

在超長指令集程式壓縮上，根據利益評估建造分離式字典

Building partitioned dictionary with benefit estimation for

VLIW code compression

研究生：楊佳原

Student : Jia-Yuan Yang

指導教授：鍾崇斌博士

Advisor : Dr. Chung-Ping Chung

國立交通大學

資訊工程學系

碩士論文



Submitted to Department of

Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月


# 在超長指令集程式壓縮上，根據利益評估建造分離式字典

學生：楊佳原

指導教授：鍾崇斌 教授

國立交通大學資訊工程學系碩士班

## 摘要



在處理器上使用 VLIW 來提昇效能是個趨勢。但在可攜式系統上，程式總是希望越小越好。VLIW 鬆散的程式結構對記憶體的使用而言是不好的。程式記憶體的保存資料和存取，占了超過 50% 的耗電，而壓縮程式碼不但可以減低處理器對程式記憶體的需求，也能減少處理器和記憶體之間的傳送，因此希望設計一個方法減少大幅減小程式碼的大小。

在本篇論文中，引用操作碼 (Opcode) 和運算元 (Operand) 序列分開建立字典的想法。為了善用 VLIW 的重複性和規則性，我們採用壓縮程式碼附加長度欄位，一次壓縮數道指令。依據我們訂定的利潤評估公式，挑選出最佳的一群 VLIW 序列放入字典中。操作碼和運算元字典的建造先後順序和空間壓力，最長序列的限制也是我們討論的重點。

實驗結果顯示，在新的序列排列方法下可以達到 42% 的壓縮率。

# Building partitioned dictionary with benefit estimation for VLIW code compression

Student : Jia-Yuan Yang

Advisor : Dr. Chung-Ping Chung

Institute of Computer Science and Information Engineering  
National Chiao-Tung University

## ABSTRACT

It is a trend that people adopt VLIW to enhance processor performance. When it comes to embedded software, smaller is better. Decreasing code size to fit into cost- or space-constrained memory systems is important business these days. On VLIW embedded system, the power consumption of code memory is about 50%. By code compression, code size and traffic between memory and CPU could be reduced. So we hope to design an approach for VLIW code compression to reduce code size greatly.

In this thesis, we adopt partitioned dictionary and codeword with length slot. Partitioned dictionary has opcode and operand dictionaries and has more chance to reuse dictionary entry. Codeword with length slot can reduce the number of codewords and let dictionary entry could be reused by different sequences. We mainly present a heuristic algorithm for building Dictionary. Depending on our benefit equation, we can judge which VLIW sequences are beneficial. Also, we discuss the relation between opcode and operand, the effect of max compress length and combine relation.

Experimental results show that 40.2% compression ratio can be achieved on average.

Chapter 1 Introduction .....	1
1.1    VLIW .....	1
1.2 Motivation.....	2
1.3    Objective .....	3
1.4    Organization of the Thesis .....	3
Chapter 2 Background .....	5
2.1 VLIW code compression .....	5
2.2 Dictionary-based Method .....	6
2.3 Partitioned Dictionary .....	8
2.3    Codeword with length slot .....	11
2.4 Summary .....	12
Chapter 3 Design.....	14
Given Environment .....	14
Base idea .....	15
3.1 Build the dictionaries .....	16
3.1.1 How to judge a sequence’s benefit.....	17
3.1.2 Build Dictionary flow .....	19
3.1.2.1 Create candidate sequence set.....	21
3.1.2.2.1 Choose the benefit one into chosen set.....	21
3.1.2.2.2 Combine relation.....	22
3.1.2.3 Stop chosen set increasing .....	22
3.1.2.4 Re-count and remove impossible sequence .....	23
3.2 Replacing VLIW line sequence with codeword .....	24
3.2.1 How to replace sequence with codeword.....	24
3.2.2 Determine the Length Bits .....	26



3.2.3 Codeword format .....	26
3.3 Place codeword into memory .....	27
Chapter 4 Simulation .....	29
4. Platform and Benchmark .....	29
4.1 The effect of Max compress length .....	30
4.2 Compare with optimal situation.....	31
4.3 The effect of OP and OPD dictionary with different sizes .....	31
Chapter 5 Conclusion.....	34
REFERENCES .....	36



Figure 1.1 we can see IA64 NOP ratio on this picture. It is about 20%~30% NOP existed in IA64 code. ....2

Figure 2-1: Dictionary-base compression example. ....7

Figure 2-4 Compression using Operand factorization ..... 10

Figure 2-5 Instruction fetch path in the proposed code compression scheme for VLIW processor-based systems ..... 10

Figure 2-6 Dictionary entries with sequential access ability ..... 12

Figure 2-7 Dictionary entries can be inquired by different sequences ..... 12

Figure:3-2 After defining benefit, sequences are chosen in benefit order one by one. 17

Figure 3-3: Benefit equation. .... 18

Figure 3-4 The flow of choosing the most beneficial sequences ..... 20

Figure 3-5 Flow of Choose algorithm..... 20

Figure 3-6: Create candidate sequences ..... 21

Figure 3-8: What is combine relation ..... 22

Figure 3-10 Re-count ..... 23

Figure 3-10 Replace VLIW line sequence with OP&OPD index and length ..... 24

Figure 3-11 How to replace lines depending on Mark..... 25

Figure 4-1 VLIW format ADSP-21535 DSP ..... 29

Figure 4-2: The effect of Max compress length..... 30

Figure 4-3 Compare with optimal situation ..... 31


Figure 4-4 OP and OPD bit pattern program cover ratio ..... 32

Figure 4-5 Compared with giving enough dictionary size, it has better cut some dictionary size to reduce codeword length..... 33

# Chapter 1 Introduction

It is a trend that people adopt VLIW to enhance DSP and processor performance. When it comes to embedded software, smaller is better. Shrinking code size to fit into cost- or space-constrained memory systems is important business these days. By code compression, code memory size and traffic between memory and CPU could be reduced.

## 1.1 VLIW



In VLIW (Very long Instruction Word) architectures where a high-bandwidth instruction prefetch mechanism is required to supply multiple operations per cycle. Just few embedded system need to deal with complex computation. But modern embedded systems, such as network terminals or PDAs, consist of simple WWW browsers or some offices tools. The ability of computation is needed more and more on the embedded system. VLIW and super-scalar are two methods that are usually used to enhance performance. But VLIW is well suit to embedded system. VLIW processor is developed over twenty years, but the improvement of VLIW compiler falls behind the improvement of VLIW hardware. Recently VLIW compiler still can't generate good density code. A lot of no operation instructions (NOP) exist in the code and some instructions, which could be parallel-execution don't be found out. Writing tight code is one thing, but a processor's instruction set affects memory footprints as well. No amount of clever tweaking of C source code will make up for a

chip that has lousy code density.

To solve these problems, some new architectures are brought up, like that HP-Intel IA64 and TI C6x series. But they just solve a little part of problem.

Generally there are about 40% NOP in TI C6x series code and 20% in IA64 code [1].

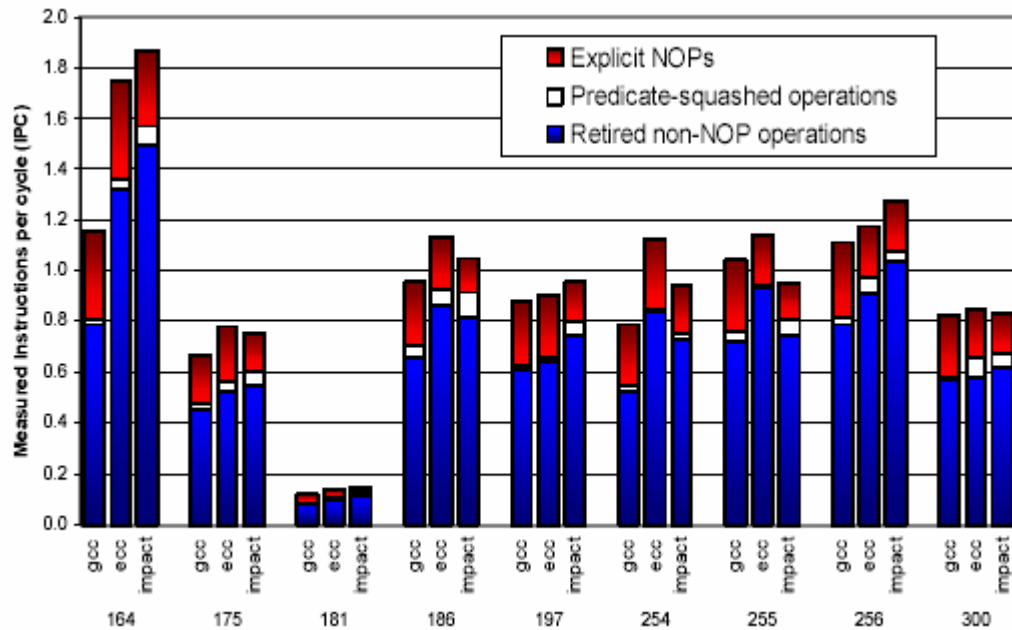


Figure 1.1 we can see IA64 NOP ratio on this picture. It is about 20%~30% NOP existed in IA64 code.

