduction in the decoding table size (and consequently further improvement in the compression ratio) was achieved by using the "Instruction Splitting Technique". This technique reduced not only the decoding table size but also the encoded instructions generated by using Huffman Coding Algorithm. The achieved compression ratios (with overhead accounted for) are superior to approaches that have been proposed so far. In addition to that, this technique is more general to code compression as it is independent of the instruction set architecture. Average compression ratios of 47%, and 49% were

achieved for ARM, MIPS, respectively.

The thesis also presented new code compression technique which is ISA-Dependent and based on specific processor architecture. It extracts the unused fields of the instruction format and then encodes them to reduce the toggle between each two consecutive instructions and consequently to improve the final compression ratio. It was shown that this technique can be applied to any processor architecture if the ISA is known by targeting two processors, namely ARM and MIPS. Average compression ratios of 45% and 48% were achieved for MIPS and ARM processors, respectively, with a little impact on performance.

The "Filled Buffer Technique" was used to improve the compression ratio of the sliding window compression algorithms. It was shown that this technique may be applied to any Lempel-Ziv family algorithms and may be combined with the previous work of Xie et al. [77] to improve their compression results. Average compression ratios of 61% and 56% were achieved for the TMS320C62x and TMS320C64x VLIW processors, respectively.

The final code compression technique presented in this thesis was "LICT: Left-uncompressed Instructions Compression Technique". It was used to improve the performance of the hardware decoder independent from the processor architecture. Applying LICT on the Burrows-Wheeler (BW) [107] code compression algorithm improves the performance explicitly (2.5x) with a little impact on the compression ratio (only 3% compression ratio loss). The evaluations are conducted using a representative set of benchmarks (from Mediabench and Mibench) and applied to the TMS320C62x VLIW processor.

7.2 Future Work

The possible further work related to this thesis can be done by improving the decoding performance of the compression technique. As the time of decoding and executing the compressed instructions is always greater than the execution time of uncompressed instructions, this results in degradation in the decoding performance. One of the methods which can be used to improve the decoding performance is to build the hardware decoder into the CPU itself as part of the instruction decode phase. Another method is to leave some instructions at the beginning of each basic block uncompressed. During executing these uncompressed instructions, the remaining compressed instructions in the basic block will be decoded. In this case, the decoding time of the compressed instructions will be hidden by the time required to execute the uncompressed ones, and hence no performance penalty will be occurred.

Power consumption is an important issue in the embedded systems, since memories consume a significant amount of an embedded system's power budget. Code compression technique reduces memory and also results in fewer access to memory. This causes reduction in memory power and bus power.
The possible future work would be the power estimation of decoding hardware and memory along with the performance evaluation for the code compression technique. This can be carried out using an cycle-accurate performance simulator coupled with a power simulator and the results can be compared with the non compressed approach.

Appendix A

MIPS Instruction Set

MIPS instructions are divided into the following functional groups [121]:

- Load and Store
- Computational Instructions
- Jump and Branch
- Miscellaneous
- Coprocessor

A.1 Load and Store

Mnemonic	Description	Defined in
LB	Load Byte	MIPS I
LBU	Load Byte Unsigned	I
SB	Store Byte	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
SH	Store Halfword	I
LW	Load Word	I
LWU	Load Word Unsigned	III
SW	Store Word	I
LD	Load Doubleword	III
SD	Store Doubleword	III
LWL	Load Word Left	I
LWR	Load Word Right	I
SWL	Store Word Left	I
SWR	Store Word Right	I
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III

A.2 Computational Instructions

Mnemonic	Description	Defined in
ADDI	Add Immediate Word	MIPS I
ADDIU	Add Immediate Unsigned Word	I
SLTI	Set on Less Than Immediate	I
SLTIU	Set on Less Than Immediate Unsigned	I
ANDI	And Immediate	I
ORI	Or Immediate	I
XORI	Exclusive Or Immediate	I
LUI	Load Upper Immediate	I
DADDI	Doubleword Add Immediate	III
DADDIU	Doubleword Add Immediate Unsigned	III
ADD	Add Word	I
ADDU	Add Unsigned Word	I
SUB	Subtract Word	I
SUBU	Subtract Unsigned Word	I

