TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 689 • Publication 689

Jari Heikkinen

Program Compression in Long Instruction Word Application-Specific Instruction-Set Processors

Tampere 2007

Tampereen teknillinen yliopisto. Julkaisu 689
Tampere University of Technology. Publication 689

Jari Heikkinen

Program Compression in Long Instruction Word Application-Specific Instruction-Set Processors

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 7th of December 2007, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2007

ISBN 978-952-15-1869-0 (printed)
ISBN 978-952-15-1917-8 (PDF)
ISSN 1459-2045

ABSTRACT

Modern day embedded systems set high requirements for the processing hardware to minimize the area, and more importantly, the power consumption. Moreover, the ever increasing complexity of embedded applications requires more and more processing power. Application-specific architectures, where the hardware resources can be tailored for a given application, have been introduced to meet these requirements.

Parallel processor architectures have also become favorable as they provide more processing power by utilizing the instruction level parallelism. However, parallel processor architectures result in large program codes, which require large memories that increase not only the area, but also the power consumption of the system due to increased memory I/O bandwidth. Program compression methods have been proposed to tackle this problem and reduce the size of the program code and, therefore, also the area and power consumption.

The focus of this Thesis is on program compression in parallel processor architectures. State-of-the-art program compression methods are surveyed and compared against the derived comparison metrics. Based on the survey, three compression methods are chosen to be evaluated on transport triggered architecture, a parallel processor architecture template used to design application-specific instruction-set processors. The methods are adapted to exploit the characteristics of the architecture. In addition to code density evaluations in terms of compression ratio, an evaluation methodology based on hardware implementations is proposed. It allows to evaluate the effects of compression on the actual area and power consumption of the system.

Program compression may also result in poor instruction-set orthogonality, which makes the programming after compression more difficult and worsens the performance. The orthogonality may turn out to be so poor that the program code cannot be modified anymore. A novel methodology with a small overhead in area and power consumption is proposed to allow to modify the program code also after compression.

PREFACE

The work presented in this Thesis has been carried out in the Institute of Digital and Computer Systems at Tampere University of Technology during the years 2001-2007.

I would like to express my gratitude to my supervisor Prof. *Jarmo Takala* for his professional guidance and encouragement towards doctoral degree. I am also thankful to Prof. *Henk Corporaal* for his invaluable support to my research work. Acknowledgements go also to my Thesis reviewers Prof. *Johan Lilius* and *Jan Hoogerbrugge*, Ph.D., for their constructive and valuable comments on the manuscript.

Many thanks to all my colleagues in the FlexDSP research group for their cooperation. Especially, *Andrea Cilio*, Ph.D., *Tommi Rantanen*, M.Sc., *Jaakko Sertamo*, M.Sc., *Teemu Pitkänen*, M.Sc., and *Pekka Jääskeläinen*, M.Sc., deserve special thanks for helping me in several matters related to my research work. I would also like to thank *Mauri Kuorilehto*, M.Sc., *Panu Hämäläinen*, Dr.Tech., *Mikko Kohvakka*, M.Sc., *Tero Kangas*, Dr.Tech., and *Erno Salminen*, M.Sc., for sharing ideas, knowledge, and opinions, both on and off the topic.

This Thesis was financially supported by Graduate School in Telecommunication System-on-Chip Integration (TELESOC), National Technology Agency (TEKES), Nokia Foundation, Ulla Tuominen Foundation, Foundation of Advancement of Technology, Heikki and Hilma Honkanen Foundation, Jenny and Antti Wihuri Foundation, and HPY Research Foundation, which are gratefully acknowledged.

I wish to express my sincere gratitude to my parents, *Hannu* and *Irma Heikkinen*, and my brother *Jarkko* for their unconditional support and encouragement throughout my academic career and, for that matter, throughout my life. Finally, I thank you *Terhi* for your love, patience, and understanding.

Tampere, November 2007

Jari Heikkinen

TABLE OF CONTENTS

LIST OF PUBLICATIONS

This Thesis is a monograph, which contains some unpublished material, but is mainly based on the following publications. In the text, these publications are referred to as [P1], [P2], . . ., [P6].

[P1] J. Heikkinen, J. Takala, and J. Sertamo, “Code Compression on Transport Triggered Architectures,” in *System-on-Chip for Real-Time Applications*, W. Badawy and G.A. Jullien Eds., pp. 203–213. Kluwer Academic Publishers, Boston, MA, USA, 2003.

[P2] J. Heikkinen, T. Rantanen, A. Cilio, J. Takala, and H. Corporaal, “Evaluating Template-Based Instruction Compression on Transport Triggered Architectures,” in *Proceedings of the International Workshop on System-on-Chip for Real-Time Applications*, Calgary, Canada, June 30 – July 2 2003, pp. 129–195.

[P3] J. Heikkinen, A. Cilio, J. Takala, and H. Corporaal, “Dictionary-Based Program Compression on Transport Triggered Architectures,” in *Proceedings of the IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May. 23–26 2005, pp. 1122–1125.

[P4] J. Heikkinen, J. Takala, and H. Corporaal, “Dictionary-Based Program Compression on TTAs: Effects on Area and Power Consumption,”in *Proceedings of IEEE Workshop on Signal Processing Systems*, Athens, Greece, Nov. 2–4 2005, pp. 479–484.

[P5] J. Heikkinen and J. Takala, “Programmability in Dictionary-Based Compression,” in *Proceedings of International Symposium on System-on-Chip*, Tampere, Finland, Nov. 13–16 2006.

[P6] J. Heikkinen and J. Takala, “Effects of Program Compression,” in *Journal of Systems Architecture*, vol. 53, no. 10, pp. 679–688, Oct. 2007.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-Set Processor
BB	Branch Block
CISC	Complex Instruction-Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DC	DeCode
DCMPR	DeCoMPRess
DCT	Discrete Cosine Transform
DMEM	Data MEMory
DSP	Digital Signal Processing *or* Digital Signal Processor
EPIC	Explicitly Parallel Instruction Computer
EOP	End of a Packet
EX	EXecute
FFT	Fast Fourier Transform
FIFO	First In First Out

FIW	Functional Instruction Word
FPGA	Field Programmable Gate Array
FU	Functional Unit
GCR	Global Connection Register
GCU	Global Control Unit
GPP	General-Purpose Processor
HBL	Hardware Block Library
HDB	Hardware DataBase
HDL	Hardware Description Language
HLL	High-Level Language
ID	IDentifier
IDF	Implementation Definition File
IEEE	the Institute of Electrical and Electronics Engineers
IF	Instruction Fetch
ILP	Instruction-Level Parallelism
IMM	IMMediate unit
JPEG	Joint Photographic Experts Group
LAT	Line Address Table
LIM	Local Instruction Memory
LISA	Language for Instruction Set Architectures
LSU	Load-Store Unit
LZW	Lempel-Ziv-Welch
MAC	Multiply-ACcumulate

MBL	MetaCore Behavioral Language
MPEG	Moving Picture Expert Group
MSL	MetaCore Structural Language
MV	MoVe
NOP	No-Operation
NRE	Non-Recurring Engineering
OSAL	Operation-Set Abstraction Layer
OTA	Operation Triggered Architecture
PC	Program Counter
PDA	Personal Digital Assistant
PICO	Program In, Chip Out
PIG	Program Image Generator
PMEM	Program MEMory
RAM	Random Access Memory
ROM	Read-Only Memory
RISC	Reduced Instruction-Set Computer
RF	Register File
RTL	Register Transfer Level
SFU	Special Functional Unit
SIMD	Single Instruction, Multiple Data
SoC	System-on-Chip
SRAM	Static Random Access Memory
TCE	TTA Codesign Environment

TTA	Transport Triggered Architecture
TPEF	TTA Program Exhange Format
TVLIW	Tagged Very Long Instruction Word
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLES	Variable-Length Execution Set
VLIW	Very Long Instruction Word
XML	EXtensible Markup Language

LIST OF SYMBOLS

CR	compression ratio
S_c	compressed code size
S_t	decoding table size
S_u	uncompressed code size
k	budget of templates
τ	set of templates
W	program code size
T_l	template l in τ
n	number of minimal templates
C_i	unique move slot and long immediate field combination i
f_i	usage frequency of C_i
$N(\tau, C_i)$	narrowest template T_l in τ to encode C_i
$w(T_l)$	width of template T_l
T_{C_j}	candidate template j
$v(C_i)$	lower bound of the width of T_{C_j}
b_{C_j}	benefit of T_{C_j}
K	number of bit patterns in program code
N	number of unique bit patterns in program code

C	connectivity of a register file
w	width of bit pattern
I_s	set of buses to which an input socket *i* is connected to
O_s	set of buses to which an output socket *i* is connected to

1. INTRODUCTION

The recent advances in the semiconductor technologies, fabrication processes, and design methodologies have allowed to implement more and more functionality on a single chip, even entire systems that are composed of programmable processors cores, memories, customized hardware accelerators, and radio frequency and analogue parts. Such a system is often referred to as System-on-Chip (SoC). The higher integration rate has allowed to reduce the size and, therefore, the cost of digital systems and has enabled the introduction of many new digital products. The growth has been explosive especially in the embedded systems market. Embedded system usually refers to a computing system that has been specifically designed to control some device, i.e., the system is not used for general purpose computing. Embedded systems are also often embedded into the device they control. Nowadays, the market share of embedded systems far surpasses that of the general purpose computing systems, such as personal computers.

Embedded systems can nowadays be found in almost every aspect of our everyday life; automobiles, home automation, banking, multimedia, and telecommunications to name a few. The most recent advances have concentrated on the market of portable handheld embedded systems. These systems, like cellular phones, personal digital assistants (PDA), game consoles, and media players, are often limited by constraints on size, weight, battery life, and cost. Hence, the area and, nowadays more importantly, the power consumption have turned out to be the most important design constraints in the embedded system development.

Apart from the area and power consumption requirements, embedded systems require nowadays more and more processing power from the computing hardware due to the increased complexity of embedded applications. The requirements for small area and low power consumption but high performance have lead to the utilization of application-specific structures where the hardware resources can be tailored according to the requirement of the application. This allows to meet the performance requirements. The area and power consumption are also reduced as the system contains only the hardware resources that are required to execute the given applications.

Recently, very long instruction word (VLIW) architectures have gained considerable popularity in embedded systems, especially in digital signal processing (DSP) tasks, due to their modularity and scalability [100]. VLIW architectures can provide more processing power through utilizing the instruction-level parallelism (ILP) available in the application by executing operations in parallel in concurrently operating functional units (FU). The FUs are controlled by a long instruction word that contains dedicated fields for each FU. This kind of an instruction encoding leads to poor code density, i.e., large size of the program code, as the full processing power of the architecture cannot always be fully utilized [29].

The size of the program code is increasing also due to increased complexity of the applications that are developed for the embedded systems. Furthermore, high-level languages (HLLs), like C and C++, are gradually replacing assembly language in writing embedded applications due to lower application development and maintenance costs. This incurs a penalty in the code size as the compiler cannot achieve the quality of the hand-written and hand-optimized assembly code when the HLL code is translated into machine code. Moreover, compilers traditionally favor performance at the cost of code size.

Large program codes require large memories, which may lead to systems where the memories consume more area than the actual processor core [14]. Large memories may also increase the power consumption of the system due to higher memory I/O bandwidth. This reduces the battery life. Therefore, methods to reduce the size of the program code need to be developed in order to preserve area, and more importantly, reduce the power consumption of the program memory and the entire system.

Program compression methods have been proposed to reduce the size of the program code to allow to use smaller memories and hence save the area and power consumption. The program is compressed during compile-time and stored in compressed form into the program memory. During execution, the compressed instructions are fetched from the program memory and decompressed back to their original form using a dedicated decompressor hardware before they are decoded into control signals that control the hardware resources of the processor.

1.1 Objective and Scope of Research

The objective of this Thesis is to develop an effective program compression methodology for a new type of customizable parallel processor architecture, the transport triggered architecture (TTA). Transport triggered architecture suffers from poor code density due to the programming model of the architecture and the minimal instruction encoding that has been utilized to simplify instruction decoding.

The first objective is to perform a survey over the proposed program compression approaches on parallel processor architectures. The survey is performed by defining the main metrics related to program compression, which are then used to perform a comparison over the methods included in the survey.

The second objective is to adapt and utilize the most effective methodologies found in the survey on TTA. In addition to code density evaluations, the processor systems executing both compressed and uncompressed instructions are implemented in hardware. This allows to evaluate the compression methods in terms of area, and more importantly, power consumption, which provide more accurate estimates on the effectiveness of the compression methods. Such measures have been rarely reported for program compression approaches.

The third objective is to design a methodology that allows to maintain the programmability, i.e., allow to make modifications to the program code also after the compression has been applied. In case the hardware decompressor is implemented using nonprogrammable structures, such as standard cell logic, the programmability may be lost as the decompression hardware is tailored for a particular application or a set of applications and cannot be modified after the processor has been implemented in hardware. The objective of the methodology is to provide full programmability with a small overhead in area, power consumption, and performance.

The fourth objective is to integrate the TTA program compression methodology to the TTA codesign environment toolset. This allows to include the program compression in the TTA processor design and code generation flow. The binary image generator of the TTA codesign environment is designed to provide a methodology that allows easy integration and modification of the compression methodology. The compressor module generates not only the compressed binary code, but also the hardware description of the decompressor hardware.

1.2 Main Contributions

In this Thesis, a design methodology for compressing the program code to reduce the area and power consumption is developed for transport triggered architecture. To summarize, the main contributions are the following:

- Study and comparison of state-of-the-art program compression methods on parallel processor architectures by comparing the methods against the defined comparison metrics
- Utilization of dictionary-based, Huffman encoding, and instruction template-based compression methods on transport triggered architecture by employing the characteristics of the architecture to achieve better compression
- Evaluation methodology based on hardware implementations of the processor designs to measure the effects of program compression on area and power consumption, which allows to include the decompression overhead in the results
- Novel methodology to maintain the programmability and increase the orthogonality of the instruction set that is affected when program compression methods are utilized
- Integration of the program compression methodology to the transport triggered architecture codesign environment, allowing to utilize plug-in compressor modules to compress the program code and generate the decompressor and integrate it to the hardware description of the processor

1.2.1 Author's Contribution

The author was responsible for making the survey of state-of-the-art program compression methodologies and to define the comparison metrics that were used to compare the proposed compression methodologies.

The author was responsible for adapting Huffman coding, dictionary-based compression, and instruction template-based compression methods on transport triggered architecture. The profiling tools for the Huffman coding and dictionary-based compression were implemented by the author. The program code profiling and template selection tool for the instruction template-based compression was implemented by M.Sc. Tommi Rantanen. The code density estimations of the utilized compression method on TTA were carried out by the author. The results of these studies have been presented in [P1, P2, P3].

The author also derived the hardware implementations for the dictionary-based and instruction template-based compression approaches and evaluated the hardware implementations in terms of area and power consumption. The results of these experiments have been reported in [P4, P6].

The author was also responsible for developing the dictionary extension methodology to maintain the programmability after the compression has been utilized. The author proposed and designed also the methodology to minimize the overhead of the dictionary extension on area and power consumption of the decompression logic. The author was responsible for performing the evaluations of the proposed methodology in terms of area, power consumption, and performance. The methodology and the results of the evaluations have been published in [P5].

The integration of the compression methodology in the TTA Codesign Environment has been designed together with Pekka Jääskeläinen, M.Sc., and Lasse Laasonen, M.Sc.. Lasse Laasonen was responsible for the implementation of the program image generator and the processor generator of the codesign environment with support for program compression.

The work reported in this Thesis has been published earlier in six publications [P1–P6]. Therefore, some Chapters contain verbatim extracts from these publications. These extracts are under copyright of respective copyright holders. None of the publications has been used in another person's academic Thesis.

1.3 Thesis Outline

Chapter 2 presents the related work on program compression. A detailed survey of the program compression methods proposed for parallel processor architectures is presented. The methods are compared in terms of the defined comparison metrics.

Processor hardware customization is discussed in Chapter 3. An overview of the related work on customizable processor architectures is given. The main objective of the Chapter is to present the details of the TTA, which is the customizable parallel processor architecture utilized in this Thesis. The Chapter presents also the MOVE framework and TTA Codesign Environment (TCE), which are semi-automatic design methodologies for designing TTA processors. Tools of the MOVE framework were used in this Thesis for designing the TTA processors and compiling the benchmark applications that were used in the evaluation of the compression methodologies on TTA. The designed program compression methodology developed in this Thesis was integrated to the tools of the TCE.

Chapter 4 describes the principles of Huffman coding, dictionary-based compression, and instruction template-based compression and how the methods were utilized on TTA. In addition, the details of the hardware implementations of the dictionary-based and instruction template-based compression methods to evaluate the effects of the compression methods on area and power consumption are presented. Implementation details of the hardware decompressor are also explained.

In Chapter 5, a description of a dictionary extension method to maintain the programmability after compression is given. The results of utilizing the three compression methods to compress program code compiled on TTA processors are presented in Chapter 6. The effectiveness of all the three compression methods is measured in terms of compression ratio. In addition, the results of the hardware implementation of the dictionary-based and instruction template-based compression methods are given. The effects of the proposed dictionary extension method to maintain the programmability on the area, power consumption, and performance are also presented. Chapter 7 concludes the Thesis.

2. PROGRAM COMPRESSION

In computer science, compression traditionally refers to data compression, which can be understood as a process of encoding data to save data storage space or reduce transmission bandwidth. Although the data has already been encoded in digital form, i.e., represented in bits, it can often be encoded more effectively using less bits. Data contains usually redundancy that can be removed by using specific encoding algorithms. This reduces the number of bits required to represent the data and, therefore, allows to reduce the cost of the data storage space or communication network as less bits need to be stored or transmitted.

In principle, compression is performed by first forming a model of the data to be compressed and then using this model in the encoder to transform the input data into compressed form. Encoding is done by utilizing the model to transform data units, denoted as symbols, into codewords that represent the original symbols with fewer bits. Symbols can be arbitrary units of data, e.g., a byte, a word, or a cache line. The model typically characterizes statistical properties of the symbols in the input data. For example, a model may represent the probabilities of the symbols in the input data, i.e., how frequently each symbol exists. This information is then utilized to perform the compression, e.g., by assigning variable-width codewords to symbols based on their probabilities of occurrence.

Compressed data cannot be interpreted directly by the receiver of the data as the encoding has changed. Hence, prior to using the compressed data, either fetched from the code storage or transmitted over the communication channel, it has to be decoded back to the format in which the data was originally encoded in. This process is denoted as decompression. An identical model to the one used in the encoding process is required to decode the codewords back to the original symbols that represent the original input data. The compression and decompression phases and the data flow in between these two phases is exemplified in Fig. 1.

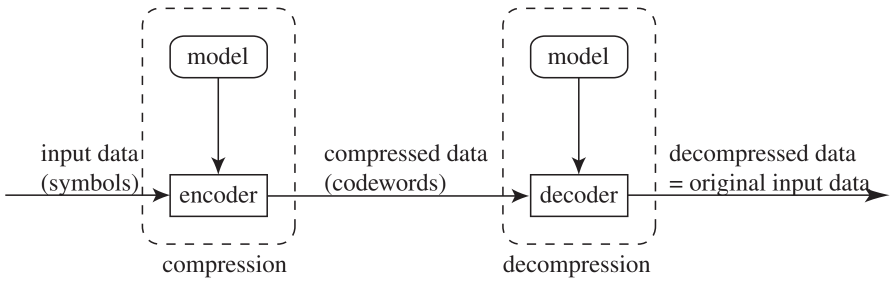


Fig. 1. *Principle of compressing and decompressing data.*

Compression algorithms may be either lossy or lossless. Lossy algorithms trade some loss of data for more effective compression while lossless compression algorithms do not allow any loss of data, i.e., the original data can be perfectly reconstructed when decompression is performed. Lossy algorithms have traditionally been applied for images, audio, and video, where some loss of quality can be tolerated without losing the essential data so that the data can still be perceived. Text compression, on the other hand, requires lossless compression so that the data corresponding to the characters of the text can be perfectly recovered during decompression.

The increased complexity of applications, especially in embedded systems, has lead to an increase in the size of the program code. This has set higher requirements for the code storage space. It has been estimated that the cost of the memory that stores the program code may nowadays be up to 50% of the total cost of the embedded system [39]. Hence, interest has been raised to compress also the program code. Program compression can be considered as a special case of data compression.

Program compression differs from the data compression in the sense that the data to be compressed is not general data. It has a hierarchical structure, which may allow to achieve better compression. Program code consists of instructions that all have a specific structure. Specific information inside the instruction word is in most cases in the same locations among all the instruction words, depending of course on the instruction format. This allows to inspect the data in smaller fragments and find more redundancy that can be removed by applying the compression methods. Furthermore, program code typically uses only a small part of the possible instructions in the instruction set. This property can also be utilized and the instructions can be encoded so that only the actual instructions used in the program code are covered in the compressed program code. This allows to encode the instructions with fewer bits.

Program compression requires also that the compression is lossless. The original representation of the program has to be perfectly reconstructable so that the program can be executed correctly. Another requirement is to guarantee random access decompression, i.e., to allow the decompression to start from any location in the program code or at some block boundaries [78]. As the execution flow may be discontinuous due to branch and jump operations, the decompression has to be capable of starting directly at the branch targets. This requirement is usually guaranteed by compressing the program in blocks inside which the execution flow is continuous, i.e., there are no branch or jump targets within the blocks. The blocks are chosen so that the branch and jump targets are directly at the beginnings of these blocks. Compression blocks need to be aligned to addressable memory locations so that the target instructions can be interpreted directly after the branch or jump has been taken.

Program compression is typically performed during compile-time. Therefore, the compression speed is of little importance. Decompression, on the other hand, is performed during run-time, so it adds an overhead to the execution time. Decompression is typically performed by a dedicated decompression hardware that is added to the system. Hence, it affects also the hardware cost of the system. The overhead in the execution time due to decompression depends not only on the complexity of the compression algorithm and the implementation of the decompression hardware, but also on the location of the decompressor in the system. The decompressor can be placed in different locations in the system. Usually, the decompressor is placed in between the program memory and the processor core. If the system contains a cache in between the processor core and the program memory, the decompressor can be placed either pre- or post-cache. The decompressor can also be placed inside the processor core. In such a case, decompression is usually performed in between the instruction fetch and decode stages of the instruction pipeline. Figure 2 illustrates the different alternatives for placing the decompressor.

Initially, program compression methods were proposed for single issue processor architectures, such as *reduced instruction-set computers (RISC)*, *complex instruction-set computers (CISC)*, and single-issue *digital signal processors (DSP)*. Later, when parallel processor architectures, such as VLIW, started to gain popularity, program compression methods started to emerge for those architectures as well. Program compression approaches on single-issue and parallel processor architectures are introduced in this Chapter. As this Thesis concentrates on parallel processor archi-

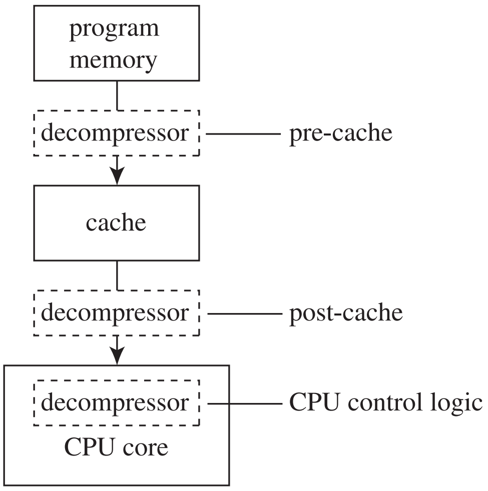


Fig. 2. *Alternative locations for the decompressor.*

tectures, the compression methods on single-issue processor architectures have been discussed only briefly. The compression methods proposed for parallel processor architectures are discussed in more detail. A comparative analysis of the program compression methods on parallel processor architectures based on the defined compression metrics is also given.

2.1 Program Compression on Single-Issue Processor Architectures

The first compression methods on single-issue processor architectures were mostly adaptations of the traditional data compression methods. In one of the earliest approaches [123], Wolfe and Chanin used Huffman coding [50] to compress the instructions of a 32-bit RISC processor architecture. Huffman coding assigns variable-width codewords to the symbols being coded based on their usage frequencies. The most frequent symbols are assigned the shortest codewords and vice versa. The work was extended in [63] by experimenting the Huffman coding on five different 32-bit RISC architectures. Huffman coding on RISC architectures has further been experimented in [36, 56, 57, 75, 77, 89, 119]. Huffman coding has also been applied on DSP architectures in [95].

Another statistical compression method, arithmetic coding [103], has been studied extensively by Lekatsas et. al. on RISC architectures in [76–82]. Arithmetic coding is similar to Huffman coding as it assigns codewords to symbols based on their usage frequencies. However, arithmetic coding is not limited to integral number of bits as Huffman encoding and can therefore perform better. The drawback of arithmetic coding is its more complex decompression procedure as it requires complex arithmetic operations. Lekatsas et. al. have applied reduced-precision arithmetic coding [49] in their work to replace the complex arithmetic operations with table look-ups to make decompression less compute-intensive and, therefore, better suitable for program compression.

Dictionary-based program compression is another extensively studied program compression method. Dictionary-based compression identifies the unique instructions in the program code and stores them into a dictionary and replaces the occurrences of these instructions in the program code with indices to the dictionary. Dictionary-based compression methodologies have been proposed for RISC architectures in [14, 17–20, 58, 72, 73, 75, 80, 132, 133]. Dictionary-based compression has also been applied on DSP architectures in [26, 84, 85, 112].

Dictionary-based compression can be applied at different symbols granularity levels. In addition to assigning codewords to single instructions, instruction sequences can be considered as symbols to be stored into a dictionary and be replaced with a single dictionary index, as proposed in [27, 53, 71]. Instructions can also be divided to smaller fields inside which unique bit patterns are searched for to be stored into the dictionary [28, 74, 89, 93, 98, 131]. The division can be made according to the existing fields in the instruction word, i.e., the opcode, operand, and immediate fields, or by dividing instructions arbitrarily to smaller fields, e.g., by dividing a 32-bit instruction to two 16-bit halves.

An adaptive dictionary-based compression scheme, proposed by Ziv and Lempel in [135, 136], has also gained success in program compression. The dictionary is generated adaptively when symbols are encountered. The incoming symbols are replaced with pointers that point to locations where the symbols occurred previously. The methodology proposed by Lempel and Ziv has been adapted and applied on RISC architectures in [15, 62, 73, 75].

The program code size can also be reduced by re-encoding the instructions in the instruction set to be represented with fewer bits. For example, ARM Thumb [16], SH3 [44], and MIPS16 [61] are architectures where the original 32-bit RISC instructions are re-encoded to 16-bit instructions. This is accomplished by, e.g., restricting the amount of registers the instructions can use as operands. A methodology is usually provided to execute also the original 32-bit wide instructions. Micro RISC [40], Heads and Tails [94], and EISC [70] are further examples of tightly encoded 16-bit RISC architectures. Simonen et. al. have experimented instruction set re-encoding on DSP architectures in mixed 32/16-bit execution mode in [110].

Aside from compressing the already compiled program code, attention can be given to reduce the code size already during the compilation phase. One example of such an alternative is to factor out frequently executed instruction patterns into subroutines and replacing these instructions with calls to the subroutine [32, 60, 107, 113]. Similar methods have also been proposed for DSP architectures in [83, 86].

2.2 Program Compression on Parallel Processor Architectures

Program compression methods have also been proposed for parallel processor architectures, where the poor code density is even a bigger problem than on single issue processor architectures. Parallel processor architectures typically require a long instruction word to control explicitly each of the concurrently operating hardware resources. This kind of an encoding usually leads to increased size of the program code and, therefore, to poor code density.

As the main topic of this Thesis is program compression on parallel processor architectures, a detailed survey of the proposed compression methods on parallel processor architecture is presented. The objective of the survey is to describe the methods proposed in the literature and provide a comparison between them. Direct comparison of the methods is fairly difficult as the methods have been proposed for different target architectures and the specific features of the architectures are often utilized in the compression methods. Furthermore, in many cases, the required details for direct comparison are not reported, or are reported using different metrics. For example, quantitative comparison between different methods is difficult as there are several ways in which the effectiveness of the method has been presented. Therefore, the comparison of the methods is more feasible when the different aspects are characterized and compared separately.

2.2.1 Comparison Metrics

In order to compare the compression methods, some metrics are defined. They try to capture the essential characteristics of the compression methods. The defined metrics to ease the comparison of the program compression methodologies on parallel processor architectures include target architecture, instruction encoding, compression granularity, decompression properties, branch handling, and effectiveness. These metrics are described in more detail in the following.

Target Architecture

Target architecture defines the details of the processor architecture on which the compression method has been applied on. All of the processor architectures included in the survey belong to the class of very long instruction word architectures that provide more processing power by executing several operations in parallel in concurrently operating function units. Special classes of VLIW architectures included in the survey are the explicitly parallel instruction computing architecture (EPIC), and TTA. TTA is the processor architecture for which program compression methods are developed in this Thesis.

The target architectures can be compared in terms of the degree of available parallelism, i.e., how many operations they can issue in parallel. This has an effect on the size of the program code and also on the effectiveness of the compression method. Parallel processor architectures are typically programmed using a wide instruction word that has dedicated fields for each of the concurrently operating functional resources. Hence, the more parallel resources there are, the wider the instruction word. Figure 3 shows an example of a VLIW instruction word that has three dedicated fields to specify operations for three functional resources. Each field is comprised of the opcode and operand fields that are required to specify a single RISC-type operation.

Most programs contain parallelism that can be exploited by the parallel resources of the architecture. However, programs contain also parts where the data dependencies limit the available parallelism. This results in sequences of instructions where only few of the parallel resources can be utilized. Null values, i.e., no-operations (NOPs) need to be explicitly defined for the unutilized functional resources. The greater the degree of parallelism, the more NOPs need to be defined. This increases the size

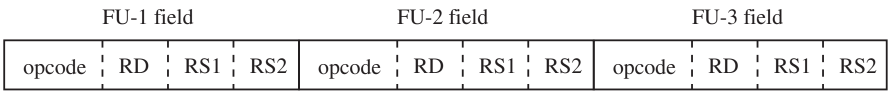


Fig. 3. *An example of VLIW instruction word. FU-(x): functional unit field. Opcode: Operation code field. RD: Destination register field. RS(y): Source register field.*

of the program code. Consequently, this also makes the compression more effective as there are more NOPs that can be exploited in the compression to achieve more compact code.

The degree of parallelism on VLIW and EPIC architectures can be expressed in terms of operation issue-rate, i.e., the number of operations that can be issued in parallel. As TTA has a different programming paradigm, operation issue-rate cannot be used to measure the degree of parallelism. TTA processors are programmed by explicitly specifying the data transports that transport data between functional resources. The number of data transports required to complete a single operation depends on the operation type. A typical three-operand RISC-like operation requires three data transports. On the other hand, jump instruction requires only one data transports. TTA allows also operand bypassing, which means that two operations can share a data transports. Therefore, it is difficult to estimate the operation issue-rate on TTA. A better measure is to express the degree of parallelism in terms of data transport issue-rate, i.e., how many data transports can be issued in parallel.

Instruction encoding

Instruction encoding in the context of this Thesis defines how the original instructions are encoded into compressed codewords. More accurately, it defines whether the original bit patterns in the program code, e.g., instructions or operations slots, are encoded to fixed- or variable-width codewords. Fixed-width codewords are simple from the instruction fetch and decompression point of view as all the codewords are of same width. Variable-width codewords complicate both the instruction fetch and decompression logic as the width of the codeword is not known during instruction fetch or at the beginning of decompression.

Fixed-width codewords are easy to fetch from the program memory. The codewords can be aligned evenly to the program memory. Usually the instruction fetch packet can be adjusted to be as wide as a single codeword. This allows to fetch one codeword at a time from the program memory. Decompression is also fairly straightforward as there is no need to first detect the width of the codeword and extract it from the bit stream fetched from the program memory before it can be decompressed. Fixed-width codewords usually result from dictionary-based compression methods. Codewords represent indices that point to unique bit patterns stored into the dictionary. Decompression is simple as the index can be used directly to access the dictionary to obtain the corresponding bit pattern.

Variable-width instructions are more problematic from the instruction fetch and decompression point of view as the codewords are of different width and the width of the codeword is known only during the decompression phase. The instruction fetch packet is usually configured to be as wide as the widest possible compressed instruction word to guarantee that during each clock cycle enough bits are fetched from the program memory to cover an entire compressed instruction word. This guarantees that one instruction word per clock cycle can be decompressed and stalling of the instruction pipeline can thus be avoided. The decompressor becomes also more complicated as the width of the compressed instruction has to be determined before the decompression can be performed. Buffers and shift registers are also needed in order to handle the incoming bit stream and to avoid overflow in case the compressed instruction consumes less bits than have been fetched from the program memory. Variable-width codewords results from the compression methods that utilize the non-uniform probability distribution to encode symbols with variable-width codewords, such as Huffman coding and arithmetic coding.

Compression granularity

Compression granularity defines the granularity of the bit patterns that are considered as symbols for compression. The granularity level affects directly the possibility to find redundancy, i.e., find repeated bit patterns from the program code. This affects the size of the generated coding tables. The more fine-grained the granularity level, i.e., the smaller the bit patterns that are considered as symbols for compression, the greater the probability to find redundancy from the program code. On the other hand, fine-grained granularity level means that the entire compressed instruction words will be represented with several codewords. This increases the width of the compressed instruction word and, therefore, also the size of the compressed program code. Higher granularity level leads to less possibilities to find repeated bit patterns from the program code, but on the other hand, to smaller size of the compressed program code as it consists of only a single or a few codewords that correspond to the original bit patterns stored into a coding table.

Compression methods included in this survey consider several different granularity levels. The highest granularity level corresponds to entire VLIW-type instruction words. The other more fine-grained levels utilized in the methods include the granularity levels of operation slots, operation slot sequences, opcode fields, and fields of arbitrary size, e.g., bytes that do not correspond to any structural fields of the instruction words.

Decompressor Implementation

Decompressor implementation defines the location of the decompression hardware in the processor system. The decompressor is in most cases placed outside the processor core, in between the core and the program memory. With the existence of a cache, the decompressor can be placed either pre- or post-cache. The decompressor can also be placed inside the control path of the processor core. Figure 2 already presented the alternative locations of the decompressor. The location of the decompressor affects both the effectiveness of the compression and the performance.

In case the decompressor is placed pre-cache, only the program memory is in compressed form. Instructions are fetched from the program memory during cache miss and decompressed back to the original format before they are placed in the cache. This alternative hides the decompression latency behind the cache miss. Decompression latency is paid only when a cache miss occurs. In post-cache implementation, also the cache is in compressed form. This means that the total size of the code storage in the system is smaller as the size of the cache can be decreases. Decompression is performed when instructions are fetched from the cache and consumed in the processor core. Decompression latency is paid whenever instructions are fetched from the cache. This tends to increase the execution time. However, as the cache contains compressed instructions, more instructions fit in the cache if its size is maintained unchanged. This improves the cache hit ratio and reduces the number of cache misses, i.e., shortens the execution time. The net effect depends on the compression parameters and the code to be compressed. Hence, there is a tradeoff in between minimizing the size of the code storage space and the performance.

When the decompressor is placed in the control path of the processor core, the complexity of the decompressor may require it to be implemented in a separate pipeline stage in between the instruction fetch and decode stages. The additional pipeline stage increases the depth of the instruction pipeline and, therefore, the branch delay. The increased branch delay is paid only when a branch is taken. In some of the compression approaches the decompression procedure is fairly simple, e.g., only a simple look-up table access in the dictionary-based compression. This allows to integrate the decompressor to the logic of the instruction fetch or decode stage without affecting the clock period too much. In such a case, the performance is maintained as the depth of the instruction pipeline is unchanged.

Random Access Support

Random access support defines how the change of the execution flow due to branches and jumps can be supported after the program code has been compressed. Random access is maintained if the decompressor can start the decompression process directly from the target instruction after a branch or a jump is executed. Program compression complicates the random access support because the program representation in the program memory is typically changed due to encoding the original instructions with compressed instructions. Change in the program representation means that the addresses of the target instructions have changed. Moreover, the target instructions may end up to be placed in unaddressable locations in the program memory if no instruction alignment is made.

Random access can be typically guaranteed by compressing the program code in blocks inside which the program flow is continuous. The beginnings of the compressed blocks, representing branch and jump targets, have to be aligned to addressable locations in the program memory to guarantee that after a branch or a jump the execution continues immediately from the correct target instruction. This alignment may require some padding bits to be inserted in the program memory before the target instructions. In addition, as the addresses of the instructions in the program memory have changed, some sort of translation between the compressed and uncompressed address spaces is required. One alternative is to modify the branch and jump target addresses in the instructions to correspond to the target addresses in the compressed address space [71]. Another alternative is to use a *Line Address Table (LAT)* that provides a mapping between the original and the compressed addresses spaces [122].

Effectiveness

Effectiveness defines how effectively the size of the program code can be reduced. Effectiveness of a compression method is typically measures in terms of compression ratio, which defines the ratio of the compressed and uncompressed code sizes. Compression ratio has to include also the overhead of the decoding tables that are used in the decompression phase. The decoding tables usually define the model that can be used to decompress the compressed instructions back to the original form. Hence, compression ratio *CR* can be defined as

$$CR = \frac{S_c + S_t}{S_u}$$

where, S_c is the size of the compressed code, S_t the size of the decoding tables, and S_u the size of the uncompressed code.

2.2.2 Comparison of Methods

Tables 1- 3 list the proposed compression approaches for parallel processor architectures and present their classification according to the comparison metrics. Table 1 shows the details of the compression method, the target architecture, and the type of instruction encoding used for the methods. Table 2 describes the compression granularity and the decompressor implementation, i.e., the location of the decompressor in the processor system. Table 3 considers the random access support mechanisms and presents statistics on the effectiveness of the compression methods in terms of compression ratio. The table illustrates also whether the presented compression ratio takes into account the decompression overhead, i.e., the size of the decoding table.