## 1.2 Motivation

Shrinking code size to fit into cost- or space-constrained memory systems is important business these days. On VLIW embedded system, the power consumption of code memory is about 50%. It's not uncommon to spend more money on memory than on the microprocessor, so choosing a processor that's thrifty with memory can pay off. Thus, to reduce code size by compressing VLIW code is important for reducing system code memory size. Furthermore, to compress code offers the same or higher instruction fetching bandwidth by using less bus width and reduces power



consumption in advance. So it will reduce power consumption greatly by reducing code memory size and memory access.

When compiling, VLIW compiler usually generates RISC code first. Then, depending on parallel rules, some instruction sequences will be combined to form VLIW line sequence. Some VLIW code optimization like loop unroll will repeat VLIW line sequences to enhance performance. Thus, the repetition of VLIW sequence is high. We could take this feature to reduce code size.

## 1.3 Objective

Design an approach for VLIW code compression to reduce code size to reduce data traffic between memory and CPU and memory size.

Replacing the most frequent sequences with smaller codeword achieves compression for a program. Dictionary-based code compression can be an instruction format dependence method that can utilize the repetition and regularity of code more efficiently. By compressing code, we hope to reduce data traffic between Memory and CPU, Memory size decrease and core size increase as little as possible.

## 1.4 Organization of the Thesis

This thesis is divided as follows. Chapter 2 discusses the factor in VLIW and dictionary-based code compression. In chapter 3 we describe our code compression algorithm based on Chapter2 code compression method. The experimental environment and benchmark suite are described in Chapter 4. Our experimental results and relative analysis are also presented in chapter 4. Then, we summarize our

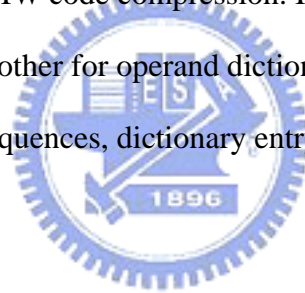
conclusions and future works in Chapter 5.



# Chapter 2

## Background

We introduce dictionary-based code compression in this Chapter. First, we introduce what is dictionary-based. Replacing the most frequent sequences with smaller codeword achieves code compression. Then partitioned dictionary-based method is well done in the VLIW code compression. It divide dictionary into two, one for opcode dictionary and the other for operand dictionary. In order to take advantage of repetitions of VLIW line sequences, dictionary entry with sequential access ability is adopted.



### 2.1 VLIW code compression

Variable-to-fixed (V2F) VLIW code compression [6] is a VLIW code compression scheme that use variable-to-fixed (V2F) length coding. It also proposes an instruction bus encoding scheme, which can electively reduce the bus power consumption. It shows that the compression ratios using memoryless V2F coding for IA-64 and TMS320C6x are around 72.7% and 82.5% respectively. Markov V2F coding can achieve better compression ratio up to 56% and 70% for IA-64 and TMS320C6x respectively. The length of codeword on V2F VLIW code compression is still various. Decompressing one VLIW line in one cycle can not be ensured. If we want to speedup its decompression, more ROM and decompressing logic is needed.

Modern VLIW ISAs adapt a VLES (various length execution set) scheme to achieve high code density. But this size of fetch bundle on IA64 or TI series are fixed, and it means dictionary-based method is still flexible to modern VLIW code compression. But convention Dictionary-based code compression scheme on RISC machines are still needed some changes to suit to VLIW compression like adding extra dictionary output port.

Considered compression ratio and decompress architecture, V2F isn't on an advantageous position. And fetch bundle which is fixed length will be various after V2F. If V2F take instruction group as its compressing element, the existence of bundle is not needed and we need to redesign whole instruction set. Our research is the code compression application using dictionary-based scheme.

## 2.2 Dictionary-based Method

Dictionary-based compression methods attempt to find out common sequences of characters and replace them with a single codeword. Codeword is a basic element of compressed code. It includes Tag (represent compressed code or non-compressed code), Index (represent which dictionary entry is quarried) and else. To improving quarried frequency of entry is the most important thing. But not whole program will be compressed; some programs which just appear once are no need to be compressed.

The reduction of code size is achieved if the code sequence in the dictionary appears more than once and can be replaced by a codeword that is smaller than the size of this code sequence.

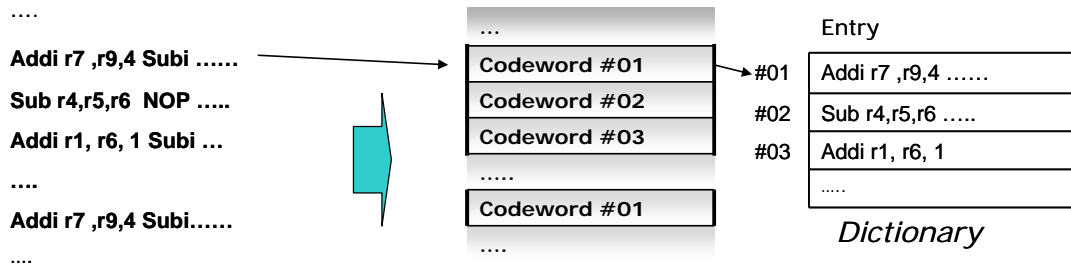


Figure 2-1: Dictionary-base compression example.

In Figure left part is program before compressing. If right dictionary have this bit pattern, we can replace code with codeword which has dictionary entry index. Some codes which is the part of (.....) can't be inquired by dictionary. These codes will not be compressed and still exist in program in original type or add some tag to present un-compressed state.

Dictionary decompression uses a codeword as an index into the dictionary table, and then inserts the dictionary entry into the decompressed code sequence. If codeword are aligned with machine words, the dictionary lookup is a constant time operation. Sometimes, in order to get more compression space, use Huffman encoding and MPEG-2 VLC encoding to encode index, and variable length index is produced. The general design for a compressed program processor is given in Figure

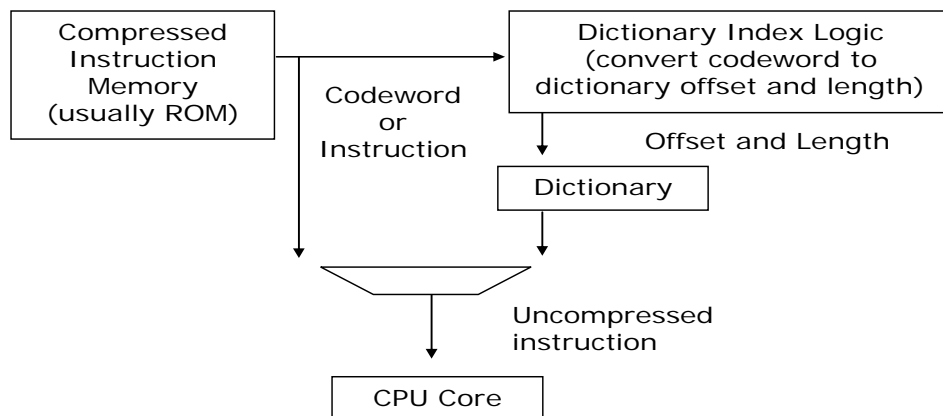


Figure 2-2 Compressed program processor

Lefurgy et al.[8] proposed a dictionary-based compression method, which stores a copy of the whole 32-bit instruction sequence, which appears frequently in the program, into the dictionary and replaces the occurrences of the sequence with shorter (fixed or variable-length) codeword. The average compression ratios of 61%, 66%, and 74% were reported for the PowerPC, ARM, and i386 processors respectively. Wolfe et al. proposed a Huffman-encoding compression method in Compressed Code RISC Processor (CCRP). Each 32-byte cache line is compressed into smaller aligned bytes or words.