Mnemonic	Description	Defined in
DADD	Doubleword Add	MIPS III
DADDU	Doubleword Add Unsigned	III
DSUB	Doubleword Subtract	III
DSUBU	Doubleword Subtract Unsigned	III
SLT	Set on Less Than	I
SLTU	Set on Less Than Unsigned	I
AND	And	I
OR	Or vI	
XOR	Exclusive Or	I
NOR	Nor	I
SLL	Shift Word Left Logical	I
SRL	Shift Word Right Logical	I
SRA	Shift Word Right Arithmetic	I
SLLV	Shift Word Left Logical Variable	I
SRLV	Shift Word Right Logical Variable	I
SRAV	Shift Word Right Arithmetic Variable	I
DSLL	Doubleword Shift Left Logical	III
DSRL	Doubleword Shift Right Logical	III
DSRA	Doubleword Shift Right Arithmetic	III
DSLL32	Doubleword Shift Left Logical + 32	III
DSRL32	Doubleword Shift Right Logical + 32	III
DSRA32	Doubleword Shift Right Arithmetic + 32	III
DSLLV	Doubleword Shift Left Logical Variable	III
DSRLV	Doubleword Shift Right Logical Variable	III
DSRAV	Doubleword Shift Right Arithmetic Variable	III
MULT	Multiply Word	I
MULTU	Multiply Unsigned Word	I
DIV	Divide Word	I
DIVU	Divide Unsigned Word	I
DMULT	Doubleword Multiply	III
DMULTU	Doubleword Multiply Unsigned	III
DDIV	Doubleword Divide	III
DDIVU	Doubleword Divide Unsigned	III
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MTLO	Move To LO	I

A.3 Jump and Branch

Mnemonic	Description	Defined in
J	Jump	MIPS I
JAL	Jump and Link	I
JR	Jump Register	I
JALR	Jump and Link Register	I
BEQ	Branch on Equal	I
BNE	Branch on Not Equal	I
BLEZ	Branch on Less Than or Equal to Zero	I
BGTZ	Branch on Greater Than Zero	I
BEQL	Branch on Equal Likely	II
BNEL	Branch on Not Equal Likely	II
BLEZL	Branch on Less Than or Equal to Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II
BLTZ	Branch on Less Than Zero	I
BGEZ	Branch on Greater Than or Equal to Zero	I
BLTZAL	Branch on Less Than Zero and Link	I
BGEZAL	Branch on Greater Than or Equal to Zero and Link	I
BLTZL	Branch on Less Than Zero Likely	II
BGEZL	Branch on Greater Than or Equal to Zero Likely	II
BLTZALL	Branch on Less Than Zero and Link Likely	II
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	II

A.4 Miscellaneous

Mnemonic	Description	Defined in
SYSCALL	System Call	MIPS I
BREAK	Breakpoint	I
TGE	Trap if Greater Than or Equal	II
TGEU	Trap if Greater Than or Equal Unsigned	II
TLT	Trap if Less Than	II
TLTU	Trap if Less Than Unsigned	II
TEQ	Trap if Equal	II
TNE	Trap if Not Equal	II
TGEI	Trap if Greater Than or Equal Immediate	II
TGEIU	Trap if Greater Than or Equal Unsigned Immediate	II
TLTI	Trap if Less Than Immediate	II
TLTIU	Trap if Less Than Unsigned Immediate	II
TEQI	Trap if Equal Immediate	II
TNEI	Trap if Not Equal Immediate	II
SYNC	Synchronize Shared Memory	II
MOVN	Move Conditional on Not Zero	IV
MOVZ	Move Conditional on Zero	IV
PREF	Prefetch Indexed	IV
PREFX	Prefetch Indexed	IV

A.5 Coprocessor

Mnemonic	Description	Defined in
LWCz	Load Word to Coprocessor-z	MIPS I
SWCz	Store Word from Coprocessor-z	I
LDCz	Load Doubleword to Coprocessor-z	II
SDCz	Store Doubleword from Coprocessor-z	II
LWXC1	Load Word Indexed to Floating Point	IV
SWXC1	Store Word Indexed from Floating Point	IV
LDXC1	Load Doubleword Indexed to Floating Point	IV
SDXC1	Store Doubleword Indexed from Floating Point	IV
COPz	Coprocessor-z Operation	I

Bibliography

[1] T. Bonny and J. Henkel. Using Lin-Kernighan Algorithm for Look-Up Table Compression to Improve Code Density. *Proc. of the 16h Great Lakes Symposium on VLSI-(GLSVLSI'06)*, pp. 259-265, Philadelphia, USA, April 2006.