The columns in the tables correspond to the comparison metrics that were presented in the previous Section. Each row in the table corresponds to a compression method. Each method is named based on the first author of the publication in which the method is described. Reference to this publication is also given. On few occasions,

Table 1. *Comparison of the target architecture and instruction encoding of the program compression methods on parallel processor architectures. NA: Not available.*

Method		Target Architecture		Instr. enc.	
Author	Ref.	Architecture	Max. issue rate	Fixed-width	Variable-width
NOP removal					
Colwell	[29]	Trace VLIW	28 ops.		x
Conte	[30]	Tinker VLIW	NA		x
Weiss	[118]	DSP16 VLIW	NA		x
Richter	[102]	M3-DSP VLIW	16 ops.		x
Starcore	[1]	SC 140 VLIW	6 ops		x
Texas Instruments	[3–5]	TMS320C6x VLIW	8 ops.		x
Suzuki	[114]	VLIW	2 ops.	x	
Aditya	[10]	VLIW	4-12 ops.		x
Haga	[42]	IA-64 EPIC	6 ops.		x
Heikkinen	This work	Move TTA	3-13 trans.		x
Dictionary-based					
Nam	[92]	Sparc-based VLIW	4-12 ops.	x	
Hoogerbugge	[48]	TM 1000 VLIW	5 ops.	x	
Ros	[104]	TMS320C6x VLIW	8 ops.	x	
Ros	[105]	TMS320C6x VLIW	8 ops.		x
Lin	[88]	TMS320C6x VLIW	8 ops.	x	
Piccinelli	[97]	ST200 VLIW	4 ops.		x
Ibrahim	[51]	VLIW	NA	x	
Heikkinen	This work	Move TTA	3-13 trans.	x	
Entropy encoding					
Larin	[68]	TEPIC VLIW	6 ops.		x
Xie	[125, 126]	TMS320C6x VLIW	8 ops.		x
Xie	[127]	TMS320C6x VLIW, IA-64 EPIC	8 ops., 6 ops.	x	
Heikkinen	This work	Move TTA	3-13 trans.		x
Instruction set re-enc.					
Larin	[68]	TEPIC VLIW	6 ops.		x
Biswas	[21]	TMS320C6x VLIW	8 ops.	x	
Liu	[90]	VLIW	4 ops.		x

two or more methods have been combined into a single row due to their tight correspondence between each other. The compression methods in the table are organized into four compression categories based on the general compression principle. Compression approaches inside each category have been arranged based on the year they were published. The compression categories include *NOP removal, dictionary-based compression, Entropy encoding*, and *instruction set re-encoding*. The compression methods included in the survey are presented in the following Subsections based on the above mentioned compression category classifications.

Table 2. *Comparison of the compression granularity and decompressor implementation of the program compression methods on parallel processor architectures. PMEM: Program Memory. CPU: Central Processing Unit.*

Method		Granularity					Decompressor impl.			
Author	Ref.	Instruction word	Operation slot	Operation slot sequence	Opcodeopcode field	Arbitrary bit sequence	Pre-cache	Post-cache	Between PMEM and CPU	CPU control path
NOP removal										
Colwell	[29]		x				x			
Conte	[30]		x				x	x		
Weiss	[118]			x						x
Richter	[102]			x						x
Starcore	[1]		x							x
Texas Instruments	[3–5]		x							x
Suzuki	[114]		x							x
Aditya	[10]		x							x
Haga	[42]		x							
Heikkinen	This		x							x
Dictionary-based										
Nam	[92]				x				x	
Hoogerbugge	[48]			x					x	
Ros	[104]		x	x					x	
Ros	[105]		x						x	
Lin	[88]					x		x		
Piccinelli	[97]		x						x	
Ibrahim	[51]	x								x
Heikkinen	This	x	x		x					x
Entropy encoding										
Larin	[68]		x		x	x		x		
Xie	[125, 126]		x	x			x			
Xie	[127]					x	x			
Heikkinen	This	x	x		x					x
Instruction set re-enc.										
Larin	[68]				x					x
Biswas	[21]		x							x
Liu	[90]			x						x

NOP Removal

VLIW architectures are typically tailored for the highly parallel sections of the program code to fully utilize the parallelism available in the application. In the less parallel sections of the program this results in large number of NOPs to be specified

Table 3. *Comparison of the random access support and effectiveness of the program compression methods on parallel processor architectures.*

Method		Random access			Effectiveness	
Author	Ref.	Address lookup table	Address patching	Alignment	Compr. ratio	Decoding table included
NOP removal						
Colwell	[29]		x	x	NA	NA
Conte	[30]		NA		NA	NA
Weiss	[118]		NA		NA	NA
Richter	[102]		NA		NA	NA
Starcore	[1]		x	x	NA	NA
Texas Instruments	[3–5]		x	x	NA	NA
Suzuki	[114]		x	x	0.56-0.65	No
Aditya	[10]		x	x	0.12-0.37	No
Haga	[42]		x	x	NA	NA
Heikkinen	This		x	x	0.36-0.56	No
Dictionary-based						
Nam	[92]		-		0.63-0.71	Yes
Hoogerbugge	[48]		NA		0.20	No
Ros	[104]		x	x	0.81-0.89	Yes
Ros	[105]	x			0.73-0.82	Yes
Lin	[88]	x			0.75	NA
Piccinelli	[97]	x			0.68	Yes
Ibrahim	[51]		NA		NA	NA
Heikkinen	This		-		0.53-0.76	Yes
Entropy encoding						
Larin	[68]	x		x	0.30-0.75	No
Xie	[125, 126]	x	x		0.67-0.80 / 0.84-0.89	No
Xie	[127]			x	0.70 - 0.83 / 0-56-0.73	No
Heikkinen	This		x	x	0.41-0.82	Yes
Instruction set re-enc.						
Larin	[68]		x	x	0.64	No
Biswas	[21]		-		0.75	No
Liu	[90]		NA		0.33-0.39	NA

for the FUs that do not perform an operation. This worsens the code density. Several approaches have been proposed to address this issue.

Colwell et. al. were the first ones to propose a program compression method on VLIW architecture [29]. They utilized their method on a 28-issue Trace VLIW architecture. They proposed to use a “mask” identifier to precede each instruction word and specify which operation slots are present in the instruction word. This method allows to remove the NOPs from the instruction words and hence improve the code density. The decompression phase to reconstruct the compressed instructions back to their original form was performed in the cache.

Conte et. al. proposed a flexible instruction encoding for VLIW architectures that was experimented in the TINKER VLIW test suite [30]. In the TINKER encoding, individual operations are combined into a MultiOp, which is a parallel unit of issue. The MultiOp is defined by a header, which is followed by the operations that are to be issued in parallel. Each operation can issue an operation in one of four types of FUs; integer, memory, floating-point, and branch. Each operation has a header and a tail bit to define the beginnings and ends of MultiOps. The first operation of a MultiOp has its header set and the last operation its tail bit set. This encoding allows to define only the actual operations in the program code, i.e., the specification of NOPs can be avoided. An *expander* is used to decompress TINKER encoding and route the operations in the MultiOp to correct locations. The expander can be placed either pre- or post-cache.

In [118], Weiss and Fettweis proposed a VLIW encoding denoted as Tagged VLIW (TVLIW). It composes VLIW instruction words from a limited set of TVLIW instructions that each contain two operation slots, defined as functional unit instruction words (FIW). Each FIW is preceded by a tag field that defines the FU the following FIW is to control. A class field is placed at the beginning of the TVLIW instruction word to indicate how many TVLIWs the actual VLIW instruction is to be assembled from. The TVLIW instruction decoder is used to assemble the entire VLIW instruction word. The FIWs are distributed to the assigned FUs and NOPs are supplemented for the remaining units. In case the full VLIW functionality is required, several TVLIWs are required. Assembling such an instruction word takes several clock cycles. As these kind of instructions often reside in loops, Weiss and Fettweis introduced a loop cache to hold the assembled VLIW instructions of the loop. This way the instruction assembly overhead has to be paid only once for the instructions inside the loop.

Richter et. al. have improved the above mentioned TVLIW approach by implementing a VLIW buffer to avoid the penalty cycles in assembling the actual VLIW instruction word [102]. The buffer is software-controlled and allows the coherence characteristics of subsequent instructions to be exploited by reusing the previously fetched instruction.

Modern VLIW architectures have addressed the code size bloat problem with architectural features that allow to omit the NOPs from the instruction words. In the Starcore SC 140 VLIW architecture, a variable-length execution set (VLES) is used to provide high code density [1]. SC 140 can fetch eight 16-bit instructions at a time and is capable of executing up to six instructions concurrently. Instructions belonging to a single execution set are identified using either serial or prefix grouping. Serial grouping reserves two bits in the operation slots to define whether the current operation is the last to belong to an execution set. Prefix grouping adds one- or two-word prefix field to an execution set to define how many instructions belong to it.

Similarly to SC 140, Texas Instruments TMS320C6xx VLIW architecture supports variable-length execution set to preserve code space [3–5]. TMS320C6xx fetch packet contains eight 32-bit operations. The processor can issue up to eight operations in parallel, i.e., the entire fetch packet. Operations issued in parallel are defined similarly to the SC 140 serial grouping. The last bit in the 32-bit operation encoding is reserved to define whether the following operation slot belongs to the same execution set. Instruction decoder checks these control bits and identifies which operations are to be issued in parallel. Operation encoding has to explicitly specify the FU on which the operation is to be executed, as the unit cannot be determined from the location of the operation slot in the execution set.

Suzuki et. al. proposed in [114] a somewhat different methodology to omit all the NOPs from the instructions of a 2-way VLIW processor. Only the actual operations are included in the instructions and they are placed in the operation slots. The instruction decoder can detect whether the operations are in the correct order and if not, it corrects the order and supplements the necessary NOPs. In addition, special NOP instructions are proposed to define instructions that contain only NOPs. The instruction format remains unaffected.

A compression method similar to the one proposed by Colwell *et. al.* was proposed by Schlansker and Rau in [108], and experiments of the proposed method on VLIW architecture were reported by Aditya *et. al.* in [10]. The proposed method is based on multiple instruction formats denoted as instruction templates, which provide operation slots for only a subset of the FUs. The rest of the FUs not having an operation slot in the instruction word obtain NOPs implicitly. The used template is identified using a template selection field at the beginning of the template. This field is inspected during the decompression to determine the fields present in the template, their widths, and their bits positions. With this information the operation slots in the template can be directed to correct FUs and NOPs can be assigned to the other units. A dedicated multi-NOP field is also proposed to specify several NOPs.

In [42], Haga and Barua proposed a new instruction scheduling algorithm for an EPIC architecture to improve the code density. The proposed algorithm tries to find an optimal instruction schedule that reduces the number of NOPs scheduled in the program code. Instruction scheduling on EPIC is somewhat different than on VLIW. The scheduling is based on using templates that identify the operations that have no dependencies and thus can be executed in parallel [111]. The templates specify the allowed operation sequences that can be used while performing the instruction scheduling. Similarly to TMS320C6x and SC 140 architectures, stop bits are used in the templates to express the set of operations that can be executed in parallel.

The NOP removal approach applied on TTA in this Thesis follows the principle of the instruction template-based scheme proposed by Schlansker and Rau [108]. Opposed to VLIW instruction words that specify the operations for the concurrently operating FUs, TTA instruction words specify the data transports that are to be performed on the available transport buses. Operations occur as side-effect of the data transports. The instruction template-based compression method can be adapted to the TTA programming model by considering the fields that specify the data transports, denoted as move slots, as the elements of the templates. As a result, the instruction templates specify data transports for a subset of all the possible transport buses. Null data transports are allocated implicitly on the buses for which a data transport is not present in the instruction template.

In general, the proposed NOP removal compression approaches result in variable-width instructions. Only the approach proposed by Suzuki et. al. result in fixed-width instructions as it is applied on a dual-issue VLIW architecture that allows this. Furthermore, most of the methods operate at the operation slot granularity level. VLIW instructions are partitioned to fields based on the operation slots and only the ones that contain valid operations are included in the compressed instruction words. The Tagged VLIW approaches [102,118] partition instructions to operations slot pairs and compose the compressed instruction words from a variable number of these pairs.

In most of the presented approaches the compressed instructions are translated back to the original format inside the control logic of the processor core. In the commercial

VLIW processors, SC 140 and TMS320cC6x [1,3–5], there is no need for a separate decompression phase. The instruction decoder identifies the operations that are to be executed in parallel. In [10,102,114,118] and in this Thesis, a separate decompression step is required before the decoder to expand the compressed instruction representation back to the original form. In [29,30], a separate decompressor is also required, but it is placed outside the processor core. In [29], reconstruction of the original VLIW instruction word is performed during the cache re-fill, i.e., pre-cache. In [30], a more flexible organization is allowed as the instruction expander can be placed either pre- or post-cache.

Random access support requires that the execution flow can continue from the branch and jump target instructions also after the compression has been made. All of the proposed NOP removal methods require aligning the branch and jump targets to addressable memory locations. Furthermore, branch and jump target addresses need to be patched to correspond to the compressed address space.

The effectiveness of the compression method has been addressed only in [10], [114], and in this Thesis by reporting the compression ratio. None of the given compression ratios include the decompressor overhead, which is fairly difficult to measure, as there is no coding table whose size could be measured. The best compression ratios have been reported for [10], where an average compression ratio of as high as 0.12 is reported. This is obtained for a 12-issue VLIW architecture using as many instruction templates as required by the program code. The complexity of the decompressor is directly related to the number of templates utilized to compress the program code. As the decompressor overhead is not included in the compression ratio, it does not express the absolute truth of the effectiveness of the compression method. The best average compression ratio for the instruction template-based compression approach utilized in this Thesis is 0.47, which is achieved for a 13-issue TTA architecture, i.e., an architecture that can issue up to 13 data transports in parallel. The best compression ratio for [114] is 0.56, which is fairly high for a dual-issue VLIW architecture.

It must be noted that the achievable compression ratios depend heavily on the utilized processor architecture and on the effectiveness of the compiler to schedule the application on to the available hardware resources. Reliable comparison of the methods would require utilizing the methods on a single processor architecture with the same set of benchmark applications.

Dictionary-Based Compression

Dictionary-based program compression methods are based on the traditional dictionary compression method that has been used widely, e.g., for text compression. Dictionary compression finds all the unique substrings, e.g., words in a text, stores them into a dictionary, and replaces them with codewords that identify that substring in the dictionary [121]. The dictionary contains a list of the substrings and their corresponding codewords. During decompression the codeword to be decompressed is searched from the dictionary and the corresponding substring is given as an output.

Dictionary-based compression is applied on program code by interpreting the instructions, or other bit patterns as substrings inside which unique bit patterns are searched for and stored into a dictionary. The original instructions are replaced with indices pointing to the dictionary. The dictionary can be implemented as a look-up table that is addressed using the dictionary indices to obtain the original instructions. More detailed description of dictionary-based compression is presented in Chapter 4.

In [92], Nam et. al. proposed a dictionary-based compression method for VLIW. They separated the opcode and operand patterns from the VLIW instructions and compressed each pattern separately. Their method was based on the fact that VLIW instructions typically perform the same set of operations for a slightly different set of operands, or vice versa. Therefore, by separating the VLIW instructions into opcode and operand streams, more repeated bit patterns could be found. This improves the effectiveness of the compression method. The method results in two dictionaries to be generated, one for the opcode patterns and one for the operand patterns. These two dictionaries are accessed with their own codewords.

Hoogerbrugge et. al. proposed to apply dictionary-based compression at a higher granularity level [48] on TriMedia TM1000 VLIW architecture. They searched for frequently occurring operation slot sequences from the program code and compressed them into superinstructions that were executed on a virtual machine using a software interpreter. The superinstructions were chosen by first forming an expression tree of the program code. All the unique subtrees were then evaluated and their static occurrence probabilities determined. The subtrees were then prioritized based on the probability of occurrence and the width of the subtree. The priority reflects the effect of the subtree on the code size. The subtrees with the highest priority were chosen to be compressed as superinstructions. The decompression was performed using a software interpreter. In addition to superinstructions, Hoogerbrugge et. al. included a dictionary for large constants and utilized relative jumps to improve the effectiveness of their method.

In [104], Ros and Sutton experimented dictionary construction for both single operation slots and operation slot sequences of VLIW instructions. They applied their method on TMS320C6xx architecture, which already has a compact encoding that avoids the explicit specification of NOPs. This affects the effectiveness of the compression method as NOPs that waste the code space have already been removed. The single operation slot compression was applied by storing the most frequent operations into a dictionary and replacing them with dictionary indices. The operation slot sequence compression was applied similarly, considering two to eight operation slots as bit patterns to be compressed. The most frequent sequences were stored to the dictionary and replaced with dictionary indices. As the sequences may be of different sizes, they may overlap. This required to recalculate the usage frequency statistics after each iteration of the selection process.

Ros and Sutton improved their dictionary-based compression method in [105] by optimizing the size of the dictionary. They utilized the property of Hamming distance [43] in the construction of the dictionary. Hamming distance defines among two strings of equal length the number of positions for which the corresponding symbols are different. In this context, the strings correspond to bit sequences and symbols to individual bits. The bit sequences stored into the dictionary were chosen so that all the bit sequences in the program code differ only by a determined Hamming distance from any dictionary entry. The original bit pattern can be restored by toggling the bits that differ. The compressed instruction word defines an index to a dictionary entry, the number of bits that need to be toggled, and the positions of the bits to toggle. The proposed dictionary size optimization method was applied on TMS320C6xx and Intel Itanium [52] architectures by experimenting different Hamming distances and different dictionary selection principles.

In [88], Lin et. al. proposed a Lempel-Ziv-Welch-based (LZW) dictionary encoding method for VLIW instructions. LZW coding, proposed by Welch in [120], is a modification to the original Ziv-Lempel coding [135, 136]. The coding is based on using the previously seen data to encode the incoming one by maintaining a dictionary of the previously seen data. The incoming symbol strings are looked up from the dictionary and encoded with references to the dictionary. Any new input string is stored into the dictionary. The method results in variable-width sequences to be encoded with fixed-width dictionary indices.

Piccinelli et. al. proposed a compression method that can be considered as a hybrid between the dictionary-based and arithmetic encoding methods [97]. The 32-bit operations are divided to 16-bit half-operations, out of which the most frequent ones are stored into a dictionary. The dictionary indices are further grouped into vectors. Each vector is then assigned a codeword based on their Laplacian statistical distribution, following the vector quantization method presented in [38].

Ibrahim et. al. experimented dictionary-based compression to reduce the power consumption on a multi-clustered VLIW in [51]. They utilized a small local instruction memory (LIM) into which they stored the most frequently executed instructions. The instructions in the local instruction memory were then accessed in the program code simply with indices to the LIM. The size of the LIM was limited to 32 instructions. Ibrahim *et. al* proposed an additional power saving method by utilizing the single instruction multiple data (SIMD) presence in the application. An original instruction operating on multiple data could be implemented as a compressed instruction. The number of such instructions was limited to 8 to avoid increasing the cycle time due to more complex decompression.

The dictionary-based compression approach utilized in this Thesis follows the basic principle of dictionary-based compression. The compression is applied on TTA program codes at different granularity levels to investigate how it affects the effectiveness of the compression method. At the highest granularity level, entire instruction words are considered as substrings inside which unique bit patterns are searched for to be stored into the dictionary. As the TTA instruction words are fairly long, the possibility to find repeated bit patterns is fairly small. This results in most of the instruction words being stored into the dictionary, resulting in large size and poor improvement in the code density.

The effectiveness of the dictionary-based compression can be improved by applying the compression to more fine-grained bit patterns. This can be achieved on TTA by dividing the instruction words to smaller fields based on the move slot boundaries. This allows to find more repeated bit patterns and results in smaller dictionary. The drawback of this is the increased size of the compressed code as compressed instruction words are composed of several dictionary indices.

The probability to find repeated bit patterns can be improved even further by dividing the move slots to more fine-grained fields, based on the internal source and destination identifier (ID) field boundaries. This results in even smaller dictionary but, consequently, increases the size of the compressed code even further.

In general, dictionary-based program compression methods encode instructions to fixed-width codewords that correspond to indices to the dictionary that contains the original bit patterns. Except for [105] and [97], all of the presented dictionary-based program compression methods follow this principle. Also in [105] and [97], where the final compressed instructions are variable-width, the dictionary-based compression phase, performed first, results in fixed-width instructions. The phases that follow, i.e., vector quantization in [97] and dictionary size optimization in [105] make the final compressed instructions variable-width.

The dictionary-based compression approaches included in the survey cover several different granularity levels at which compression is applied to, operand slot granularity level being the most often used. Apart from this Thesis, which applied dictionary-based compression at three different granularity levels, only in [104] more than one granularity level is considered.

The decompressor, i.e., the dictionary, is in most of the methods placed outside the processor core, i.e., in between the processor core and the program memory. This means that the decompression overhead is paid always when instructions are fetched from the program memory. Cache is included in the system only in [88]. In that approach, the decompressor is placed post-cache. In [51] and in this Thesis, the decompressor is placed in the control path of the processor core. In [51], the decompressor is implemented in a separate pipeline stage in between the instruction fetch and decode stages. In this Thesis, in addition to implementing the decompressor in a separate pipeline stage, an integrated decompressor alternative is also evaluated where the decompressor is integrated with the decoder into a single pipeline stage.

Ideally, the dictionary-based compression does not break the program representation as the compressed instruction words are typically fixed-width. This allows to maintain the random access support without any modifications, as is the case in [92] and in this Thesis. In these approaches, the program memory is adjusted to be as wide as the compressed instruction word. The program memory is word addressable, which means that the original addresses apply also for the compressed instruction words.

In [104], branch targets need to be aligned to addressable memory locations. The original branch addresses do not apply anymore. Therefore, address patching is required. Branch target alignment is required also in [105] and in [97] as they result in variable-width instructions. Mapping between the compressed and uncompressed address spaces in provided through an LAT.

Except for [51], which targets to reduce the energy consumption of the program memory, all of the presented methods evaluate the effectiveness of the compression method in terms of compression ratio. The best compression ratios, on average 0.20, are reported in [48]. However, the decompression overhead, i.e., the size of the dictionary is not included in the reported compression ratio. Furthermore, the compression ratio is given only for the compressed region, i.e., the code region left uncompressed is not included in the results. Out of the methods that include the decompression overhead in the compression results, best compression ratio is achieved in this Thesis, on average 0.53-0.76, depending on the utilized granularity level. The best compression ratio is obtained at move slot granularity level.

In [51], where the target was to minimize the energy consumption of the program memory, energy savings of 17% to 38% are reported. Out of the compression approaches included in this survey, except for this Thesis, this is the only approach to consider the effects of the program compression on the energy consumption. The area and power consumption results obtained in this Thesis are presented in Chapter 6.

Entropy encoding

Entropy encoding methods utilize the entropy, i.e., the property that some symbols are used more frequently than others. This results in non-uniform probability distribution. Entropy encoding methods utilize this property by assigning the most frequent symbols with shorter codewords. The least frequent symbols need to be assigned with longer codewords to describe all the symbols with unique codewords. Even though the least frequent symbols are assigned with codewords that are longer than the original bit patterns, compression is achieved as these codewords occur only few times compared to the shorter codewords that occur frequently and, therefore, save more code space.

The most commonly used entropy encoding methods are Huffman coding and arithmetic coding. Huffman coding assigns variable-width codewords of integral number of bits to the symbols based on their probability distribution. The codewords are prefix-free, i.e., no codewords is a prefix of another one. Huffman coding is discussed in more detail in Chapter 4 when the utilization of Huffman coding on TTA is discussed.

Larin and Conte applied Huffman coding on TEPIC VLIW in [68]. They applied Huffman coding at three different symbol granularity levels; at byte level, at stream level, and at whole operation level. At byte level, the program code is partitioned into symbols based on byte boundaries. A probability distribution is formed for all the bytes used in the program code. The probability model is then used to encode the bytes with variable-width codewords following the Huffman coding principle. At stream level, operations are partitioned into parallel streams based on the field boundaries that exist in the operations, e.g., the opcode, operand, and immediate fields. Each parallel stream is encoded separately. At whole operation level, entire operations are considered as symbols to be Huffman encoded. Larin and Conte observed that compression at higher granularity levels results in better code size reduction, but also to more complex decompressor that implies a large hardware overhead.

Huffman coding approach utilized in this Thesis follows the basic principles of Huffman coding. The program code is divided into symbols based on which a probability distribution is calculated. The most frequent symbols are assigned to short codewords and vice versa, following the Huffman coding principle. Huffman coding is applied at the same three granularity levels as in the dictionary-based compression, i.e., at full instruction, move slot, and ID field levels.

Arithmetic coding is similar to Huffman coding but performs better as it can assign codewords to a fraction of a bit. Arithmetic coding encodes symbols as real numbers in the interval of $[0,1)$. The interval is divided to sub-intervals based on the probability distribution of the symbols. Arithmetic coding proceeds by finding a sub-interval corresponding to the symbol and then dividing the sub-interval to new sub-intervals based on the updated probability distribution. This process is repeated until the last symbol of the string is reached. A real number inside the final interval is then assigned to the string. The number is expressed in bits using fractional bit representation. This represents the compressed codeword.

In [125], Xie et. al. proposed a compression method based on arithmetic coding for the TMS320C6x VLIW architecture. Even though arithmetic coding methods have been reported to result in high compression ratios, they have not been widely used for program compression due to complex and time consuming decompression procedure that makes it difficult to fit the decompressor into the processor hardware. In their arithmetic compression approach, Xie et. al. utilized reduced-precision arithmetic coding proposed by Howard and Vitter in [49] to approximate the complex floating-point calculations with lookup-table accesses to reduce the complexity of the decompression and hence make the method more practical for program compression.

Xie et. al. utilized dynamic Markov modeling [33] in their arithmetic compression approach to provide a statistical probability distribution that is utilized in the arithmetic coding process to encode symbols into variable-width codewords. Coding is performed separately for each VLIW fetch packet. Each packet is further divided into smaller sub-blocks at the operation slot boundaries to allow parallel decompression. Markov model is built either for all the sub-blocks in the program code, or the fetch packets are divided into parallel streams based on sub-block boundaries and a Markov model is built separately for each stream.

Markov modeling results in fairly large decompressor that introduces an overhead in the hardware of the processor and hence affects the effectiveness of the compression method. In [126], Xie et. al. studied different modeling approaches for arithmetic coding to reduce the decompression hardware overhead. They noted that the overhead could be reduced when simpler models were used for compression.

The simplest model calculates the probabilities of ones and zeroes across the entire program code and assigns fixed probabilities for them regardless of their positions in the instruction words. Positional information may improve the effectiveness of the compression. For example, opcode and operand fields are typically in the same locations within the instruction words. This information can be utilized by calculating the bit probabilities for the model within certain boundaries, e.g. inside operation slots, and forming a model for each bit position. This improves the compression ratio, but results in more complex hardware decompressor.

In [127], Xie et. al. studied a somewhat different entropy encoding scheme on VLIW architecture. This scheme compresses variable-width bit sequences to fixed-width codewords. The method is based on Tunstall coding, which was proposed by Tunstall in [116]. Tunstall coding utilizes the probabilities of ones and zeroes to find variable-width bit sequences that are to be assigned with fixed-width codewords.

In general, entropy encoding methods result in variable-width codewords. This is the case for most of the presented entropy encoding methods. Only in [127] fixed-width codewords are assigned to variable-width bit sequences. In terms of compression granularity, all the different granularity alternatives are covered in the methods. Most of the methods cover more than one granularity level. Huffman coding approach utilized in this Thesis covers three different granularity levels.

Decompressor is placed outside the processor core in all of the approaches except for the Huffman coding approach presented in this Thesis. In [68] the decompressor is placed pre-cache, and in [125–127] post-cache. In this Thesis, the decompressor is considered to be implemented inside the control path of the processor core even though the actual hardware implementation of the decompressor is not made.

As the entropy encoding methods in most cases result in variable-width instructions, the branch targets need to be aligned in all of the presented methods to addressable boundaries so that the branch target instructions can be accessed directly in the program memory. Due to variable-width instructions and the branch target alignment, address mapping between the compressed and uncompressed address spaces is also required. This is accomplished either with an LAT as in [68, 125, 126], or by patching the branch target addresses to correspond to the compressed address space, as in [127] and in this Thesis.

Compression ratios are reported for all of the presented entropy encoding methods. Only in this Thesis the decompression overhead, i.e., the decoding table, is taken into account in the compression ratio. The Huffman coding approach presented in [68] results in best compression ratio, at best 0.30 when the compression is performed at operation slot level. However, this would lead to a very large decoder and would make the approach less effective. Therefore, the compression ratios obtained in this Thesis, which are at best 0.41, are very competitive.

Instruction Re-Encoding

Instruction re-encoding methods provide means for code size reduction by modifying the instruction encoding so that operations are encoded with fewer bits. Instruction re-encoding is motivated by the fact that program codes typically use only a subset of the entire instruction set and some operations are used more frequently than others. Furthermore, there are also operations in the instruction set that do not utilize all the bits that are reserved for an operation, e.g., one operand operations. This suggests that the code size can be reduced by encoding these instructions with fewer bits. As the compact encoding needs to support only a subset of the entire operation set, the width of the opcode field, identifying the operation to be performed, can be reduced. The widths of the operand fields, specifying the operands for the operation, can also be reduced as all of the possible registers in the register file (RF) are not used as operands. Also the width of the immediate field can typically be shortened to make the instructions fit to the re-encoded size.

As the opcode and operand fields are shortened, the number of possible operations and the number of registers that can be used as operands for an operation become limited. Furthermore, due to the limited width of the immediate field, the range of possible immediate values becomes smaller and complicates, e.g., branching. Therefore, processor architectures with re-encoded instruction set typically also support the original instruction set. The re-encoded instruction set is utilized to save code space where applicable, and the original instruction set is used elsewhere in the program code. A distinction has to be made between these two instruction encoding alternatives. A dedicated bit can be used to define the used encoding, or alternatively, a special instruction can be introduced to toggle between the two encodings.

The execution of the re-encoded instructions is typically supported by implementing a dedicated hardware in the processor core that expands the re-encoded instructions to the original format before decoding. This way the original instruction decoder can be used for both the original and re-encoded instructions. This avoids the need to implement a separate parallel decoder for the re-encoded instructions.

In [68], Larin and Conte proposed an instruction set re-encoding method for the TEPIC VLIW architecture where the re-encoded instruction set was formed based on the profile of the instructions used in the given application. The opcode and operand fields were encoded with only as many bits as were required. The number of different operations used in the program was profiled and the width of the opcode field was adjusted based on that number. Similarly, the widths of the operand fields were adjusted based on the number of registers alive simultaneously. A dedicated instruction decoder had to be implemented in the processor control path to decode the re-encoded instructions back to the original format so that they could then be decoded using the original decoder.

In [21], Biswas and Dutt proposed a method that achieves code size reduction on TMS320C6x VLIW architecture by supplementing the instruction set with new complex operations that combine two base operations that share and operand, i.e., have a read-after-write dependency. A heuristic-based algorithm is used to generate these operations in the compilation step. The proposed algorithm converts two three-operand operations into one four-operand operations, i.e., operations *x = a op1 b* and *y = x op2 c* are combined into operation *y = (a op1 b) op2 c*. Execution of the combined operations is simple. The four-operand operation is split back to two consecutive three-operand operations in the decode phase of the dispatch stage of the instruction pipeline, after which the operations can be executed using the original datapath of the processor.

Liu et. al. proposed a mix of NOP removal and instruction re-encoding approach for a quad-issue VLIW architecture in [90]. The encoding is performed hierarchically in three steps. At first, single operations are re-encoded as variable-width operations based on the operation types and the required operands. The re-encoded operation consists of a fixed-width “head” and a variable-width “tail”. The head contains fields for the opcode, control signals, and the possible operands. The proposed encoding follows the HAT (heads-and-tails) principle, proposed by Pan and Asanović for RISC architecture in [94]. To support the parallel issue of several operations, the variable-width operations belonging to an execution set are packed into an instruction packet. An identifier preceding the packet is used to identify for which functional units the packet contains operations. It also provides information on the instruction execution type. The variable-width instruction packets are further packed into fixed-width bundles to simplify memory accesses.

Out of the presented instruction set re-encoding methods, [68] and [90] result in variable-width and [21] in fixed-width instructions. In [68], the compression is performed at opcode and operand field level. In [90], the instruction set re-encoding is performed at operation slot level. In [21], bundles of two operations are re-encoded into single operations.

The decompressor, i.e., the logic that translates the re-encoded operation representations back to their original form is performed in the control logic of the processor core in all of the evaluated re-encoding approaches. In [21], there are no requirements to modify the addressing or instruction alignment to support random accesses to the compressed program code. In [68], the re-encoded instructions need to be aligned to addressable locations and the branch addresses have to be patched. In [90], random access support methods have not been discussed.

Out of the presented approaches, best compression ratios, at best 0.33, are presented in [90]. Information whether the decompressor overhead is included is not available. All the other instruction re-encoding methods, which have significantly worse compression ratios, include the decompression overhead in the compression ratio.

2.2.3 Comparison Summary

The compared program compression approaches were classified into four categories. NOP removal methods [1,3–5,10,29,30,42,102,114,118] concentrate on avoiding the explicit specification of NOPs. Dictionary-based compression methods [48, 51, 88, 92, 97, 104, 105] search the program code for unique bit patterns that are stored into a dictionary and replaced with indices pointing to the entries in the dictionary. Entropy encoding methods [68, 124, 126, 127] utilize the entropy, i.e., the fact that some bit patterns occur more frequently than others, and encode the most frequent bit patterns with shorter codewords and vice versa. Instruction set re-encoding methods [21, 68, 90] modify the instruction encoding to encode the operations with less bits. The program compression methods developed for and evaluated on TTA belong to the first three categories.

Generally, dictionary-based compression methods encode instructions with fixed-width codewords, which are easier to decompress than variable-width codewords. Methods in the other categories usually result in variable-width codewords that make the instruction fetch and decompress logic more complex. Variable-width codewords typically require that branch targets are aligned to addressable memory locations. Address mapping or address patching is also required to correct the branch target addresses point to the correct compressed instructions. Fixed-width codewords usually do not need branch target alignment nor address patching as the width of the program memory can be adjusted according to the width of the compressed instruction.

Most of the methods perform compression at operation slot or operation slot sequence level. This is a fairly natural choice as operation slots are the basic elements of VLIW instruction words. Several approaches consider also smaller granularity levels, i.e., opcode and operands fields, or arbitrary bit sequences, e.g., bytes. Smaller granularity level increases the probability to find redundancy from the program code.

However, it increases the size of the compressed program code due to compressed instruction words being composed of several codewords. The trade-offs of the compression granularity on the effectiveness of the compression methods are evaluated only in the entropy encoding approach in [68] and in the dictionary-based compression and entropy encoding approaches in this Thesis. However, only in this Thesis the sizes of the decoding tables are taken into account, which makes the evaluation more accurate.

Most of the compression methods included in the comparison present the effectiveness of the compression method in terms of compression ratio. However, as some methods include the decompression overhead in the reported compression ratio while others do not, comparison of the effectiveness of the methods is difficult. In addition, compression ratio does not entirely represent the effectiveness of the compression method. Aside from reducing the size of the program code, which allows to use smaller memories, also the power consumption is affected. Fetching fewer bits from the program memory consumes less power. Out of the compression approaches included in this survey, only [51] in addition to this Thesis considers this issue and presents estimates on the energy saving achievable in the program memory.

In order to fully characterize all the aspects of a compression method, evaluation of the effectiveness of a compression method should be performed by implementing in hardware both the original system and the system that executes compressed program code. This would allow the decompression overhead to be taken into account and the effectiveness of the compression method could be measured in terms of silicon area rather than compression ratio. Hardware implementations would also allow to estimate the effects of the compression on the power consumption. In this Thesis, the instruction template-based and dictionary-based compression approaches on TTA have been implemented in hardware, allowing closer analysis of all the aspects of the compression methods. Such measures are rarely reported for the program compression methods.

3. PROCESSOR HARDWARE CUSTOMIZATION

Modern day embedded systems require more and more performance from the underlying processing hardware due to the increased complexity and the tight real-time requirements, e.g., in video processing. At the same time, there are requirements for fast time to market, lower cost and, especially, lower power consumption. General-purpose processors (GPP), such as RISC and CISC, developed generality and simplicity in mind, often cannot meet these requirements. The processor may not provide sufficient performance as the hardware has not been optimized for any particular application or application domain. Furthermore, as the processors need to be capable of executing all kinds of applications, the chip area is usually large and consumes significant amount of power.

The above mentioned requirements can be met by customizing the processor hardware for the given application. Such systems are often denoted as application-specific systems. Customization allows to tailor the hardware resources of the system to provide enough performance. Moreover, as the system is tailored for a particular application or application domain, the system does not have to contain hardware resources to support execution of all kinds of applications. This may allow to reduce the chip area and, therefore, also the power consumption.

The highest level of application-specific customization is offered by chips that contain only hard-wired logic, i.e., they do not execute any software. Within the context of this Thesis, such systems are referred to as application-specific integrated circuits (ASIC). Such systems usually provide high performance and are also area and power efficient. However, the time to market is usually long as the system has to be designed manually. The development costs are also high due to high non-recurring engineering (NRE) and mask set-up costs for chip fabrication. Mask set-up costs are currently measured in millions of dollars. As ASICs are implemented purely in hardware, they lack in flexibility. Hence, if the application changes, a new version of the ASIC has to be developed. This increases the costs even further, especially due to the need to develop a new photomask for fabrication. Therefore, ASICs are practical only for high-volume products.