The compressed code size will have three parts as follows:

1. Compressed size: After compression, most original program will be replaced by codeword and some program which don't be compressed: Not all code can be compressed. So compressed size includes codeword size and un-compressed code size.
2. Dictionary Size: We need dictionary to achieve compress code
3. According above, the estimate factor is:

Compression ratio =  $\frac{\text{Compressed Size} + \text{Dictionary size}}{\text{Original Size}}$

Compressed Size = Codes which could not be compressed + codeword size

## 2.3 Partitioned Dictionary

Improving Dictionary-Based Code Compression in VLIW Architectures

[Sang-Joon NAM 1999][7], which divides one VLIW line into two groups, one for opcode group and the other for operand group In Figure two indexes are used to indicate different dictionary. Frequent-used VLIW lines are extracted from the original code to be mapped into two dictionaries, an opcode dictionary and an operand dictionary. An average code compression ratio is 63%. In program, OP[opcode] or

OPD[operand] part have more repetition opportunities than whole VLIW line. Maybe we can proceed to find out other way to split dictionary but this paper shows OP and OPD is a good way to split.

Their algorithm has 2 steps as follows.

1. Building entries of two dictionaries

Building a dictionary that can achieve maximum compression is known as an NP-complete problem. This code compression scheme replaces an instruction word by an opcode sequence and an operand sequence, and limits their total bit-width to be the same as that normal operation. Thus, the maximum compression problem is changed from NP-complete problem to a simple greedy one.

2. Replacing instruction words with the opcode dictionary index and operand dictionary index

The occurrence of each opcode and operand in the entry of two dictionaries is simply represented by fixed length opcode index and operand index. Specifically, the total bit width as required for the opcode index and operand index is made equal to that of an uncompressed operation in order to align the compressed VLIW line with the cache boundary. This can result in worse compression than a variable-length opcode index and operand index encoding, but makes instruction-fetching and decoding mechanism simple and fast. In general, variable-length encoding methods such as Huffman encoding are expensive to decode.

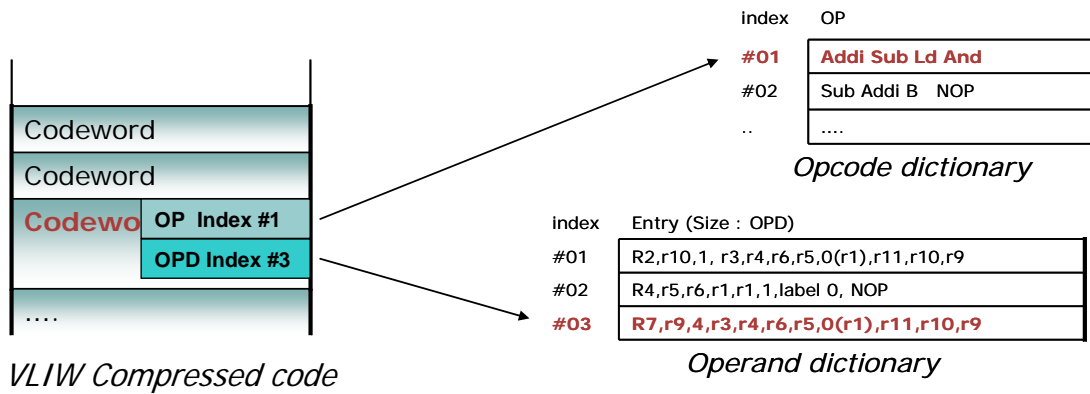


Figure 2-4 Compression using Operand factorization

In figure 2-4, one VLIW line is replaced by one codeword which has two indexes. In partitioned-based dictionary code compression, each codeword must be added one extra index. But in program, we have few chances to find out two totally the same VLIW lines. Using program character, the part of opcode or operand in the program is similar to each other; we can get more chance to reuse dictionary entries.

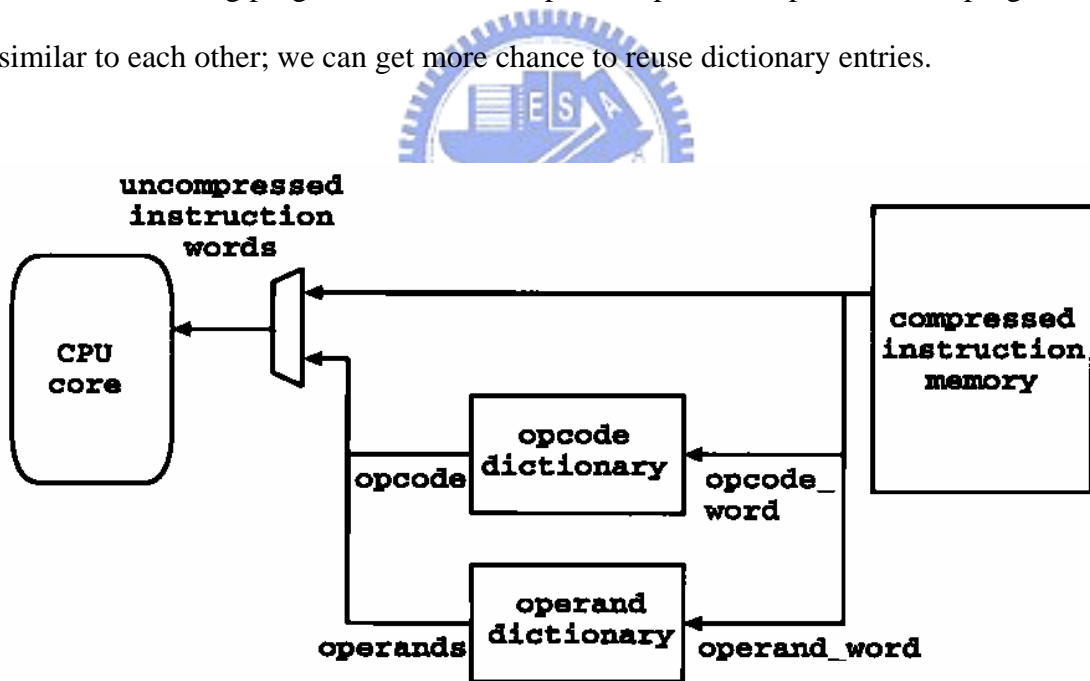


Figure 2-5 Instruction fetch path in the proposed code compression scheme for VLIW processor-based systems

Compressed program include compressed code and un-compressed code. When decompressing, we will inquire dictionary to decode codeword which is a shorter bit string to present compressed code. If un-compressed code is fetched, it will take the



bypass to enter processor directly.

## 2.3 Codeword with length slot

In Figure length slot is added to codeword. We adopt the dictionary to implement multiple-length common sequence. Sequences can start at any entry of dictionary and end at any following entry. The codeword is used as an index into the dictionary entry originally. But we give the ability to access dictionary with multiple entries. Length slot is added to codeword. According to length slot, Codeword can access any entry and its followings sequentially. A codeword takes two arguments: index and length. During decompression, the decompressor jumps to the point in the dictionary indicated by index and fetches length opcode or operands, and at next cycle decompressor would depend on length slot to access next codeword or increase index automatically.

When compiling, VLIW compiler usually generates RISC code first. Then, depending on parallel rules, some instruction sequences will be combined to form VLIW line sequence. Some VLIW code optimization like loop unroll will repeat VLIW line sequences to enhance performance. Thus, the repetition of VLIW sequence is high. In order to use the advantage, we take dictionary entry with sequential ability. Several sequential VLIW lines could be just compressed by one codeword.

By this way, the sum of codeword can be reduced.