[2] T. Bonny and J. Henkel. Efficient Code Density Through Look-up Table Compression. *IEEE/ACM Proc. of Design Automation and Test in Europe Conference (DATE07)*, pp. 809-814, Nice, France, April 2007.

[3] T. Bonny and J. Henkel. Instruction Splitting for Efficient Code Compression. *In 44th ACM/EDA/IEEE Design Automation Conference (DAC'07)*, pp. 646-651, San Diego CA, USA, June 2007.

[4] T. Bonny and J. Henkel. Instruction Re-encoding Facilitating Dense Embedded Code. *IEEE/ACM Proc. of Design Automation and Test in Europe Conference (DATE08)*, pp. 770-775, Munich, Germany, March 2008.

[5] T. Bonny and J. Henkel. FBT: Filed Buffer Technique to Reduce Code Size for VLIW Processors. *26th IEEE/ACM International Conference on Computer-Aided Design (ICCAD08)*, pp. 549-554, San Jose, CA, USA, November, 2008.

[6] T. Bonny and J. Henkel. LICT: Left-uncompressed Instructions Compression Technique to Improve the Decoding Performance of VLIW Processors. *In 46th ACM/EDA/IEEE Design Automation Conference (DAC'09)*, pp. 903-906, San Francisco CA, USA, July 2009.

[7] T. Bonny and J. Henkel. Efficient Code Compression for Embedded Processors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, volume 16, issue 12, pp. 1696-1707, December 2008.

[8] S. Parameswaran, J. Henkel, A. Janapsatya, T. Bonny and A. Ignjatovic. Book Chapter: Design and Run Time Code Compression for Embedded Systems. in "Designing Embedded Processors", J. Henkel and S. Parameswaran (Eds.), Springer, pp. 97-128, 2007.

[9] M. Broy. Automotive software and systems engineering. *Formal Methods and Models for Co-Design*, pp. 143-149, 2005.

[10] *http://www.wsts.org.*

[11] *Report RG-229R Future of Embedded Systems Technology from Business Communications Company. www.bccresearch.com*, April 20, 2005.

[12] M. Collin and M. Brorsson, Low Power Instruction Fetch using Profiled Variable Length Instructions. *in Proceedings of the IEEE International SoC Conference*, pp. 183-188, 2003.

[13] L. Benini, A. Macii, E. Macii and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. *In Proceedings of International Symposium on Low Power Electronics and Design*, pp. 206-211, 1999.

[14] L. Benini, D. Bruni, A. Macii and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. *Proceedings of the conference on Design, Automation & Test in Europe DATE02*, pp. 449-453, 2002.

[15] L. Benini, A. Macii and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. *International Symposium on Low Power Electronics and Design*, pp. 322-327, Aug. 2001.

[16] L. Benini, F. Menichelli, and M. Olivieri. A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems. *IEEE TRANSACTIONS ON COMPUTERS*, vol. 53, No. 4, April. 2004.

[17] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on CAD*, volume 18, number 12, pp. 1689-1701. Dec 1999.

[18] H. Lekatsas, J. Henkel, V. Jakkula and S. Chakradhar. A unified architecture for adaptive compression of data and code on embedded systems. *Proc. of 18th. the International Conference on VLSI Design*, pp. 117-123, 2005.

[19] A. Wolfe and C. Chanin. Executing compressed programs on an embedded risc processor. *in Proc. 25th Annual Int. Symp. Microarchitecture*. pp. 81-91,1992.

[20] M. Guthaus and J. R. et al. MiBench: a free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2002.

[21] J. Fritts. MediaBench Project. *http://euler.slu.edu/ fritts/mediabench/.*

[22] H. Lekatsas, J. Henkel, and W. Wolf. Code Compression for Low Power Embedded Systems Design. *Design Automation Conference DAC-00*, pages 294-299, 2000.