Field-programmable gate arrays (FPGA) can be used to avoid the high mask set-up costs. FPGAs contain reconfigurable hardware that can implement the user-defined functionality. FPGAs do not achieve the performance or area and power consumption of ASICs, but they provide flexibility as the hardware can be reconfigured to correspond to any changes in the functionality. Hence, FPGAs have been widely used for prototyping and also for low-volume products. Recently, FPGAs with embedded processor cores have emerged. This allows to shorten the design time and provide faster time to market as some parts of the application can be implemented in software. Embedded processor cores may be implemented as hard macros aside the configurable logic, such as the PowerPC 405 processor core [129] that can be included in the Xilinx Virtex-II Pro FPGA board [9]. Alternatively, the processor may be described in structural hardware description language (HDL), e.g., very high-speed integrated circuit hardware description language (VHDL) or Verilog, and implemented using the reconfigurable resources of the FPGA. Nios II processor core [13] from Altera, or the MicroBlaze processor core [128] from Xilinx exemplify such systems. Despite their flexibility, FPGAs are usually impractical in high-volume embedded systems mostly due to their large area and power consumption and high cost.

Application-specific instruction set processors (ASIP) are considered as processors that contain customized hardware and instructions for a specific application or set of applications to improve the performance. Generally, there are two alternatives for an ASIP. It can either be a GPP that is extended with additional hardware resources, e.g., a DSP that has additional resources for digital signal processing applications, or the entire processor can be designed for a particular application. The latter allows to optimize also the area and power consumption as the ASIP in that case can be optimized to contain only the hardware resources that are needed. Compared to ASICs, ASIPs are also more flexible as they are programmable. When the application changes, the software can be modified without the need to modify the hardware resources and fabricate a new chip, as in ASICs. ASIPs offer also a faster time to market as the application is developed in software. Compared to GPPs, the time to market is still longer as the customized hardware resources need to be designed and implemented in hardware.

As the characteristics of GPPs, ASICs, and ASIPs discussed above indicate, there is a tradeoff between cost, performance, time to market, and flexibility between different architecture alternatives. GPPs provide fast time to market with fairly low cost. However, they may not provide enough performance. ASICs in their turn usually provide high performance with small area and low power consumption, but the design times are long and the development costs are high. ASIPs can be considered as a compromise between GPPs and ASICs in terms of their characteristics. Figure 4 classifies different processing architectures based on their characteristics.

ASIP architectures have raised a lot of interested in the recent years. ASIP design is often difficult as the design space of possible architecture configurations is large. Finding an optimal processor configuration requires several different alternatives to be designed and implemented to find the most suitable ones. Efficient evaluation requires a set of software tools, such as HLL compiler, assembler, linker, and instruction set simulator to be developed for each different configuration. If these tools are developed manually, NRE costs and time to market are increasing.

Several different ASIP design methodologies and architectures have been proposed to ease the ASIP design. This chapter begins with on overview of the design methodologies and architectures presented in the literature for designing ASIPs. The overview is followed by a detailed description of TTA, which is a customizable architecture template that is used in this Thesis for designing ASIPs on which the proposed program compression methods are utilized. The ASIP development methodologies utilizing the TTA paradigm, the Move framework and its successor, TTA Codesign Environment, are also introduced.

3.1 Customizable Processor Architectures

Processor hardware customization can be divided into two subclasses: instruction-set extensions and fully customizable processor architectures. Instruction-set extension methodologies extend the instruction-set with special operations and add corresponding hardware to the processor core to speed up execution. Customizable processor architectures allow to design the processor hardware resources according to the requirements of the application. Architectures and methodologies belonging to these two classes are introduced in the following Subsections.

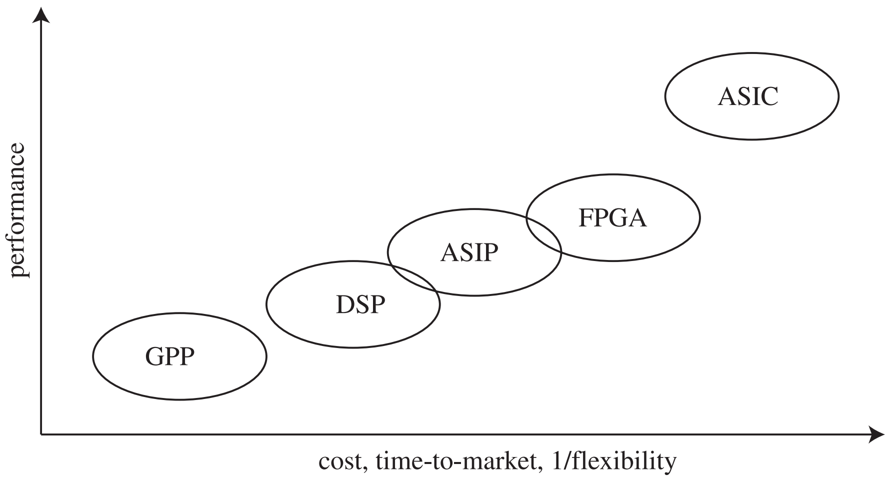


Fig. 4. *Classification of processing architectures based on their characteristics.*

3.1.1 Instruction-Set Extensions

Application-specific instruction set processors have traditionally been created by extending a pre-designed processor core with features that are customized for the target application. This involves extending the instruction set with new used-defined instructions and adding hardware to the processor core to support these instructions. This approach provides improved performance for the particular application for which the customization is made. The drawback of this approach is that the area and power consumption are high as the processor architecture still contains all the original hardware resources that might not be needed in the application for which the customization is performed.

In [64], Kucukcakar et. al. proposed a method for designing ASIPs by adding application-specific instructions to the instruction set of the Motorola MC68HC11 processor. The method uses a profiler to identify the performance bottlenecks from a software implementation of the target application. Application-specific instructions are then developed based on this information. New instructions are introduced either for the frequently used subroutines or for the commonly used instruction sequences. Support for the new instructions is implemented to the control and datapath of the processor using reconfigurable logic. This allows to utilize the new instructions directly in the assembly representation of the program code. The compiler of the tool framework can also be modified to support HLL code to utilize the new instructions.

In [130], Yang et. al. proposed a MetaCore design environment to design ASIPs with DSP specific functionality. Customization in the environment is based on a predefined microarchitecture and the basic operation set that provides the basic operations and the functionality required in DSP applications. The instruction set can be customized by selecting instructions from the predefined instruction set and by adding new application-specific instructions. For the new application-specific instructions, the corresponding FU has to be added to the processor design. The processor hardware resources, such as buses, latches, multiplexers, FUs and interconnections, are declared using *MetaCore Structural Language* (MSL) description. A *MetaCore Behavioral Language* (MBL) description is used to describe the hardware parameters of the target architecture and the low-level bit operations and timing information.

The first phase of the MetaCore design environment is the design space exploration. The target design is evaluated with the help of the provided software tools, such as compiler, assembler, instruction set simulator, and performance analyzer. The designer can modify the target design based on speed, area, and power estimations provided by the performance analyzer. After the final target design is found, SMART, the HDL generator of the MetaCore environment, can be used to translate the target design into HDL description.

A customizable RISC processor core for DSP was proposed by Kang et. al. in [59]. The architecture can be tailored for a specific DSP application by adding several DSP specific features to the architecture. These features include single-cycle multiply-accumulate (MAC) operation, direct memory addressing, hardware looping, and address generation units. The compiler of the RISC processor can be modified to support the DSP-specific features. The tool flow contains a code-converter that analyzes the data flow graphs of the application codes programmed in SPARC assembly and transforms the control flow graphs to exploit the DSP-specific architectural features that were added to the RISC core.

Lx architecture [37], developed jointly by Hewlett-Packard and ST Microelectronics, provides a scalable and customizable VLIW architecture platform to be used in embedded processing systems. A processor configuration is constructed from a set of clusters. Each cluster is a 4-issue VLIW core that contains four arithmetic-logic units (ALU), two 16×32 multipliers, one load/store unit, one branch unit, 64 general-purpose registers, and eight 1-bit branch registers. The number of clusters can be varied to scale the architecture according to the performance requirements.

The Lx architecture is customized by adding application-specific instructions. The Lx architecture is said to be more favorable to be customized for an application domain rather than for a specific application. The architecture is accompanied with a commercial software tool chain where the changes due to scaling or customization are not exposed to the programmer. The heart of the tool flow is an instruction-level parallel compiler, based on the Multiflow compiler [91].

The Xtensa processor core [41] from Tensilica provides another customizable architecture template. Similarly to MetaCore, Xtensa provides a basic instruction set architecture with a set of base operations and a basic architecture to execute these operations. The base ISA contains approximately 80 instructions, a superset of the traditional RISC operation sets. The architecture is configurable and the designer has several options to modify the architecture, e.g., determine the number of registers, the size of the instruction and data caches, and the set of FUs included in the implementation. The architecture supports adding user-defined custom instructions and corresponding FUs to implement the used-defined operation functionality.

The main tool of the Xtensa design flow is the EXPRES processor extension compiler that can create a tailored configuration based on C/C++ language code of the algorithm. The EXPRES compiler can explore the design space to find a suitable configuration for the application. The processor generator of the design flow can then be used to create the HDL description of the target processor.

The ARCtangent microprocessor architecture from ARC provides a configurable, extendable, and synthesizable 32-bit RISC core that can be customized to meet the performance, area, and cost requirements [6]. The architecture provides a mixed 16/32-bit instruction set to optimize the code size. The instruction set, register file configuration, caches, buses, and other architectural features can be customized. A library with some of the common DSP filters and algorithms is also provided. Processors are customized using ARChitect Processor-Configuration Tool that can automatically generate the register transfer-level (RTL) description of the processor. The tool set also includes compiler, assembler, linker, profiler, and instruction set simulator to compile and evaluate the application on the designed processor configuration.

Adelante Technologies, acquired by ARM Ltd. in 2003, provides a Saturn 16-bit VLIW DSP core that can be configured. The core uses a mixed 16/32-bit instruction set like ARCtangent to minimize the code size. The base architecture consists of two multipliers, four ALUs, and of several other parallel resources. The architecture is customized by adding used-defined operations to the instruction set and providing execution units that support the user-defined operations. Customization is also supported at a higher level with a capability to include application-specific co-processors. The tool set contains the basic software development and simulation tools.

The Jazz DSP processor core of Improv Systems is another customizable VLIW processor architecture [8, 87]. Jazz allows the user to add custom instructions and/or custom execution units using the Jazz PSA Composer Tool Suite. The tool suite is used to build the processor configuration and to compile and simulate the target application on the customized processor configuration. Processors designed with the Jazz tool suite can range from a single configured Jazz processor to systems with many Jazz processor implementations. The compiler of the tool suite can optimize the application for parallel execution and supports also task-level parallelism using multiple-processor cores. The Jazz PSA tool suite contains a processor generator that can create the HDL description of the target processor.

3.1.2 Fully Customizable Processor Architectures

Apart from customizing an existing processor core according to the requirements of a particular application, the processor architecture can also be designed from scratch. This allows more effective hardware customization as the entire processor will be customized particularly for the given application or a set of applications. This allows the architecture to contain only the necessary hardware resources to execute the target application, therefore saving both area and power consumption.

APE2 [23] from Cambridge Consultants is a customizable DSP architecture that is aimed to be used as a co-processor for a microcontroller core. The architecture is based on VLIW architecture. It allows to configure the hardware according to the requirements of the application. The processor consists of parallel modules that can be selected from a library of predefined processing blocks. In addition, user-defined modules can be included. The combination of the used modules is entirely free. Data is transferred in between the modules through a dedicated routing bus. A complete software development and HDL generation toolkit is provided for fast processor design. The hardware modules and connections in between them are defined in software. The application is described in assembly language. The assembler provides statistics on the hardware performance, which aids the designer to modify the configuration to meet the requirements of the application. The HDL description of the processor is created automatically in Verilog. The tool set contains a code compression tool that compresses the program code and creates a HDL description of the decompression logic that will be included in the control path of the processor.

CHESS/CHECKERS from Target Compiler Technologies is a retargetable tool set for designing embedded processors that are customized for a particular application [2]. Customization is done by modifying the programmer's view of the processor, i.e., the instruction set, FUs, registers, and buses. The processor architecture is modeled using a proprietary modeling language, nML, which serves as an input to all the tools of the design environment. The instruction set and the structural information about the data path are described in the nML specification. The nML specification of the processor configuration is given to the retargetable compiler, CHESS, which translates the source program, written in C, to optimized machine code. CHECKERS is an instruction set simulator that can simulate the execution of the compiled code on the target processor. The synthesizable hardware description of the target processor is generated using a HDL generator, GO. The drawback of the tool set is the lack of the tool-assisted design space exploration. This makes the design process tedious as the designer has to manually modify the architecture description and evaluate each configuration to find the most suitable one.

The CoWare Processor Designer [7] provides an automated ASIP design environment for embedded processor design. The design flow [46] is based on a Language for Instruction Set Architectures (LISA) [96,137], that was created at Aachen University of Technology. A LISA processor description describes the instruction-set, pipelines, register files, pins, memories, and caches of the target processor. The description is used to generate a complete set of software tools for compiling and simulating applications on the target processor. The instruction set simulator provides profiling data that can be used to tailor the processor architecture and instruction set to meet the performance and cost requirements. Once the requirements are met, RTL level HDL description of the processor can be created with an automated HDL generator.

The architectural synthesis subsystem of the PICO (Program In, Chip Out) design tool from Hewlett-Packard [11] provides a fully automated design process for designing ASIPs. The architectural synthesis subsystem can be used to design VLIW and EPIC processors. The main tool of the system is a Spacewalker tool that searches the design space of architectural choices for an optimal configuration for a given target application, which is described in C-language. The Spacewalker evaluates several architecture configurations by varying the architectural resources of the VLIW or EPIC processor, such as FUs, RFs, read/write ports on each RF, and by evaluating various interconnection topologies, cache and memory hierarchies, instruction formats and instruction fetch and decode hardware alternatives. The Spacewalker provides statistics of the processor configuration in terms of area and performance.

The hardware synthesis subsystem creates the architecture of the processor and a machine-description database that is used to retarget the compiler for the designed processor configuration. The architecture subsystem outputs RTL HDL description of the processor and provides statistics of the chip area for the Spacewalker. The retargetable compiler, Elcor, compiles the application on the designed processor configuration. Elcor estimates also the performance and hardware resources usage and provides this statistics to the Spacewalker to guide its search of the design space. The design framework also considers the code bloat problem of parallel processor architectures and provides means to improve the code density.

MOVE framework [35], developed at Delft University of Technology, The Netherlands, provides a semi-automatic design process for designing ASIPs that utilize the transport triggered architecture paradigm [34]. The modularity, flexibility, and scalability of the TTA are utilized in the MOVE framework to provide a semi-automatic design process for designing processors that are tailored for a particular application or a set of applications. The toolset provides a design space explorer to search for optimal processor configurations for the target application. A retargetable compiler is provided to generate the ILP code for the target processor, whose synthesizable HDL description can be generated automatically with a hardware generator. A successor of the MOVE framework, TTA Codesign Environment (TCE) [55], following the design principles of the MOVE framework, has been developed at Tampere University of Technology, Finland. It contains several new features and tools, such as processor designer and instruction set simulator and debugger.

As the TTA architecture and the TTA-based ASIP design methodologies are tightly involved in the work presented in this Thesis, they are discussed in more detail in the following two Sections. Section 3.2 introduces the details of TTA. Section 3.3 presents in more details the TTA-based ASIP design methodologies, the MOVE framework and the TTA Codesign Environment.

3.2 Transport Triggered Architecture

Transport triggered architecture is a class of statically programmed ILP architectures that reminds VLIW architectures. TTA forms a superclass of VLIW architectures by exploiting in addition to operation level parallelism also the parallelism available at the data transport level. A VLIW instruction defines the operations that are executed in parallel whereas TTA instruction defines the data transports that are to be executed concurrently between the hardware resources of the architecture.

3.2.1 Principles

Transport triggered architecture was proposed to overcome the limitations of VLIW architectures that have been widely used in modern day embedded systems due to their modularity and scalability [100]. As VLIW architecture is modular, performance can be scaled up by adding more FUs. Some architectures even support the inclusion of user-defined FUs that are designed specifically for a particular task. However, though modular and scalable, VLIW architectures have been criticized for the increased complexity of the RF and the bypass network, especially when the number of FUs becomes large [29]. This is due to VLIW architectures being designed for the worst case, which requires that each FU needs to have three ports to the RF and each FU output needs to be connected to all the inputs of all the other FUs.

Few approaches have been proposed to avoid the increased complexity of the register file. Capitanio et. al. proposed a clustered architecture to ease the complexity of the register file [25]. In this approach, the VLIW processor is partitioned into smaller clusters inside which the worst case requirements for the register file ports are met. However, special move instructions need to be used to transfer data in between the clusters, implying a penalty in the performance. Zalamea et. al. proposed another alternative to ease the register file pressure by using a two-level hierarchical register file [134]. This approach provides a large number of register ports with low access time. The first level has a small capacity but several ports. The second level has higher capacity but smaller number of ports. The second level interacts with the first level and the data memory. The drawback of this approach is the increased latency in between the memory and the functional units as there is one more register level in between them.

Corporaal proposed in [34] an alternative approach to overcome the register file and bypass network complexity. He stated that there are situations when not all of the register file ports and the bypass network connections are required. This was based on the following findings:

- All operations do not require two operands, e.g., register-to-register copies, immediate operations, jumps and calls.
- All operations do not produce a result, e.g., jumps and calls.
- Values can be bypassed between FUs, so they do not need to be stored into, or read from the RF. If all the usages of a value can be bypassed, there is not need to store the value into the RF.
- An operand may be used multiple times by consecutive operations. This means that the value needs to be read from the RF only once.
- RF ports may be shared by multiple reads from the RF. This happens when a register is used in multiple operations that are executed concurrently.

The above cases indicate that the register file can be implement in the architecture as a functional unit that has significantly less read and write ports than a normal register file in VLIW architectures. The utilization of the register file is determined at compile-time by the compiler. This allows to ease the register file pressure [34].

In addition to reducing the complexity of the register file, Corporaal noted that the complexity of the bypass network could also be reduced. Bypasses can be made visible at the architectural level by assigning them to the inputs and outputs of the FUs. This way spilling of the bypass values to RF is made under the program control. The bypass complexity can also be reduced by reducing the number of read and write connections, and therefore, the number of bypass buses. This implies that, besides the operations, also the operand transfers (transports) need to be assigned at compile-time. This way the bypass transports become visible at the architectures level, allowing to hide the operands. This mirrored programming model defines the concept of transport triggering. [34]

3.2.2 Software Characteristics

The programming model is the main difference between TTA and the traditional operation triggered architectures (OTA), such as VLIW. In OTA, the program specifies the operations the processor is to perform, e.g., addition, subtraction, multiplication, and shifting, and identifies the associated operands, i.e., registers and immediate values. In TTA, the program specifies explicitly the data transports, also denoted as moves, to be performed by the interconnection network. Operations occur as side effects. Therefore, TTA has also a flavor of dataflow architectures.

The execution of an operation on TTA involves transporting the operands of the operation to the inputs ports of an FU that is able to perform the desired operation, and transporting the results from the output port of the FU to the desired location when the result is available. The above mentioned data transports are classified into three classes: operand, trigger, and result moves.

Operand move is responsible for transporting an operand of an operation to the operand port of an FU. Trigger move transports another operand to the trigger port of the FU. Data transport to the trigger port of the FU fires the execution of the operation. If the FU can perform more than one operation, an opcode is transported in the trigger move to specify the operation to be performed. Once the operation is performed, the result of the operation is transported from the result port of the FU to some other hardware resource in the architecture with a result move. TTA architecture does not limit the number of operand ports nor result ports. This allows to design special functional units (SFU) that can perform operations with multiple operands and produce multiple results.

There are some limitations in the order in which the data moves are to be executed. The operand moves need to precede or be executed in the same clock cycle as the trigger move in order to include all the operands in the operation. The trigger move must precede the result moves to transport the correct output data from the FU to another resource.

Data transports with regards to RFs are somewhat different. Data transports in and out of the RFs are denoted as input and output moves. They both include an opcode to specify the register that is to be written in case of an input move, or to be read in case of an output move.

A typical RISC-type operation corresponds to three data transports. For example, a RISC-type add operation that adds together the values of two registers,

```
add R1, R2, R3
```

corresponds to three move operations:

```
R1 -> FU_o.add;
R2 -> FU_t.add;
FU_r.add -> R3;
```

The first move (operand), transports the data from register R1 to the operand port of the FU. The second move (trigger) transports the data from the register R2 to the trigger port of the FU and fires the operation execution. The last move (result) transports the result data from the output of the FU to register R1.

TTA concept utilizes ILP by executing data transports in parallel. The number of the data transports executed during a single clock cycle is upper bounded by the number of data transport buses. Each data transport bus can execute a single data transport during one clock cycle. Limitations in the order of the data transports and the availability of the move buses and FUs with correct functionality need to be taken into account when the data transports are scheduled on the available buses. Programming data transports manually, i.e., assigning the operations to the FUs and allocating the data transports on the various buses would become too complex. Therefore, a retargetable compiler that can compile HLL code on a TTA processor architecture has been developed. The retargetable compiler is described in more detail in Subsection 3.3.1.

3.2.3 Hardware Characteristics

TTA processors are constructed out of a set of basic building blocks. These include:

- *Functional units* that execute operations
- *Register files* that contain registers to store data locally
- *Buses* that transfer data between functional hardware resources
- *Sockets* that establish connections between the functional hardware resources and the buses

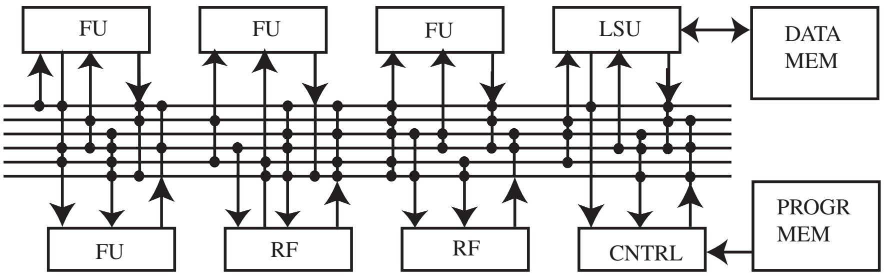


Fig. 5. *TTA processor organization FU: functional unit. RF: register file. LSU: load-store unit. CNTRL: control unit. Dots represent connections between buses and sockets.*

Figure 5 depicts an example of a TTA processor configuration. The architecture consists of a set of functional resources, i.e., FUs and RFs, a control unit, and program and data memories. An interconnection network consisting of buses and input and output sockets performs the data transports in between the architectural resources. Buses are used to transport data in between the functional resources. Input and output sockets are used to connect the functional resources to the buses. An input socket contains a multiplexer that chooses one of the buses from which the data is written to the input port of the attached functional resource. An output socket contains a de-multiplexer that chooses the bus to which the output data of the attached functional resource through the output port is to be written to. Sockets do not need to be connected to all the available buses.

The design of FUs and RFs is separated from the design of the interconnection network; both can be designed independently. FUs can implement any functionality and they can be added to a processor configuration without a need to modify the interconnection network. In addition to the FUs and RFs, TTA processor contains a control unit that is responsible for controlling the execution of the processor. The control unit is responsible for fetching instructions from the program memory and decoding them into control signals that control the sockets in the interconnection network to transfer data on the buses. One or more FUs are configured as load-store units (LSU) that provide an interface to the data memory. As the FUs can implement any functionality, in addition to FUs executing operations from the basic operation set of the architecture, SFUs can be designed and included in the processor architecture to implement user-defined functionality. The modularity of the architecture allows the SFUs to have as many input and output ports as seen necessary.

3.2.4 Pipelining

TTA supports two levels of pipelining. In addition to transport pipelining, which means that the execution of the data transports is pipelined, also the FUs can be pipelined. Decisions for these two pipelining schemes can be made independently.

TTA supports two different transport pipelining schemes, three- and two-stage, out of which the three-stage pipeline is typically used. The three-stage pipeline consists of three stages: instruction fetch (IF), decode (DC), and move (MV). The three-stage transport pipeline is depicted in Fig. 6. [34]

In the IF stage, the instruction fetch logic fetches the next instruction from the program memory or the cache based on the value of the program counter (PC) and stores it to the instruction register. In the DC stage, the instruction is decoded and the control signals controlling the sockets of the interconnection network and possible opcodes are generated and registered. In the MV stage, the actual data transports take place. FUs and RFs place data on the buses through output sockets and receive data from the buses through input sockets. Figure 7 illustrates the organization of the transport pipeline logic in the control unit of the processor. The figure illustrates the logic related to the transport pipeline stages and the corresponding pipeline registers.

Pipelining in FUs is made to split the execution of complex operations into smaller parts so that the cycle time can be reduced. FU pipeline stages contain combinatorial logic and a pipeline register. Each pipeline stage increases the latency of the FU, i.e., the number of clock cycles it takes for the result value to be available on the output of the unit after the operation has been triggered.

3.2.5 Instruction Format

TTA instruction words are in principle similar to VLIW instruction words. However, instead of specifying operations in operation slots, TTA instructions words contain dedicated fields to define data transports to be performed on the buses. These fields are denoted as move slots [34]. A TTA instruction word contains as many move slots as there are move buses in the architecture. Each of the move slots is further divided into three fields; guard, source ID, and destinations ID field, as depicted in Fig. 8.

The guard field is used to specify a guard value that is used in the decode stage to control whether the data transport on that specific bus is to be executed or squashed.

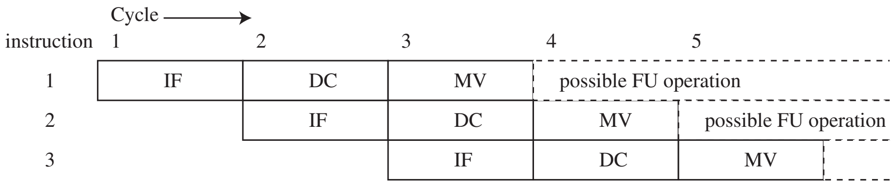


Fig. 6. *Organization of the three-stage TTA transport pipeline.*

The guard value usually refers to a Boolean register or its inverse. If the guard value is true, a squash signal is asserted, resulting in squashing of the data transport. This provides means for conditional execution. The source ID field is used to define the address of the socket that is to write data on the bus. The destination ID field is used to define the address of the socket that reads data from the bus. During the decode stage, the values in the source and destination ID fields are compared against the hardwired socket IDs. When the IDs match, the matching sockets are activated to transform data between the source and the destination. The source and destination ID fields may also contain an optional opcode that is sent to the resource the socket is connected to. The opcode specifies the operation an FU is to perform, or the register in an RF that is to be written or read.

TTA instruction encoding provides two means to specify immediate values that are used, e.g., for constants and jump addresses. When the immediate value is small, consuming only few bits, it can be specified in the source ID field. The immediate value is extracted from the source ID field in the decode stage and placed on the bus in the move stage. A dedicated immediate flag bit is used to control whether the source ID field contains an immediate value or a socket address. Apart from using source ID fields to define the immediate bits, larger immediate values can be specified in a dedicated long immediate field that is included in the end of the instruction word, as shown in Fig. 8. This field cannot be used for any other purpose. Hence, if a long immediate value is not needed, the bits in the long immediate field are wasted.

The width of the TTA instruction word depends on the processor configuration. The instruction word contains as many move slots as there are buses. The width of each move slot depends on the number of functional resources and the number of connections through the sockets to the corresponding bus. The width of the guard field is determined by the number of possible guard terms used for conditional execution. In a typical configuration, guard field is three to four bits wide. The width of the

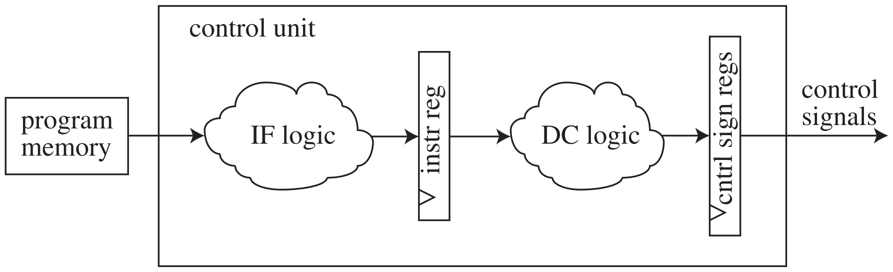


Fig. 7. *Organization of the transport pipeline logic in the control unit of the processor. instr reg: instruction register. cntrl sign regs: control signal registers.*

destination ID field is determined by the number of destination connections on the bus and the widths of the opcodes. The width of the source ID field is determined by the number of source connections on the bus and the widths of the opcodes, or on the width of the short immediate, which typically determines the width of the source ID field. The destination ID fields are typically five to seven bits. The source ID field is typically 9 bits wide as the short immediate is normally configured to be eight bits. The width of the long immediate field can be configured freely, although in MOVE framework, described in Section 3.3.1, the width is restricted to 32 bits.

3.3 TTA-Based ASIP Design Methodologies

As discussed above, two design methodologies have been developed for designing TTA-based ASIPs; MOVE framework in Delft University of Technology, The Netherland, and its successor, TTA Codesign Environment in Tampere University of Technology, Finland. The tools of the MOVE framework were used in this Thesis to design TTA processor configurations for a set of DSP and multimedia applications to evaluate the developed program compression methodologies on TTA. The developed program compression methodology was then integrated to the tools of the TTA Codesign Environment to provide a complete design methodology that can address the code size bloat problem. The integrated compression module can be used to generate the compressed binary code and to generate the structural hardware description of the hardware decompressor and integrate it automatically to the structural hardware description of the processor core. The MOVE framework and the TTA Codesign Environment are discussed in more detail in the following two Subsection.

move slot 0			move slot 1			move slot 2			immediate
G(3)	S(9)	D(6)	G(3)	S(9)	D(5)	G(3)	S(9)	D(7)	LI(32)

Fig. 8. *Structure of the TTA instruction word. G: Guard field. S: Source ID field. D: Destination ID field. LI: Long immediate field. (x): x-bit field.*

3.3.1 Move Framework

MOVE framework is a set of software tools that can automate the design of ASIPs that utilize the TTA paradigm [35]. MOVE framework provides a semi-automatic design methodology that allows to shorten the design-time. The flexibility and scalability of TTA allow to customize the hardware resources of the processor to meet the performance requirements of the target application. Hardware resource optimization allows also to minimize the cost, area, and power consumption of the processor.

MOVE framework consists of three main components, as depicted in Fig. 9. Design space explorer searches the design space for a processor configuration that yields the best cost/performance ratio for a given application. Software subsystem generates the instruction-level parallel code for the designed processor configuration. Hardware subsystem is responsible for generating the hardware implementation of the processor and estimating the costs of a processor configurations in terms of area and power consumption. These main components are described in the following Subsections.

Design Space Explorer

ASIP design requires several iterative steps to be taken to develop and evaluate different processor configurations to find the most optimal configuration that meets the cost and performance requirements of the target application. Performing these steps manually would become too tedious, time consuming and costly, especially if the design space is large. MOVE framework provides a design space explorer tool that can automatically modify the architectural parameters of a processor configuration and evaluate each configuration in terms of cost and performance [47]. This allows the designer to land into the most interesting part of the design space rapidly.

The design space exploration process consists of two phases: resource optimization and connectivity optimization. Resource optimization, executed first, varies the num-

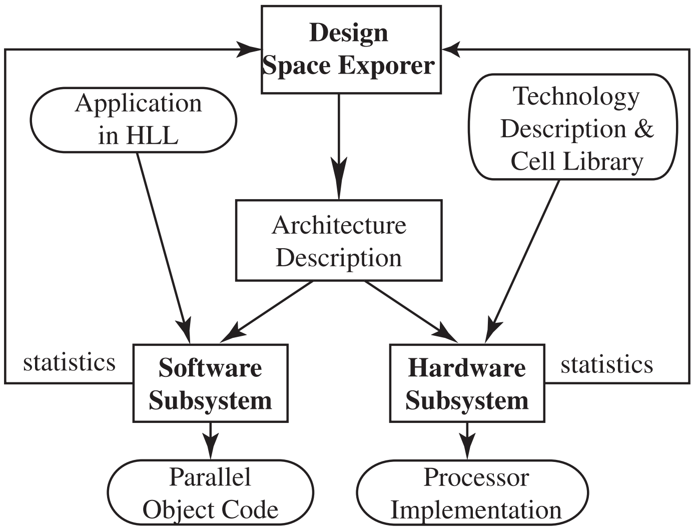


Fig. 9. *Principal design flow in the MOVE framework.*

ber of different hardware resources, such as buses, FUs, and RFs. These hardware resources are described in an architecture description file. Each processor configuration the design space explorer tries out is evaluated in terms of performance, area, and power consumption. Statistics are obtained from the software and hardware subsystems that are invoked within the design space explorer. The simulator of the software subsystem provides statistics on performance and the hardware resources utilization. The cost estimator of the hardware subsystem evaluates the area and power consumption of the given processor configuration. The most promising candidate configurations are chosen for the connectivity optimization where unnecessary connections of the sockets to the data transport buses are removed to reduce the area and shorten the critical path.

Software Subsystem

Software subsystem of the MOVE framework is responsible for compiling the given application into an object code that is executable on the developed processor configuration. The software subsystem includes a simulator that can provide statistics, e.g., on code execution and hardware resource utilization. This information is used in the design space exploration to evaluate processor configurations.

The code generation flow starts from a HLL description of the target application. MOVE framework supports C/C++ languages. The compiler front-end is based on GNU gcc, which is a compiler collection that transforms the HLL code into sequential TTA assembly code. In sequential TTA code, all the RISC type operations of the assembly code generated by the front-end compiler have been translated to corresponding data transports, as was exemplified in Subsection 3.2.2. The compiler back-end, also denoted as the scheduler, is responsible for generating the parallel TTA code from the sequential code. It maps the sequential TTA code onto the available hardware resources of the target architecture that are described in the architecture description file. The generated parallel TTA code can be simulated with a parallel simulator that provides statistics of code execution, code size, hardware resource utilization, and immediate value usage. It can also verify the correctness of the generated code by comparing the results of the simulation against the results of the sequential simulation.

Hardware Subsystem

The hardware subsystem of the MOVE framework is responsible for implementing the designed TTA processor in hardware. The hardware subsystem contains a processor generator that can automatically generate a synthesizable hardware description of the processor configuration. In addition, the hardware subsystem contains a cost estimator to evaluate a processor configuration in terms of area, power consumption, and timing, which are given to the design space explorer.

The processor generator produces a synthesizable RTL VHDL description based on the architecture description file. The basic building blocks, such as FUs, RFs, input and output sockets, and bus drivers are pre-described in VHDL and placed in a hardware database, from where they can be obtained to be instantiated in the processor implementation. The top-level architecture, port mapping, interconnection network, and the control logic are then created automatically. A testbench that can be used to simulate the functionality of the designed processor is also created. Once verified, commercial tools can be used to synthesize the processor into standard cell logic. The processor generator of the MOVE framework was redesigned at Tampere University of Technology, providing more automated VHDL generation of the processor configuration based on the architecture description [109].

The cost estimator of the MOVE framework was also redesigned at Tampere University of Technology [99,101]. Compared to the original estimator, it provided statistics also on power consumption based on a priori information on area, power consumption, and timing of the basic building blocks. Each hardware resource is characterized in terms of the above mentioned criteria and this information is stored in a cost database that is created for the used target technology. The area and critical path delay can be calculated by summing up the values in the database. Power consumption cannot be obtained directly from the database but must be linearly approximated according to the utilization statistics for each of the functional resources.

3.3.2 TTA Codesign Environment

TTA Codesign Environment, follows the design flow introduced in the MOVE framework, but it includes several new tools and features, such as a new processor designer, compiler, instruction set simulator with debugging features [54], and processor hardware and binary image generator [66]. In TCE, the architecture description, operation set extension features, and hardware databases have also been redesigned. The program compression methodology proposed in this Thesis has also been incorporated in the tools of the TCE. The TCE tool set is currently still under development.

The TCE design flow consists of four main phases [55]. Initialization phase provides the sequential TTA program code and the initial architecture description. Processor design and exploration phase offers a semi-automatic design process for tailoring the hardware resources according to the requirements of the target application. Code generation and analysis phase schedules the program to the available hardware resources and analyzes the program execution. In the processor and program image generation phase, the structural hardware description and the final program images are generated. These main design phases are described in the following Subsections.

Initialization

In the initialization step, a front-end compiler, provided by a third party is used to compile the target applications, described in C/C++ language into sequential TTA code, which is stored into a TTA Program Exchange Format (TPEF) binary file. The initial processor configuration can be designed with a Processor Designer that provides a graphical user interface for building the processor configuration from the set of basic building blocks. The hardware resources, such as FUs, RFs, sockets, and buses are added to the processor description and their parameters are defined. The configuration is stored in extensible markup language (XML) format.

Design Space Exploration

The generated sequential TTA code and the initial architecture description from the initialization phase are given as input to the design space exploration phase. The principle of the design space exploration is the same as in MOVE framework. However, in addition to varying the number of functional resources, different implementation alternatives can also be evaluated. E.g., an adder may be implemented either by a ripple-carry or carry-look-ahead added. To support this, all the different implementation alternatives of the resources need to be characterized based on the evaluation criteria, i.e., area, power consumption, and timing. A hardware database (HDB) is used to organize this information.

The used hardware resources are described in an architecture definition file (ADF). It describes the functional characteristics of the architecture, e.g., the number of buses, FUs, and RFs and their connectivity. This information is sufficient for the code generation. However, as the architecture resources can have different implementations, ADF is not sufficient for generating the HDL of the processor configuration. Hence, an implementation definition file (IDF) has been introduces to specify the implementation details of the functional units to allow HDL generation.

Code Generation and Analysis

Code generation and analysis phase is responsible for generating the parallel TTA code from the sequential one. An instruction scheduler is used to map the data transports of the sequential TTA onto the hardware resources of the target processor and generate the parallel TTA code. The instruction scheduler in TCE has been generated modularity in mind, which allows to implement and experiment new scheduling and optimization ideas with simple plug-ins.