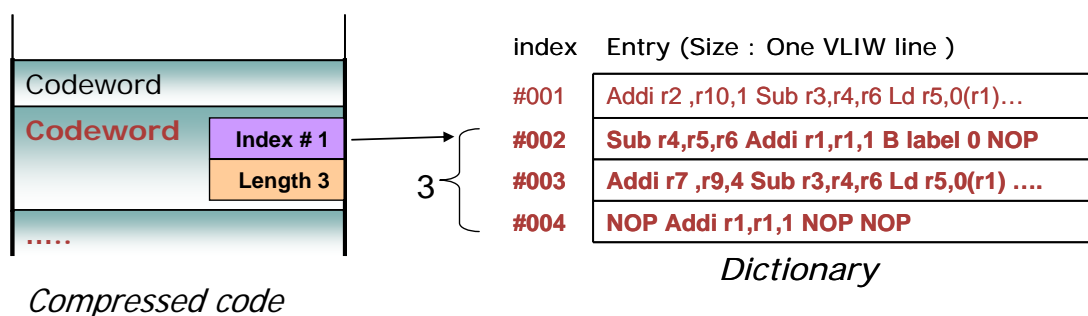


Figure 2-6 Dictionary entries with sequential access ability

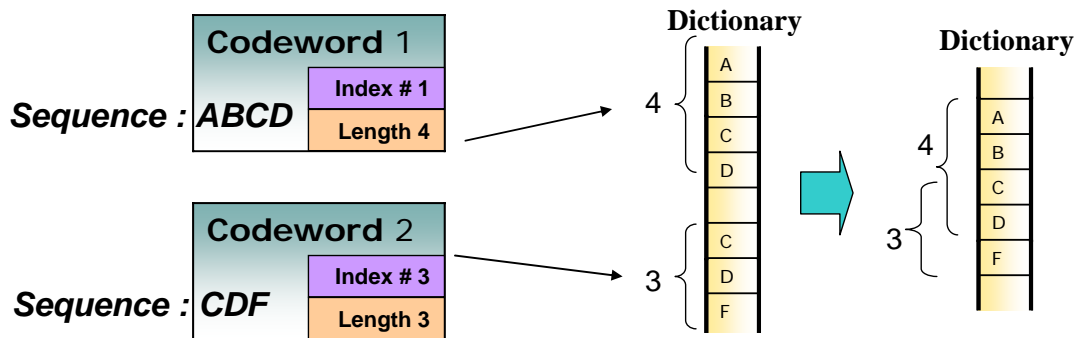


Figure 2-7 Dictionary entries can be inquired by different sequences

Sequence can start at any entry in the dictionary. Sequence can end at any entry after start entry in the dictionary. Dictionary entry could be used efficiently. That can reduce repetition entry as possible. Small sequences can combine to a large one and it will be more beneficial.



## 2.4 Summary

Dictionary-based is an instruction format dependence method that can utilize the repetition and regularity of code more efficiently. To increase the dictionary entry quarried frequency, we take improving dictionary-based code compression. To take advantage of VLIW sequence, we let dictionary entry with sequential ability. We adopt partition dictionary to increase repetition of dictionary entry. We could have more opportunity to reuse entry, but extra index is needed. Dictionary entry with sequential ability could take advantage of frequency-used sequences. Several VLIW lines just need one codeword to represent. It reduces the sum of codeword and

dictionary entry could be used efficiently. But length slot is needed. Most research just take partial advantage: use greedy algorithm to find the most frequency sequences and sequentially put them into dictionary. We hope to take more advantage of this.



# Chapter 3 Design

We mainly present a heuristic algorithm for building Dictionaries. Goal of building dictionaries is present dictionaries for compression ratio. By benefit equation, we just allow most beneficial sequences exist in the dictionary. By combination, dictionary entry can be covered by different sequences to reduce repetition entry as possible. During building dictionary, compressed sequence in program is decided.

## Given Environment

We have OP and OPD dictionaries, and they have more chance to be inquired in the same size. It is low frequency that we can find two the same VLIW lines in the program. But separating one VLIW line into OP and OPD can take well use of dictionary entry. Compared with one dictionary, two dictionaries cost more space and logic but it gets worthy compression ratio.

We also adopt length slot is added to Codeword. It is commonly replacing compressing multiple with one codeword. Reduce the Number of codeword and Dictionary entry can be reused by different sequences.

For decoding efficiently, OP dictionary entry size is full OP of one VLIW line and the same as OPD. We add decompressor to decode codeword, and some performance-constrain must keep. We hope that the minimum ability of decompressor is to decode one codeword per cycle. One dictionary port can output one dictionary entry per cycle. If one VLIW line is separated into two parts and both parts are placed

in the same dictionary, two output ports of dictionary are needed. We don't assume dictionary hardware. Dictionary may be placed in the processor, memory, or some special hardware. So we don't know how many output ports which can supply. From conservative view, we assume that one OP dictionary entry size is total OP of one VLIW line. A sequence which length is two means this sequence has full OP or OPD of two VLIW line.

## Base idea

The key idea of the building algorithm is to select the most frequent sequences, which should be inserted into the dictionary, and reduce redundancy as possible.

Our goal is to build an efficient dictionary. An efficient dictionary could use each dictionary entry as well as possible. Partitioned dictionary-based is used and it is well done on many compression methods. Depending on the high frequency of VLIW line sequence, we want sequences can start at any entry of dictionary and end at any following entry.

The task of determining an optimal dictionary for a given text is known to be an NP-complete in the size of the text. However, many heuristics have sprung up that find near optimal solutions to the problem, and most are quite similar. The modified algorithm proceeds as follows.

We have OP and OPD dictionaries, and compression ratio is affected by both. When building dictionary, how to compress program will be decided. 1. We build one of each first and mark program which could be inquired in this dictionary. 2. Depending on marked program, we build another one dictionary. 3. After building dictionary, we compress program depending on mark. Some un-marked program may have change to be compressed lucky.

# 3.1 Build the dictionaries

In Figure we separate all VLIW lines sequences into opcode sequences and operand sequences. And build each one of OP or OPD dictionaries first. OPD dictionary pressure is always bigger than OP dictionary. In ADSP-21535 DSP, about 6.7~10.5% OP bit pattern dominate all program Ops. About 15.2~21.7% OPD bit pattern dominate all program OPDs. Build one dictionary first, and mark inquired-able sequences in the program. Another dictionary is built depending on marked program. Which dictionary should be built first will be discussed in Chapter 4. Now if we build OP dictionary first, we have an equation to judge a sequence is beneficial or not. After defining benefit, sequences are chosen in benefit order one by one.

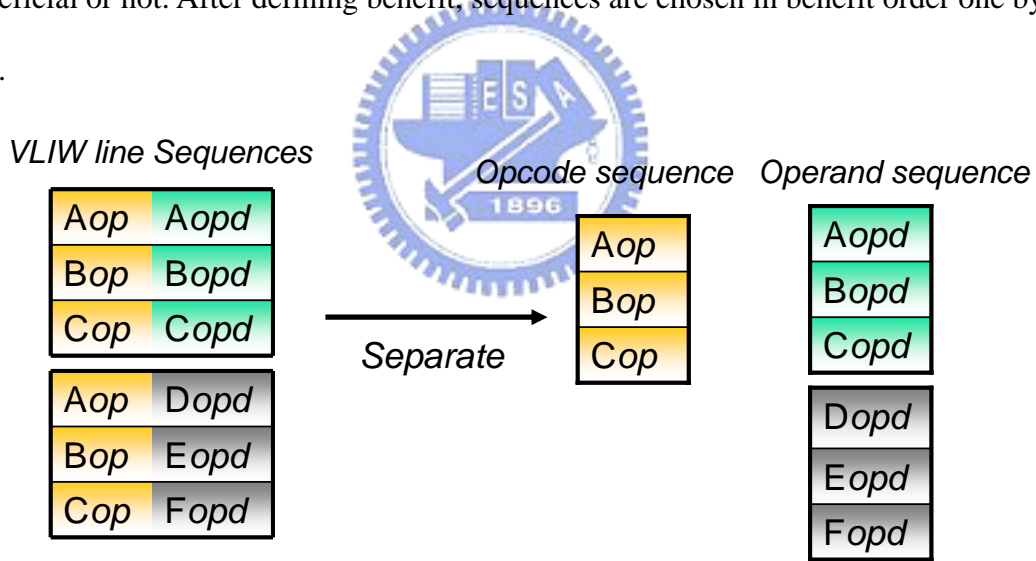
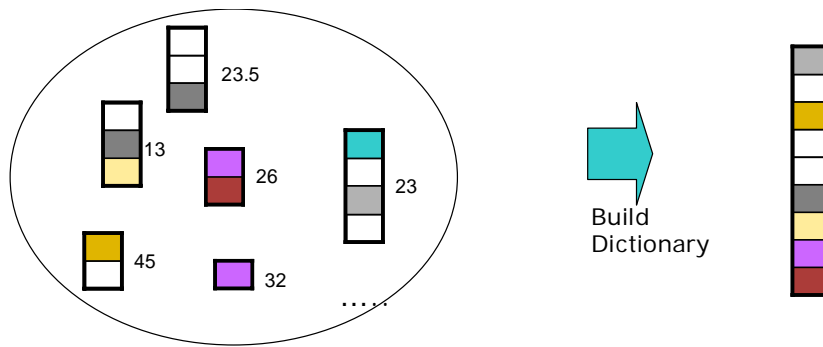


Figure 3-1 Separate VLIW line sequence.