[23] H. Lekatsas, J. Henkel, and W. Wolf. Arithmetic Coding for Low Power Embedded System Design. *Princeton University, NEC USA*, 2000.

[24] H. Lekatsas, J. Henkel, and W. Wolf. Code Compression as a Variable in Hardware/-Software Co-Design. *International Workshop on Hardware/Software Co-Design*, 2000.

[25] H. Lekatsas, J. Henkel, and W. Wolf. H/S Embedded Systems: Design and simulation of pipelined decompression architecture for embedded systems. *Proceedings of the international symposium on systems synthesis*, 2001.

[26] T. Bell, J. Cleary, and I. Witten. Text Compression. *Prentice-Hall, Englewood Cliffs*, 1990.

[27] Y. Yoshida, B. Song, H. Okuhata, T. Onoye and I. Shirakawa. An object code compression approach to embedded processors. *international symposium on Low power electronics and design*, pp. 265-268, 1997.

[28] C. Lefurgy, P. Bird, I. Chen and T. Mudge. Improving code density using compression techniques. *Proceedings of the Annual International Symposium on Microarchitecture (Micro-30)*, pp. 194-203, 1997.

[29] C. Lefurgy and T. Mudge. Code Compression for DSP. *In Compiler and Architecture Support for Embedded Computing Systems*, George Washington University, 1998.

[30] C. Lefurgy, E. Piccininni and T. Mudge. Evaluation of high performance code compression method. *Proceedings of the Annual International Symposium on Microarchitecture (Micro-32)*, pp. 93-102, 1999.

[31] C. Lefurgy, E. Piccininni and T. Mudge. Reducing code size with run-time decompression. *In proceedings Sixth International Symposium on High Performance Computer Architecture*, pp. 218-222, 1999.

[32] C. Lefurgy. Efficient execution of compressed programs. *PhD, University of Michigan*, 2000.

[33] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, Vol. 126, Issue 1, pp. 106-130, 2000.

[34] B. Chazelle, A Minimum Spanning Tree Algorithm with Inverse Ackermann Type Complexity. *Journal of the ACM (JACM)*, Vol. 47, No. 6, pp. 10281047, November 2000.

[35] Y. Nekritch. Decoding of Canonical Huffman codes with Look-up Tables. *Proceedings of the Conf. on Data Compression*, pp. 566, 2000.

[36] X. Kavousianos, E. Kalligeros and D. Nikolos. Multilevel Huffman Coding: An Efficient Test-Data Compression Method for IP Cores. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 6*, pp. 1070-1083, June 2007.

[37] ChipIt Platinum Edition. http://www.prodesigncad.com.

[38] Virtex-II platform FPGA user guide. http://www.xilinx.com/support/documentation/.

[39] S. Klein. Space- and time-efficient decoding with Canonical Huffman trees. *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, pp. 65-75, 1997.

[40] D.C. Burger and T.M. Austin. The SimpleScaler Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News* , vol. 25, issue 3, pp. 13-25. 1997.

[41] D. Sweetman. See MIPS Run. *Morgan Kaufmann, ISBN 1558604103*, 1999.

[42] S. Furber. ARM System-on-Chip Architecture (2nd Edition). 2000.

[43] H. Lekatsas, J. Henkel, and V. Jakkula. Design of an One-cycle Decompression Hardware for Performance Increase in Embedded Systems. *in Design Automation Conference (DAC'02)*, pp. 34-39, 2002.

[44] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, vol. 23, No. 3, pp. 337-343, 1977.

[45] T. Welch. A technique for high-performance data compression. *IEEE computer*, pp. 8-19, 1984.

[46] J. Storer and T. Szymanski. Data Compression Via Textual Substitution. *Journal of the ACM*, vol. 29, No. 4, pp. 928-951, 1982.

[47] A. Aho, J. Ullman and J. Hopcroft. Data Structures and Algorithms. *Addison Wesley, ISBN-10: 0201000237, (January 11, 1983).*

[48] T. Krazit. ARMed for the living room. *CNET News*, 2006, http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html.