Code generation and analysis phase provides also statistics on the program execution. An instruction set simulator is used to simulate the parallel TTA code and provide statistics about the execution, such as cycle count and hardware resource utilization. In addition, the simulator provides debugging capabilities with a graphical user-interface. The simulator allows, e.g., to set breakpoints and inspect the state of the processor at any clock cycle. The contents of the registers in the RFs, input and output registers of FUs, and the contents on the memory can also be inspected. [54]

The management of the operation set is also improved in TCE. The operation definitions are stored into a database, denoted as operation set abstraction layer (OSAL). OSAL specifies for each operation their static properties, such as the number of operands and results and their simulation behavior, which is linked dynamically to the simulator during run-time. An operation set editor is also provided to manage the OSAL and to allow to design user-defined custom operations.

Processor and Program Image Generation

The final phase of the TCE design flow involves generating the structural hardware description of the processor configuration and the final program image. The design flow of this phase is depicted in Fig. 10.

A processor generator is used to generate the structural HDL description of the target processor. The ADF and IDF describe the details of the hardware resources of the target processor configuration. This information is used by the processor generator to obtain the HDL descriptions of the hardware resources, such as FUs, RFs, sockets, and bus drivers from a hardware block library (HBL). These blocks are instantiated at the top-level of the processor design. Top-level architecture, interconnection network, and control logic are then created automatically by the processor generator. [66]

For the program image generation a binary encoding map, defining how to encode the TTA instructions, has to be generated. The map defines for each move slot the addresses of the source and destination sockets and the opcodes for the functional units and register files. In addition, it defines the possible guard and immediate value encodings. The binary encoding, together with the scheduled parallel TTA code are used by the program image generator (PIG) to create the final binary executable. [66]

The program compression methodology developed in this Thesis is integrated to the processor and program image generation tools. PIG includes a dedicated program

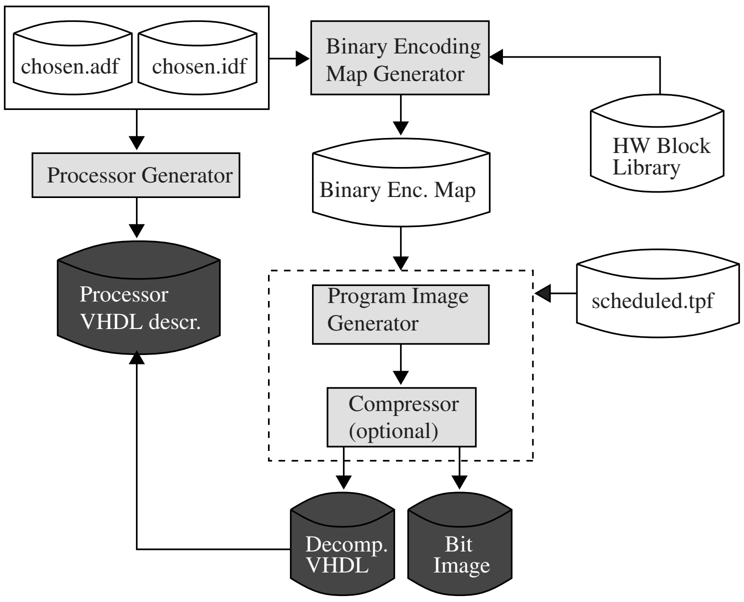


Fig. 10. *Principal design flow of processor and program image generation phase in the TCE design flow.*

compression step in the binary generation flow to compress the program code. The compressor is included as a plug-in module in the PIG. This allows to use different compression methodologies for the program compression. In order to execute the compressed program code, a hardware decompressor has to be included in the hardware of the processor. The compressor plug-in can be used to generate the HDL description of the decompression hardware, which can be included automatically in the final HDL description of the target processor. The presence of the hardware decompressor in the HDL description of the processor is defined in the IDF.

4. PROGRAM COMPRESSION ON TTA

Parallel processor architectures, like TTA, suffer from poor code density. These architectures are typically programmed using a long instruction word than controls all the concurrently operating functional resources. The long instruction word increases the size of the program code and requires larger memories, hence increasing the cost of the chip. In addition, long instruction words increase the power consumption of the chip due to increased program memory bandwidth and increased width of the control flow datapath, i.e., the width of the instruction pipeline inside the control logic.

TTA has even worse code density compared to other parallel architectures, e.g., VLIW, due to its minimal instruction encoding and the TTA programming paradigm. Initial studies on code size reported three times as large code sizes on TTA processors compared to VLIW processors [45]. TTA instructions are minimally encoded in order to reduce the complexity of the instruction decoding procedure. Higher encoding would increase significantly the complexity of the decode logic and would also lengthen the critical path. In the TTA instruction encoding, a typical two-operand RISC operation corresponds to three data transports that all need to be explicitly specified. TTA encoding requires that the functional resources involved in the data transports, either as a source or a destination, need to be explicitly specified as they cannot be determined implicitly as in VLIW encoding, where the location of the field in the instruction word can be used to identify the functional unit. In TTA encoding, only the data bus on which the transport is to be performed is obtained implicitly from the location of the field in the instruction word.

In this Thesis, program compression methodologies presented in the literature were adapted and utilized on TTA processors. The utilized compression methods were chosen based on the findings of the state-of-the-art program compression methodology survey, presented in Chapter 2. The chosen methods included Huffman coding, instruction template-based compression, and dictionary-based compression. These three compression methodologies were chosen as they, in general, showed the best compression ratios. They also allow to estimate the effect of fixed- and variable-width codewords on the decompressor complexity as Huffman coding and instruction template-based compression methodologies result in variable-width and dictionary-based compression in fixed-width codewords.

The adapted compression methods were applied on benchmark applications that were compiled for TTA processors in order to evaluate the effects of program compression on the code density. The results of the code density evaluations are presented in Chapter 6. Code density is measured in terms of compression ratio, which defines the ratio of the compressed and uncompressed program codes. Compression ratio should include also the size of the decoding table, as it is required to perform decompression, but only few compression methods take this overhead into account.

There are also some methodologies, e.g., instruction template-based compression, where there is no decoding table whose size could be included in the compression ratio. Therefore, to fully characterize all the aspects of the compression method, hardware implementations of the processor with and without the support for compression are required. This allows to compare directly the area, and nowadays even more importantly, the power consumption of the uncompressed and compressed implementations. In this Thesis, hardware implementations and area and power consumption estimations were performed for the dictionary-based and instruction template-based compression methods. The results of the area and power consumption evaluations are also presented in Chapter 6.

Section 4.1 describes the principles of the three compression methods and explains how they were adapted to be utilized on TTA. Section 4.2 presents the details of the hardware implementations of the dictionary-based and instruction template-based compression methods, concentrating mostly on the dictionary-based compression.

4.1 Utilized Compression Methods

The following Subsections describe the principles of the three compression methods and how they how were adapted on TTA. Subsection 4.1.1 introduces Huffman coding. Subsection 4.1.2 gives introduction to instruction template-based compression, and Subsection 4.1.3 presents the details the dictionary-based compression.

4.1.1 Huffman Coding

Huffman coding assigns codewords to symbols based on their probabilities of occurrence. Symbols with high probability are assigned short codewords and the less frequent symbols with longer codewords. The generated symbols are prefix-free, i.e., no codeword is a prefix of a codeword of another symbol. Huffman codes for the symbols are assigned by constructing a coding tree based on the symbol probabilities. The tree is constructed by generating first a leaf node for each symbol. The probability of the symbol is assigned to the node. The tree construction process is then initialized by constructing a parent node for the two nodes that have the smallest probability. The sum of the probabilities of the two leaf nodes is assigned for the parent mode. The process of combining two nodes with the smallest probability is repeated until only one parent node remains. That node becomes the root node.

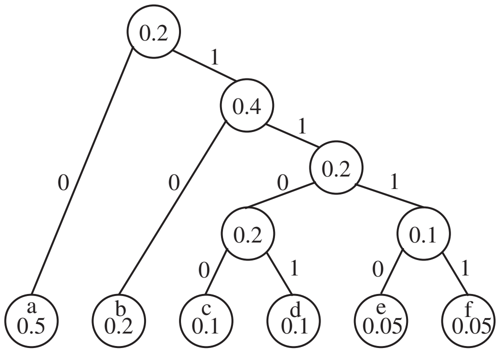
The branches of the non-leaf nodes are labeled as 0 and 1. The codewords for the symbols are obtained by traversing from the root node to each leaf node and by picking up the labels from the branches along the route. An example of a Huffman coding is presented in Fig. 11. Figure 11(a) presents the symbols with their probabilities and the assigned Huffman codewords. Figure 11(b) presents the Huffman tree.

Huffman codewords, their widths, and the original symbols are stored into a decoding table that is used for decoding purposed. An example of a decoding table is illustrated in Fig. 11(c). The widths of the codewords need to be aligned to be the same, i.e., as wide as the widest codewords. Therefore, padding bits need to be added for the shorter codewords. The padding bits are highlighted in grey in Fig. 11(c). Decoding is performed by extracting one bit at a time from the compressed bit stream to a candidate codeword that is matched against the codeword in the decoding table. The codeword length is used to identify the significant bits in the decoding table entries. The padding bits are ignored. The bits from the compressed bit stream are extracted to the to the candidate codeword until a match is found in the decoding table. The symbol corresponding to the codewords is then given as output.

When Huffman coding is applied on program codes, bit patterns in the instruction stream are considered as symbols for coding. Symbols can represent entire instruction words or smaller bit patterns inside the instruction words. In the case of parallel processor architectures, such as VLIW and TTA, instruction words are long and are composed of several fields. In case entire instruction words are considered as sym-

Symbol	Probablity	Codeword
a	0.5	0
b	0.2	10
c	0.1	1100
d	0.1	1101
e	0.05	1110
f	0.05	1111

(a)



(b)

Codeword	Length	Symbol
0000	1	a
1000	2	b
1100	4	c
1101	4	d
1110	4	e
1111	4	f

(c)

Fig. 11. *a) Original symbols with their probabilities and the assigned Huffman codewords. b) Huffman tree. c) Huffman decoding table.*

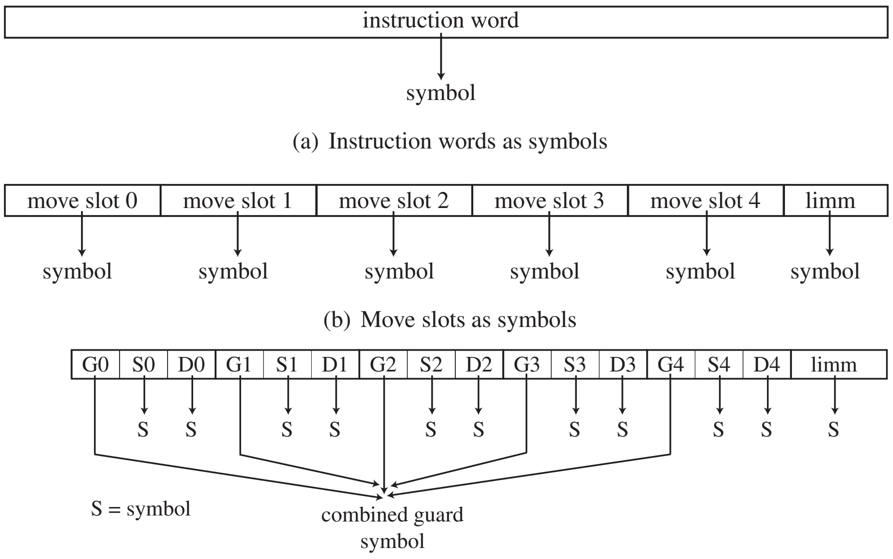
bols, the number of symbols becomes large as there are only few instructions that exist more than once. This leads to almost uniform probability distribution and results in compressed codewords being of the same width. This implies poor compressibility. Hence, a more effective alternative is to divide instruction words into smaller bit patterns that are considered as symbols for the Huffman coding. By having smaller bit patterns as symbols, there are better possibilities to find symbols that are used more frequently that others, i.e., find a non-uniform probability distribution. This approach provides more effective compression.

The long instruction words are typically composed of several fields that control the concurrently operation architectural resources, e.g., functional units in the VLIW architecture. Typically, each field in the instruction word has its own encoding, which is independent of the encoding of other fields. Therefore, as Huffman encoding is based on the probability distribution of the symbols, symbols should be chosen according to the boundaries of the instruction word fields as the bits in the adjacent fields do not usually have any correspondence with each other.

Instructions words are typically encoded hierarchically. For example, VLIW instruction words are composed of operation slots that specify the operations for the concurrently operating functional units. Each operation slot is further composed of smaller fields, such as the opcode and operand fields that define the operation and the registers involved in the execution of the operation. Hence, this kind of instruction encoding allows Huffman encoding to be applied at different granularity levels. The more fine grained the granularity, the better the possibilities to find symbols that are used more frequently than others. Furthermore, at smaller granularity levels the number of possible symbols is smaller. This eases both the coding and decoding. As the coding is done during compile-time, the complexity and the time to perform the coding is of little importance. What matters is the complexity and execution time of the decoding as it is performed during run-time.

As discussed in Section 3.2, TTA instructions are composed move slots, specifying the data transports on the buses, and long immediate fields specifying long immediate values, such as jump addresses and large constants. Each move slot is further composed of guard and source and destination ID fields. This allows to experiment alternative symbol granularities for the Huffman coding. Three symbol granularity levels were experimented in this Thesis.

At the first granularity level, entire instruction words were considered as symbols for Huffman coding. At the second granularity level, move slots were considered as symbols. The long immediate field was considered as a separate symbol stream and was coded independently. At the third granularity level, symbols were chosen according to the source and destination ID field boundaries. The source and destination ID fields were considered as separate symbol streams as they do not have any correspondence between each other. Similarly to the second granularity level, long immediate field was considered as a separate stream and was coded independently. Furthermore, as the guard fields are fairly small, all the guard fields of an instruction word were combined into a single bit pattern. These combined guard field bit patterns of the instruction words were then considered as a single stream that was compressed independently from the source and destination ID fields. Figure 12 illustrates the three symbol granularity levels that were used for the Huffman coding on TTA.



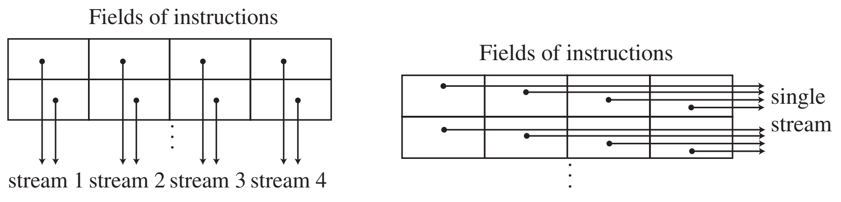
(c) Source and destination ID fields as symbols

Fig. 12. *Different granularity levels for selecting the symbols for Huffman coding.*

For the move slot and ID field granularity levels, coding can be done either vertically or horizontally, as illustrated in Figure 13. In the vertical coding, the fields of the instruction words are considered as parallel streams, as shown in Fig. 13(a). Each parallel stream is encoded independently. This results in as many decoding tables as there are parallel streams. In the horizontal coding, depicted in Fig. 13(b), all the fields are considered as a single stream and a single decoding table is generated. At move slot level, vertical compression considers all the move slots as separate streams whereas horizontal compression combines all the move slots into a single stream. The long immediate field is considered as a separate stream in both cases. Similarly, at ID field level, vertical compression considers all the source and destination ID fields as separate streams. Horizontal compression combines all the source ID fields into a single stream as well as all the destination ID fields. The combined guard field and the long immediate field are considered as separate streams in both cases.

4.1.2 Instruction Template-Based Compression

The instruction template-based compression approach is based on utilizing different instruction formats, denoted as templates, to encode instruction words. These in-



(a) Vertical compression (b) Horizontal compression

Fig. 13. *Compression alternatives for the move slot and ID field granularity levels.*

struction formats contain fields for only a subset of all the fields of the instruction word. The fields not present in the instruction word obtain null encoding, i.e., a NOP, implicitly. This method requires a template selection field at the beginning of each compressed instruction word to define the used template. [108]

The method can be applied on TTA program codes by considering the move slots and the long immediate fields as fields of the templates. For the move slots not present in a template, a null data transport is obtained implicitly. If a long immediate field is not present in a template, an immediate value of zero is obtained implicitly.

The templates are chosen based on the profile of the program code. The profile identifies all the used move slot and long immediate field combinations in the program code, i.e., the combinations of the move slots that carry an actual data transport and the long immediate fields that specify an immediate value that is used in one of the data transports. The best reduction in code size could be obtained by having a template for all of the possible move slot and long immediate combinations used in the program code. However, this would not be very cost-effective as such a large number of templates would increase the complexity of the instruction fetch and decode logic. Therefore, the number of templates has to be limited.

If there are n fields in the instruction word, there are 2^n possible field combinations. Typically, only a small fraction of the possible field combinations are used in the program code. The used combinations can be found by profiling the program code. In addition to finding all the used field combinations, the probabilities of their occurrence are profiled. Based of the profiling information, a limited set of the field combinations can be chosen as templates. The templates need to be chosen so that all the used field combinations in the program code can be covered with them. For the field combinations of the program code that do not have an exact match in the template set, a superset template needs to be chosen, i.e., a template that has at least all the required fields present. On the unused fields, null data transport for the move slots, or zero immediates for the long immediate fields need to be explicitly specified.

In [12], a greedy heuristic algorithm was proposed for the selection of a limited set of templates in VLIW and EPIC architectures. A slightly modified version of the proposed algorithm is used in this Thesis to choose the most beneficial templates. The selection process can be defined as follows. With the target of a set of k templates, a template set τ needs to be found that minimizes the program code size W, given by

$$W = \sum_{i=1}^{m} f_i \times w(N(\tau, C_i))$$

where τ is $\{T_l | l = 1, \ldots, n\} \cup \{T_l | l = n+1, \ldots, k\}$, n being the number of minimal templates that are always present, C_i is a unique move slot and long immediate field combination used in the program code, f_i is the usage frequency of C_i, $N(\tau, C_i)$ is the narrowest template T_l in τ to encode C_i, and $w(T_l)$ is the width of the template T_l. m is the number of instructions that are encoded with the templates.

Before the template selection takes place, the minimal template set needs to be defined. The minimal template set is a set of templates that are needed in order to encode all the field combinations that exist in the program code. In the case of TTA, the minimal set contains only one template, namely a template that has all the possible move slot and long immediate fields present.

With the budget of k templates, the template selection process turns into a process of selecting $k-n$ custom templates. The selection of the custom templates is performed in a loop that, during each iteration, estimates which of the field combinations is the best combination to be included in the template set, i.e., reduces the code size the most. After all the field combinations are evaluated, the most beneficial template is added to the template set. Hence, as $k-n$ custom templates need to be chosen, the loop is iterated $k-n$ times. In the case of TTA there is only one minimal template, i.e., $n = 1$, meaning that the loop is iterated $k-1$ times. The field combinations are evaluated by adding each field combination in its turn to the template set τ as a candidate template T_{C_j} and calculating the benefit of that field combination. The benefit defines the number of bits saved by including T_{C_j} in the template set compared to the original template set that consists of the original τ that contains the minimal templates and the custom template already chosen on the previous iteration rounds. T_{C_j} is removed from τ before the next field combination is defined as T_{C_j} and added to τ to calculate its benefit.

The benefit for T_{C_j} is obtained as follows. τ is re-defined as the current set of templates at any given time in the selection process, containing the minimal templates and the custom templates that have been chosen on the previous rounds. $v(C_i)$ is defined as a lower bound on the width of a template candidate T_{C_j}, corresponding to a field combination C_i, and is obtained by adding together the widths of the fields that are present in C_i. When a candidate template T_{C_j} is added to the template set τ, its benefit can be calculated by comparing the size of the program code encoded with only the current template set τ, and with the T_{C_j} included in τ, i.e., $\tau \cup T_{C_j}$. Hence, the benefit b_{C_j} can be defined as

$$b_{C_j} = \sum_{i=i}^{m} f_i \times [v(N(\tau, C_i)) - v(N(\tau \cup T_{C_j}, C_i))]$$

After calculating the benefits for all the template candidates corresponding to the move slot and long immediate field combinations found from the program code, the combination C_i with the largest benefit is chosen and added to the template set τ. As the chosen template may be used as a superset template for some other field combinations as well, the benefits for the remaining field combinations need to be reset and recalculated again during the next iteration of the selection loop.

Example. The template selection process is exemplified on a TTA processor with three move buses and support for one long immediate value. The instruction word consists of three move slots and one dedicated long immediate field. Figure 14(a) depicts the profile of the program code. It illustrates the used move slot and long immediate combinations with their usage frequencies, i.e., how many times each field combination is used in the program code. The combinations are arranged for clarity according to descending usage frequencies. An unused field is marked with a "-". The instructions are 86 bits wide. The widths of the move slots and the long immediate field are also shown. In the example code, 12 field combinations out of the $2^4 = 16$ possible were used in the program code.

The template selection process begins by initializing the template set τ with the minimal template set, i.e., in the case of TTA with a template that has all the three move

	Combinations				Frequency
1	move1	move2	move3	-	75
2	move1	move2	move3	limm	42
3	-	-	-	-	15
4	move1	move2	-	-	15
5	move1	-	-	-	14
6	move1	-	move3	-	13
7	move1	move2	-	limm	12
8	-	-	move3	-	7
9	move1	-	-	limm	6
10	-	move2	move3	-	5
11	move1	-	move3	limm	3
12	-	move2	-	-	1

Field widths			
18	18	18	32

(a)

Chosen templates					Width (bits)
	18	18	18	32	
00	move1	move2	move3	limm	88
01	move1	move2	move3		56
10	move1				20
11	move1	move3			38

(b)

Fig. 14. *a) Profile of the used move slot and long immediate field combinations with their usage frequencies. b) The chosen templates and their widths.*

slots and the one long immediate field present. The selection of the custom templates is then performed in the selection loop. In this example, a budget of four templates ($k = 4$) was used, i.e., three custom templates could be chosen. During the first iteration, the field combination that has all the fields except for the long immediate field present turns out have the highest benefit, so it is chosen as the first custom template. In addition to using the chosen template for the first field combination, it can be used also for the combinations 2, 3, 4, 5, 6, 8, 10, and 12. The minimal template has to be used for the remaining field combinations. The chosen template saves 32 bits times the frequency of the field combinations the template can be used for.

During the second iteration of the loop, the most beneficial field combination is the field combination with only the first move slot present. It can be used for the combinations 3 and 5. On these combinations it saves 36 bits times their frequency as it has two move slot fields less compared to next narrowest template in τ, i.e., the template that was chosen on the previous iteration. During the last iteration of the loop, the field combination that has the first and third move slots present becomes the most beneficial combination and is added to the template set. It can be used for the combinations 6 and 8, and it saves 18 bits times the usage frequency of these combinations as it now has one move slot less that the next narrowest template.

Once the templates are chosen, the template selection fields need to be assigned for the templates to identify each template. As four templates were used in the example, 2-bit wide selection field is needed. Figure 14(b) illustrates the chosen templates and their widths. The three custom templates chosen in this example reduce the code size already by 31%.

4.1.3 Dictionary-Based Compression

Dictionary-based program compression is based on the fact there are bit patterns, e.g., instructions, in a program code that occur more than once. In addition, often only a small part of all the possible bit patterns is used. These properties can be utilized by storing all the unique bit patterns of a program code into a dictionary and replacing the bit patterns in the program code with indices that point to the dictionary. Given a program with N unique bit patterns, the length of the dictionary index becomes $\lceil log_2|N|\rceil$ bits. The dictionary introduces an overhead that has to be taken into account when the effectiveness of the method is of concern. In case there are only few repeated instructions, the method may turn out not to reduce the code size as the dictionary becomes large, containing most of the bit patterns of the original program code. Compression is effective when the following equation is fulfilled:

$$(K-N)w - K\lceil log_2|N|\rceil > 0$$

K is the number of bit patterns in the program code, N is the number of unique bit patterns in the program code that are stored into the dictionary, and w is the width of the original bit pattern. The left-hand side of the equation defines the number of bits saved when the dictionary-based compression is applied, and it needs to be greater that zero to make the method effective.

Program code

	31 0
0	addi, r1, r4, #16
1	mul, r4, r2, r1
2	shr, r2, r3, #5
3	sub, r5, r2, r1
4	mul r4, r2, r1
5	add, r1, r4, #16
6	bneqz r1, #10c
7	add, r1, r4, #16

(a)

Dictionary

	31 0
0	addi, r1, r4, #16
1	mul, r4, r2, r1
2	shr, r2, r3, #5
3	sub, r5, r2, r1
4	bneqz r1, #10c

Program code

	2 0
0	index 0
1	index 1
2	index 2
3	index 3
4	index 1
5	index 0
6	index 4
7	index 0

(b)

Fig. 15. *a) The original uncompressed program code. b) The generated dictionary and the compressed program code.*

Figure 15 illustrates an example of dictionary-based compression. Figure 15(a) illustrates the original program code, consisting of RISC type 32-bit wide instructions. The unique instructions of the program code are stored into a dictionary and replaced with indices that point to the dictionary. As there are five unique instructions, 3-bit indices are requires to access the dictionary. The dictionary and the compressed program code are depicted in Fig. 15(b). The decompression procedure is fairly straightforward. The dictionary index, fetched from the program memory, is used to access the dictionary, from where the original bit pattern is obtained and can be sent forward in the instruction pipeline.

Dictionary-based compression has been typically applied to entire instruction words on single issue processors, such as CISC or RISC. Compressing entire instruction words does not result in effective compression on parallel processor architecture as the instruction words are typically long and are composed of several fields. There are only feq unique instructions, which results in most of the instructions to be stored into the dictionary. Even if two instructions differ in only one bit, they are both unique and need to be stored. Better compression can be achieved when the instructions are divided into smaller bit patterns to which compression is applied to.

TTA instructions can be divided into smaller bit patterns at different granularity levels. The same granularity levels as in the case of Huffman coding are utilized. That is, unique bit patterns to be stored into the dictionary and to be replaced with dictionary indices are searched at the level of full instruction words, at the level of move slots, and at the level of ID fields. At move slot and ID field levels, the vertical and horizontal compression approaches can be applied. Vertical compression divides instructions into parallel streams according to the field boundaries and searches unique bit patterns inside these streams. A unique dictionary is created separately for each stream. Horizontal compression searches for unique bit patterns across all the fields in the instruction words and only a single dictionary is generated. At both move slot and ID field levels the long immediate field is considered as a separate stream. At ID field level the source ID and destination ID fields are considered separately in horizontal compression and two dictionaries are generated, one for the unique source IDs and one for the unique destination IDs. Further on, all the guard fields of an instruction word are combined into a single bit pattern. Unique bit patterns are then searched among these guard field combinations in the program code.

4.2 Hardware Implementations

In order to take the implementation details of the decompression hardware and the program memory into account for better evaluation of the compression methods and to obtain statistics in terms of area and power consumption, the TTA processors used in the evaluations were implemented in hardware. The area and power consumption were evaluated on both the uncompressed and compressed implementations. The processor generator of the MOVE framework was used to create the VHDL descriptions of the TTA processors. For the compressed designs, the decompression circuitry was designed manually and implemented in the control path of the processor core. Program memories, both uncompressed and compressed, and data memories were implemented using random access memories (RAM). The memories were included in the design as pre-synthesized macro cells provided by the technology vendor. The processor designs were synthesized on a low-power 130 nm CMOS standard-cell technology with 5 metal layers using the Synopsys Design Compiler version 2003.06. Area statistics were obtained from the synthesis results. The power consumption values were obtained from the Design Compiler using the switching activity information obtained from gate-level simulations that were performed using Mentor Graphics Modelsim version 5.31.00.

In order to implement the compressed systems in hardware, the program memory has to be adjusted for the compressed codewords and the control path of the processor must be modified to handle fetching compressed instructions and decompressing them back to the original format before they are decoded. The design principles of the decompressor for the compression methods are presented in Subsection 4.2.1.

The hardware implementations and area and power consumption estimations were done for the dictionary-based and instruction template-based compression approaches. Huffman coding method was not implemented in hardware as conclusions for its effectiveness could be drawn based on the results of the instruction template-based compression as both methods result in variable-width instructions and therefore need to address the same issues, such as buffering the incoming instruction fetch packets and aligning branch and jump targets to addressable memory locations. The details of the hardware implementations of the dictionary-based and instruction template-based compression are discussed in more detail in subsections 4.2.2 and 4.2.3.

4.2.1 Implementation Principles

For the execution of the compressed program codes, the most important hardware module is the decompressor that is responsible for decompressing the compressed instructions back to their original form. The complexity of the decompression circuitry depends highly on the compression method. The decompressor of the dictionary-based compression methods is extremely simple as it consists of only the dictionary that contains the original bit patterns. The index that is used to access the dictionary is fixed-width, so there is no need for any additional logic to obtain the codeword from the incoming bit stream. Huffman coding and instruction template-based compression result in more complex decompression procedure due to variable-width codewords. Before the decompression takes place, the codewords need to be identified from the incoming bit stream. This requires either an identifier before each codeword to specify the length of the codeword, or the bits of the input stream need to be investigated one by one until the entire codeword is identified.

In addition to more complex decompressor, also the instruction fetch logic becomes more complex when compressed instructions are variable-width. The IF stage has to forward enough bits each clock cycle to the decompressor so that an entire instruction word per clock cycle can be obtained from the decompressor for decoding. If this is not the case, the pipeline of the processor needs to be stalled. The width of the instruction fetch packet needs to be adjusted to cover the worst case situation, i.e., that all the codewords belonging to an instruction word are the widest possible. In case the codewords are smaller, the fetch packet contains bits of the next instruction word. These bits need to be buffered and concatenated with the bits of the next instruction fetch packet at the beginning of next clock cycle to construct the next instruction word. The buffer holding the excessive bits fetched from the program memory may eventually fill up. To avoid overflow in the buffer, instruction fetching has to be stalled for awhile until the buffer can accept new fetch packets.

One of the main design decisions is the location of the decompressor. As discussed in previous Chapters, there are several alternatives for this, each having their advantages and drawbacks. In the evaluations performed in this Thesis, the decompressor is placed inside the control path of the processor core. In such as case, the decompressor is typically implemented in an additional pipeline stage. This increases the depth of the pipeline and affects the jump delay as it takes one clock cycle longer for the pipeline to fill up again after a branch or jump has been taken. If the decompression procedure is simple enough, the decompressor can be integrated with either the IF or the DC stage into a single pipeline stage and avoid the increase in pipeline depth.

Compression may also influence branching as the instruction addresses may change due to compression. This is especially the case when compressed instructions become variable-width. As the variable-width instructions are placed one after another in the program memory to preserve code space, they are typically not aligned to addressable locations. However, to fulfill random access requirements, branch target instructions need to be aligned to addressable locations so that they can accessed immediately after a branch is taken. In addition, target addresses in the branch instructions need to be corrected. One possibility is to patch the branch target addresses to point to the compressed address space. Another alternative is to provide a mapping between the compressed and uncompressed address spaces with a dedicated LAT. All these issues affect the effectiveness of the compression method.

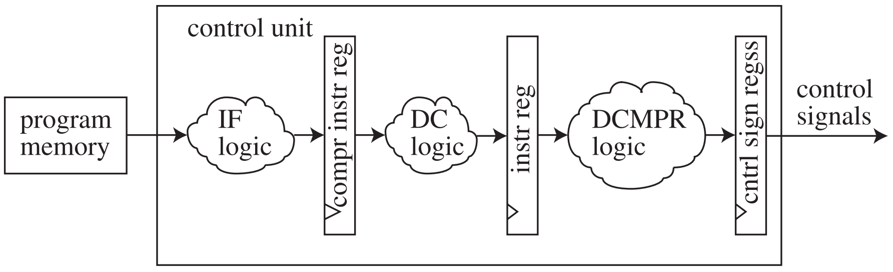
4.2.2 Implementing Dictionary-Based Compression

The decompression procedure for the dictionary-based compression is extremely simple. It involves accessing a look-up table of unique bit patterns with an index that is fetched from the program memory. TTA processors allow to implement the decompressor inside the control path of the processor core as the processors are described in VHDL.

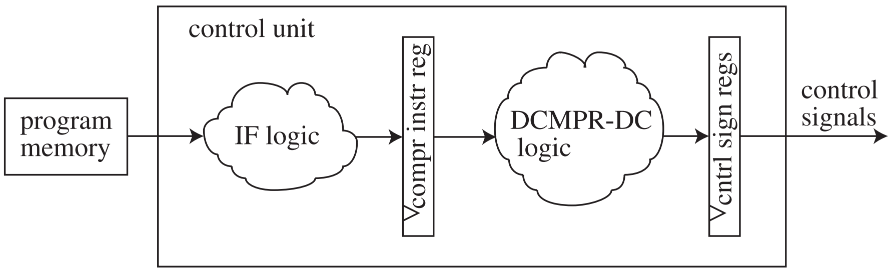
One alternative is to add a dedicated pipeline stage for the decompression (DCMPR). The additional pipeline stage needs to be placed in between the IF and the DC stages of the three-stage TTA transport pipeline, which was introduced in Subsection 3.2.4. The organization of the transport pipeline with the additional DCMPR stage is illustrated in Fig. 16(a). The figure illustrates the combinatorial logic related to the pipeline stages and the corresponding pipeline registers. The addition of a dedicated DCMPR stage increases the depth of the transport pipeline by one. This implies an increased jump latency as one additional clock cycle is required for the branch target instruction to reach the execution stage. The penalty of the increased jump delay is paid whenever a branch is taken.

As the dictionary access is a fairly simple operation, the decompressor can also be integrated to one of the existing pipeline stages. This allows to maintain the depth of the pipeline and the jump delay unchanged. Instruction fetching involves modifying the contents of the program counter and accessing the memory, whereas decoding simply compares the source and destination IDs of the instructions word to the hardwired socket IDs and generates the control signals for the transport network. The timing of the IF stage is more critical as the delay of the program memory is along the critical path. Therefore, the timing of the processor is affected less when the decompressor is integrated to the DC stage. Furthermore, as there is a register in between the IF and DC stages, it is more favorable to perform decompression in the DC stage as it allows to maintain the register as wide as the compressed instruction word. Figure 16(b) illustrates the organization of the transport pipeline with the integrated DCMPR-DC stage.

The decompressor consists of only the dictionary that contains the original bit patterns. The dictionary is accessed with the compressed codewords that are fetched from the program memory. The dictionary can be implemented using RAM, which allows to modify the contents of the dictionary, e.g., to adapt to changes in the pro-



(a) Separate DCMPR stage



(b) Integrated DCMPR-DC stage

Fig. 16. *Transport pipeline organization for the alternative decompressor implementations.*

gram code. Hence, full programmability can be maintained with this approach. However, using RAM requires that the decompressor is implemented in a separate pipeline stage as it takes at least one clock cycle to obtain data from RAM. An alternative to RAM is to use read only memory (ROM). It requires less transistors to implement, i.e., it provides smaller area and consumes less power than RAM. However, the contents of ROM cannot be modified, which limits the allowed instructions the processor can execute.

Another alternative to RAM or ROM implementation is to describe the dictionary as a matrix as standard logic vectors and let the synthesis tool optimize the dictionary into logic cells. This approach can exploit the fact that dictionary-based compression cannot fully utilize all the redundancy that exists in the program code. For example, the bit patterns stored into the dictionary may differ only in few bit positions. The synthesis tool can detect this redundancy and optimize the logic of the dictionary to achieve even smaller area than would be achieved with ROM. This approach makes the dictionary compression effective also at full instruction level as the dictionary entries, i.e., entire instruction words, contain a lot of redundancy that can be utilized.

In the evaluation of the dictionary-based program compression method on TTA, the dictionary was implemented both using RAM and standard cells. When RAM was used to implement the dictionary, a separate pipeline stage had to be added for the decompression. Standard cell implementation of the dictionary allowed to experiment both the separated and integrated decompressor alternatives.

4.2.3 Implementing Instruction Template-Based Compression

The implementation of the instruction template-based compression is more complex compared to the implementation of the dictionary-based compression. The main reason is that the compressed instructions become variable-width. In addition to including the decompressor in the control path of the processor, the instruction fetch logic needs to be modified.

The decompressor for the instruction template-based compression is responsible for detecting from the input bit stream the templates that have been used the encode the instructions of the program code. The templates specify move slot and long immediate fields for a subset of all the possible fields. The guard and source and destination ID fields of the move slots that are present in the template need to be dispatched to the correct field decoders. For the remaining decoders, IDs corresponding to a null data transport need to dispatched implicitly. The long immediate values need to be dispatched to the immediate registers in the immediate unit. In case a long immediate field is not included in the template, a value 0 is transported to the immediate register. As the width of the compressed instruction word is not known, each template precedes a template-selection field that defines the used template and its width.

The decompressor has to be capable of decompressing a single instruction word per clock cycle. As the width of the compressed instruction word is not known during instruction fetch, the instruction fetch packet needs to be configured as wide as the widest compressed instruction words, i.e., as wide as the widest template, to guarantee that enough bits to decompress a single instruction are obtained from the program memory each clock cycle. The width of the program memory is adjusted according to the width of the fetch packet.

During decompression, the template-selection field is first investigated to determine the width of the current template and the fields present in the template. This information is used to extract the bits of the move slots from the instruction fetch packet and dispatch them to the corresponding source and destination ID field decoders. For the remaining move slots, null data transport IDs are dispatched implicitly. The bits of the long immediates need also the be extracted from the templates and stored into the immediate registers inside the immediate unit. In case the template does not consume all the bits of the instruction fetch packet, the remaining bits belong to the following template or templates. The decompressor contains a shift register to store the remaining bits, which are consumed in the following clock cycle. The remaining bits of the fetch packets are stored left-aligned in the shift register. Left-aligning is made to always obtain the template-selection field from the leftmost bits of the shift register. For the decompression of the next template, the bits of the shift register are consumed before the bits of the next instruction fetch packet.