Sequences look like sticks and each has a different color combination and value

Now we choose some sticks to make a new one

Figure:3-2 After defining benefit, sequences are chosen in benefit order one by one.

### 3.1.1 How to judge a sequence's benefit

We will use a benefit function to judge a sequence which should be inserted into dictionary or not. Benefit is that if the sequence is inserted into dictionary, how much memory requirement per dictionary entry which represents the sequence could be reduced.

First, we calculate how much memory requirement could be reduced by the Sequence. If it is an OP sequence, we just take care about OP in the program. Reduce-able memory requirement by the sequence is Size of OP program inquired by the one – Size of index and offset which present those OP program after compressing – Size of sequence. If another dictionary is built, we just calculate the program which is marked. OPD Benefit is

$$= \sum_{Occurrence} (OPD \text{ inquired sequence size} - OPD \text{ Codeword size}) - Dictionary \text{ Cost}$$

Then, in order to judge benefit between different lengths of sequences, we must take sequence length into consideration. So we add sequence length as denominator to calculate how much memory requirement can be reduced at per sequence length. But

in our dictionary, dictionary entry can be cover by different sequences. If partial of sequence are existed in the dictionary and can be reuse by this sequence, the benefit denominator of this sequence is just extra needed dictionary size. Benefit ratio is

$$= \frac{\sum (OPD \text{ inquired sequence size} - OPD \text{ Codeword size}) - \text{Dictionary Cost}}{\text{Dictionary Cost}}$$

So there will exists OP sequence S in the dictionary, sequence S' benefit = (Size of OP program inquired by the one - Size of index and offset which present those OP program after compressing - Size of sequence) / extra needed dictionary size.

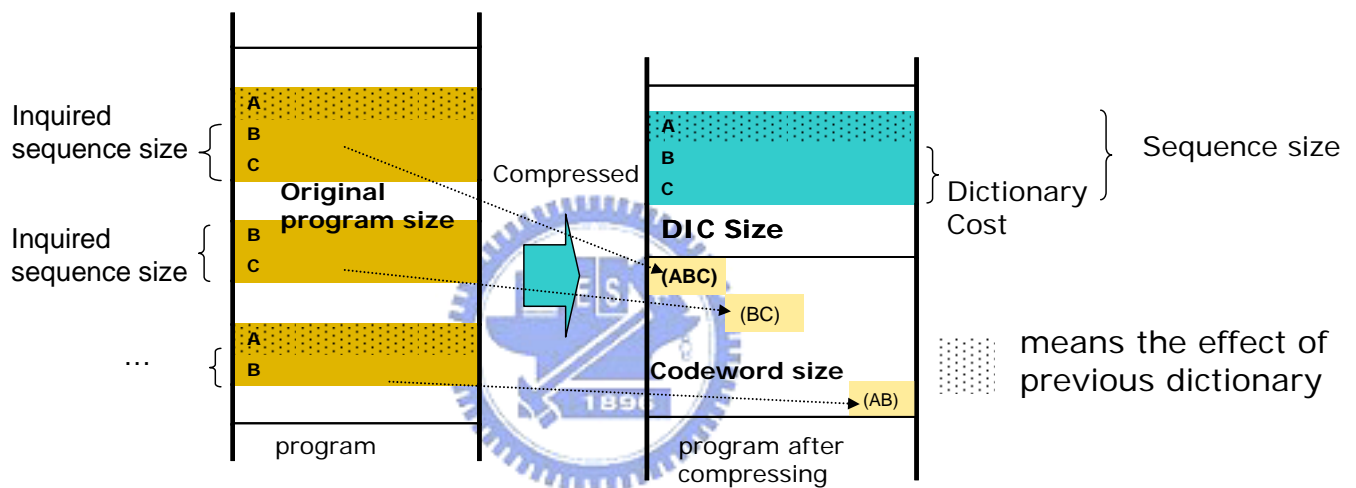


Figure 3-3: Benefit equation.

Reduce-able memory requirement: If the sequence S exists in the dictionary, the size of program which could be inquired by dictionary entry is reduce-able. If sequence S is ABC, reduce-able program size is the sum of total A, B and C in the OP part of program. This value shows that Max reduce-able memory requirement could be got by this sequence.

Codeword size: It is that how much codeword size is needed to represent reduce-able program. If three dictionary entries A, B, C, exist in the dictionary and are not placed sequentially, it would need three codeword to represent sequence ABC in the program. But if three VLIW lines are placed sequentially, we just need one



codeword to represent ABC. Therefore, the number of sequences and its sub-sequence will affect the sum of codeword.

Sequence length: It means how many VLIW lines is presented by this sequence. Long sequence has more chance to replace more sequence in the program, but it takes more dictionary size. To represent this situation, benefit must divide sequence length to generate correct value.

Benefit example is as follow:

$f(x)$  is the repetition of length  $x$ . If the opcode sequence is  $ABCD$ ,  $f(3)$  is the number of ABC and BCD in the program. We assume that  $f(4) = 2$ ,  $f(3) = 2$ ,  $f(2) = 0$ ,  $f(1) = 0$ , length = 4. So (Reduce-able Program size - Codeword size) = Opcode size of one line \* ( $f(4)*4 + f(3)*3 + f(2)*2 + f(1)*1$ ) - one codeword size \* ( $f(4) + f(3) + f(2) + f(1)$ ). Then that value which is divided by sequence length is benefit of sequence ABCD.



### 3.1.2 Build Dictionary flow

Now, we have benefit equation to judge sequence. “Candidate set”: sequences in candidate set have changes to enter chosen set. “Chosen set”: sequences in chosen set will load into the dictionary.