[49] T. Krazit. Intel has ARM in its crosshairs. *CNET News*, 2007, http://news.cnet.com/Intel-has-ARM-in-its-crosshairs/2100-1006_3-6210033.html.

[50] B. Frey. PowerPC Architecture Book, Version 2.02. *PowerPC Architect, IBM*, 2005, http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html.

[51] TMS320C62x DSP CPU and Instruction Set Reference Guide.

[52] TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.

[53] GCC online documentation. http:// gcc.gnu.org/onlinedocs/, 2004.

[54] D. Seal. ARM Architecture Reference Manual, 2nd edition. *Addison-Wesley Longman Publishing*, ISBN:0201737191, 2000.

[55] K. KISSELL. MIPS16: high-density MIPS for the embedded market. *Silicon Graphics MIPS Group*, 1997.

[56] Thumb squeezes ARM code size. *Microprocessor Report*, 1995.

[57] Joe Lemieux. Introduction to ARM thumb. *Embedded System Design Report*, 2003.

[58] D. A. Huffman. A method for the constructing of minimum redundancy codes. *Proceedings of the IRE, 4D:1098-110*, 1952.

[59] M. Kozuch and A. Wolfe. Compression of Embedded System Programs. *Proceedings of the IEEE International Conference on ICCD*, 1994.

[60] S. Y. Larin and T. M. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. *Proceedings of the Annual International Symposium on Microarchitecture MICRO-32.*, pp. 82-92, 1999.

[61] S. Liao, S. Devadas and T. K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transaction on Design Automation of Electronic Systems*, 1999.

[62] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. *In ACM SIGPLAN Notices*, Volume 34, Issue 5, pp. 139-149, 1999.

[63] J. Ernst, W. Evans, C. W. Fraser, S. Lucco and T. A. Proebsting. Code Compression. *In ACM SIGPLAN Notices*, Volume 32, Issue 5, pp. 358-365, 1997.

[64] H. Lekatsas and W. Wolf. Code Compression for Embedded Systems. *Design Automation Conference (DAC'98)*, pp. 516-521, 1998.

[65] H. Lekatsas. Code Compression for Embedded Systems. *PhD, Princeton University, 2000.*

[66] I. Chen, P. Bird and T. Mudge. The impact of Instruction Compression on I-cache Performance. *Technical Report CSE-TR-330-97, University of Michigan*, 1997.

[67] L. Clausen, U. Schultz, C. Consel and G. Muller. Java Bytecode Compression for Low-End Embedded Systems. *ACM Transactions on Programming Languages and Systems*, 2000.

[68] G. Aho, P. Centoducatte, R. Azevedo and R. Pannain. Expression-tree-based algorithms for code compression on embedded RISC architecture. *In IEEE Transactions on Very Large Scale Integration VLSI Systems*, VOL. 8, NO. 5, pp. 530-533, 2000.

[69] G. Aho, P. Centoducatte, M. Cortes, and R. Pannain. Code compression based on operand factorization. *In ACM/IEEE International Symposium on Microarchitecture*, pp. 194-201, 1998.

[70] SPEC CPU2000 Benchmarks. http://www.spec.org/cpu2000/, 2000.

[71] IBM. *CodePack (TM) PowerPC Code Compression Utility User's Manual, Version 4.1*, IBM 1998.

[72] M. Game and A. Booker. Code Pack code Compression for PowerPC Processors. *PowerPC Embedded Processor Solutions, IBM*, 2000.

[73] J. Turley. Code compression under the microscope. *EE Times Asia Report*, MAR 2005.

[74] Y. Xie, H. Lekatsas and W. Wolf. Code compression for VLIW Processors. *In Proceedings of Data Compression Conference DCC*, page 525, 2001.

[75] Y. Xie, W. Wolf and H. Lekatsas. A code decompression architecture for VLIW Processors. *In Proceedings 34th ACM/IEEE International Symposium on Microarchitecture*, pp. 66-75, 2001.

[76] Y. Xie, W. Wolf and H. Lekatsas. Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures. *In International Conference on ASIC*, pp. 337-340, 2001.

[77] Y. Xie, W. Wolf and H. Lekatsas. Code Compression Using Variable-to-fixed Coding Based on Arithmetic Coding. *in Proceedings of the Conference on Data Compression*, pp. 382-391, 2003.