Whenever the template is narrower than the instruction fetch packet, the remaining bits accumulate the number of bits in the shift register. After awhile, it may turn out that all the bits of a template are found from the shift register. As the size of the shift register needs to be limited, in the implementation used in this Thesis to two times the width of the instruction fetch packet, an overflow may occur in the shift register. To avoid the overflow, a pre-fetch buffer is implemented in the instruction fetch logic. The bits fetched from the program memory are forwarded to the decompressor from the pre-fetch buffer only when the shift register can accommodate an entire instruction fetch packet. In case there is no space, the bits fetched from the program memory are stored into the pre-fetch buffer, implemented as a first-in-first-out (FIFO) with a depth of three instruction fetch packets.

It may also turn out that if the basic block is large and several instructions in the basic block can be encoded with a template that is significantly smaller that the widest template, the FIFO may also fill up. In order to avoid overflow in the FIFO, the instruction fetch needs to be stalled. This is done by disabling the memory enable signal, which also disables the increment of the program counter. Hence, when the instruction fetch is re-enabled, fetching continues from the correct memory address. Due to the latency of the program memory, one more instruction fetch packet is received from the program memory after the memory is disabled. Therefore, the IF stage has to be stalled when the FIFO still has capability to store one instruction fetch packet. The transport pipeline can still proceed normally.

The instruction fetch is re-enabled when all the instruction fetch packets stored into the FIFO are consumed. The management of the FIFO is made so that the requirement for decompressing one instruction word per clock is satisfied. This requires the memory enable to be activated before the FIFO is totally empty, i.e., it still contains one instruction fetch packet, as there is a latency of one clock cycle before new data can be obtained from the program memory after the memory has been enabled.

The complexity of the decompressor hardware is further increased due to the shifting network for the shift register and the alignment network for dispatching the guard, the source and destination ID fields to the corresponding field decoders and the long immediates to the immediate registers. Each bit of the shift register can come from all the possible bit locations of the shift register or from all the bit location of the input of the decompressor. This implies a large multiplexer for each bit location of the shift register. Similar situation happens for aligning the fields of the instruction words to the field decoders. They can come from several different locations, which results in need for a large multiplexer. The complexity of the shifting and alignment network can be reduced by limiting the instruction words to a multiple of a fixed number of bits, referred in [12] as *quantum*. This reduces the number of inputs for the multiplexers of the shifting and alignment network and reduces the complexity of the decompressor. For the TTA implementations, a quantum of 16 bit was chosen. The templates need to be aligned to the next multiple of a quantum. This may require padding bits to be inserted to the compressed program code.

Instruction template-based compression complicates also branching. As the variable width templates are aligned to the program memory one after another, branch targets may not be aligned to addressable memory locations by default. Hence, modifications to the instruction alignment in the program memory has to be made. Branch target alignment may require additional padding bits to be placed at the end of the memory lines preceding the branch targets. On the other hand, this introduces another problem when the branch is not taken. In such as case, the padding bits will be interpreted as bits of a template. This will result in incorrect execution. To avoid such a situation, an end of a packet (EOP) bit, similarly as in [10], is added to each instruction template to identify whether the bits of the next template will follow right after, or are aligned at the beginning of the next addressable memory location. When a branch has not been taken, the EOP identifies whether there are padding bits that need to be flushed from the shift register.

For the evaluation of the instruction template-based compression on TTA, the cases of using four and 16 instruction templates to encode the instructions of the program code were implemented. The decompressor was integrated in both cases to the DC stage of the instruction pipeline to avoid additional pipeline stage. The decompression procedure, even though more complex than in dictionary-based compression, is still simple enough that it can be integrated with the decoder into a single pipeline stage without affecting the cycle time too much.

4.3 Summary of the Program Compression Evaluations

The code density evaluations, based on the compression ratio that measures the ratio of the compressed and uncompressed code size, were performed for all of the three program compression methodologies. For the Huffman and dictionary-based compression methods, the code densities were evaluated at three different granularity levels; at full instruction, at move slot, and at ID field levels. The area and power consumption evaluations were performed only for the instruction template-based and dictionary-based compression methodologies. Huffman coding method was not evaluated as conclusions for its effectiveness could be drawn based on the results of the instruction template-based compression methodology. Table 4 summarizes the code density and area and power consumption evaluations that were performed for the three program compression methods that were adapted on TTA.

As the dictionary-based compression is the main focus in this Thesis, it was evaluated in more detail. For the area and power consumption evaluations, two alternative implementations of the dictionary were evaluated. In the first case, the dictionary was implemented using RAM. In the second case, the dictionary was implemented using standard cell logic to allow the synthesis tool to optimize the logic of the dictionary.

Table 4. *Summary of the program compression methodology evaluations on TTA.*

Methodology	Granularity level	Code density	Area and power consumption		
			Separated DCMPR-stage		Integrated DCMPR-DC-stage (Std cell)
			RAM dict	Std cell dict.	
Huffman coding	Instruction	x	-	-	-
	Move slot	x	-	-	-
	ID field	x	-	-	-
Instruction template-based compression		x	-	-	x
Dictionary-based compression	Instruction	x	x	x	x
	Move slot	x	x	x	x
	ID field	x	x	x	x

In addition to evaluating two alternative implementation methods of the dictionary, two alternative locations of the decompressor in the transport pipeline of the control logic of the processor were evaluated. First, the decompressor was implemented in a separate pipeline stage in between the IF and DC stages. Then, the decompressor logic was integrated to the logic of the DC stage to avoid an additional stage in the transport pipeline. The area and power consumption evaluations for the instruction template-based compression methods were made by selecting four and 16 instruction templates to encode the instruction of the program code.

5. MAINTAINING PROGRAMMABILITY AFTER COMPRESSION

In general, programmability can be defined as a capability of a processor to respond to changes in software, i.e., to be able to execute the modified program with new instructions. This requires that the instruction set contains such instructions that can be used to execute all the operations in the source program. Instructions in the instruction set have to support at least all the fundamental operations, with which the other, possibly more complex operations, can be executed. Hence, the instruction set does not necessarily have to contain instructions for all the possible operations. For example, support for a multiplication operation can be covered with an add operation.

Moreover, the instructions do not necessarily need to be capable to operate on data from all the possible locations. For example, an add operation may be allowed to operate directly on the data that is in a register file, but not on data that is in the data memory. In the latter case, the data has to be loaded from the data memory to the register file before the add operation can be executed. In both of these cases, the performance of the processor is negatively affected due to incompleteness and inconsistency of the instruction set.

The extent to which the instruction set is consistent and complete is often referred to as orthogonality [67]. It characterizes the range of the supported operations in the instruction set and the degree of the supported operands and addressing modes for these operations. An instruction set is said to be fully orthogonal if any instruction can use any type of data as operands through any addressing mode. In general, processors with more orthogonal instruction set provide better performance as the processor is easier to program.

Even though highly orthogonal instruction set provides better performance, high orthogonality has its drawbacks as it tends to increase the size of the program code. More bits are required to encode the instructions as the instruction set has to cover a large set of different operations that all support all the possible operands and addressing modes. Furthermore, not all applications can take the advantage of the full orthogonality. Therefore, instruction sets are usually designed to provide better code density, i.e., some orthogonality is sacrificed for more compact instruction encoding.

The orthogonality is usually reduced when program compression is applied. Program compression usually utilizes the redundancy that exists in a given program code and the instructions are encoded in such a way that only the ones that are used in the given program code are supported after the compression has been applied. This results in poor orthogonality and may result in limited programmability, i.e., into a situation where the processor cannot anymore execute the entire set of new instructions that were added to the program code when the program was modified.

Orthogonality and programmability are also affected by the compression granularity. At high granularity levels, large bit patterns, such as entire instruction words are considered as symbols for compression. These bit patterns are usually composed of several smaller independently encode fields, such as opcode and operand fields. When the large bit patterns are compressed, only the combinations of the smaller fields used in the original program code are supported. This results in poor orthogonality and highly limited programmability. Compression at more fine-grained granularity level provides means for better orthogonality and programmability. Bit patterns considered as symbols for compression are smaller and usually correspond to the smaller fields inside the instruction words, e.g., opcodes and operands. This allows to support a larger set of operation with different operands and addressing modes as the instructions can be composed of the smaller fields that are all uniquely compressed.

The implementation of the decompressor, i.e., the decoding table, may also affect the programmability. As program compression is usually tailored for a particular application, the decoding tables will contain only the bit patterns that were obtained from the original application. This results in limited programmability. However, if the decoding tables are implemented using RAM, they can be reloaded with all the new bit patterns when the program code has been modified. This allows to maintain full programmability. All that is needed is to compress the modified program code and reload the RAM with the new decoding table.

The size of the RAM has to be designed carefully. If it is too small, all the bit patterns of the modified program code may not fit into it. On the other hand, if the RAM is too large, there is unused space, which reduces the compression effectiveness.

In case area and power consumption are of great concern, decoding tables can be implemented using ROM or standard cells. As ROM is denser than RAM, it has smaller area and power consumption. Standard cell implementation of the decoding table allows optimization of the decoding table logic during synthesis. This provides even smaller area and power consumption. The drawback of both of these approaches is that the decoding tables cannot be modified after they have been implemented in hardware. This results in limited programmability.

Traditionally, full programmability has been supported by allowing the program code to contain both compressed and uncompressed instructions. Instructions that have a match in the decoding table can be compressed; the others are stored into the program code in uncompressed format. During execution, uncompressed instructions pass the decompressor and are forwarded directly to the decoder whereas compressed instructions pass through the decompressor before they can be decoded. Therefore, additional information has to be added to the program code to make a distinction between the two instruction formats. This additional information increases the size of the program code, and therefore, reduces the effectiveness of the compression.

One alternative to make a distinction between compressed and uncompressed instructions is to reserve one of the compressed instructions as a "mark" that precedes each uncompressed instruction [19]. Instructions can also be divided into compressed and uncompressed regions that are identified with two different marks [18, 73]. Fetch packets may also be configured to precede a flag bit that is used to identify whether the bits of the packet describe a compressed or uncompressed instruction.

Support for both compressed and uncompressed instructions may also affect the performance. In case the program memory width is adjusted according to the compressed instructions, fetching uncompressed instructions from the program memory requires several clock cycles during which the transport pipeline has to be stalled. This degrades the performance. On the other hand, if the program memory width is adjusted according to uncompressed instructions, several compressed instructions can be fetched in a single fetch packet. Instruction fetching has to be stalled until all of them have been executed. A pre-fetch buffer is also required to avoid overflow in the instruction register. In case a compressed instruction is encountered, the fetch packet has more bits to store than are extracted from the instruction register. Moreover, uncompressed instructions typically need to be aligned to addressable memory locations. This increases the code size as padding bits are needed.

In case the decompressor is implemented in its own pipeline stage, bubbles may emerge in the transport pipeline degrading the performance. As there is a separate decompress stage, uncompressed instructions require one clock cycle less to reach the execute stage as they bypass the decompressor. Hence, when a compressed instruction is to traverse through the transport pipeline after the uncompressed one, it takes one clock cycle longer for it to reach the execute stage as it has to pass through the decompress stage. Therefore, one bubble is created in the pipeline.

Instruction execution in such a case illustrated in Fig. 17(a). Compressed instructions are denoted as *C(i)*, uncompressed instructions as *U(j)*, and the mask informing that an uncompressed instruction will follow as *CM*. The example in Fig. 17(a) assumes an ideal usage of a pre-fetch buffer so that enough bits are always available to represent a compressed or an uncompressed instruction. In the example, the compressed instruction *C3* is to be executed after the uncompressed instruction *U1*. As *C3* has to pass through the decompress stage, which *U1* bypasses, a bubble is created in the decode stage, and it proceeds to the execution stage in the next clock cycle. There is one idle cycle seen in the decompress stage as *U1* bypasses the decompress stage.

Region compression [18, 73] can avoid the insertion of a bubble after each uncompressed instruction by dividing the program code into compressed and uncompressed regions. Two marks are required to identify these regions. A mark, *CM*, that is as wide as a compressed instruction is used to indicate the beginning of an uncompressed region. The mark is examined in the decompress stage. Another mark, *UM*, which is as wide as an uncompressed instruction identifies the end of an uncompressed region and the beginning of a compressed region. This mark is examined in the decode stage. In region compression, stall cycles are paid only when a transition from an uncompressed region to a compressed region takes place. As the example of the transport pipeline execution for region compression shows in Fig. 17(b), two bubbles are created in the execute stage when transferring from an uncompressed region back to a compressed region.

In the example in Fig. 17(b), a compressed mark *CM*, decompressed after the compressed instruction *C1*, marks the beginning of an uncompressed region, which consists of two uncompressed instructions, *U1* and *U2*. The uncompressed region is defined to end with an uncompressed mark *UM*, that is identified in the decode stage. Due to the presence of *UM*, the next instruction to be executed, *C2*, is fetched two clock cycles after the last uncompressed instruction *U2*. As it has to go through the

Cycle	IF	DCMPR	DC	EX
1	C1	-	-	-
2	C2	C1	-	-
3	M	C2	C1	-
4	U1	M	C2	C1
5	C3	-	U1	C2
6	C4	C3	Bubble	U1
7	C5	C4	C3	Bubble
8	C6	C5	C4	C3
9	C7	C7	C5	C4

(a)

Cycle	IF	DCMPR	DC	EX
1	C1	-	-	-
2	CM	C1	-	-
3	U1	CM	C1	-
4	U2	-	U1	C1
5	UM	-	U2	U1
6	C2	-	UM	U2
7	C3	C2	Bubble	Bubble
8	C4	C3	C2	Bubble
9	C5	C4	C3	C2

(b)

Fig. 17. *a) An example of transport pipeline execution when an uncompressed instruction is xexecuted among compressed instructions. b) An example of transport pipeline execution in region compression.*

decompress phase, a bubble is created in the decode stage and it propagates to the executed stage. Furthermore, as the *UM* did not proceed any further from the decode stage, there are two bubbles seen in the execute stage. There are three idle cycles seen in the decompress stage as the uncompressed instructions bypass the decompressor.

As described above, maintaining programmability by supporting the execution of both compressed and uncompressed instructions has some drawbacks. These drawbacks are more severe on parallel processor architectures as the compressed instructions may be significantly smaller than the uncompressed ones. Performance is often affected, especially when the program memory is adjusted according to compressed instructions. Furthermore, instruction fetching and pipeline organization may become more complex.

5.1 Programmability on TTA

Out of the three compression methods that have been evaluated on TTA, programmability and instruction set orthogonality are affected when Huffman coding and dictionary-based compression methods are applied. The instruction template-based compression method maintains the full programmability and the original orthogonality of the instruction set as the minimal template, containing all the fields of the instruction words is always included in the template set. The minimal template corresponds to the original instruction word and, therefore, supports the same set of operations and operand and addressing combinations as the original instructions.

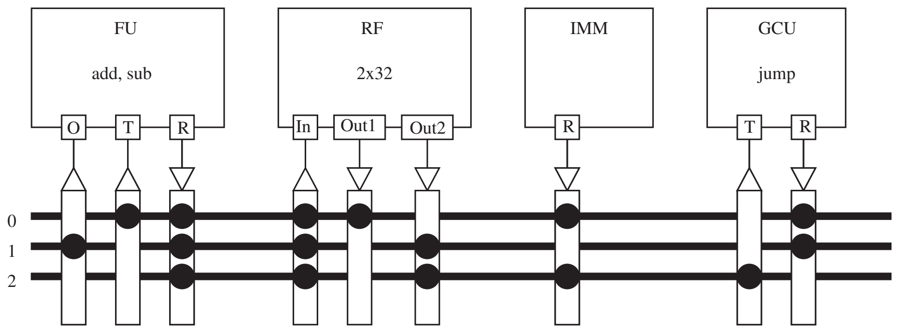
As discussed in the previous Section, programmability can be maintained by supporting the execution of both compressed and uncompressed instructions. The drawback of this method is that it degrades the performance and adds complexity to the control logic, especially on parallel processor architectures, such as TTA. However, the programming model of TTA allows to maintain the programmability also without the support for uncompressed instructions. In this Thesis, such a methodology has been proposed for the dictionary-based compression, but the method could also be extended quite easily for the Huffman coding compression method. The methodology was originally introduced by the author of this Thesis in [P5].

The requirements for maintaining full programmability on TTA are presented in Subsection 5.1.1. The proposed methodology for maintaining the programmability for the dictionary-based compression is introduced in Subsection 5.1.2. Evaluations of the proposed methodology in terms of performance, area, and power consumption are presented in Chapter 6.

5.1.1 Requirements for Programmability

Due to the mirrored programming paradigm of TTA, there is only one type of operation that can be programmed; a data transport from a source to a destination. Therefore, the programmability can be maintained on TTA if the data transports from all the possible sources to all the possible destinations can be executed. As the source and destination ID fields may contain opcodes, all possible opcodes need to be supported in the required data transports. In case of an FU, the opcode specifies the operation the unit is to perform. These opcodes can be supported by considering the trigger input as many destinations as there as opcodes. In case of an RF, the opcode specifies the register to be written or read. Similarly to FUs, the RF inputs have to be considered as many destinations as there are opcodes, and the RF outputs as many sources as there are opcodes.

Figure 18 shows an example TTA processor configuration and the required data transports to maintain the programmability. Figure. 18(a) depicts the architecture of the processor. The processor contains one FU that can perform two operations, add and subtract, one RF with two 32-bit registers, an immediate unit (IMM) placing large immediates on the buses, and a global control unit (GCU). The architecture has three



(a)

FU_R -> FU_O
FU_R -> FU_T(add)
FU_R -> FU_T(sub)
FU_R -> RF_In(0)
FU_R -> RF_In(1)
FU_R -> GCU_T(jump)

RF_Out2(0) -> FU_O
RF_Out1(0) -> FU_T(add)
RF_Out1(0) -> FU_T(sub)
RF_Out1(0) -> RF_In(0)
RF_Out1(0) -> RF_In(1)
RF_Out2(0) -> GCU_T(jump)

RF_Out2(1) -> FU_O
RF_Out1(1) -> FU_T(add)
RF_Out1(1) -> FU_T(sub)
RF_Out1(1) -> RF_In(0)
RF_Out1(1) -> RF_In(1)
RF_Out2(1) -> GCU_T(jump)

IMM_R -> FU_O
IMM_R -> FU_T(add)
IMM_R -> FU_T(sub)
IMM_R -> RF_In(0)
IMM_R -> RF_In(1)
IMM_R -> GCU_T(jump)

GCU_R -> FU_O
GCU_R -> FU_T(add)
GCU_R -> FU_T(sub)
GCU_R -> RF_In(0)
GCU_R -> RF_In(1)
GCU_R -> GCU_T(jump)

(b)

Fig. 18. *a) An example TTA processor configuration. b) All the possible data transports that can be executed on the example TTA processor.*

move buses to transport data in between the resources. Connections between the buses and the functional resources are established through input and output sockets. The FU inputs and outputs are denoted as *T* (trigger), *O* (operand), and *R* (result). The RF inputs and outputs are denoted as *In* and *Out*.

As the FU has two inputs ports and one output port and there are two possible opcodes for the trigger input, the FU has in total three destinations and one source. The RF has two 32-bit registers, which correspond to two distinct destinations and sources. The number of input our output ports on the RF does not affect the number of sources or destinations as all the registers are accessible through all the input and output ports, so any of then can used for the transports. The IMM unit has one source and the GCU to one destination and one source.

When all the possible sources and destinations have been identified, all the required data transports for full programmability can be defined. The number of the required data transports depends on the number of sources and destinations. In case there are n sources and m destinations, there are $n \times m$ possible data transports that need to be supported. In the example processor configuration, depicted in Fig. 18(a), there are five sources and six destinations meaning that in total 30 data transports are required for full programmability. These data transports are illustrated in Fig. 18(b).

To perform each of the data transports, a bus needs to be assigned for each data transport to which both the source and the destination are connected to through the output and input sockets. E.g., the data transport from the FU result output to the trigger input of the GCU with an opcode for jump, *FU_R –> GCU_T(jump)*, has to be executed on bus two as the input socket of the GCU trigger input is not connected to any other bus. In case there is no bus on which the required data transport can be executed, i.e., the input and output sockets involved in the data transport are not connected to the same bus, a connection to either of these two sockets needs to be added so that the data transport can be made.

In addition to being capable of transporting data in between all the sources and destinations, all possible immediate values used to specify, e.g., large constants and jump addresses, should also be supported. In TTA, immediates may be up to 32 bits, which implies a huge number of possible immediate values that would need to be supported.

5.1.2 Programmability Support Methodology

Dictionary Extension

Dictionary-based program compression stores unique bit patterns into a dictionary and replaces them in the program code with indices to the dictionary. As the compression is usually tailored for a particular application, the bit patterns stored into the dictionary represent only a small subset of all the possible bit patterns. This leads to poor orthogonality, which usually means that full programmability cannot be guaranteed. This makes it difficult or even impossible to modify the program code so that it could still be executed.

As discussed in the previous Subsection, full programmability can be guaranteed on TTA if data can be transported from all the sources to all the destinations in the architecture. For the dictionary-based compression, this requirement can be fulfilled by extending the dictionary with such bit patterns that the required data transports can be executed. The drawback of the dictionary extension is that it adds an overhead to the size of the dictionary and increases also the power consumption due to increased amount of logic. The overhead depends on how many of the required data transports are already supported in the dictionary, and also on the compression granularity.

When the compression is performed at instruction level, the dictionary contains entire instruction words. This means that the additional entries supporting the execution of the required data transports need to be added as entire instruction words. As each of the required data transports need to be independently executable, they cannot be combined into the entries. Hence, each of the required data transports requires its own entry, i.e., an instruction word, where only a single data transport is specified on one of the move slots. Null data transports need to be specified on the other move slots. This implies a large overhead in the size of the dictionary. The overhead depends also on the number of buses in the architecture; the more buses in the architecture, the wider the instruction word and the larger the overhead.

The overhead of extending the dictionary to maintain the programmability can be decreased significantly if the compression is applied at smaller granularity levels, i.e., at move slot or ID field levels. At move slot level, the bit patterns corresponding to the required data transports can be added to the move slot dictionaries. This avoids the need to specify the null data transports in the dictionary entries, as is the case at instruction level. Furthermore, the move slot dictionaries may already contain plenty of the bit patterns that are required for full programmability. This reduces the amount of bit patterns that need to be added. Therefore, the overhead of the dictionary extension is significantly smaller than at instruction level.

The overhead is even smaller when the compression is applied at ID field level, where the bit patterns stored into the dictionaries correspond to source and destination IDs. Support for the required data transports can be provided by adding the source and destination IDs of the required data transports to the corresponding source and destination ID dictionaries. The same IDs may be used in several data transports. In case these transports are performed on the same buses, the IDs need to be added to the dictionaries only once. In addition, the source and destination ID field dictionaries may already contain quite a large set of the required source and destination IDs, which reduces the number of entries that need to be added.

Dictionary extension at the three different granularity levels is exemplified in Fig. 19. Figure 19(a) shows the set of data transports that are to be added to the dictionaries to maintain the programmability on the example TTA processor, depicted in Fig. 18(a). For each data transport, the assigned bus is also shown. Figures 19(b)- 19(d) illustrate how the dictionary extension is performed at the three different granularity levels. The new dictionary entries are added after the original entries that are stored into the dictionaries during compression. For clarity, dictionary extension for the move slot and ID field granularity levels is shown only for the vertical compression approach. Dictionary extension for the horizontal compression proceeds similarly with the exception that only a single dictionary is used for all the move slots at move slot granularity level. At ID field level, one dictionary is created separately for the source and destination IDs. Long immediate fields and the combined guard fields at ID field level are ignored in the example.

The orthogonality after dictionary extension is still significantly worse compared to the orthogonality of the original uncompressed instruction set, even though the method allows to maintain the programmability. The orthogonality, and hence also the performance are highly affected by the compression granularity. At instruction level, the extended entries allow to specify only a single data transport per instruction. This results in poor performance when these entries are utilized to cover instruction that were not included in the original dictionary. The orthogonality and performance are better at smaller granularity levels where it is possible to execute several data transports in parallel, provided that the extended entries are distributed somewhat evenly to the dictionaries so that they can be executed in parallel.

When the application has been modified after the compression has been applied and the dictionaries have been implemented, the extended dictionaries are given to the compiler as a constraint so that a correct program code can be created. The compiler first tries to create instructions that match the original entries that were created during the compression. This is possible in case the modifications to the program code are local and most of the instructions remain unchanged. In case a matching instructions cannot be found among the original dictionary entries, the desired functionality has to be described using the extended dictionary entries.

At instruction level, the extended entries will provide fairly poor performance as only one data transport per instruction can be performed. This means that the modifications to the program code need to be small, e.g., bug fixes. In case there is a need

Data transport	Bus
FU_R -> FU_T(add)	0
FU_R -> RF_In(0)	1
RF_Out1(0) -> FU_T(add)	0
RF_Out2(1) -> FU_O	1
IMM_R -> FU_T(sub)	0
IMM_R -> RF_In(1)	2
GCU_R -> FU_O	1
GCU_R -> RF_In(0)	0

(a)

Instruction word

	...	
FU_R -> FU_T(add)	null	null
null	FU_R -> RF_In(0)	null
RF_Out1(0) -> FU_T(add)	null	null
null	RF_Out2(1) -> FU_O	null
IMM_R -> FU_T(sub)	null	null
null	null	IMM_R -> RF_In(1)
null	GCU_R -> FU_O	null
GCU_R -> RF_In(0)	null	null

(b)

move slot 0	move slot 1	move slot 2
...	...	...
FU_R -> FU_T(add)	FU_R -> RF_In(0)	IMM_R -> RF_In(1)
RF_Out1(0) -> FU_T(add)	RF_Out2(1) -> FU_O	
IMM_R -> FU_T(sub)	GCU_R -> FU_O	
GCU_R -> RF_In(0)		

(c)

src ID 0	dst ID 0	src ID 1	dst ID 1	src ID 2	dst ID 2
...	...	...	...	...	...
FU_R	FU_T(add)	FU_R	RF_In(0)	IMM_R	RF_In(1)
RF_Out1(0)	FU_T(sub)	RF_Out2(1)	FU_O		
IMM_R	RF_In(0)	GCU_R			
GCU_R					

(d)

Fig. 19. *An example of dictionary extension. a) an example set of data transports to be added. The extended dictionaries for compression at b) instruction, c) move slot, and d) ID field levels. The three dots (...) represent the entries that correspond to bit patterns already stored into the dictionary during compression.*

to prepare for larger changes in the code and to maintain a feasible performance, some reduction in area and power consumption through compression has to be sacrificed by applying compression at smaller granularity levels, e.g., at move slot level.

In this Thesis, further study of the dictionary extension methodology was limited to compression at instruction level as it provided the best reduction in area and power consumption, which were the main objectives.

Minimizing the Dictionary Extension Overhead

The overhead of dictionary extension can be reduced by minimizing the number of entries that need to be added. This is crucial especially for the compression at instruction level where each additional data transport requires a separate entry in the dictionary. The overhead becomes large especially on large processor configurations as the number of possible destinations and sources is large. This increases the number of the required data transports. Moreover, instruction words on such large processor configurations are long, which increases the dictionary overhead even further as each dictionary entry requires more bits.

The number of the required data transports can be reduced by introducing a dedicated register, denoted as *global connection register (GCR)* through which all the data transports for maintaining the programmability are to be executed. Utilization of the GCR allows to split the source-to-destination data transports into two distinct data transports; a data transport from a source to the GCR and from the GCR to a destination. This reduces the number of the required data transports. In case there are n sources and m destinations, with the utilization of the GCR the number of the required entries can be reduced from $n \times m$ to $n + m$. The larger the n and m, the greater the reduction. Fig. 20 illustrates the required data transports in the example TTA processor, shown in Fig. 18(a), when the GCR is utilized. The number of the required data transports is reduced in the example from 30 to 11.

To introduce the GCR in the architecture, a dedicated RF to hold the GCR can be added or, alternatively, a register in one of the existing RFs can be chosen as GCR. As the GCR is used to transport data in between the sources and the destinations, the input and output sockets of the RF that hosts the GCR should be connected to as many buses possible. If a dedicated RF for the GCR is added, the sockets of the RF can be adjusted to be connected to as many buses as required. If one of the existing RFs is used to host the GCR, an RF that has the best connectivity is chosen. The connectivity C of an RF can be defined as

FU_R -> GCR	GCR -> -> FU_O
RF_Out1(0) -> GCR	GCR -> FU_T(add)
RF_Out1(1) -> GCR	GCR -> FU_T(sub)
IMM_R -> GCR	GCR -> RF_In(0)
GCU_R -> GCR	GCR -> RF_In(1)
	GCR -> GCU_T(jump)

Fig. 20. *The optimized set of the required data transports with the utilization of the GCR in the example TTA processor configuration illustrated in Fig. 18(a).*

$$C = 2 \times min\{|\bigcup_{i=1}^{n} Is_i|, |\bigcup_{j=1}^{m} Os_j|\}$$

where Is_i is the set of buses an input socket i of the RF is connected to, Os_j is the set of buses an output socket j of the RF is connected to, and n the number of input sockets and m the number of output sockets in the RF.

Hence, the connectivity takes into account in addition to the number of connections to the buses also the balance between the number of buses the input and output sockets are connected to. This guarantees that there are better possibilities to move data from all the sources to the GCR and from the GCR to all the destinations without a need to add a large number of connections to the input and output sockets of the chosen RF. The RF with the best connectivity is chosen to host the GCR. In case the connectivities are equal, the RF with the larger number of pure connections to the move buses is chosen to host the GCR. Any register inside the chosen register can be chosen as GCR. In practice, the register with index *0* is chosen as the GCR. That register cannot be used anymore to hold data as the register is used to execute all the source-to-destination data transports.

The utilization of the GCR to minimize the number of additional entries in the dictionaries reduces the orthogonality even further, and therefore, also the performance. The extended dictionary entries cannot transport data directly from the source to the desired destination in a single clock cycle. Instead, the data is transported during one clock cycle from the source to the GCR, and then in the next clock cycle from the GCR to the destination. Hence, two clock cycles are required to complete a single source-to-destination data transport.

Immediate Support

As was mentioned in Subsection 5.1.1, in addition to transporting data from all the possible sources to all the possible destinations, all the possible immediate values need to be supported as well in order to maintain full programmability. After compression, only the immediate values in the original program code are supported. They represent only a small subset of all the possible immediates.

All the possible immediate values can be supported by specifying them along the compressed instructions in the program memory and extracting them during the decode phase in the transport pipeline. In order to do this, the immediate values need to be distinguished from the compressed instructions. This can be accomplished by reserving one of the compressed codewords as an *immediate specifier* that is placed before the immediate bits in the program memory.

TTA supports up to 32-bit wide immediates. However, all the 32 bits are rarely used. Using only 32-bit immediates in the program would be impractical and would lead to unnecessary increase in the code size. Furthermore, the performance would decrease as the processor would always have to be stalled until all the 32 bits of the immediate would have been fetched from the program memory. The increase in the code size and the decrease in performance can be minimized by introducing a set of immediate formats that are of different width. A unique immediate specifier is required for each immediate format. The widths of the distinct immediate formats are chosen to be multiples of the width of the program memory, which is typically adjusted to the width of the compressed instruction. This type of immediate format organization allows the immediate bits to be aligned evenly to the program memory. Immediate formats at least up to 32-bit need to be supported. For example, in case the compressed instructions are 9 bits wide, four different immediate formats are needed; 9, 18, 27, and 36 bits.

Figure 21 shows an example of placing immediate values along the compressed instructions. Compressed instructions are defined as Ci, where i is the index of the instruction. Immediate specifiers preceding the immediate bits are denoted as Is_k, where k is the index of the specifier, specifying also out of how many instruction fetch packets the immediate value is to be assembled from. Immediate packets are denoted as Il_p, where l is the index of the immediate value and p the index of the packet. Whenever an immediate specifier is encountered, the processor is stalled and

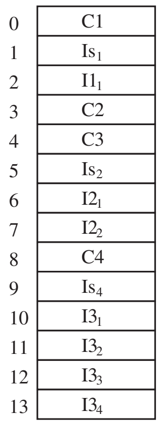


Fig. 21. *Example of a compressed program code including immediate values.*

the bits on the following k memory lines are fetched one line at a time and assembled together into a single immediate value that is placed in the 32-bit wide immediate register in the immediate unit. The immediate value can then be transported from the immediate unit during the following clock cycle to the desired destination. In case the GCR is utilized, a data transport from the immediate register of the immediate unit to the GCR is implicitly obtained and executed once all the immediate bits have been fetched. This avoids the need to explicitly specify the data transport to the GCR.

In the example shown in Fig. 21, compressed instruction $C1$ is followed by an immediate specifier Is_1, which identifies that the following immediate consists of a single fetch packet, $I1_1$, which follows the immediate specifier. The second immediate value, following the compressed instructions $C3$, is identified with an immediate specifier Is_2, and defined in fetch packets $I2_1$ and $I2_2$. The last immediate, following instruction $C4$, is composed out of four fetch packets, $I3_1$, $I3_2$, $I3_3$ and $I3_4$ and identified with an immediate specifier Is_4.

Conditional Execution

The guard fields in the move slots of TTA instruction words provide means for conditional execution. The guard value that is specified in the guard field defines whether the data transport specified in the move slot is to be executed on the bus or to be squashed. In case the data transport is to be executed unconditionally, a true value is specified in the guard field. In case conditional execution is required, the guard field is assigned a value that specifies a Boolean register whose value, or its inverse, is used to guard the data transport. The number of guard values that can be assigned for a data transport depends on the number of Boolean registers. The more Boolean registers in the processor configuration, the larger the number of possible guard values. Supporting all of these different guard values for all the required data transports would increase the size of the dictionary as entries with all the possible guard values would need to be added for all the possible data transports.

However, conditional execution can be supported with a single guard value that can be used to guard all of the required data transports. This provides that only a single entry per data transport has to be added to the dictionary. The chosen guard value is chosen to correspond to one of the Boolean registers, typically the register with index “0”. For unconditional execution, the Boolean register chosen to be used as the guard value has to be set true so that the data transport will be executed. In case conditional execution is required, the Boolean register has to be updated just before the required data transport takes place so that the Boolean valued stored into the chosen Boolean register can be used to guard that particular data transport. Before the next unconditional instruction is executed, a true value has to be stored in the chosen Boolean register.

This methodology worsens the orthogonality as the range of the supported Boolean registers is limited to one. Performance is also affected as for each conditional data transports the Boolean register has to be set to contain the guard value before the data transport is executed, and then cleared before the next uncoditional data transport is to be executed.

6. RESULTS

The three program compression methods were evaluated on TTA by using the MOVE framework to design TTA processors for a set of benchmarks from the digital signal processing and multimedia application domains. The three program compression methods that were adapted on TTA were then applied to the program codes of the benchmark applications that were compiled on the designed TTA processors. The compression methods were utilized as discussed in Chapter 4.

Besides evaluating the code density in terms of compression ratio, which is based on the bit sizes of the program memory and the decoding tables, the processor designs were implemented in hardware to obtain more accurate statistics of the effectiveness of the compression methods in terms of area and power consumption. Alternative possibilities on how to implement the dictionary and how to place the decompressor in the control logic were also evaluated, as discussed in Chapter 4.

Section 6.1 presents the evaluation methodology, i.e., describes the used benchmarks and explains the taken design methodology to design TTA processors for the benchmark applications. Section 6.2 presents the results of the code density evaluations. The results of the hardware implementations are presented in Section 6.3. Results for the code density evaluations were originally published in [P1, P2, P3], and the results for the hardware implementations in [P4, P6].

The methodology to improve the orthogonality of the instructions set was also evaluated. The effects of the proposed methodology on the performance, area, and power consumption were measured. Section 6.4 presents the results of these evaluations.

6.1 Evaluation Methodology

A set of benchmark applications were chosen and implemented to be used in the evaluations. The benchmarks were chosen mostly from the digital signal processing domain as it is the domain where the advantages of TTA, i.e., hardware customization and low area and low power consumption can be utilized effectively. Few benchmarks were also taken from the multimedia application domain. The benchmarks used in the evaluations are discussed in more detail in Subsection 6.1.1.

The TTA processors for the evaluations were designed using the MOVE framework. The design space explorer was used to evaluate the design space of the possible processor configurations for the benchmark applications and provide statistics on their performance, area, and power consumption. Processor configurations for the benchmarks were chosen based on the exploration statistics. The benchmark applications were then compiled on the chosen processor configurations using the retargetable compiler of the MOVE framework tool set. For the area and power consumption evaluations, HDL descriptions of the chosen processor configurations were generated using the processor generator of the MOVE framework. The HDL descriptions were then synthesized to standard cells logic on a 130 nm low-power technology to evaluate the processors in terms of area and power consumption. The processor design for benchmarking is presented in details in Subsection 6.1.2.

6.1.1 Benchmarks

For the code density evaluations, six benchmarks from the digital signal processing and multimedia application domains were used. The first four benchmarks are applications from the DSP application domain. The first two benchmarks realize discrete cosine transform (DCT), a kernel widely used in video, image, and audio coding. Here the constant geometry two-dimensional (2-D) DCT algorithm proposed in [65] and one-dimensional (1-D) DCT proposed in [115] have been considered. Constant geometry algorithms are regular and modular. Thus, they allow efficient exploitation of parallelism. The 2-D 8×8 DCT is realized with row-column approach, i.e., the entire 2-D transform is realized with the aid of 1-D transforms. The 1-D 32-point DCT contains five functions, each corresponding to one processing stage of the transform. Each processing stage is written totally unrolled, i.e., without loops. In general, this type of code results in large program code size but provides large enough basic blocks for the compiler to utilize the parallelism available in the application.

The third DSP benchmark is Viterbi decoding [117], which is a widely used algorithm in many decoding and estimation applications in the communications and signal processing domain. The algorithm decodes 256-state 1/2-rate convolutional codes and, contains path metric computation and survivor path search. The last DSP benchmark performs edge detection that is used to process digital images and mark the points where the luminous intensity changes sharply [24].

The last two benchmarks are applications from the multimedia application domain. They realize MPEG2 decoding and JPEG compression applications that have been taken from the MediaBench benchmark set [69]. The MPEG2 decoding application has been written in a way to emphasize the correct implementation of the moving picture expect group (MPEG) standard. The JPEG compression application performs JPEG compression from a variety of graphic file formats using the joint picture expert group (JPEG) standard.