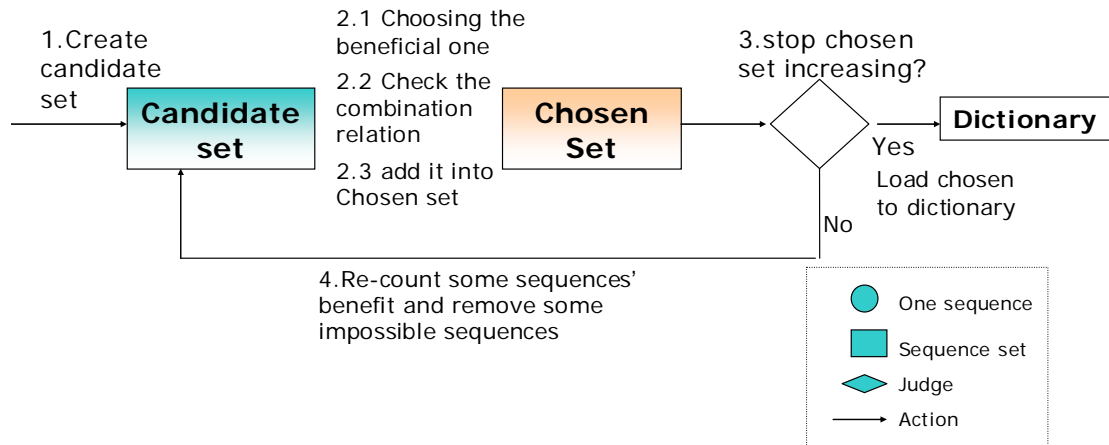


Figure 3-4 The flow of choosing the most beneficial sequences

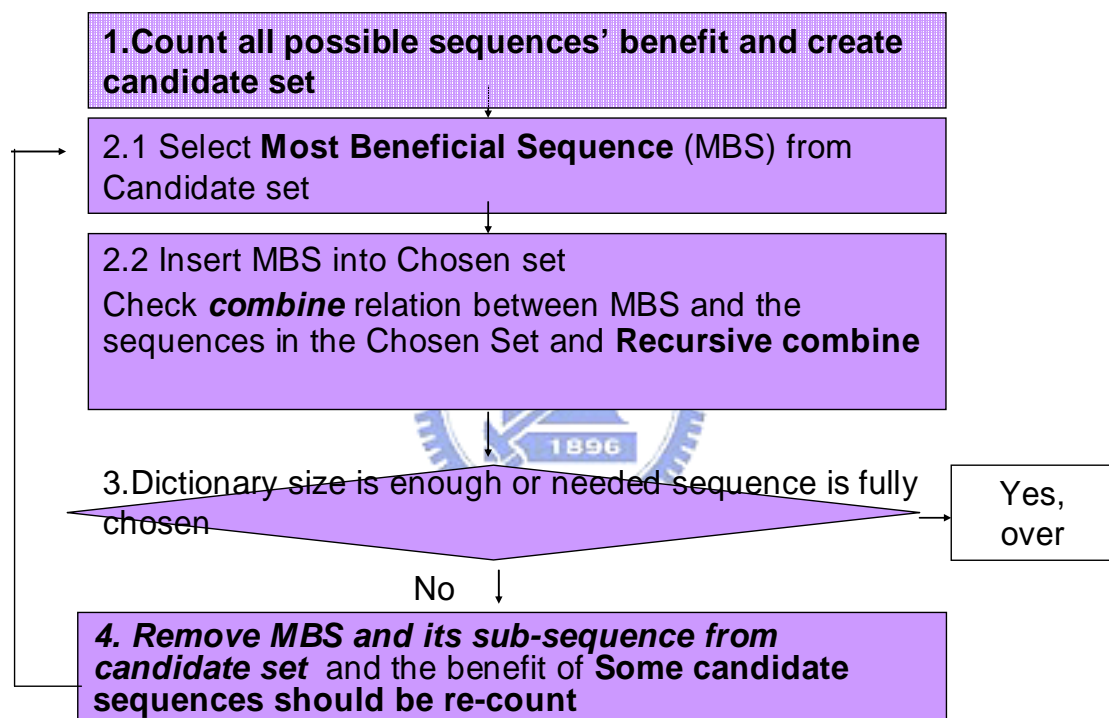
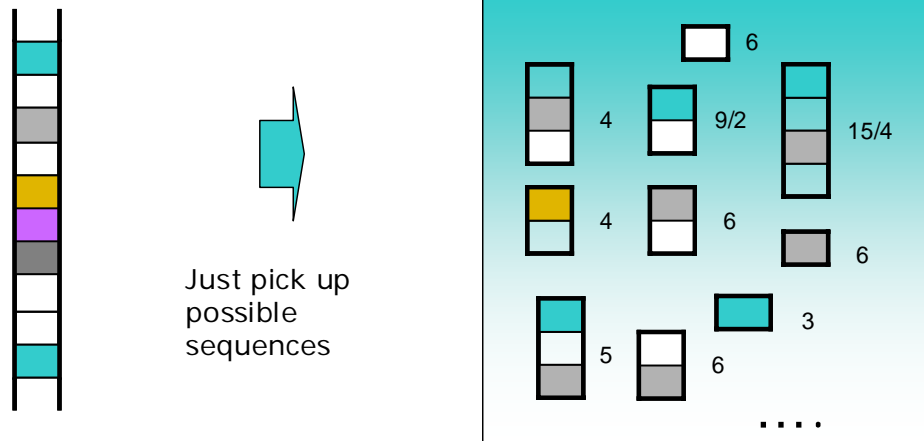


Figure 3-5 Flow of Choose algorithm

### 3.1.2.1 Create candidate sequence set



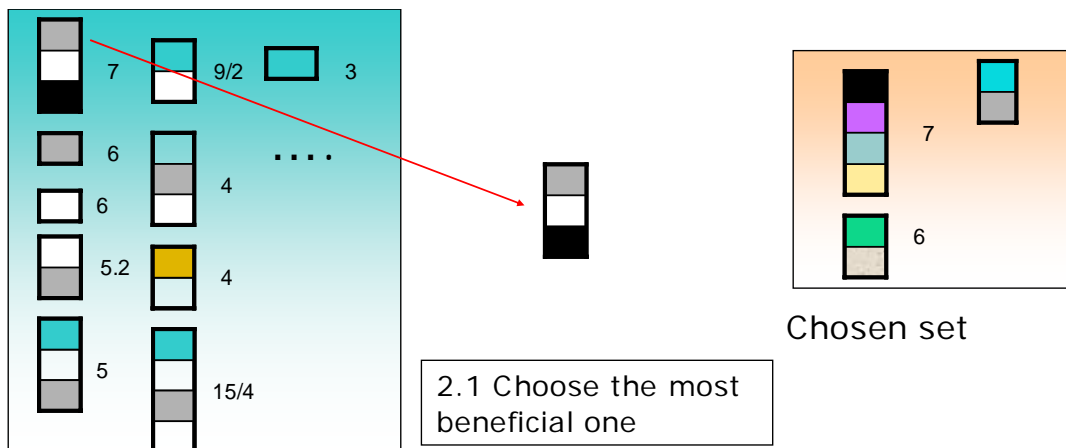
Original program

Candidate sequence set

Figure 3-6: Create candidate sequences

If this dictionary is built first, we try all possible combination. If one dictionary is built, other one's candidate set just consider program which is marked by previous dictionary. The Goal of candidate set is reduce computing time. We remove sequences which don't be considered, like sequence's benefit is smaller than 0. We also limit Max candidate sequence length. It is easier to simulate and we will test length from 1 to 8.

### 3.1.2.2.1 Choose the benefit one into chosen set



Candidate sequence set

Chosen set

Figure 3-7: Choose a beneficial one from candidate set

### 3.1.2.2.2 Combine relation

When one sequence is inserted into chosen set, it is a problem of repetition bit pattern. To avoid repetitions of bit pattern, we need combining sequences. By combining, the repetition could be reduced as possible. In dictionary, sequence ABC and BCD could be replaced by just ABCD (size=4), not ABC and BCD (size=6)

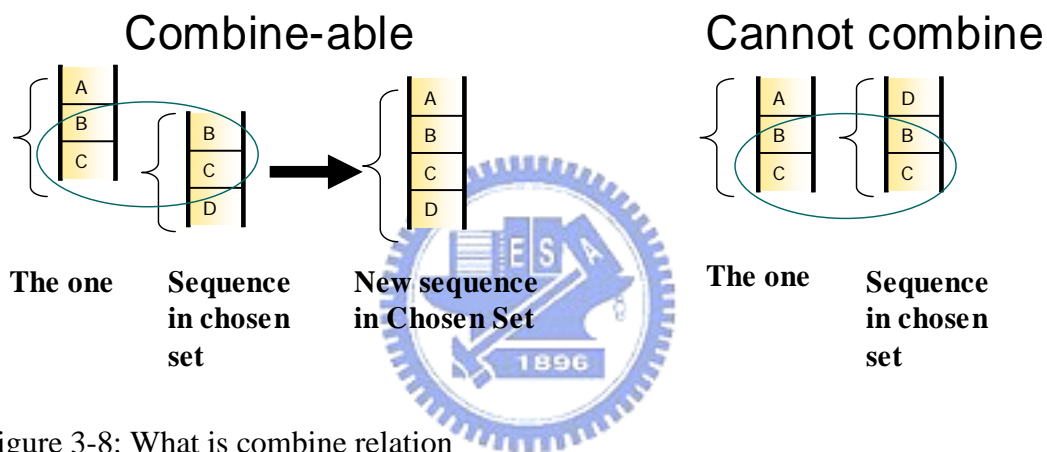


Figure 3-8: What is combine relation

If a new chosen sequence is created, the one will check combine relation with others sequences in chosen set until no possibility of combining.