[78] Y. Xie, W. Wolf and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. *ACM Proceedings of 15th International Symposium on System Synthesis*, pp. 138-143, 2002.

[79] Y. Xie. Code compression algorithms and architectures for embedded systems. *PhD, Princeton University*, 2002.

[80] Y. Xie, W. Wolf and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. *In IEEE Transactions on Very Large Scale Integration (VLSI) System*, Vol. 14, No. 5, pp. 525-536, 2006.

[81] C. Lin, Y. Xie and W. Wolf. LZW-based code compression for VLIW embedded systems. *Proceedings of the Design, Automation and Test in Europe conf. (DATE04)*, pp. 76-81, 2004.

[82] C. Lin, Y. Xie and W. Wolf. Code Compression for VLIW Embedded Systems Using a Self-Generating Table. *In IEEE Transactions on Very Large Scale Integration (VLSI) System*, VOL. 15, NO. 10, pp. 1160-1171, 2007.

[83] R. Benes, S.M. Nowick, and A. Wolfe. *A fast asynchronous compressed-code embedded processors. 1998 Fourth International Research in Asynchronous Circuits and Systems*, pp. 43-56, 1998.

[84] S. Seong and P. Mishra. *A Bitmask-based Code Compression Technique for Embedded Systems. 24th IEEE/ACM International Conference on Computer-Aided Design (ICCAD06)*, pp. 251254, 2006.

[85] S. Seong and P. Mishra. *An efficient code compression technique using application-aware bitmask and dictionary selection methods. IEEE/ACM Proc. of Design Automation and Test in Europe Conference (DATE07)*, pp. 582-587, 2007.

[86] I. Pavlov. *LZMA SDK. http://www.7-zip.org/sdk.html.*

[87] *Philips Semiconductors: An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture. http://www.nxp.com/acrobatdownload/other/vliw-wp.pdf.*

[88] I. Tuduce and T. Gross. *Adaptive Main Memory Compression. In Proceedings of Technical Conference on USENIX*, pp. 237-250, 2005.

[89] X. Xu, C. Clarke, and S. Jones. High performance code compression architecture for the embedded ARM/Thumb processor. *Proceedings Conference Computing Frontiers*, pp. 451-456, 2004.

[90] I. Tuduce. *Adaptive Main Memory Compression. PhD, Swiss Federal Institute of Technology Zurich*, 2005.

[91] H. Lekatsas, R. Dick, S. Chakradhar and Y. Lei. *CRAMES: Compressed RAM for Embedded Systems. In IEEE/ACM Proceedings on Hardware/Software Co-design and System Synthesis, CODES+ISSS'05*, pp. 93-98, 2005.

[92] Heidi Pan. *High Performance, Variable-Length Instruction Encodings. PhD, Massachusetts Institute of Technology*, 2002.

[93] P. Howard and J. Vitter. Practical implementations of arithmetic coding. *In J. A. Storer, editor, Image and text compression*, pp. 85-112, 1992.

[94] N. Abramson. Information theory and coding. *McGraw-Hill electronic sciences. McGraw-Hill, New York*, 1963.

[95] I. H. Witten, R. M. Neal and J. G. Cleary. Arithmetic coding for data compression. *Communications of ACM*, VOL. 30, NO. 6, pp. 520-540, 1987.

[96] M. Corliss, E. Lewis and A. Roth. DISE: a programmable macro engine for customizing applications. *In Proceedings of the International Symposium on Computer Architecture, ISCA 03*, pp. 362-373. June 2003.

[97] M. Corliss, E. Lewis and A. Roth. The Implementation and Evaluation of Dynamic Code Decompression using DISE. *In ACM Transactions on Embedded Computing Systems*, VOL. 4, NO. 1, pp. 38-72. February 2005.

[98] D. J. Auerbach J. D. Harper T. M. Kemp, R. K. Montoye and J. D. Palmer. A decompression core for PowerPC. *IBM Journal of Research and Development*, VOL. 42, NO. 6, November 1998.

[99] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder. Reducing code size with echo instructions. *In Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, pp. 8494, 2003.