These six benchmarks represent applications from DSP and multimedia domains. All of them are written in C-language using integer data types. This allows to omit floating-point arithmetics, which is not perfectly supported in the current MOVE framework tool set. Furthermore, the DCT applications are realized using fixed-point representation with normalized number ranges, which is typical for DSP realizations. The DCT applications, Viterbi decoding, and edge detection algorithms have been optimized for DSP. The MPEG2 decoding and JPEG compression applications have been written as precise specifications of the MPEG and JPEG standards and, are not optimized for DSP. The applications also vary widely in program code size. The applications optimized for DSP are small whereas MPEG2 decoding and JPEG compression are fairly large.

All of the six applications that were used to evaluate to code density could not be used for the area and power consumption evaluations. In order to simulate the correct functionality of the processor, and more importantly, to obtain statistics on the switching activity for power consumption estimation, applications that utilize the services of the operating system, such as memory allocation, through system calls, or utilize files to send data to or receive data from the processor could not be used. Therefore, the multimedia applications MPEG2 decoding and JPEG compression and the edge detection applications had to be omitted, leaving three DSP benchmarks in the benchmark set. A 1024-point Radix-4 fast Fourier transform (FFT) [31] was added for this reason to the benchmark set, as it could be used in the actual hardware simulations. FFT algorithms have gained an important role in many DSP systems, e.g., communication systems. They are used e.g., in speech and image processing.

6.1.2 Processor Design

The TTA processor configurations for the benchmark applications were designed using the design space explorer of the MOVE framework that searches the design space by evaluating several alternative processor configurations in terms of area, power consumption, and performance by varying the number of functional resources, such as FUs, RFs, and buses. The local optimal configurations, denoted also as pareto points [22], together with the evaluation statistics are given to the designer. On the basis of the evaluation data, the designer can choose the most suitable processor configuration for the second step of the exploration, the connectivity optimization, where the unnecessary connections from the interconnection network are removed.

For the code density evaluations, the design space explorer was used to obtain processor configurations for the six benchmark applications. Each of the benchmark applications was explored separately. On the basis of the exploration data, three processor configurations were chosen for each application; a *large*, a *medium*, and a *small* configuration. As the names suggest, these three configurations vary in the number of functional resources, i.e., FUs, RFs, and buses.

The large configuration was chosen to represent a configuration with large number of buses, FUs, and RFs and good performance. The drawback of this configuration is that is consumes a significant amount of area and power. Also, the instruction word becomes fairly long. As the full parallelism provided by the architecture cannot always be fully utilize, the resulting program code becomes large.

The small configuration was chosen from the other end of the design space. It contains only the minimal resources that are required to perform the application. The advantage of this configuration is its small area and power consumption, but the performance is poor. The size of the program code is fairly small as most of the parallel functional resources can be effectively utilized, resulting in only few null data transports to be specified on the move slots.

The medium configuration was chosen as a compromise between cost and performance. The chosen configuration has a decent performance but with modest area and power consumption. The configuration was chosen from the point in the design space where the effect of including more functional resources in the architecture on the performance starts to diminish. Figure 22 shows an example pareto curve for the 32-point DCT application and the locations of the chosen processor configurations

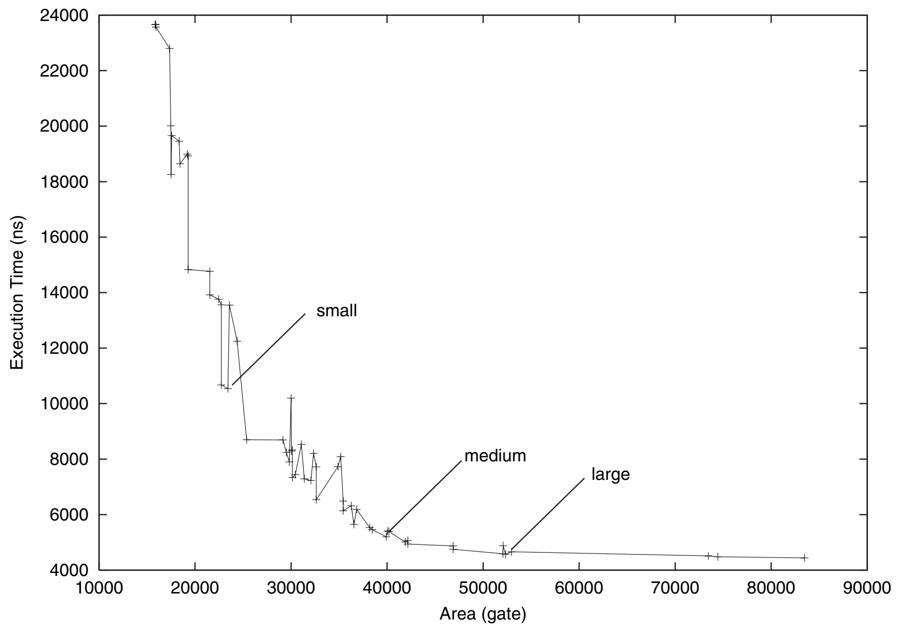


Fig. 22. *The chosen processor configurations for the 32-point DCT application on the pareto curve.*

on the curve. The hardware resources of the chosen processor configurations for the six benchmark applications, identified with letters A-R, are tabulated in Table 5.

After the processor configurations were chosen for the benchmark applications, the compiler of the MOVE framework was used to compile the benchmarks on the target processors to obtain statistics on performance and code size. The highest optimization level of the compiler (-O2) was utilized. The statistics of the benchmark applications compiled on the target TTA processor configurations A-R are presented in Table 6. The statistics clearly show that larger processor configurations provide better performance but result also in significantly larger program code.

Table 6 depicts also the number of actual data transports in the program code. There are more data transports on the bigger processor configurations, but the difference is not as big as the difference in the program code sizes. This indicates that the increase in the code size on the larger processor configurations is mostly due to the increased number of null data transports. This is due to larger processor configurations having more move slots, which results in larger number of null data transports to be specified in the less parallel sections of the program code.

Table 5. *Hardware resources of the chosen processor configurations. LSU: Load-stored unit. MUL: Multiplier. ARITH: Arithmetic unit. SHIFT: Shifter. LOGIC: Logic unit. CMP: Compare unit. Regs.: Registers. Iwidth: Instruction width.*

Application	Conf.	Buses	FUs						Regs.	Iwidth
			LSU	MUL	ARITH	SHIFT	LOGIC	CMP		[bits]
2-D 8×8 DCT	A	13	4	1	2	2	-	1	60	248
	B	8	3	1	1	2	-	1	60	171
	C	3	2	1	1	1	-	1	16	83
32-point DCT	D	10	2	1	3	1	-	-	34	196
	E	8	2	1	2	1	-	-	30	160
	F	4	1	1	2	1	-	-	30	99
Viterbi dec.	G	9	2	-	2	2	2	1	48	200
	H	6	2	-	2	1	1	1	46	146
	I	3	2	-	1	1	1	1	34	89
Edge detect.	J	9	4	1	4	1	-	3	60	198
	K	5	1	1	1	1	-	1	60	126
	L	3	1	1	1	1	-	1	31	89
MPEG2 decompr.	M	8	3	1	3	3	3	1	40	184
	N	6	4	1	2	1	3	1	40	148
	O	5	1	1	1	1	1	1	40	127
JPEG compr.	P	13	1	1	2	1	1	1	60	272
	Q	6	1	1	1	1	1	1	60	146
	R	3	1	1	1	2	1	1	46	91

For the hardware implementations, the benchmark set had to be restricted to 32-point DCT, 8×8 DCT, Viterbi decoding, and 1024-point FFT applications. These applications do not use system calls or access files, so they could be simulated at gate-level using the hardware simulator to obtain switching activities for the power consumption estimations.

Designing the same set of three processor configurations for each of the benchmark applications for the hardware evaluations would have resulted in a large number of processor configuration to be described in VHDL and synthesized into logic. To reduce the number of processors to be implemented hardware, the design space explorer was used to design such processor configurations that could execute all the four benchmark applications. All the benchmark applications were given simultaneously for the design space explorer, which then searched the design space for processor configurations that would be capable of executing all the four applications effectively. This meant that each processor configuration had to be evaluated separately for each of the applications and then use the average over all the benchmarks to make a decision how to proceed in the exploration, i.e., what resource to add or remove next. As

Table 6. *Performance and code size statistics of the benchmark applications, each compiled on the three different TTA processor configurations.*

Application	Conf.	Clock cycles	Data transports	Instr. count	Code size [bytes]
2-D 8×8 DCT	A	18827	434	127	3937
	B	19420	450	138	2950
	C	33229	469	208	2158
32-point DCT	D	456	2644	477	11687
	E	481	2686	502	10040
	F	822	2856	843	10432
Viterbi dec.	G	1441938	906	253	6325
	H	1512212	888	262	4782
	I	2468580	883	385	4283
Edge detect.	J	348054	16091	565	13984
	K	349663	15764	605	9529
	L	361483	15755	644	7165
MPEG2 decompr.	M	2825344	986	7652	175996
	N	2827945	950	8018	148333
	O	2830314	953	8550	135731
JPEG compr.	P	9044490	29214	12198	414732
	Q	9089799	25778	12308	224621
	R	9783150	23910	12909	146840

the applications were of different sizes, weighting factors were used to balance the effects of the applications.

From the results of the design space exploration, two processor configurations were chosen for the hardware evaluations; a small (A) and a medium (B) processor configuration. The small configuration contains the minimal set of resources to execute all the four applications. It provides small area and low power consumption, but with poor performance. The medium configuration was chosen to provide a good performance for all the applications with a modest number of hardware resources. This configuration was chosen close to the point in the design space where the performance increase with additional resources starts to diminish. The functional resources of the two processor configurations are tabulated in Table 7.

In order to place the code sizes into the context of parallel processor architectures, the four benchmarks were compiled in addition to the two TTA processor configurations also on the Texas Instruments TMS320C62x VLIW processor [3]. Statistics on the code size and performance of the benchmark applications compiled on the two TTA processor configurations and the TMS320C62x are tabulated in Table 8. The program codes on TMS32062x were compiled using two different compiler optimiza-

Table 7. *Hardware resources of the two TTA processor configurations.*

Conf.	Buses	Functional units	Registers	Instr. width [bits]
A	5	1 multiplier, 1 load-store, 1 ALU, 1 compare, 1 shifter, 1 logic, 1 sign extend	19	127
B	8	1 multiplier, 2 load-stores, 2 ALUs, 1 compare, 3 shifters, 1 logic, 2 sign extend	52	192

tions; one optimizing the code for speed and the other for size. The results indicate that the TTA processors provide significantly better performance than TMS32C62x. In terms of the number of clock cycles, TTA processor configuration A has six times better performance than TMS320C62x and configuration B more than eight times better performance. This indicates the TTA paradigm allows to tailor the hardware resources is such a way that the parallelism available in the applications can be utilized effectively to achieve high performance.

From the code size perspective the two TTA processors result in larger code sizes than the TMS320C62x. TTA code sizes are on average 75% larger compared to the speed optimized code on TMS320C6x. The effect of the size optimization in the TMS320C62x is negligible. The differences in the code sizes between the two TTA processor configurations depend on the application. 32-point DCT application has larger code size on configuration B whereas in the 1024-point FFT the situation is the opposite. The codes sizes in the two other applications are more or less the same. Larger processor configurations should, in general, result in larger program codes as the instruction words are longer. However, in 1024-point FFT the compiler can effectively utilize the available parallelism. The number of instructions is reduced significantly, from 315 to 149, which reduces the size of the program code even though the instruction words are wider.

The chosen processor configurations were described in VHDL using the processor generator of the MOVE framework. It generates automatically the VHDL description for the control logic and the interconnection network, fetches VHDL descriptions of the basic building blocks, i.e., FUs, RFs, and sockets, from a database and maps them to the architecture by creating a top-level VHDL description. Data and program memories were included in the processor implementations as pre-synthesized static RAM (SRAM) macro blocks from the technology vendor libraries. They were configured in such a way that the same memories could be used for all the four benchmarks. A single-ported, 32kB data memory was chosen for the configuration A and

Table 8. *Benchmark statistics on the two TTA processor configurations and on the TMS320C62x.*

Application	Target	Clock cycles	Instr. count	Code size [bytes]
32-point DCT	TTA (A)	466	484	7684
	TTA (B)	423	441	10584
	C62x (speed)	2382	229	7332
	C62x (size)	2415	225	7200
2-D 8 $\times$ 8 DCT	TTA (A)	22959	163	2588
	TTA (B)	19455	137	3288
	C62x (speed)	258992	65	2072
	C62x (size)	275649	63	2008
1024-point FFT	TTA (A)	282547	315	5001
	TTA (B)	123667	149	3576
	C62x (speed)	745876	62	1972
	C62x (size)	755277	61	1948
Viterbi dec.	TTA (A)	2710738	367	5827
	TTA (B)	1568227	253	6072
	C62x (speed)	13481571	89	2836
	C62x (size)	13500053	88	2824

a 32kB dual-ported data memory for the configuration B. For the uncompressed program memories, a 512 word, 128-bit memory (8kB) was used for the configuration A, and a 512 word, 192-bit memory (12kB) for the configuration B. The program memories were connected directly to the processor core, i.e., cache was not used.

The processors were synthesized on a 130 nm CMOS standard cell low-power 1.5V technology using the Synopsys Design Compiler version 2003.06. A timing constraint of 200 MHz was used, although higher clock frequencies could have been achieved. Area statistics were obtained from the Design Compiler synthesis. The switching activities for the power analysis were obtained from the gate-level simulations run on ModelSim. Power consumption estimates were obtained from the Design Compiler by utilizing the gate-level switching activity information. Table 9 shows the average area and power consumption for the two processor configurations.

Table 9. *Area and power consumption of the reference designs. PMEM: Program memory. DMEM: Data memory.*

	Area [kgates]				Power consumption [mW]			
Conf.	PMEM	DMEM	Core	Total	PMEM	DMEM	Core	Total
A	32 ($512 \times 128bit$)	104 ($8192 \times 32bit$)	31	167	6	17	16	41
B	48 ($512 \times 192bit$)	410 ($8192 \times 32bit$)	80	538	13	26	30	71

6.2 Code Density

The code density estimations were carried out by utilizing the three compression methods on the program codes of the six benchmark applications that were each compiled on the three TTA processor configurations that were designed separately for each benchmark. The results of the code density evaluation were presented in terms of compression ratio. Subsection 6.2.1 presents the results for the dictionary-based compression. The results for the Huffman coding approach are given in Subsection 6.2.2, and for the instruction template-based compression in Subsection 6.2.3.

6.2.1 Dictionary-Based Compression

The code density estimations for the dictionary-based compression method were performed at the three different granularity levels that were introduced in Subsection 4.1.3, i.e., at full instruction level, at move slot level, and at ID field level. The code densities were reported in terms of compression ratio. The results for the three granularity levels are reported in the following three Subsections.

Instruction Level

At first, the dictionary-based compression was applied at the level of full instructions. The compiled programs were searched for unique instructions to be stored into a dictionary and replaced with indices that point to the dictionary. Figure 23 illustrates the obtained compression ratios as proportions of the compressed program code and the dictionary to where all the unique instructions were stored into. The results are given for the six benchmark applications, each compiled on the three processor configurations designed separately for each application. The compression ratios are expressed in percentage values.

An average compression ratio of 76.3% is achieved across all the benchmark applications compiled on the different processor configurations. The processor configuration does not affect the achievable compression ratio much. The size of the program code has been reduced significantly, but the size of the dictionary becomes large. This indicates that most of the instructions in the program code are stored into the dictionary, i.e., there are only few repeated instructions found in the program code.

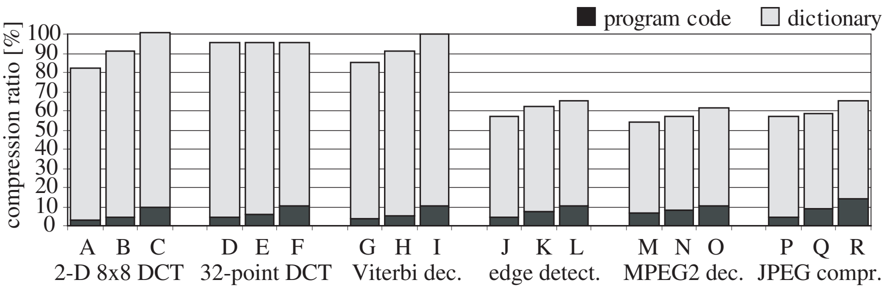


Fig. 23. *The results of applying dictionary-based compression at instruction level.*

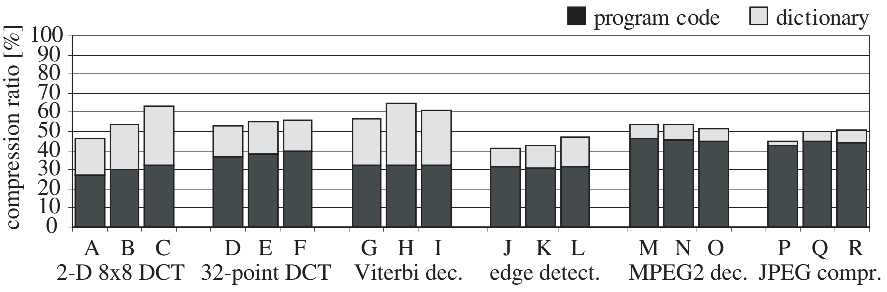
This is quite expected, as TTA instructions are fairly long and are composed of several independently encoded fields, i.e., move slots that are further composed of the guard, source ID, and destination ID fields. This results in low probability to find identical instructions from the program code from which the dictionary-based compression could benefit. The identical instructions that are found in the program code are mostly instructions that specify nothing but null data transports, i.e., are totally empty, or specify data transports in only few of the possible buses.

For the edge detection, MPEG2 decoding, and JPEG compression, the compression ratios are somewhat better. This results from the fact that the compiler could not effectively compile these applications, which resulted in large number of totally empty instructions that could be compressed effectively.

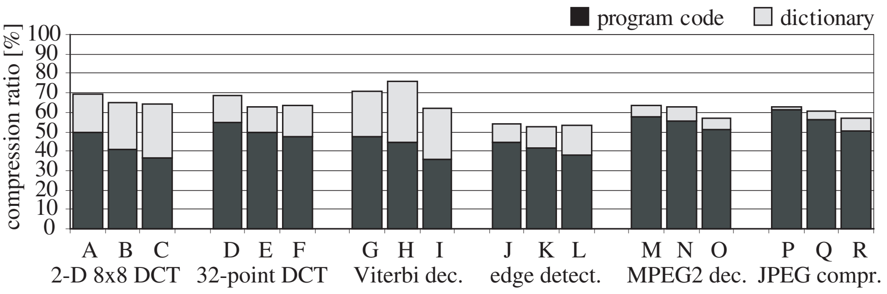
Move Slot Level

In the second phase the dictionary compression was evaluated at move slot granularity level where the instructions were divided into fields according to the move slot boundaries. Both the vertical and horizontal compression approaches were evaluated. Vertical compression considers all the move slots as parallel streams that are compressed separately whereas in horizontal compression all the move slots across all the instructions words are considered as a single stream. Long immediate fields in both cases are considered as a separate stream that is compressed separately.

Compression ratios for the compression at move slot granularity level are depicted if Fig. 24. Figure 24(a) shows the results for the vertical compression, and Fig. 24(b) for the horizontal compression. The vertical compression approach results in average



(a) Vertical compression



(b) Horizontal compression

Fig. 24. *The results of applying dictionary-based compression at move slot level.*

compression ratio of 52.5%, whereas the horizontal compression approach achieves an average compression ratio of 62.5%. The results show that the size of the dictionary has been reduced significantly. The reduced size of the dictionary implies that the division to smaller field inside which unique bit patterns are searched for allows to find more repeated bit patterns. This improves the effectiveness of the compression.

On the other hand, as the instructions are divided into several smaller fields, the compressed instruction word becomes wider than at full instruction granularity level. Compressed instruction words consists of several dictionary index fields that access either the parallel dictionaries in the vertical approach or the single dictionary in the horizontal approach. Similarly to compression at instruction level, the edge detection, MPEG2 decoding, and JPEG compression benchmarks result in smaller dictionaries as the original instructions contain a large number of null data transports, i.e., significant amount of redundancy that can be reduced through compression.

Vertical compression approach results in better compression ratio compared to the horizontal approach. The dictionaries are somewhat the same size, but the compressed program codes are bigger in horizontal compression. This is due to storing all the unique instructions into a single dictionary, which becomes large and results in wider dictionary index. The wide dictionary index increases the width of the compressed instruction word. In vertical compression, each parallel move slot stream has its own dictionary. This makes the dictionaries smaller as they need to cover unique move slots only for a single move slot stream but not across the entire program code. Smaller dictionaries results in smaller dictionary indices, which make the compressed instruction word narrower.

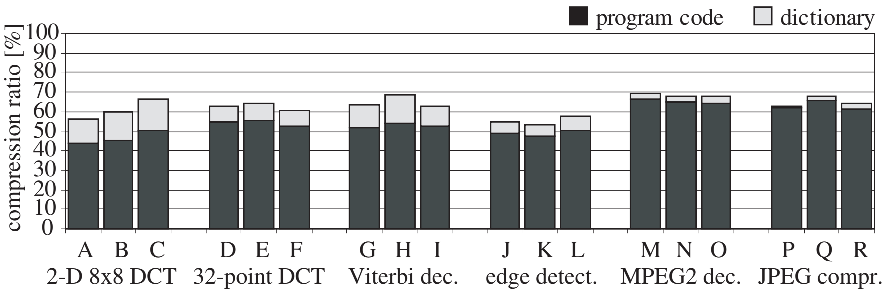
ID Field level

Compression at ID field granularity level divides instruction words to even smaller fields based on the guard, source ID, and destination ID field boundaries inside the move slots. Unique bit patterns to be stored into the dictionaries are searched inside these fields. Both the vertical and horizontal compression alternatives were utilized. Vertical compression considers all the source and destination ID fields as separate streams that are compressed independently. In the horizontal compression, all source ID fields as well as all destination ID fields are considered as a single stream. In both approaches, all the guard fields are combined into a single stream that is compressed independently, as also the long immediate field.

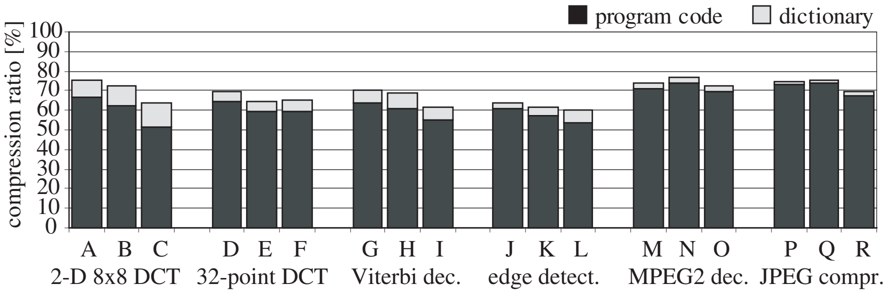
Figure 25 illustrates the results for the compression at ID field level. Figure 25(a) shows the results for the vertical and Fig. 25(b) for the horizontal compression approach. Average compression ratios of 62.8% and 68.9% are achieved, respectively. The dictionaries are even smaller than at move slot level. This is because more repeated bit patterns are found, implying that less entries need to be stored into the dictionaries. However, the size of the compressed program code is increased due to wide instructions as they are composed of several dictionary index fields.

6.2.2 Huffman Coding

The effectiveness of the Huffman coding approach in terms of code density was evaluated at the same three granularity levels as in the dictionary-based compression. Code densities, measured in terms of compression ratio, are reported for these three granularity levels in the following three Subsections.



(a) Vertical compression



(b) Horizontal compression

Fig. 25. *The results of applying dictionary-based compression at ID field level.*

Instruction Level

The Huffman coding was first evaluated at instruction level, i.e., entire instruction words were considered as symbols to be encoded to variable-width codewords. The obtained compression ratios are presented in Fig. 26. The compression ratios are presented as proportions of the compressed program code and the decoding table used for decompression. The decoding table contains the original symbols, i.e., the instruction words, the corresponding codewords, and their lengths. This information is required for the decompression of the Huffman coded codewords to obtain the original instruction words.

The achieved compression ratios are fairly poor, on average 81.9%. The compression ratios are especially poor for the first three DSP benchmark applications, 2-D 8×8 DCT, 32-point DCT, and Viterbi decoding applications, where the Huffman coding may actually result in larger code size compared to the uncompressed code size. Only the last three benchmarks, edge detection, MPEG2 decompression, and JPEG compression achieve feasible reduction in code size. The poor compression ratios for the

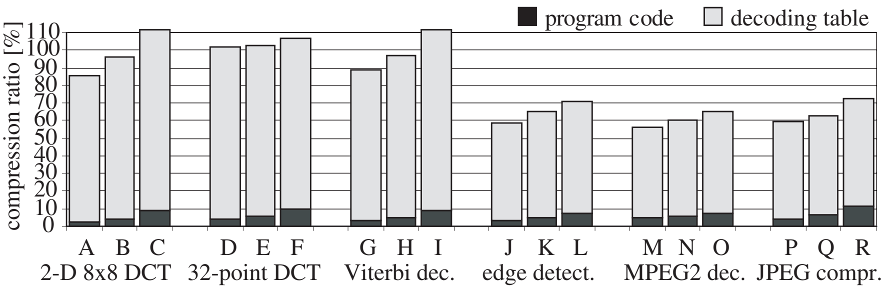


Fig. 26. *The results of applying Huffman coding at instruction level.*

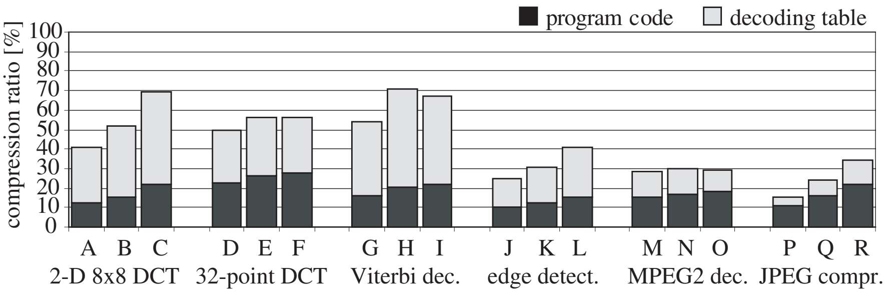
first three benchmarks are due to the large size of the decoding table. The decoding table becomes large as there are only few repeated instruction words in the program code, which results in storing most of the instruction words with their assigned codewords and their lengths to the decoding table. For the last three benchmarks there is a lot of redundancy in the original program code due to the inability of the compiler to utilize the parallel resources. This redundancy can be utilized during Huffman coding, resulting in smaller decoding table and more effective compression.

The achieved compression ratios are worse compared to the results of dictionary-based compression at the same granularity level. This indicates that the probability distribution is nearly uniform meaning that there is no profit in assigning variable-length codewords to the instruction words based on their frequency of occurrence. Assigning fixed-width codewords that can be used directly to access the dictionary that contains the unique instruction words is therefore more profitable.

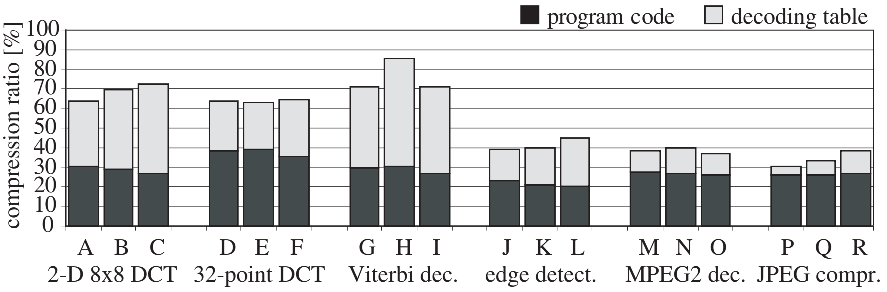
Move Slot Level

In the second granularity level, the instruction words were divided to smaller bit patterns based on the move slot boundaries to increase the probability of finding repeatedly existing bit patterns. This reduces the size of the decoding table. Both the vertical and horizontal compression approaches were experimented, similarly as in the dictionary-based compression.

The results for the Huffman coding at move slot level are presented in Fig. 27. Figure 27(a) depicts the results of the vertical compression and Fig. 27(b) the results of the horizontal compression. An average compression ratio of 43.1% is obtained for



(a) Vertical compression



(b) Horizontal compression

Fig. 27. *The results of applying Huffman coding at move slot level.*

the vertical compression and 53.7% for the horizontal compression approach. The results are significantly better compared to compression at instruction level. The results indicate that even though the size of the compressed program code is larger due to compressed instructions being wider as they are composed of several Huffman codewords, the size of the decoding table has been reduced significantly. This is due to considering smaller bit patterns as symbols for compression. This allows to find more repeated bit patterns and results in non-uniform usage probability, which makes the compression more effective.

Vertical compression results in better compression ratio compared to horizontal compression, similarly as in dictionary-based compression. The reason is the smaller size of the compressed program code in the vertical approach. The sizes of the decoding tables are more or less the same. The larger size of the compressed program code in the horizontal compression is due to performing Huffman coding for all the move slots across the program code. As there are more symbols for which codewords are to be assigned to, the codewords become on average wider as the depth of the Huffman tree is larger. In the vertical approach, each move slot stream is encoded separately, which results in smaller average width of the codewords. As the entire instructions are composed of as many codewords as there are move slots, instructions in the horizontal compression become wider and imply a larger code size.

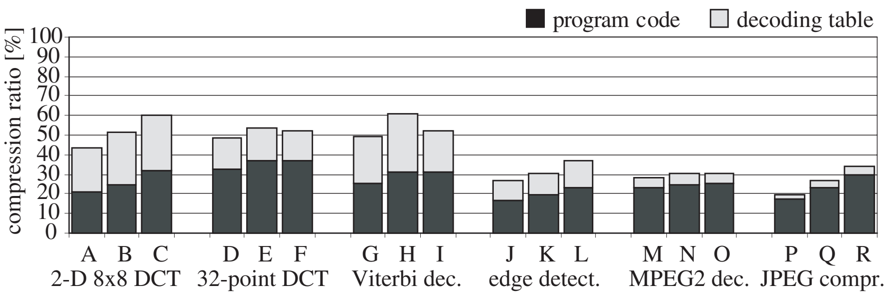
Compared to dictionary-based compression, the obtained results for the Huffman coding are notably better. Hence, the non-uniform probability distribution can be utilized effectively in Huffman coding to reduce the size of the program code even further compared to dictionary-based compression. The decoding tables become larger in Huffman coding than the dictionaries in the dictionary-based compression, but the reduction in the size of the program code is far greater, making the Huffman coding approach more effective at move slot level.

ID Field Level

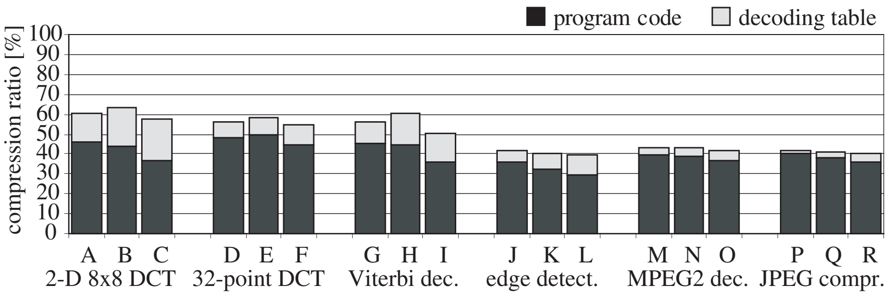
At the third granularity level, instruction words were divided to even smaller fields according to the guard, source ID, and destination ID field boundaries. This increases even further the probability of finding repeated bit patterns and allows to reduce the size of the decoding tables. Similarly to move slot level, both the vertical and horizontal compression approaches were utilized.

The obtained results are illustrated in Fig 28. Figure 28(a) shows the results for the vertical, and Fig. 28(b) for the horizontal compression. The compression ratios are on average 40.9% for the vertical and 49.4% for the horizontal compression approach. Compared to compression at move slot level, the achieved compression ratios are few percents better. The decoding tables are significantly smaller as more repeated bit patterns can be found inside the source and destination ID fields. This reduced the number of symbols that are stored into the decoding tables. The drawback of dividing instruction to even smaller fields is the increased size of the compressed program code as the compressed instructions become on average wider as they are composed of several codewords. However, the reduction in the dictionary size is greater than the increase in the program code size, which makes the compression at ID field level more effective.

Similarly to move slot level, vertical approach results in better compression ratios as there are more symbols to be assigned variable-width codewords in the horizontal approach. This increases the average width of the compressed codewords and



(a) Vertical compression



(b) Horizontal compression

Fig. 28. *The results of applying Huffman coding at ID field level.*

implies larger program code size. Even though the overall size of the decoding tables is smaller in horizontal compression, the program codes are significantly larger, resulting in worse compression ratio compared to the vertical compression approach.

At ID field level, Huffman coding improves the compression ratio even greater than at move slot level compared to the compression ratios achieved using the dictionary-based compression. As the bit patterns for the Huffman coding are smaller, the probability of finding repeatedly occurring bit patterns increases and hence improves the effectiveness of the Huffman coding. The overhead due to the larger decoding tables is small compared to the reduction in the size of the compressed program code.

6.2.3 Instruction Template-Based Compression

The effects of the instruction template-based compression on the code density were measured by varying the number of templates used to encode the instructions of the program code. The number of templates was varied according to powers of two from two to 32. In addition, the case of selecting as many templates as there are move slot and long immediate field combinations used in the program code was evaluated.

The achieved compression ratios are illustrated in Fig. 29. Compression ratios are presented for each of the benchmark applications, each compiled on the three TTA processor configurations designed specifically for the application. The results indicate that the code density can be significantly improved when the instruction templates are utilized. The program codes contain large number of null data transports that need to be explicitly specified, resulting in waste of code space. Instruction templates can be effectively utilized to avoid the null data transport specifications. The results show that the compression ratios are better with larger number of templates. This is due to the fact that the more templates are used, the more of the move slot and long immediate field combinations occurring in the program code can be covered with the templates. This allows to reduce the number of the explicit null data transport specifications. With the maximum number of templates, each move slot and long immediate field combination used in the program code has its own template, i.e., null data transport specifications can be avoided entirely. This results in an average compression ratio of 46.5%.

The results also show that the compression ratios are better on the larger processor configurations. This is quite obvious as these configurations have more buses. As the full parallelism cannot always be fully utilized, null data transports need to be specified on the buses. The more buses there are, the larger is the number of null data transports that need to be explicitly specified.

Figure 30 depicts the obtained relative program code sizes of the six benchmark applications on the different processor configurations. The presented program code sizes are presented separately for each application in relation to the largest uncompressed code size *(Ref)*, i.e., the uncompressed code size of the application on the largest processor configuration. In addition to presenting the results for the different number of templates, the lower bound on the achievable program code size, the effective code size *(eff. code)* is illustrated. This is the theoretical lower bound on the program code size when all the null data transports have been removed without any overhead, i.e., the template selection information.

As was shown by the compression ratios, the more templates are used, the better is the compression ratio, i.e., the smaller the size of the compressed program code, ap-

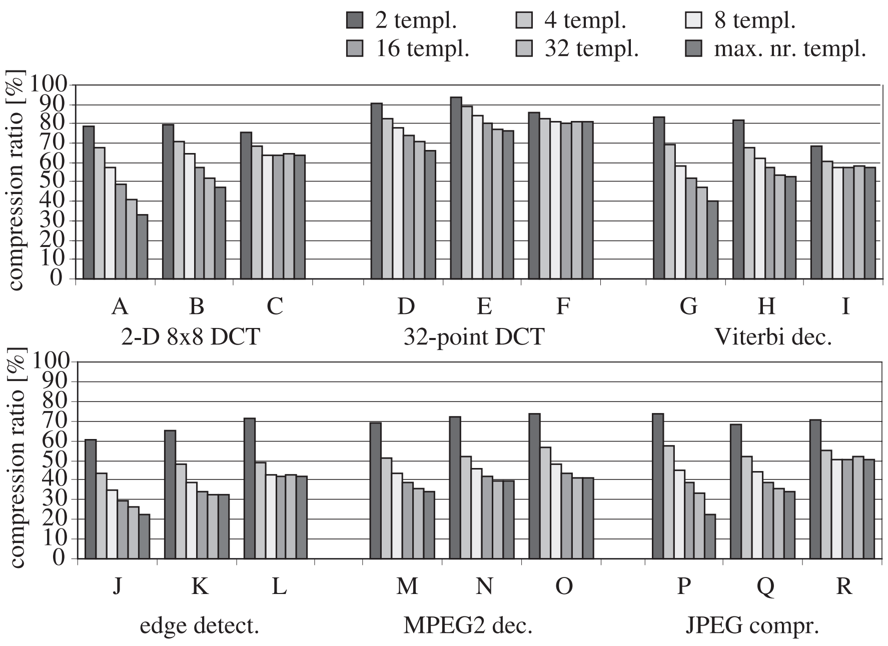


Fig. 29. *The results of applying instruction template-based compression.*

proaching the effective program code size that is the lower bound on the achievable code size. Obviously, this cannot be reached as the template-selection field assigned to each template introduces an overhead in the program code size. The relative program code sizes also illustrate that the more templates are used, the closer are the code sizes on the different processor configurations. With the maximum number of instruction templates, the resulting program code sizes are more or less the same on the different processor configurations. This means that by utilizing the instruction template-based compression, the effect of the processor configuration on the code size becomes negligible. This eases the selection of the processor configuration to be used for a given application. Large processor configurations become favorable as they turn out to have the same code size as the smaller processor configurations but can provide better performance.