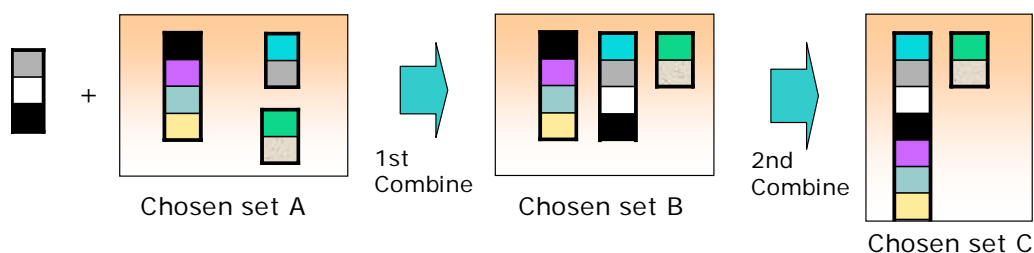


Figure 3-9 Recursive combine

### 3.1.2.3 Stop chosen set increasing

After combining, we check candidate and chosen sets' size. If chosen set's size arriving limited size or candidate set is empty, load chosen sequences into dictionary

in high benefit order. This is the last step to build dictionary. Else, re-count some sequences' benefit and repeat previous actions. If index has 8 bits, it means available dictionary size is 256 basic dictionary entry sizes. Sometimes we just need 200 entries, and strongly filling dictionary with 256 entries would cause adverse effects on compression ratio. By benefit, we can avoid this problem.

### 3.1.2.4 Re-count and remove impossible sequence

After inserting one sequence into Chosen Set, some sequences' benefit are not accurate. Ex: We suppose that BC is chosen and the benefit of ABCD, BCD, CDE and else should be re-count because their benefit has been take off a part by BC. So we recount some sequences' benefit.

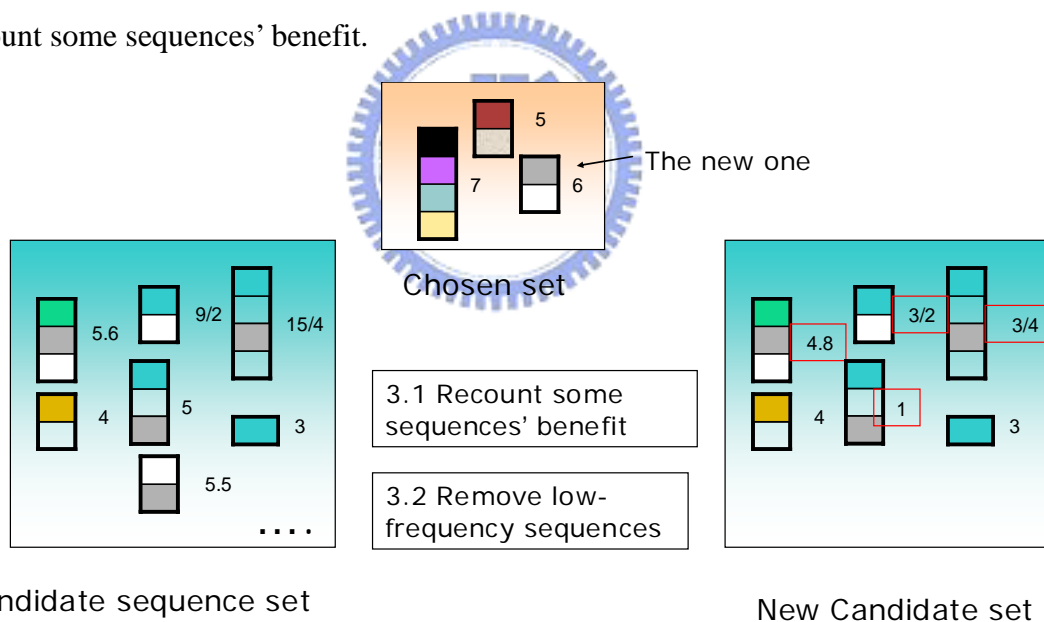


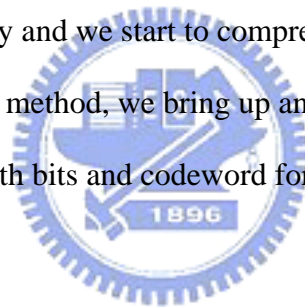
Figure 3-10 Re-count

Because of dictionary entry with sequential ability, long sequence could replace whole benefit of its sub-sequence. Remove the one which just is inserted into chosen set and its sub-sequence from candidate. So, if one sequence is inserted, its sub-sequences could be seemed as being inserted. If sequence ABC is inserted into B set, A, B, C, AB and BC would be removed from Set C. But why we need to re-count

some sequence's benefit? It is a problem that some sequence's benefit are not accurate after inserting one sequence into B Set. Because their partial benefit is taken off. For example, if ABC is inserted, any sequence, which includes A, B, C, AB and BC, should be re-count. Because sequence EBC includes sequence BC, the benefit of sequence EBC must remove the part benefit from sequence BC. By removing part benefit, the benefit of sequence EBC shows real value.

## 3.2 Replacing VLIW line sequence with codeword

Now, dictionary is already and we start to compress program. To improve traditional partition dictionary method, we bring up an idea about opcode or operand match. We also introduce length bits and codeword format in this chapter.



### 3.2.1 How to replace sequence with codeword

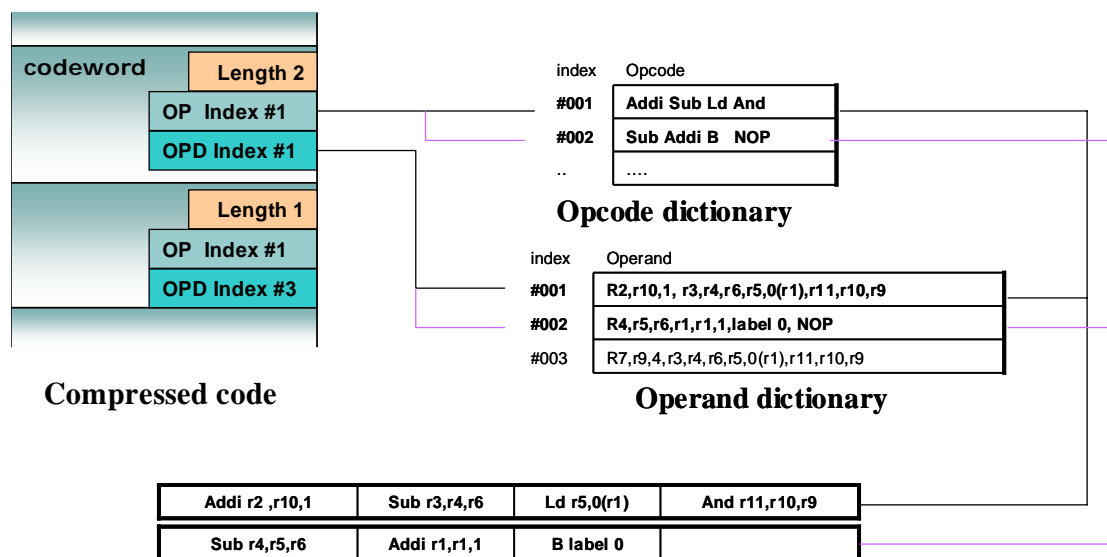


Figure 3-10 Replace VLIW line sequence with OP&OPD index and length

We assume that the VLIW line is split into several parts. The rule of traditional partitioned dictionary-based compression code method is that one VLIW line will be compressed only when all parts can be queried by dictionaries. In other words, if some part of VLIW line cannot be queried by dictionary, this VLIW line still can't be compressed. This limits the space of compression ratio. We hope to use dictionary more efficiently.

When compressing, we have some limits of VLIW line sequence. One is branch target only appear at the start of sequence. If branch target is not at start, the branch instruction would not jump to other instruction correctly.