[100] C. W. Fraser. An instruction for direct interpretation of LZ77-compressed programs. *Technical Report MSR-TR-2002-90, Microsoft Research, USA*, September 2002.

[101] M. Thuresson and P. Stenstrom. Evaluation of extended dictionary-based static code compression schemes. *Proceedings of the 2nd conference on Computing frontiers*, pp. 77-86. 2005.

[102] M. Ros and P. Sutton. Compiler Optimization and Ordering Effects on VLIW Code Compression. *Proceedings of the 2003 International Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES 2003)*, pp. 95-103. 2003.

[103] M. Ros and P. Sutton. Code Compression Optimization for VLIW Processors. *PhD, University of Queensland*, 2007.

[104] D. Das, R. Kumar and P. Chakrabarti. Code compression using unused encoding space for variable length instruction encodings. *Proceedings of the VLSI Design & Test Workshop (VDAT)*, 2004.

[105] D. Das, R. Kumar and P. Chakrabarti. Dictionary based code compression for variable length instruction encodings. *Proceedings of the 18th International Conference on VLSI Design*, pp. 545-550, 2005.

[106] T. Okuma, H. Tomiyama, A. Inoue, E. Fajar and H. Yasuura. Instruction encoding techniques for area minimization of instruction ROM. *11th International Symposium on System Synthesis*, pp. 125-130, 1998.

[107] D. Salomon. Data Compression: The Complete Reference. *Springer*, 2007.

[108] V. Cuppu. Data Compression: The Complete Reference. *Cycle Accurate Simulator for TMS320C62x,*, http://www.cs.cmu.edu/afs/cs/academic/class/15745-s07/www/c6xref.

[109] S. J. Nam, I. C. Park and C. M. Kyung. Improving dictionary-based code compression in VLIW architectures. *IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences*, pp. 2318-2324, 1999.

[110] N. Ishiura and M. Yamaguchi. Instruction Code Compression for Application Specific VLIW Processors based on Automatic Field Partitioning. *In Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies*, pp. 105-109, 1997.

[111] J. Prakash and C. Sandeep. A Simple and Fast Scheme for Code Compression for VLIW Processors. *Masters, Indian Institute of Science*, 2003.

[112] J. Prakash, C. Sandeep, P. Shankar and Y. N. Srikant. A Simple and Fast Scheme for Code Compression for VLIW Processors. *Proceedings of the Conference on Data Compression*, pp. 444, 2003.

[113] N. Kadri, S. Niar and A. Baba-Ali. Impact of Code Compression on the Power Consumption in Embedded Systems. *International Conference on Embedded Systems and Applications (ESA)*, pp. 197-203, 2003.

[114] M. Ros and P. Sutton. A hamming distance based VLIW/EPIC code compression technique. *Proceedings of the 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES04)*, pp. 132-139, 2004.

[115] M. Ros and P. Sutton. A post-compilation register reassignment technique for improving hamming distance code compression. *Proceedings of the 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES05)*, pp. 97-104, 2005.

[116] S. K. Menon and P. Shankar. Space/time Tradeoffs in Code Compression for the TMS320C62x Processor. *Technical Report IISc-CSA-TR-2004-4, Indian Institute of Science*, 2004.

[117] C. Lin and C. Chung. Code Compression Techniques Using Operand Field Remapping. *IEE Proceedings on Computers and Digital Techniques, Vol. 149, Issue 1, pp. 25-31*, 2002.

[118] Texas Instruments, Code Composer Studio IDE overview from Texas Instruments. http://focus.ti.com/dsp/docs/dspsupportatn.tsp?familyId=44&sectionId=3&tabId=415&toolTypeId=30, 2006.

[119] S. Debray, W. Evans and R. Muth. Compiler techniques for code compression. *In ACM Proceedings of the 2nd Workshop on Compiler Support for System Software*, pp. 11-24, 1999.

[120] S. Debray, W. Evans, R. Muth and B. Sutter. Compiler techniques for code compression. *ACM Transaction on Programming Languages and Systems*, 22(2), pp. 378-415, 2000.

[121] Charles Price. MIPS IV Instruction Set, Revision 3.2. http://math-atlas.sourceforge.net/devel/assembly/mips-iv.pdf, 1995.

[122] John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 2007.