The effectiveness of the instruction template-based compression depends also on the effectiveness of the scheduler to utilize the functional resources and the buses of the processor. The relative code sizes, presented in Fig. 30 shows worse results for the 32-point DCT application compared to the other applications. This was due to the scheduler being capable of effectively utilizing all the buses of the architecture and

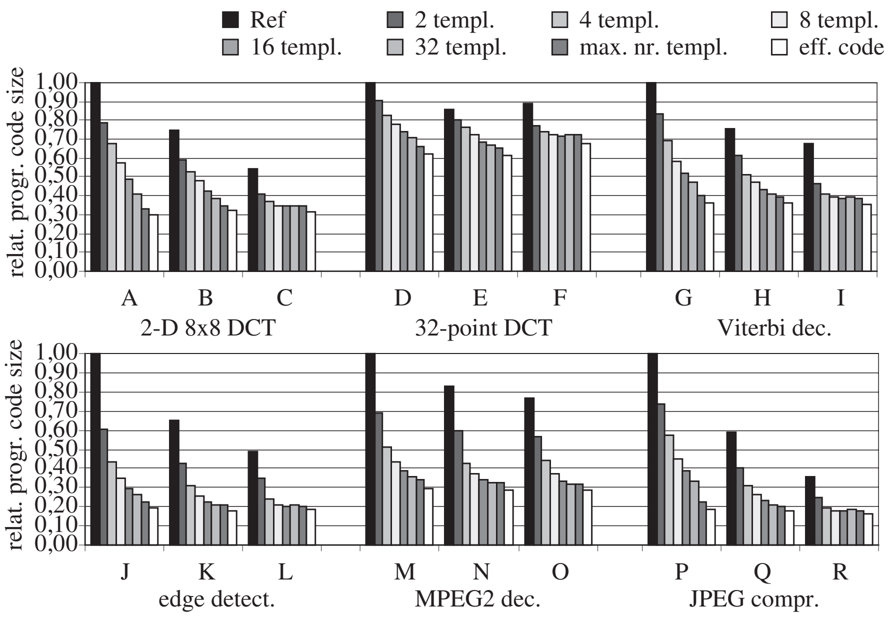


Fig. 30. *The relative code sizes of the benchmark applications when instruction template-based compression is utilized.*

result in small number of null data transports in the program code. This was due to writing the application totally unrolled, i.e., no loops were used. This resulted in larger basic blocks and better possibilities for the scheduler to utilize the parallelism. The opposite examples are the edge detection, MPEG2 decoding, and JPEG compression applications, which the scheduler could not schedule effectively. This resulted in large amount of instructions containing nothing but null data transports. On account of this, the code size reductions were better for these three applications.

6.3 Area and Power Consumption

The hardware implementations were made for the dictionary-based and instruction template-based compression methods to obtain more accurate statistics in terms of area and power consumption to evaluate more accurately all the aspects of the compression when applied on TTA processors. The Huffman coding approach was not implemented as it resembles the instruction template-based compression in the sense that it also results in variable-width instructions that turned out to increase the complexity of the instruction fetch and decompression logic significantly. This results in poor area and power consumption, as demonstrated later in this Section. The hardware implementations were made for the two processor configurations that were designed for the four DSP benchmark applications. The results for the dictionary-based compression are presented in Subsection 6.3.1, and for the instruction template-based compression in Subsection 6.3.2.

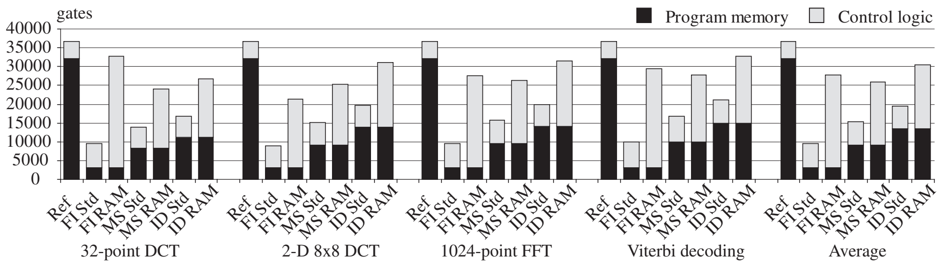
6.3.1 Dictionary-Based Compression

As discussed in Chapter 4, the decompressor for the dictionary-based compression can be implemented in several ways. The decompressor can be implemented in a separate pipeline stage, or alternatively, integrated together with the decoder into a single pipeline stage. Furthermore, the dictionary in the decompressor can be implemented using either RAM to maintain the programmability, or using standard cell logic to achieve higher reduction in area and power consumption. The results of these implementation alternatives are presented in the following.

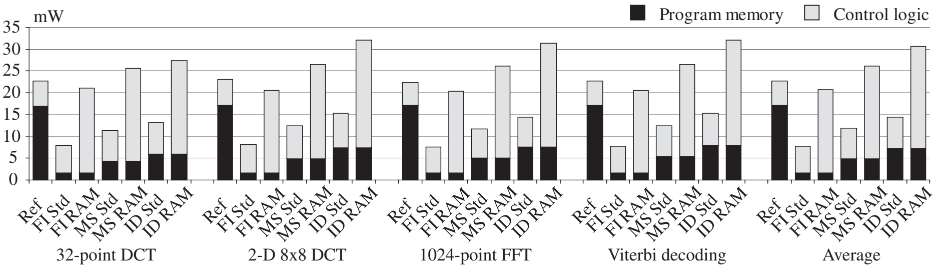
Decompression in an additional pipeline stage

Implementing the decompressor in an additional pipeline stage allows to use either standard cell logic or RAM to implement the dictionary as an entire clock cycle is devoted for decompression. The results of these two dictionary implementation alternatives are illustrated in Fig. 31 for the configuration A and in Fig. 32 for the configuration B. Figures 31(a) and 32(a) illustrate the area and Figs. 31(b) and 32(b) the power consumption.

The results are given for the uncompressed reference implementation *(ref)*, and for the compressed implementation alternatives with the dictionary implemented using either standard cells *(Std)* or RAM *(RAM)*. Results for the compressed implementations are given for the compression at full instruction level *(FI)*, at move slot level *(MS)*, and at ID field level *(ID)*. The area and power consumption are presented for the program memory and the control logic of the processor core, as these are the only parts affected by the compression. The results are given for all the four benchmark applications. In addition, an average over all the benchmarks is given.



(a) Area

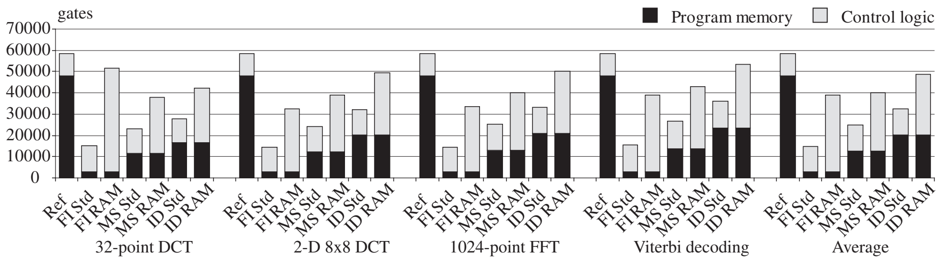


(b) Power consumption

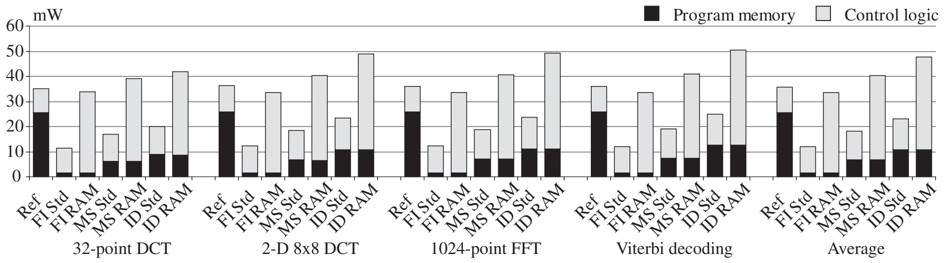
Fig. 31. *Area and power consumption results on configuration A.*

The results indicate that the area and power consumption of the program memory and the control logic can be decreased significantly at all granularity levels when standard cells are used to implement the dictionary. The best reduction in both area and power consumption is obtained when the compression is made at instruction level. The area is reduced on average 74% on configuration A and 75% on configuration B. The power consumption is reduced on average 66% on both configurations.

The reason for obtaining the best area reduction at instruction level is due to the small size of the program memory and the control logic. Replacing instructions with dictionary indices reduces the width, and therefore, the size of the program memory significantly. Even though the dictionary contains most of the instruction as only few repeated instructions are found, which was demonstrated by the code density evaluations in Section 6.2, the synthesis tool can find the redundancy from the bit patterns in the dictionary and utilize Boolean logic optimization to minimize the logic and result in small area. Furthermore, the area of the instruction fetch stage in the control logic is reduced significantly due to the reduced width of the program memory, as the width of the instruction fetch datapath is reduced. Overall, the area of the control logic is not affected that significantly even though the decompressor is added to



(a) Area



(b) Power consumption

Fig. 32. *Area and power consumption results on configuration B.*

the control logic. Power consumption follows the same trend. The power consumption of the control logic remains mostly unchanged but the power consumption of the program memory decreases significantly. This is due to the reduced width of the program memory and the fact that the power consumption of the program memory is mostly dependent on its width rather than its length [106].

The drawback of applying the dictionary-based program compression at instruction level and implementing the dictionary using standard cells is the highly limited programmability of the architecture. The program can be modified only if all the instructions of the modified code can be found from the original dictionaries. As TTA instructions are long and they are composed of several smaller fields, the number of possible bit pattern combinations in the instruction words becomes huge. Therefore, the probability that all the instructions of the modified code could be found from the original dictionaries is extremely small. As the programmability is lost anyway, one could claim that better reduction in area could be obtained by simple implementing the original program memory using standard cells.

Programmability can be maintained better at smaller granularity levels. As the bit patterns considered for compression are smaller, the probability of finding the modified bit patterns from the original dictionaries is better. The more fine-grained the granularity level, the better the programmability. Thus, from the experimented granularity levels, the programmability is maintained best at ID field level where the instructions are broken up to fields according to ID field boundaries.

However, applying dictionary-based compression at smaller granularity levels results in worse area and power consumption reduction. The worse results are due to the increased area of the program memory. The original instruction words are divided to smaller fields that are compressed separately, meaning that the compressed instruction word consists of several dictionary index fields. This increases the width of the compressed instruction word. The area of the control logic remains mostly unchanged in between different granularity levels. At move slot level, the area is reduced on average 58% on configuration A and 57% on configuration B. The power consumption is reduced on average 47% and 49%, respectively. At ID field, the results are slightly worse. The area is reduced on average 47% and 45%, respectively, and power consumption 36% on both configurations.

Programmability can be fully maintained when the dictionary is implemented using RAM as the contents of the dictionary can be reprogrammed when the program code changes. However, using RAM to implement the dictionary results in fairly poor area and power consumption reduction, as is illustrated in Fig. 31 and Fig. 32. At instruction level, the area is reduced on average 25% on configuration A and 33% on configuration B. The power consumption is reduced 9% and 7%, respectively. At move slot level the area can be reduced somewhat better, 30% on configuration A and 31% on configuration B. However, at move slot level the power consumption turns out to increase rather than decrease. The power consumption of the program memory and the control logic increases on average 15% on configuration A and 12% on configuration B.

The increase in the power consumption results from using several independently accessed RAM dictionaries inside the processor core. At move slot granularity level there is a RAM dictionary for each of the move slots and an additional one for the long immediate field. As each RAM dictionary is accessed separately, i.e., each RAM has its own address lines, there are several address lines that control the loads from the RAM dictionaries. This increases the power consumption. The increase in the power consumption turns out to be even worse when the compression is applied at ID field level, as there are even more independently accessed RAM dictionaries inside the processor core. The obtained area and power consumption results in the program memory and the control logic at the three distinct granularity levels for the separated decompressor implementation approach are summarized in Table 10. Reduction in area and power consumption is identified with a negative value, increase with a positive value.

Devoting an entire transport pipeline stage for the decompression has also a consequence on the performance. Due to the additional decompression stage, the depth of the transport pipeline is increased by one. This implies an increase in the branch delay as it takes one more clock cycle before the instruction of the branch target reaches the move stage. The increase in cycle count due to increased branch delay depends on the number of taken branches. The increased branch delay penalty is paid only when the program execution does not continue at the next instruction, i.e., the execution of the program jumps to an address that is loaded to the program counter.

Table 11 shows the increase in the cycle count of the four benchmark applications on the two processor configurations when the branch delay is increased due to implementing the decompressor in a separate pipeline stage. For the 32-point DCT application the increase in the cycle count on both configurations is less than 1%. An opposite example is the Viterbi decoding application, where the cycle count is increased 15.3% on configuration A and 7.6% on configuration B due to several taken branches. The 32-point DCT application was written totally unrolled, i.e., the loops in the application were opened and each iteration was programmed separately. This decreased the number of branches, and therefore, resulted in only small increase in the cycle count. Hence, the impact of the increased branch delay on the performance of the other applications could be reduced through loop unrolling. On the other hand, loop unrolling would increase the size of the original uncompressed code, but this overhead could again be decreased through program compression.

Table 10. *Summary of the area and power consumption evaluations with a separate pipeline stage for the decompressor.*

Compression granularity	RAM solution		Standard cell solution	
	Area	Power cons.	Area	Power cons.
Instruction level	-25 / -33%	-9 / -6%	-74 / -75%	-66 / -66%
Move slot level	-30 / -31%	+15 / +12%	-58 / -57%	-47 / - 49%
ID field level	-17 / -16%	+35 / +33/%	-47 / -45%	-36 / -36%

Table 11. *Increase in cycle count due to increased branch latency.*

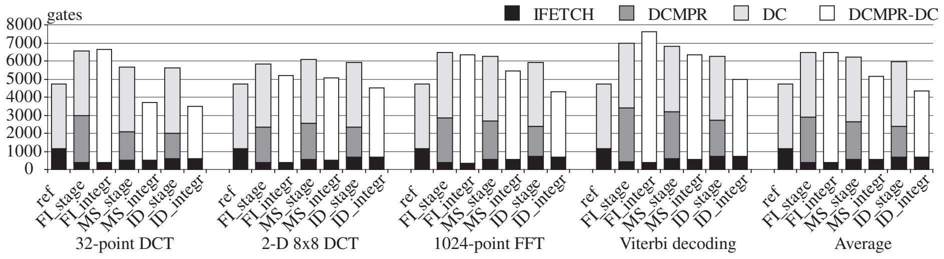
Benchmark	Increase on conf. A	Increase on conf. B
32-point DCT	0.8%	0.5%
2-D 8×8 DCT	5.8%	1.4%
1024-point FFT	7.4%	6.3%
Viterbi decoding	15.3%	7.6%
Average	7.3%	4.0%

Integrated decompressor

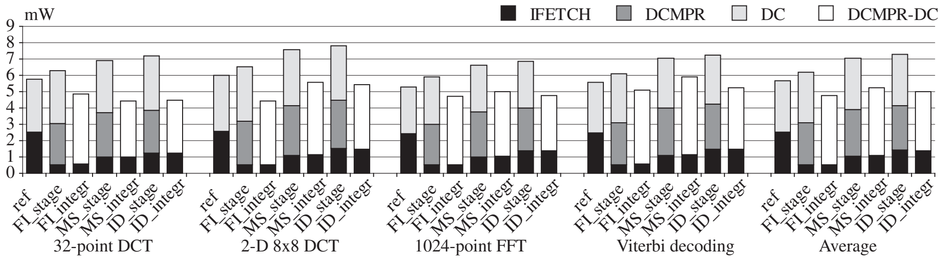
The decompressor can also be integrated to the decode stage to avoid increasing the depth of the pipeline that would lead to increased branch delay and decreased performance. The integration of the decompressor requires that the dictionary is implemented using standard cells as the contents of RAM cannot be obtained during the same clock cycle when the address is given.

The integration of the decompressor affects also the area and power consumption of the control logic. The program memory is not affected. Figures 33 and 34 illustrate the area and power consumption of the control logic of the two processor configurations, A and B, for the uncompressed reference implementation *(ref)*, for the implementation with separate decompress stage *(stage)*, and for the integrated decompressor *(integr)*. The results are presented for the three evaluated granularity levels. Figures 33(a) and 34(a) illustrate the area, and Figures 33(b) and 34(b) the power consumption. The area and power consumption for the control logic are presented in terms of their components, i.e., the logic that corresponds to the stages of the TTA transport pipeline. In the uncompressed reference implementation there are two pipeline stages in the control logic; instruction fetch *(IF)* and decode *(DC)* stages. In the separated decompressor implementation an additional pipeline stage is added for the decompression *(DCMPR)* in between the IF and DC stages. In the integrated decompressor implementation, the DCMPR and DC stages are integrated into a single decompress-decode *(DCMPR-DC)* stage.

The results show that the integrated decompressor results in smaller area of the control logic at all granularity levels. The more fine-grained the granularity, the greater the difference between the two decompressor implementation alternatives. At full instruction level the area of the control logic of the integrated decompressor is on average equal to the area of the implementation with an additional decompress stage, but 14% smaller on configuration B. At move slot level, the area of the integrated de-



(a) Area



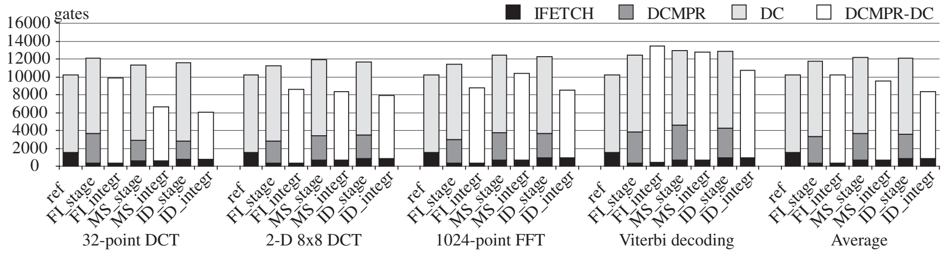
(b) Power consumption

Fig. 33. *Area and power consumption of the control logic on configuration A with different transport pipeline organizations.*

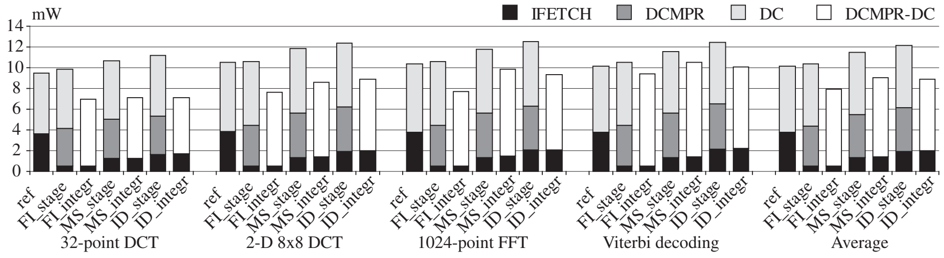
compressor is on average 17% smaller on configuration A, and 22% on configuration B. At ID field level, the area is on average 27% and 31% smaller, respectively.

The area of the instruction fetch logic is equal in both decompressor implementation alternatives as the decompressor does not influence the implementation of the instruction fetch logic. However, implementing the decompressor in an additional pipeline stage implies a need for an instruction-wide register in between the decompressor and the decoder to store the decompressed instruction before the decode stage. Integrating the decompressor and decoder into a single pipeline stage avoids the need for such a register. In addition, the integrated decompressor provides better possibilities for the synthesis tools to minimize the logic as the module to be synthesized into logic is larger, allowing better possibilities for Boolean logic optimization.

The power consumption of the integrated decompressor implementation is significantly smaller at all granularity levels compared to the implementation with a separate DCMPR stage. At instruction level the power consumption of the control logic of the integrated decompressor is on average 23% smaller on configuration A, and 24% on configuration B. At move slot level it is on average 26% smaller on configuration



(a) Area



(b) Power consumption

Fig. 34. *Area and power consumption of the control logic on configuration B with different transport pipeline organizations.*

A, and 21% smaller on configuration B, and at D field level on average 31% and 27% smaller, respectively. For both decompressor implementation alternatives, the input and output functions are exactly the same, i.e., the same bit vectors (dictionary indices) are sent to the decompress and decode logic and the same bit vectors (control signals) are obtained as output. Therefore, the difference in power consumption is due to the smaller logic of the integrated decompressor implementation.

The results also indicate that with the integrated decompressor the overhead of the decompression logic in the area and power consumption can be effectively minimized. With the integrated decompressor, the power consumption of the control logic becomes actually smaller than that of the uncompressed implementation at all granularity levels. Also the area of the control logic becomes smaller when the granularity level is small enough; on configuration A this happens only at ID field level, but on configuration B already at full instruction level. Smaller area and power consumption are due to the reduced width of the program memory datapath and the instruction register in the instruction fetch logic. The decompressor introduces an overhead to the area of the control logic but this overhead is smaller than the area reduction in the

IF stage. Integration of the decompressor and decoder into a single module allows also more efficient logic optimization, reducing the area even further.

The overall area and power consumption reductions, including the program memory and the control logic, at the three distinct granularity levels for the integrated decompressor implementation are summarized in Table 12. Compared to the results of the separated decompressor implementation, presented in Table 10, the integrated decompressor results on average 4% better reduction in area and power consumption.

The timing of the processor is not affected when the decompressor is implemented in a separate pipeline stage because an entire clock cycle is devoted for the decompression. However, the timing is affected when the decompressor is integrated to the decode stage as both the decompression and decoding have to be performed during a single clock cycle. This lengthens the critical path and, therefore, limits the achievable clock frequency. The effect of the decompressor implementation on the timing of the processor was evaluated by trying out different clock period times in the synthesis of the three alternative processor implementations, i.e., the uncompressed implementation and the separated and integrated decompressor implementations, and to find the smallest clock period time with which the timing can still be met.

In the processor configuration A, the uncompressed and separated decompressor implementations achieved a maximum clock frequency of 267MHz. The critical path went through the interconnection network and the squashing and decoding logic in the control unit to the control signal registers. Integrating the decompressors to the decode stage limited the achievable clock frequency to 250MHz. The critical path went from the instruction register through the decompression and decode logic to the control signal registers, as expected. In the processor configuration B, which has more functional resources, a maximum clock frequency of 222MHz was obtained for the uncompressed implementation. Clock frequencies in between 200MHz and

Table 12. *Summary of the area and power consumption evaluations with the integrated decompressor implementation.*

Compression granularity	Standard cell implementation	
	Area	Power cons.
Instruction level	-73 / -77%	-72 / -73%
Move slot level	-62 / -62%	-56 / -56%
ID field level	-52 / -51%	-46 / -45%

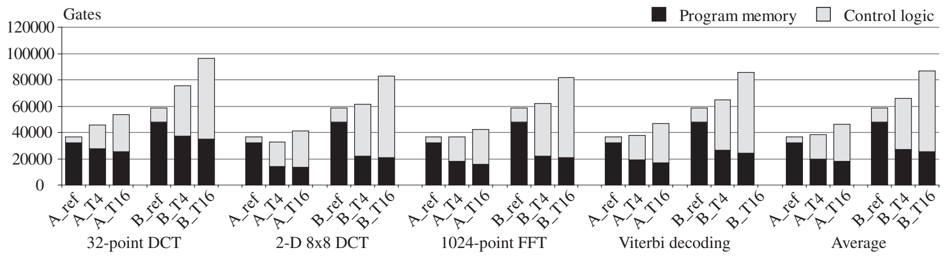
222MHz, depending on the benchmark application, were obtained for the two compressed implementations. The critical path in all the implementation alternatives went through the interconnection network and the squashing logic and decoding logic in the control unit to the control signal register. Hence, in processor configuration B, the integration of the decompressor to the decode stage did not affect the achievable clock frequency. The effect of the integrated decompressor on the achievable clock frequency on processor configuration A was also fairly small.

6.3.2 Instruction Template-Based Compression

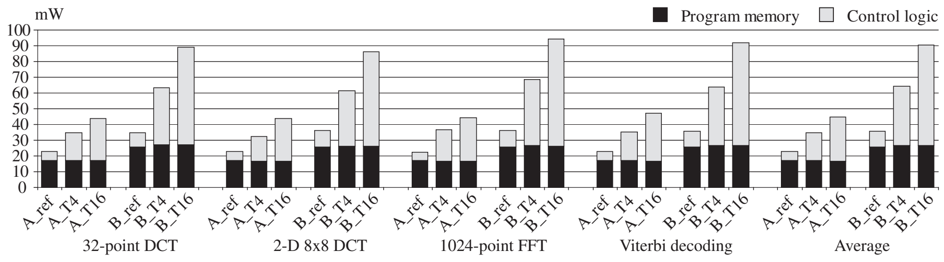
The effects of the instruction template-based compression on area and power consumption were evaluated by using four and 16 instruction templates. Figure 35 illustrates the results of applying the instruction template-based compression on the two processor configurations, A and B. Figure 35(a) illustrates the area, and Fig. 35(b) the power consumption of the program memory and the control logic on the two evaluated processor configurations. The results are given for the uncompressed reference implementation *(ref)*, and for the instruction template-based compression implementation using either four *(T4)*, or 16 *(T16)* templates to encode the original instructions.

The obtained results indicate a significant increase in both the area and power consumption when the instruction template-based compression is applied. The results also show that the more templates are used, the larger is the increase in both the area and power consumption. The total area of the program memory and the control logic increases on average 4% on configuration A and 25% on configuration B, when four templates are used. With 16 templates, the area increases 13% and 48%, respectively. The impact on the power consumption is even more dramatic. With four templates the total power consumption of the program memory and the control logic increases 53% and 97%, and with 16 templates 80% and 152%, respectively.

The results in area, depicted in Fig. 35(a), demonstrate that the area of the program memory can be reduced by applying the instruction template-based compression, but the decompression logic introduces such a large overhead that the total area turns out to be larger than in the uncompressed reference implementation. The decompressor has to handle variable-width instructions, which result in need for a shift register and a pre-fetch buffer in the control logic. Large shifting and alignment networks are also needed in order to construct the original uncompressed instruction words from



(a) Area



(b) Power consumption

Fig. 35. *Area and power consumption of the program memory and the control logic when the instruction template-based compression is applied.*

the move slots that may be in several different locations inside the templates. Large multiplexers are also required to place the remaining bits from the instruction fetch packet to the shift register from where the bits are interpreted in the decompressor during the next clock cycle. Furthermore, as the program memory is adjusted to the width of the widest compressed instruction words that are wider than the original instruction words, the width of the instruction datapath has to be increased in the instruction fetch logic. All this additional logic implies a large increase in the area of the control logic. The more templates are used, the larger the increase in area.

The power consumption results in Fig 35(b) demonstrate that the power consumption of the program memory and the control logic turns out to increase when instruction templates are applied. This is due to the increased width of the program memory as it has to be adjusted to the width of the widest compressed instruction word. Wider program memories consume more power as the power consumption of a memory is more dependent on its width rather than its length [106]. Hence, even though the size of the program memory is reduced, the power consumption of the program memory is at the same level than that of the program memory used for the uncompressed reference implementation. As the power consumption of the control logic also becomes significantly greater than in the uncompressed implementation, the total power consumption turns out to be greater than in the reference implementation.

Hence, the results demonstrate that the instruction template-based compression approach turns out to be impractical even though it resulted in high compression ratios. The compression ratio does not consider the decompression circuitry nor the implementation details of the program memory. Therefore, this proves that the compression ratio is not accurate enough to fully characterize the compression methods and evaluate their effectiveness.

6.4 Programmability Evaluation

The proposed dictionary extension method to maintain the programmability was evaluated by utilizing it for the dictionary-based compression approach that was applied on two TTA processor configurations that were designed for the hardware implementations. The proposed method was applied for the compression at instruction level as that level turned out to be the most effective granularity level to decrease the area and power consumption, but on the other hand, resulted in the most limited programmability. The extension method was evaluated by measuring its effect on area and power consumption. The effects on the performance were also estimated.

Subsection 6.4.1 describes the details of designing the extended dictionaries for the two processor configurations based on the methodology proposed in Chapter 5. The results for the performed area and power consumption estimations are presented Subsection 6.4.2. Finally, the estimates of the proposed extension method on the performance are discussed in Subsection 6.4.3.

6.4.1 Design of the Extended Dictionary

The dictionary extension requires to identify all the possible sources and destinations in the processor architecture. The required data transports to maintain the programmability can be identified based on this information. Table 13 tabulates for the two processor configurations the number of sources and destinations, the number of data transports required to maintain the programmability, and the number of the

Table 13. *Statistics of the number of data transports required to maintain the programmability on the two TTA processor configurations. Src: source. Dst: destination.*

Conf.	Nr. src	Nr. dst	Number of required data transports			
			Src − > Dst		Src − > GCR and GCR − > Dst	
			Required	Supported	Required	Supported
A	29	58	1682	1579	87	87
B	66	113	7458	7337	179	179

required data transports that can be executed in the interconnection network of the given processor configuration, i.e., the data transports that have the required connections to the buses. The table illustrates also the number of the required and supported data transports in case the GCR is utilized to minimize the number of the required data transports by splitting the required data transports into two parts; from all the sources to the GCR and from the GCR to all the destinations.

The numbers in Table 13 illustrate that even though the configuration A is a fairly small, containing only the minimal set of resources to execute all the four applications, the number of sources and destinations in the architecture is fairly large. This implies a large number of data transports that are required to maintain the programmability. For the configuration B, which has more hardware resources than configuration A, i.e., also more sources and destinations, the number of the required data transports in even larger. Extending the dictionaries with all these data transports would results in large overhead in the area of the control logic. The table also illustrates that the interconnection network does not support all of the required data transports, i.e., there are not enough connections in the input and output sockets to the move buses to execute all of the required data transports. The missing connections need to be added to the corresponding sockets to support the execution of all of the required data transports. This increases the area even further.

Therefore, it is better to utilize the GCR to minimize the number of the required data transports. As the table shows, utilization of the GCR reduces significantly the number of the required data transports that need to be supported. In addition, all of the required data transports are already supported in the example processor configurations. This avoids the need for any additional connections in the interconnection network. In the evaluations of the proposed programmability method, the GCR was implemented in one of the existing register files, following the selection procedure based on measuring the connectivity of the register files, as discussed in Chapter 5.

To utilize the GCR to minimize the number of the additional dictionary entries, the data transports from all the sources to the GCR and from the GCR to all the destinations need to be identified and assigned a bus on which the data transport is to be executed. As the dictionary-based compression at instruction level limits the performance of the additional entries to one transport per instruction, the data transports can be allocated freely on the buses as long as both the input and output sockets involved in the transport have a connection on the same bus.

Next, the data transports need to be added to the dictionary. As the dictionary-based compression has been performed at instruction level, the dictionary extension involves adding an entire instruction word for each data transport. The data transport is allocated on the move slot that corresponds to the bus on which the transport is to be executed. Null data transports are allocated on all the other buses in that specific dictionary entry. Table 14 tabulates the sizes of the original and extended dictionaries for the four benchmark applications on the two evaluated processor configurations in terms of bytes. The dictionary sizes illustrate that the extension may imply a large overhead. The overhead is at most 58.8% on configuration A, but up to 161.3% on configuration B. Hence, the larger the processor configuration, the greater the overhead in the dictionary size. This is quite obvious as there are more sources and destination on the larger processor configuration, i.e., also more additional entries to be added. The overhead depends also quite heavily on the used application, or the size of the original dictionary as the number of entries added to the dictionary is the same for each application. The 2-D 8×8 DCT application has the smallest original dictionary, i.e., also the largest overhead.

In addition to extending the dictionaries, the full programmability requires that all possible immediate values can be used in the program code. This can be accomplished by specifying the immediate bits along the compressed instructions. Immediate specifiers are used to identify the presence of the immediate values in the program code. A set of immediate formats are designed to avoid always specifying 32 bit wide immediates. Each format has its own immediate specifier that identifies how many of the following memory lines contain the immediate bits.

For the two processor configurations used in the evaluation, the width of the compressed instruction words was 9 bits on configuration A, and 10 bits on configuration B. This resulted in four different immediate formats; 9, 18, 27, and 36 bit formats on configuration A and 10, 20, 30, and 40 bit formats on configuration B. Each format

Table 14. *Statistics of the original and extended dictionaries for the four benchmarks.*

Benchmark	Conf.	Original Dict. [bytes]	Extended Dict. [bytes]	Overhead [%]
32-point DCT	A	6620	8001	20.9
	B	9168	13464	46.6
2-D 8×8 DCT	A	2350	3731	58.8
	B	2664	6960	161.3
1024-point FFT	A	4159	5540	32.3
	B	2928	7224	146.7
Viterbi decoding	A	4985	6366	27.7
	B	4944	9240	86.9

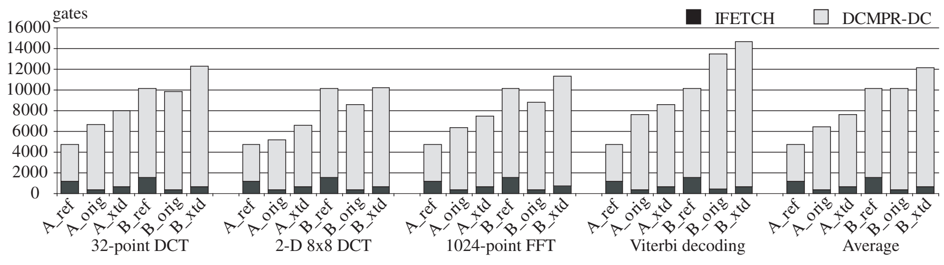
was assigned a unique immediate specifier, 9 bits wide on configuration A and 10 bits wide on configuration B. The instruction fetch logic inspects each fetched packet for the existence of an immediate specifier. Once an identifier is found, the instruction pipeline is stalled until all the immediate bits have been fetched from the program memory. The immediate value is then assembled and stored into the immediate register, from where the value is implicitly transported to the GCR.

6.4.2 Area and Power Consumption

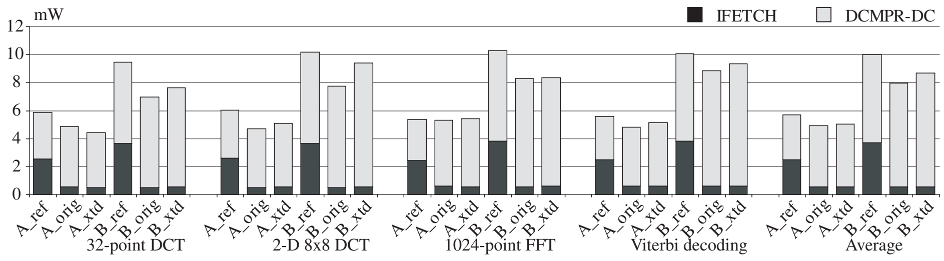
Maintaining programmability with the proposed methodology affects the area and power consumption of the control logic as the dictionary becomes larger and additional logic is added to the instruction fetch stage. The effects on area and power consumption of the control logic are depicted in Fig. 36 for the two evaluated processor configurations. For the control logic, the integrated decompressor implementation was utilized, i.e., the decompressor and decoder were integrated together into a single pipeline stage. Figure 36(a) illustrates the area and Fig. 36(b) the power consumption.

The results are shown for the original implementation *(orig)* where the compression is performed at instruction level, and for the implementation where the dictionary extension is used to maintain the programmability after compression *(extd)*. For reference, the results are also illustrated for the uncompressed implementation *(ref)*. The results are given for the two stages of the control logic: the instruction fetch *(IF)* and the integrated decompress-decode *(DCMPR-DC)* stages. In the uncompressed implementation, the results for the DCMPR-DC stage include only the decoder.

The results in Fig. 36 demonstrate that the dictionary extension and the immediate support logic result in fairly small overhead in the area and power consumption of



(a) Area



(b) Power consumption

Fig. 36. *The effects of the dictionary extension and immediate support on area and power consumption of the control logic.*

the control logic. The area of the control logic increases on average 18.9% on configuration A and 19.3% on configuration B compared to the original compressed case. The increase in the area of the control logic is mostly due to extending the dictionary with additional instructions. The effect of adding the immediate support logic on the area of the instruction fetch stage is negligible. Compared to the overhead of the dictionary extension, measured in bytes and illustrated in Table 14, the obtained results show that the synthesis tool can effectively minimize the logic of the dictionary and reduce the overhead of the additional dictionary entries.

The power consumption of the control logic is also increased due to the additional logic. The power consumption increases on average 8.1% on configuration A, and 9.8% on configuration B, compared to the original compressed implementation. This increase is also fairly small. Compared to the results of the uncompressed implementation, the power consumption of the control logic still remains smaller even though the dictionary has been extended.

The results presented in Fig. 36 are shown only for the control logic, which is the only part that is affected by the extension method. In case the program memory is included, the overhead of the dictionary extension on the total area and power consumption is only few percents. Even though the dictionary has been extended, the total area, including the control logic and the program memory, can still be reduced on average 71% on configuration A and 74% on configuration B. The power consumption can be reduced 70% and 71%, respectively.

6.4.3 Performance

The utilization of the extended dictionary entries affects also the performance. In case the GCR is utilized, a single entry can execute only a half of the data transport from a source to a destination. Therefore, it takes two clock cycles to complete a single source-to-destination data transport. The original TTA instructions can perform as many data transports in parallel as there are move slots. Hence, the effect on the performance depends on the number, the usage frequencies, and the level of the utilized parallelism in the instructions that need to be programmed with the extended entries. The performance is also affected by the number and width of the immediates used in the modified instructions as the datapath has to be stalled until all the immediate bits assigned for a data transport have been fetched from the program memory.

The effects of the dictionary extension on the performance were estimated by programming the instructions in the original program code with the extended entries and using the immediate support method to define the immediate values used in the instructions. To ease the estimation, the original instructions were modified to be programmed with the extended dictionary entries one basic block at a time. Table 15 presents for each benchmark application the number of basic blocks, the average number of instructions per basic block, the average number of data transports executed per clock cycle, which defines the utilized ILP, and the relative execution time when the basic blocks are programmed using the extended entries. The effect of modifying a single basic block on the performance is presented as an average and a maximum effect. The table illustrates also the worst case execution time when all the basic blocks are modified to be programmed using the extended entries.

The relative code sizes show that by modifying only a single basic block to be programmed using the extended entries, the execution time may be affected dramatically. On average, the execution time increases 61% on configuration A and 91% on configuration B when a single basic block is programmed entirely using the extended

Table 15. *The effects on the performance of programming basic blocks (BB) using the extended entries.*

Application	Conf.	#BBs	#instr. / BB	moves/ cycle	BB eff. on ex. time		Worst case ex. time
					avg.	max.	
32-point DCT	A	7	69.1	2.94	2.18	2.48	8.24
	B	7	63.0	3.21	2.46	2.84	10.23
2-D 8×8 DCT	A	18	9.1	2.92	1.57	5.84	10.21
	B	17	8.1	3.40	1.72	6.87	12.25
1024-point FFT	A	13	24.2	2.03	1.50	5.31	6.51
	B	12	12.4	2.91	1.85	7.85	10.21
Viterbi decoding	A	39	9.4	2.59	1.18	2.34	7.14
	B	36	7.0	3.73	1.26	3.17	9.23
Average	A	19.25	28.0	2.63	1.61	4.00	8.03
	B	18	22.6	3.04	1.91	5.39	10.48

entries. The penalty in performance depends on the utilized ILP, i.e., how many data transports are executed in parallel in the instructions that need to be programmed using the extended entries. In general, the higher the utilization of the ILP, the larger the increase in the execution time when the extended entries are used to program instructions. Large processor configurations have more functional resources, which provides means to utilize the ILP better and execute several data transports per instruction. The penalty in performance on such processor configurations is larger when the extended entries are used, as can be seen from the obtained results.

In addition to the utilized ILP, the performance is also affected by the size and the usage frequency of the basic block that has been modified. Large and frequently executed basic blocks may lead to a large increase in the execution time, as shown by the maximum effects on the execution time in Table 15. The execution time may become on average quadruple on configuration A and more than fivefold on configuration B when a single basic block is programmed using the extended dictionary entries.

The results in Table 15 show that depending on the application, the effect on the performance may vary significantly. The 32-point DCT application has only few but large basic blocks that are each executed only once. This leads to almost the same average and maximum effect on the performance. On the other hand, the 2-D 8×8 DCT and 1024-point FFT applications have a modest average effect on the performance, but as a maximum effect, a single basic block may result in dramatic increase in the execution time. This is due to both applications having a kernel basic block that is executed several thousand times, leading to large impact on the performance. In Viterbi decoding, the number of basic blocks is large but the basic blocks are fairly small and have small execution frequencies. Therefore, the average effect of programming a basic block with the extended entries on the execution time is fairly small for the Viterbi decoding.

The obtained results demonstrate that the proposed method is practical in case only a small part of the instructions in the program code are programmed with the extended dictionary entries. Hence, the method is best suitable for the case where the modifications required to the original program code are small, e.g., bug fixes. Better performance could be obtained if the compression would be applied at smaller granularity level. However, the reduction in area and power consumption would be worse compared to compression at full instruction level.

6.5 Summary of Results

Figure 37 summarizes the obtained average compression ratios for the dictionary-based, Huffman coding, and instruction template-based compression methods. For the dictionary-based compression and Huffman coding, the compression ratios are reported at the three different granularity levels, i.e., at full instruction (*FI*), move slot (*MS*), and ID field (*ID*) levels. At move slot and ID field granularity levels, the results are reported for both the vertical and horizontal compression approaches. For the instruction template-based compression, the results are reported for four, 16, and the maximum number of templates used to encode the original instructions.

The obtained results show that the best compression ratio, 40.9%, can be achieved by utilizing Huffman coding vertically at ID field granularity level. Instruction template-based compression with maximum number of templates reaches a compression ratio of 46.5%. Dictionary-based compression reaches at best a compression ratio of 52.5% when it is applied at move slot level. The compression ratios for the Huffman coding and dictionary-based compression include the overhead of the decompression, i.e., the size of the decoding table. For the instruction template-based compression the overhead of the decompression cannot be evaluated in terms of bits, so the compression ratios for it are too optimistic.

Figure 38 illustrates the average area and power consumption reductions for the dictionary-based and instruction template-based compression on the two TTA processor configurations. The results are shown for the dictionary-based compression at the

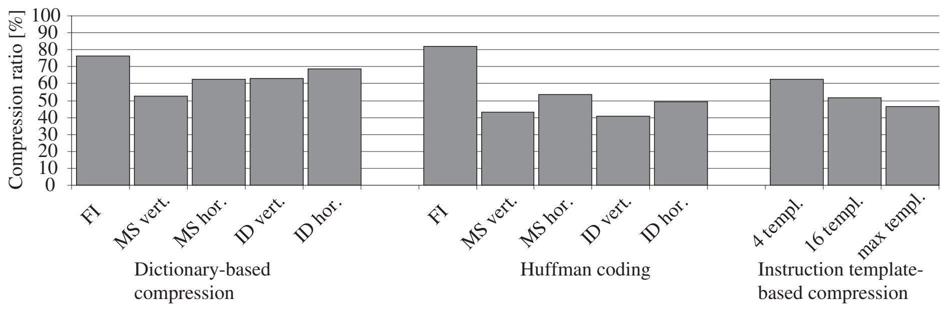
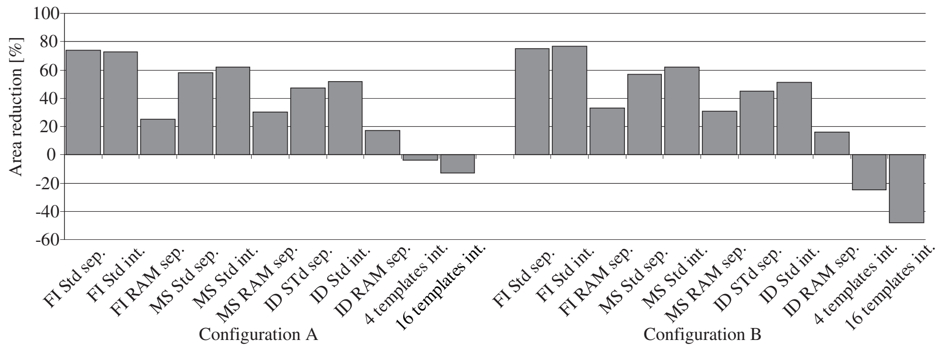


Fig. 37. *Summary of the code density evaluations in terms of average compresion ratio.*

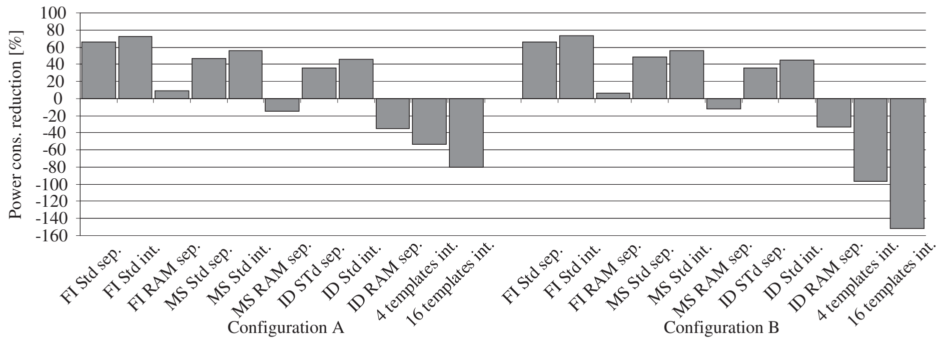
three different granularity levels. The results at these three granularity levels are summarized for the three decompressor implementation alternatives, i.e., separated decompression stage (*sep.*) with the decompressor implemented using RAM (*RAM*) or standard cells (*Std*) and integrated decompressor (*int.*) with the decompressor implemented using standard cells. For the instruction template-based compression, results are shown for four and 16 templates using an integrated decompressor.

The area and power consumption results clearly show that the dictionary-based compression is significantly more effective than the instruction template-based compression, which actually turned out to increase both the area and power consumption when it was implemented in hardware. Dictionary-based compressed performed the best at full instruction level with the integrated decompressor with the dictionary implemented using standard cells. This allowed to reduced the area at best 77% and power consumption at best 73%. Such a good results were obtained as the synthesis tool could be utilized effectively to optimize the logic of the dictionary. Integrating the decompressor to the decode stage allowed to reduce the area and power consumption even more effectively. This allowed also to maintain the performance as the length of the transport pipeline, and thus the jump delay, could be maintained unchanged.

As the obtained results from the code density and area and power consumption evaluations demonstrate, the traditional approach of evaluating the effectiveness of program compression methods based on compression ratios cannot fully characterize all the aspects related to compression. The overhead of the decompressor, i.e., the decoding tables are often not included in the compression ratio and for some compression methods, such as the instruction template-based compression, the overhead



(a) Area



(b) Power consumption

Fig. 38. *Summary of the area and power consumption evaluations based on the average area and power consumption reductions.*

of the decompressor cannot even be measured in bits. Even more severe deficiency in code density based estimations is that the details of the decompressor hardware implementation cannot be taken into account. For some methods the decoding tables may be small but the actual hardware cost of the decompressor may turn out to be large, especially if the compressed instructions become variable-width. Therefore, evaluations based on the actual area and power consumption are needed to provide more accurate estimates on the effectiveness of a compression method.

7. CONCLUSIONS

In this Thesis, program compression methods for improving the code density and minimizing the area and power consumption on a customizable processor architecture with parallel hardware resources have been studied. Minimizing the area and power consumption is crucial especially in handheld embedded systems, which are constrained by size, weight, battery life, and cost. Based on the review of previous work, three alternative compression methods were chosen to be adapted, utilized, and evaluated on transport triggered architecture, which is a customizable parallel processor architecture. The compression methods included dictionary-based compression, Huffman coding, and instruction template-based compression.

In order to evaluate the chosen compression methods on the customizable processor architecture, a case study of designing a set of customized processors for applications from the DSP and multimedia application domains was made. In addition to evaluating the code density, based on compression ratio, the customized processor designs were implemented in hardware and synthesized into standard cell logic using 130 nm low-power CMOS technology to evaluate the effects of program compression on area and power consumption, which allowed to obtain more accurate estimates on the effectiveness of the compression methods.

The evaluations demonstrated that measuring the code density in terms of compression ratio, which is the approach taken in most of the approaches published in the literature, is not enough to estimate the effectiveness of a compression approach. Compression ratio is based on the bit sizes of the program memory and the possible decoding table. The actual implementation details of the program memory and the decompression logic are not taken into account. Therefore, hardware implementations and measures in terms of area and power consumption are needed.

It was shown in the evaluations that the dictionary-based program compression is the most suitable compression method for the transport triggered architecture, the parallel processor architecture studied in this Thesis. Dictionary-based compression results in fixed-width compressed instructions that are easier to decompress. The other evaluated compression approaches, Huffman coding and instruction template-based compression, result in variable-width compressed instructions that make the instruction fetch and decompression logic significantly more complex, which results in large overhead in both area and power consumption.

The conducted experiments on alternative implementation strategies of the decompression logic showed that the decompressor, i.e., the dictionary, in the dictionary-based compression approach is best to be implemented in standard cell logic and integrated with the decoder into a single pipeline stage. Up to 77% reduction in area and 73% reduction in power consumption of the program memory and the associated control logic were achieved. The drawback of implementing the dictionary using standard cells is low orthogonality of the instruction set, and in the worst case, highly limited programmability, which may make the modifications to the program code impossible. Implementing the dictionary using RAM allowed to modify the contents of the dictionary and, therefore, maintain the programmability and full orthogonality. However, the achieved area and power consumption results of this approach were poor. Therefore, a method was proposed to extend the dictionaries holding the unique bit patterns with such bit pattern entries that the processor can be reprogrammed even after the decompressor has been implemented in hardware. The orthogonality of the instruction set remains poor, implying poor performance. Hence, the modifications to the program code are limited to be fairly small.

Based on the studies in this Thesis, it can be concluded that the area and power consumption can be efficiently minimized through program compression, especially on a customizable processor architecture where the decompression logic can be implemented in the control path of the processor core. The proposed compression system, based on the dictionary-based compression, can reduce the area and power consumption of the program memory effectively with a negligible overhead due to decompression logic compared to the reviewed state-of-the-art compression approaches. The proposed method to maintain the programmability improves the usability of the dictionary-based compression as the program code can be modified also after the program compression has been applied and the processor has been implemented in hardware.

BIBLIOGRAPHY

[1] “Excerpts from inside the StarCore SC140,” Berkley Design Technology Inc., Tech. Rep., 2000.

[2] “CHESS/CHECKERS: A retargetable tool-suite for embedded processors,” Target Compiler Technologies n.v., Leuven, Belgium, Technical White Paper, June 2003.

[3] *TMS320C62x DSP CPU and Instruction set Reference Guide*. Houston, TX, USA: Texas Instruments Inc., July 2006.

[4] *TMS320C64x/C64x+ DSP CPU and Instruction set Reference Guide*. Houston, TX, USA: Texas Instruments Inc., Aug. 2006.

[5] *TMS320C67x/C67x+ DSP CPU and Instruction set Reference Guide*. Houston, TX, USA: Texas Instruments Inc., Nov. 2006.

[6] “ARC International homepage,” http://www.arc.com, June 2007.

[7] “CoWare processor designer homepage,” http://www.coware.com/products/processordesigner.php, June 2007.

[8] “The homepage of Improv Systems,” http://www.improvsys.com, June 2007.

[9] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. San Jose, CA, USA: Xilinx Inc., Mar. 5 2007.

[10] S. Aditya, S. A. Mahlke, and B. R. Rau, “Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats,” *ACM Trans. Design Automation of Electron. Syst.*, vol. 5, no. 4, pp. 752–773, Oct. 2000.

[11] S. Aditya and B. R. Rau, “Automatic architecture synthesis and compiler retargeting for VLIW and EPIC processors,” Hewlett-Packard Laboratories, Tech. Rep. HPL-1999-93, Jan. 2000.

[12] S. Aditya, B. R. Rau, and R. C. Johnson, “Automatic design of VLIW and EPIC instruction formats,” Hewlett-Packard Laboratories, Tech. Rep. HPL-1999-94, Apr. 2000.

[13] Altera, *Nios II Processor Reference Handbook*, http://www.altera.com, 2003.

[14] G. Araújo, P. Centoducatte, R. Azevedo, and R. Pannain, “Expression-tree based algorithms for code compression on embedded RISC architectures,” *IEEE Trans. Very Large Scale Integration VLSI Systems*, vol. 8, no. 5, pp. 530–533, Oct. 2000.

[15] G. Araújo, P. Centoducatte, M. Côrtes, and R. Pannain, “Code compression based on operand factorization,” in *Proc. 11th Great Lakes Symp. VLSI*, West Lafayette, IN, USA, Mar. 22 – 23 2001, pp. 89–92.

[16] ARM, *An Introduction to Thumb*. Advanced RISC Machines Ltd., Mar. 1995.

[17] L. Benini, A. Macii, and E. Macii, “Static footprint control in code compression for low-energy embedded systems,” in *Proc. Int. Workshop Power and Timing Modeling, Optimization and Simulation*, Yverdon-les-Baines, Switzerland, Sept. 26–28 2001, pp. 206–211.

[18] L. Benini, A. Macii, E. Macii, and M. Poncino, “Region compression: a new scheme for memory energy minimization in embedded systems,” in *Proc. 25th EUROMICRO Conf.*, Milan, Italy, Sept. 8–10 1999, pp. 311–317.

[19] ——, “Selective instruction compression for memory energy reduction in embedded system,” in *Proc. Int. Symp. Low-Power Electronics Design*, San Diego, CA, USA, Aug. 16–17 1999, pp. 206–211.

[20] L. Benini, A. Macii, and A. Nannarelli, “Cached-code compression for energy minimization in embedded processors,” in *Proc. of 2001 Int. Symp. Low Power Electronics and Design*, Huntington Beach, CA, USA, Aug. 6–7 2001, pp. 322–327.

[21] P. Biswas and N. Dutt, “Reducing code size for heterogenous-connectivity-based WLIW DSP through synthesis of instruction set extensions,” in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, San Jose, CA, USA, Oct. 30 – Nov. 1 2003, pp. 104–112.

[22] R. K. Brayton and R. Spence, *Sensitivity and Optimization*. Amsterdam, The Netherlands: Elsevier, 1980.

[23] Cambridge Consultants, *APE2 Digital Signal Processor*, http://www.cambridgeconsultants.com, June 2007.

[24] J. Canny, “A computational approach to edge detection,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 8, pp. 679–714, Jan. 1986.

[25] A. Capitanio, N. Dutt, and A. Nicolau, “Partitioned register files for VLIWs: A preliminary analysis of tradeoffs,” in *Proc. IEEE Symp. Microarchitecture*, Portland, OR, USA, Dec. 1–4 1992, pp. 292–300.

[26] P. Centoducatte, R. Pannain, and G. Araújo, “Compressed code execution on DSP architectures,” in *Proc. ACM/IEEE Int. Symp. System Synthesis*, San Jose, CA, USA, Nov. 10–12 1999, pp. 55–61.

[27] I. Chen, P. Bird, and T. Mudge, “The impact of instruction compression on I-cache performance,” University of Michigan, Technical Report CSE-TR-330-97, 1996.

[28] M. Collon and M. Brorsson, “The design and performance of a variable length instruction set for low-power instruction fetch,” Royal Institute of Technology, Stockholm, Sweden, Tech. Rep., 2002.

[29] R. P. Colwell, R. P. Nix, J. J. O’Connel, D. B. Papworth, and P. K. Rodman, “A VLIW architecture for a trace scheduling compiler,” *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 967–679, Aug. 1988.

[30] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, “Instruction fetch mechanisms for VLIW architectures with compressed encodings,” in *Proc. 29th Ann. IEEE/ACM Int. Symp. Microarchitecture*, Paris, France, Dec. 2–4 1996, pp. 201–211.

[31] J. Cooley and J. Tukey, “An algorithm for the machine calculation of the complex Fourier series,” *Mathematics of Computation*, vol. 19, pp. 297–301, Apr. 1965.

[32] K. D. Cooper and N. McIntosh, “Enhanced code compression for embedded RISC processors,” in *Proc. Conf. Programming Languages Design and Implementation*, Atlanta, GA, USA, May 1–4 1999, pp. 139–149.

[33] G. Cormack and R. Horspool, “Data compression using dynamic Markov modeling,” *The Computer Journal*, vol. 30, no. 6, pp. 541–550, Dec. 1987.

[34] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[35] H. Corporaal and M. Arnold, “Using transport triggered architectures for embedded processor design,” *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

[36] S. Debray and W. Evans, “Profile-guided code compression,” in *Proc. Conf. Programming Languages Design and Implementation*, Berlin, Germany, June 17–19 2002, pp. 95–105.

[37] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. M. O. Homewood, “Lx: A technology platform for customizable VLIW embedded processing,” in *Proc. 27th Int. Symp. Computer Architecture*, New York, NY, USA, June 10–14 2000, pp. 203–213.

[38] T. Fischer, “A pyramid vector quantizer,” *IEEE Trans. Information Theory*, vol. 32, pp. 568–583, July 4 1986.

[39] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architectures, Compilers and Tools*. Morgan Kaufmann Publishers, 1998.

[40] D. R. Gonzales, “Micro-RISC architecture for the wireless market,” *IEEE Micro*, vol. 19, no. 4, 1999.

[41] R. E. Gonzalez, “Xtensa — A configurable and extensible processor,” *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[42] S. Haga and R. Barua, “EPIC instruction scheduling based on optimal approaches,” in *Proc. 1st Ann. Workshop Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, Austin, TX, USA, Dec. 2 2001.

[43] R. W. Hamming, “Error-detecting and error-correcting codes,” *Bell System Technical Journal*, vol. 29, pp. 147–160, Apr. 1950.

[44] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, “SH3: High code density, low power,” *IEEE Micro*, vol. 15, no. 6, pp. 11–19, Dec. 1995.

[45] J. Heikkinen, J. Takala, A. Cilio, and H. Corporaal, “On efficiency of transport triggered architectures in DSP applications,” in *Advances in Systems Engineering, Signal Processing and Communications*, N. Mastorakis, Ed. New York, NY, USA: WSEAS Press, 2002, pp. 25–29.

[46] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, “A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, pp. 1338–1354, 2001.

[47] J. Hoogerbrugge, “Code generation for transport triggered architectures,” Ph.D. Thesis, Delft University of Technology, The Netherlands, Feb. 1996.

[48] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel, “A code compression system based on pipelined interpreters,” *Software - Practice and Experience*, vol. 29, no. 11, pp. 1005–1023, 1999.

[49] P. G. Howard and J. S. Witter, *Practical Implementations of Arithmetic Coding*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[50] D. A. Huffman, “A method for the construction of minimum redundancy codes,” *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sept. 1952.

[51] A. Ibrahim, A. Davis, and M. Parker, “ACT: a low power VLIW cluster coprocessor for DSP applications,” in *Proc. Workshop Optimizations for DSP and Embedded Systems*, New York, NY, USA, Mar. 26 2006.

[52] Intel, *Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture, Revision 2.2*, Jan. 2006.

[53] T. Ishihara and H. Yasuura, "A power reduction technique with object code merging for application specific embedded processors," in *Proc. Design, Automation and Test in Europe Conf.*, Paris, France, March. 27–30 2000, pp. 617–623.

[54] P. Jääskeläinen, "Instruction set simulator for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, Aug. 2005.

[55] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Multimedia on Mobile Devices*, ser. Proceedings of SPIE, R. Creutzburg, J. Takala, and J. Cai, Eds., San Jose, CA, USA, Jan. 28 – Feb. 1 2007, vol. 6507.

[56] Y. Jin and R. Chen, "Instruction cache compression for embedded systems," Berkley Wireless Research Center," Technical Report, 2000.

[57] M. B. Jr. and R. Smith, "Enhanced compression techniques to simplify program decompression and execution," in *Proc. Int. Conf. Computer Design*, Austin, TX, USA, Oct. 12–15 1997, pp. 170–176.

[58] I. Kadayif and M. T. Kandemir, "Instruction compression and encoding for low-power systems," in *Proc. 15th Ann. IEEE Int. ASIC/SOC Conf.*, Rochester, NY, USA, Sept. 25–28 2002, pp. 301–305.

[59] J. Kang, J. Lee, and W. Sung, "A compiler-friendly RISC-based digital signal processor synthesis and performance evaluation," *Journal of VLSI Signal Processing*, no. 27, pp. 297–312, 2001.

[60] D.-H. Kim and H. J. Lee, "Iterative procedural abstraction for code size reduction," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Grenoble, France, Oct. 8–11 2002, pp. 277–279.

[61] K. Kissell, *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[62] M. Kjelsø, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *Proc. EUROMICRO Conf.*, Prague, Czech Republic, Sep. 2–5 1996, pp. 423–430.

[63] M. Kozuch and A. Wolfe, "Compression of embedded system programs," in *Proc. Int. Conf. Computer Design*, Cambridge, MA, USA, Oct. 10–12 1994, pp. 270–277.

[64] K. Kucukcakar, "An ASIP design methodology for embedded systems," in *Proc. IEEE Symp. Microarchitecture*, Rome, Italy, Dec. 10–13 1999, pp. 17–21.

[65] J. Kwak and J. You, "One- and two-dimensional constant geometry fast cosine transform algorithms and architectures," *IEEE Trans. Signal Processing*, vol. 47, no. 7, pp. 2023–2034, July 1999.

[66] L. Laasonen, "Program image and processor generator for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, May 2007.

[67] P. Lapsley, J. Bier, A. Shinham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features*. Wiley-IEE Press, 1996.

[68] S. Y. Larin and T. M. Conte, "Compiler-driven cached code compression schemes for embedded ILP processors," in *Proc. 32nd Ann. Int. Symp. Microarchitecture*, Gold Coast, Australia, Jan. 29–21 1999, pp. 82–92.

[69] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia communications systems," in *Proc. 30th Ann. IEEE/ACM Int. Symp. Microarchitecture*, Research Triangle Park, NC, USA, Dec. 1–3 1997, pp. 330–335.

[70] H. Lee, P. Beckett, and B. Appelbe, "High-performance extendable instruction set computing," in *Proc. 6th Australasian Computer Systems Architecture Conf.*, Haifa, Israel, Nov. 16–18 2001, pp. 89–94.

[71] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proc. 30th Ann. Int. Symp. Microarchitecture*, Research Triangle Park, NC, USA, Dec 1–3 1997, pp. 194–203.

[72] C. Lefurgy and T. Mudge, "Code compression for DSP," University of Michigan, Technical Report CSE-TR-380-98, Nov. 1998.

[73] ——, "Fast software-managed code decompression," in *Proc. Int. Workshop Compiler and Architecture Support for Embedded Systems*, Washington, DC, USA, Oct. 1–3 1999, pp. 139–143.

[74] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a high performance code compression method," in *Proc. 32nd Ann. Int. Symp. Microarchitecture*, Haifa, Israel, Nov. 16–18 1999, pp. 93–102.

[75] ——, "Reducing code size with run-time decompression," in *Proc. 6th Int. Symp. High-Performance Computer Architecture*, Toulouse, France, Jan. 8–12 2000, pp. 218–228.

[76] H. Lekatsas, J. Henkel, and W. Wolf, "Arithmetic coding for low power embedded system design," in *Proc. Data Compression Conf.*, Snowbird, UT, USA, Mar. 28–30 2000, pp. 430–439.

[77] ——, "Code compression as a variable in hardware/software co-design," in *Proc. 8th Int. Workshop Hardware/Software Co-Design*, San Diego, CA, USA, May 3–5 2000, pp. 120–124.

[78] ——, "Code compression for low power embedded system design," in *Proc. 37th Conf. Design Automation.*, Los Angeles, CA, USA, June 5–9 2000, pp. 294–299.

[79] ——, "Design and simulation of a pipelined decompression architecture for embedded systems," in *Proc. ACM/IEEE Int. Symp. System Synthesis*, Montreal, Quebec, Canada, Sept. 30 – Oct. 3 2001, pp. 63–68.

[80] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *Proc. 35th Conf. Design Automation*, San Francisco, CA, USA, June 15–19 1998, pp. 516–521.

[81] ——, "Random access decompression using binary arithmetic coding," in *Proc. Data Compression Conf.*, Snowbird, UT, USA, Mar. 29–31 1999, pp. 306–315.

[82] ——, “SAMC: A code compression algorithm for embedded processors,” *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, vol. 18, no. 12, pp. 1689–1701, Dec. 1999.

[83] S. Liao, “Code generation and optimization for embedded digital signal processors,” Ph.D. dissertation, Massachusetts Institute of Technology, MA, USA, June 1996.

[84] S. Liao, S. Devadas, and K. Keutzer, “Code density optimization for embedded DSP processors using data compression techniques,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 7, pp. 601–608, July 1998.

[85] ——, “A text compression-based method for code size minimization in embedded systems,” *ACM Trans. Design Automation of Electronic Systems*, vol. 4, no. 1, pp. 12–38, Jan. 1999.

[86] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, “Storage assignment to decrease code size,” *SIGPLAN Notices*, vol. 30, no. 6, pp. 138–143, June 1995.

[87] C. Liem, F. Breant, S. Jadhav, R. O’Farrell, R. Ryan, and O. Levia, “Embedded tools for a customizable DSP architecture,” *IEEE Design and Test of Computers*, vol. 19, no. 6, pp. 27–35, Nov.-Dec. 2002.

[88] C. H. Lin, Y. Xie, and W. Wolf, “LZW-based code compression for VLIW embedded systems,” in *Proc. Design, Automation and Test in Europe Conf. and Exhib.*, vol. 3, Paris, France, Feb. 16–20 2004, pp. 76–81.

[89] K. Lin, J.-J. Shann, and C.-P. Chung, “Code compression by register operand dependency,” in *Proc. 6th Ann. Workshop Interaction between Compilers and Computer Architectures*, Cambridge, MA, USA, Feb. 3 2002, pp. 91–101.

[90] C.-H. Liu, T.-J. Lin, C.-M. Chao, P.-C. Hsiao, L.-C. Lin, S.-K. Chen, C.-W. Huang, C.-W. Liu, and C.-W. Jen, “Hierarchical instruction encoding for VLIW digital signal processors,” in *Proc. IEEE Int. Symp. Circuits and Systems*, Kobe, Japan, May. 23–26 2005, pp. 3503–3506.

[91] P. G. Lowney, “The Multiflow trace scheduling compiler,” *The Journal of Supercomputing*, vol. 7, no. 1–2, pp. 51–143, May 1993.

[92] S. J. Nam, I. C. Park, and C. M. Kyung, “Improving dictionary-based code compression in VLIW architectures,” *IEICE Trans. Fundamentals of Electronics, Commun. and Comput. Sciences*, vol. E82-A, no. 11, pp. 2318–2124, Nov. 1999.

[93] T. Okuma, H. Tomiyama, A. Inoue, E. Fajar, and H. Yasuura, “Instruction encoding techniques for area minimization of instruction ROM,” in *Proc. 11th Int. Symp. System Synthesis*, Hsinchu, Taiwan, Dec. 2–4 1998, pp. 125–130.

[94] H. Pan and K. Asanovic, “Heads and tails: a variable-length instruction format supporting parallel fetch and decode,” in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Atlanta, GA, USA, Nov. 16–17 2001, pp. 168–175.

[95] R. Pannain, G. Araújo, and P. Centoducatte, “Using operand factorization to compress DSP programs,” in *Proc. 11th Symp. Computer Architecture and High Performance Computing*, Natal, Brazil, Sept. 29 – Oct. 2 1999, pp. 223–229.

[96] S. Pees, A. Hoffman, V. Zivojnovic, and H. Meyer, “LISA – machine description language for cycle-accurate models of programmable DSP architectures,” in *Proc. Design Automation Conf.*, New Orleans, LA, USA, June 21–25. 1999, pp. 933–938.

[97] E. Piccinelli and R. Sannino, “Code compression for VLIW embedded processors,” *ST Journal of Research*, vol. 1, no. 2, pp. 32–46, 2004.

[98] C. Piquet, P. Volet, and J.-M. Masgonty, “Code memory compression with online decompression,” in *Proc. 27th European Solid-State Circuits Conf.*, Villach, Austria, Sept. 18–20 2001, pp. 150–152.

[99] T. Pitkänen, T. Rantanen, A. Cilio, and J. Takala, “Hardware cost estimation for application-specific processor design,” in *Embedded Comput. Syst.: Architectures, Modeling, and Simulation, Proc. 5th Int. Workshop SAMOS V*, ser. Lecture Notes in Computer Science, T. Hämäläinen, A. Pimentel, J. Takala, and S. Vassiliadis, Eds. Berlin, Germany: Springer-Verlag, 2005, vol. LNCS 3553, pp. 212–221.

[100] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, and T. Chen, "VLSI design and implementation fuels the signal-processing revolution," *IEEE Signal Processing Mag.*, vol. 15, no. 1, pp. 22–37, Jan. 1998.

[101] T. Rantanen, "Cost estimation for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, May. 2004.

[102] T. Richter, W. Drescher, E. Engel, S. Kobayashi, V. Nikolajevic, M. Weiss, and G. Fettweis, "A platform-based highly parallel digital signal processor," in *Proc. Custom Integrated Circuits Conf.*, San Diego, CA, USA, May 6–9 2001, pp. 305–305.

[103] J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[104] M. Ros and P. Sutton, "Compiler optimization and ordering effects on VLIW code compression," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, San Jose, CA, USA, Oct. 30 – Nov. 1 2003, pp. 95–103.

[105] ——, "A Hamming distance based WLIW/EPIC code compression technique," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, Washington, DC, USA, Sept. 22–25 2004, pp. 132–139.

[106] C. Rowen and S. Leibson, *Engineering the Complex SOC: Fast Flexible design with Configurable Processors*. Upper Sadle River, NJ, USA: Prentice Hall Professional Technical Reference, June 2004.

[107] J. Runeson, "Code compression through procedural abstraction before register allocation," Master's thesis, University of Uppsala, Sweden, Mar. 2000.

[108] M. S. Schlansker and B. R. Rau, "EPIC: an architecture for instruction-level parallel processors," Hewlett-Packard Laboratories, Tech. Rep. HPL-1999-111, Feb. 2000.

[109] J. Sertamo, "Processor generator for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, Sept. 2003.

[110] P. Simonen, I. Saastamoinen, M. Kuulusa, and J. Nurmi, "Advanced instruction set architectures for reducing program memory usage in a DSP processor,"

in *Proc. 1st Int. Workshop Electronic Design, Test, and Applications*, Christenchurch, New Zealand, Jan. 29–31 2002, pp. 477–479.

[111] P. Song, "Demystifying EPIC and IA-64," *Microprocessor Report*, vol. 12, no. 1, pp. 24–30, Jan. 28 1998.

[112] C. G. Subash, M. Mahesh, and R. Gpvomdarajan, "Area and power reduction of embedded DSP systems using instruction compression and re-configurable encoding," in *Proc. Int. Conf. Comput. Aided Design*, San Jose, CA, USA, Nov. 4–8 2001, pp. 631–634.

[113] B. D. Sutter, K. D. Bosschere, B. D. Bus, B. Demoen, and P. Keyngnaert, "Whole-program optimization of binary executables," in *Proc. Computer and E-business Conf.*, Rome, Italy, July 31 – Aug. 6 2000.

[114] H. Suzuki, H. Makino, and Y. Matsuda, "Novel VLIW code compaction method for 3D geometry processors," in *Proc. IEEE Custom Integrated Circuits Conf.*, Orlando, FL, USA, May 21–24 2000, pp. 555–558.

[115] J. Takala, D. Akopian, J. Astola, and J. Saarinen, "Constant geometry algorithm for discrete cosine transform," *IEEE Trans. Signal Processing*, vol. 48, no. 6, pp. 1840–1843, June 2000.

[116] B. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1967.

[117] A. J. Viterbi, "Error bounds for convolutional coding and an asymptotically optimum decoding algorithm," *IEEE Trans. Information Theory*, vol. 13, pp. 260–269, Apr. 1967.

[118] M. H. Weiss and G. P. Fettweis, "Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processing," in *Proc. 3rd Int. Workshop Image and Signal Processing on the Theme of Advances in Computational Intelligence*, Manchester, UK, Nov. 4–7 1996, pp. 517–520.

[119] S. Weiss and S. Beren, "HW/SW partitioning of an embedded instruction memory decompressor," in *Proc. 9th Int. Symp. Hardware/Software Codesign*, Copenhagen, Denmark, Apr. 25–27 2001, pp. 36–41.

[120] T. A. Welch, “A technique for high-performance data compression,” *Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[121] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1999.

[122] A. Wolfe and A. Chanin, “Executing compressed programs on an embedded RISC architecture,” in *Proc. 25th Ann. Int. Symp. Microarchitecture*, Portland, OR, USA, Dec. 1–4 1992, pp. 81–91.

[123] A. Wolfe and C. Chanin, “Executing compressed programs on an embedded RISC processor,” in *Proc. 25th Annual Symp. Microarchitecture*, Portland, OR, USA, Dec. 1–4 1992, pp. 81–91.

[124] Y. Xie, H. Lekatsas, and W. Wolf, “Code compression for VLIW processors,” in *Proc. Data Compression Conf.*, Snowbird, UT, USA, Mar. 27–29 2001, p. 525.

[125] ——, “A code decompression architecture for VLIW processors,” in *Proc. 34th ACM/IEEE Int. Symp. Microarchitecture*, Austin, TX, USA, Dec. 1–5 2001, pp. 66–75.

[126] ——, “Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures,” in *Proc. 4th Int. Conf. ASIC*, Shanghai, China, Oct. 23–25 2001, pp. 337–340.

[127] Y. Xie, W. Wolf, and H. Lekatsas, “Code compression for VLIW processors using variable-to-fixed coding,” in *Proc. ACM/IEEE Int. Symp. System Synthesis*, Kyoto, Japan, Oct. 2–4 2002, pp. 138–143.

[128] Xilinx, *MicroBlaze Processor Reference Guide*, http://www.xilinx.com, Oct. 2005.

[129] ——, *PowerPC Processor Reference Guide*, http://www.xilinx.com, Jan. 2007.

[130] J.-H. Yang, B.-W. Kim, S.-J. Nam, Y.-S. Kwon, D.-H. Lee, J.-Y. Lee, C.-S. Hwang, Y.-H. Lee, S.-H. Hwang, I.-C. Park, and C.-M. Kyung, “MetaCore:

an application-specific programmable DSP development system," *IEEE Trans. VLSI Systems*, vol. 8, no. 2, pp. 173–183, 2000.

[131] C. Yeh and C.-S. Wang, "A program compression technique supporting IP-centric SOC design," in *Proc. 13th Ann. IEEE Int. ASIC/SOC Conf.*, Arlington, VA, USA, Sept. 13–16 2000, pp. 226–230.

[132] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa, "Low-power consumption architecture for embedded processor," in *Proc. 2nd Int. Conf. ASIC*, Sanghai, China, Oct. 21–24 1996, pp. 77–80.

[133] ——, "An object code compression approach to embedded processors," in *Proc. Int. Symp. Low-Power Electronics and Design*, Monterey, CA, USA, Aug. 18–20 1997, pp. 265–268.

[134] J. Zalamea, J. Llosa, E. Ayquade, and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *Proc. IEEE Symp. Microarchitecture*, Monterey, CA, USA, Dec. 10–13 2000, pp. 137–146.

[135] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[136] ——, "Compression of individual sequences via variable rate coding," *IEEE Trans. Information Theory*, vol. 24, no. 5, pp. 530–536, Sept. 1978.

[137] V. Zivojnovic, S. Pees, and H. Meyer, "LISA – machine description language and generic machine model for HW/SW co-design," in *Proc. IEEE Workshop VLSI Signal Processing*, San Francisco, CA, USA, Oct. 30 – Nov. 1 1996, pp. 127–136.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O. Box 527
FIN-33101 Tampere, Finland