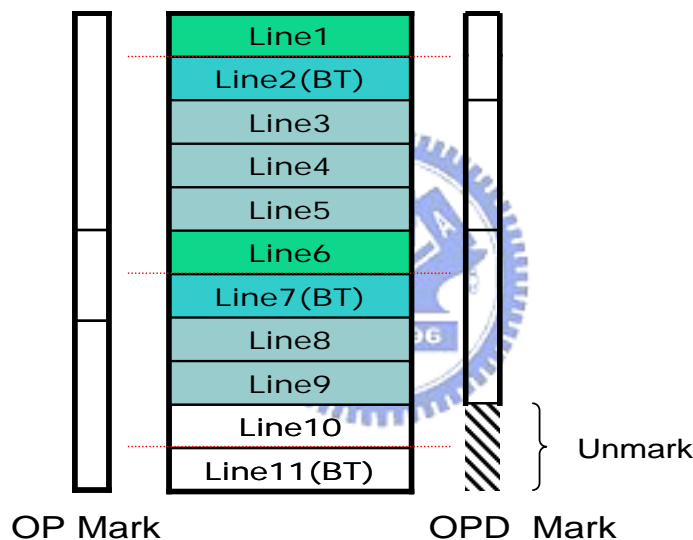


Figure 3-11 How to replace lines depending on Mark

In Figure 3-11, continuous mark means they can be inquire in dictionary continuously. We have two type marks, and continuous mark must be tied to scattered mark. After we compress program which is marked by both, program has been colored. Each continuous color means they can be just replace by one codeword. Program which is colored by white means they cannot be compressed because OP and OPD don't match at the same time.

### 3.2.2 Determine the Length Bits

Length bits determine Max compress length. In our design, length bit slot is added to codeword. Max compress length is that how many VLIW lines could be compressed by one codeword. Max compress length ( $= 2^{(\text{Length bits})}$ ) must be equal to or greater than candidate sequence length. The more length bit, the bigger codeword size. It is an overhead because that whatever sequence length any codeword need the same bits. But the more length bit, the longer the possible sequence. It is good for compression ratio because we could use just one codeword to replace long sequence not several. That is a tradeoff about length bits. Most branch target VLIW lines are at a distance of three or four VLIW lines. Longer sequence cannot get more advantage. So, we will test the length bits from 0 to 2. When building the dictionary, the benefit function is just designed for one candidate sequence length. If max compress length is not equal to candidate sequence length, the benefit is not correct and dictionary is not efficient, too. And, after experiment, we have proved this point. So, candidate sequence and length bits should be the same. We will test length bits from 0 to 2 and candidate sequence length is  $2^{\text{length bits}}$ .

### 3.2.3 Codeword format

We use two type codeword to represent program. Many compressing paper have many definition of codeword. Whole compressed program size is the sum of codeword and un-compressed code. But here, we use a type of codeword to represent un-compressed code.

After compressing, we need a tag slot to separate different type codeword. One bit is added to codeword and it different two type codeword: Both match and program which don't be compressed.



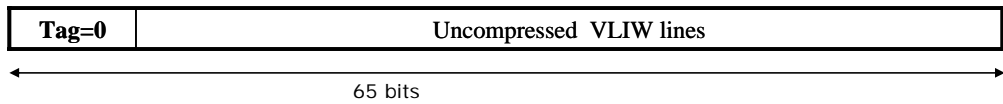
Both match codeword: this codeword includes a tag, length slot and opcode index and operand index.

Program which doesn't be compressed: a tag is added in the front of original VLIW line.

Compressed : *OP and OPD can both inquired by dictionaries*



Code which don't be compressed



$$\text{Round down (Compressed codeword length/un-compressed)} = 3$$

Figure 3-12: Codeword format

## 3.3 Place codeword into memory

One obvious side effect of a dictionary-based code compression scheme is that it alters the locations of instructions in the program. This presents a special problem for branch instructions, since branch targets change as a result of program compression.

To avoid this problem, we do not compress relative branch instructions (i.e. those containing an offset field used to compute a branch target) and leave those instructions in the compressed code instead of in the dictionary. After compression, the targets of branch instructions are patched to the new location in the compressed code.

Some branch target address is got from data memory. These instructions can be seemed as a normal instruction because its value doesn't need change. But after compression, we should modify the target address in data memory. I assumes that memory is **byte-align**. The codeword, which includes **branch target or PC-relative**

**branch instruction** must be byte-aligned. So some space cannot be used. Other codeword follow previous codeword successively



# Chapter 4 Simulation

## 4. Platform and Benchmark

We adopt ADSP-21535 DSP as our simulation platform.

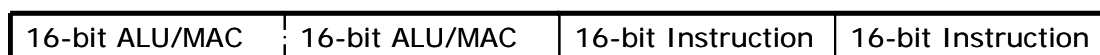


Figure 4-1 VLIW format ADSP-21535 DSP

One VLIW line has four slots. First two instructions can work as MIMD. They are ALU/MAC and control flow instructions. Last two instructions handle Immediate add, decrement, increment and memory access.

Benchmark: Programs for this DSP (download from <http://www.analog.com>) include Mpeg4 decoder, jpeg2000, FFT, Full-search mpeg2, r8x8invdct, corr\_3x3, dilation and isadc. Program is written by C and inline assembly. The codes are compiled by the Visual DSP++ 3.0 with Enable Optimization and Interprocedural Optimization and using hand code library.

By memory dump, we get all branch targets, machine code, and its assembly. This is offline work. If figure is without special mark which shows which benchmark is used, it is average of all benchmark.

# 4.1 The effect of Max compress length

We test all benchmark and found that the lowest compression is always at dictionary 4K byte. 4K means OP and OPD index's index-able size, and don't mean that we must exhaust full 4K dictionary. Some bigger benchmark, like Mpeg4 or JPG, have the best compression ratio at max length =4. And some smaller program prefer max length =2. Others max compress length all cannot get better compression ratio than 2 and 4.

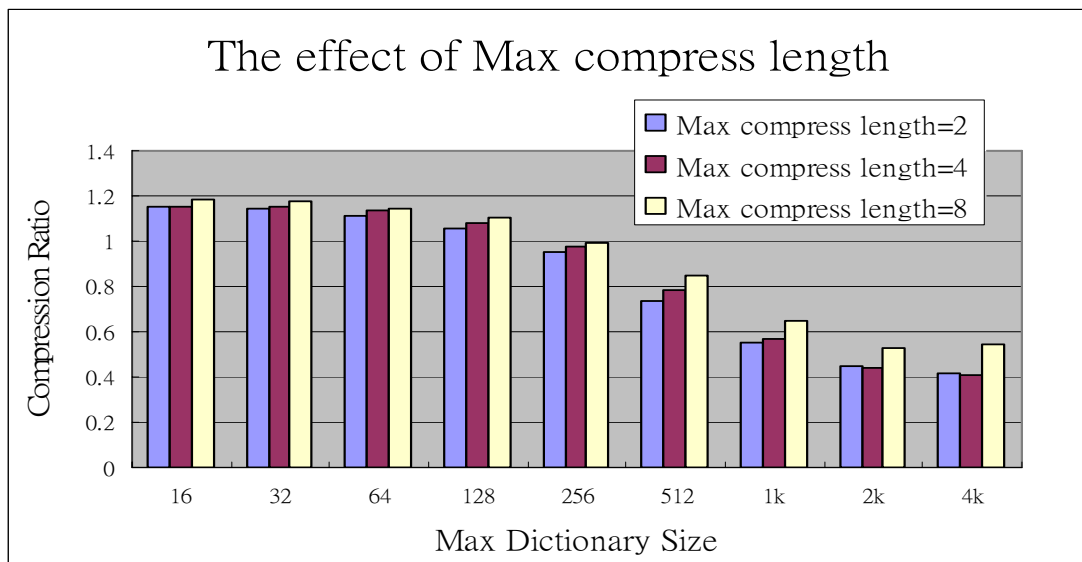


Figure 4-2: The effect of Max compress length

When length slot bit is one or two, we can get better compression. We don't show the situation which is without length slot because that always gets higher compression ratio than the situation with lengths slot. And adding length slot, we also need to consider the overhead by decompression machine.

Because branch target appear every four or five lines, we can easily explain why one or two length slot bits is better.

## 4.2 Compare with optimal situation

We mainly present a heuristic algorithm for VLIW code compression and we try to compare with optimal situation under our hardware constrain. Contract's complexity is  $O(N^M)$ ,.  $N$  is all possible OP bit pattern and  $M$  is the number of entries. We test all dictionary combination and all compressing possible. Because of branch target, each time we just need to search small area to find optimal compressing sequence. In Max 4K dictionary size (OP+OPD) and Max compress length= 4, on average contract's compression ratio have advantage about 1.78%. But it pays more time to compute. Our approach's complexity is  $O(N^2)$ .

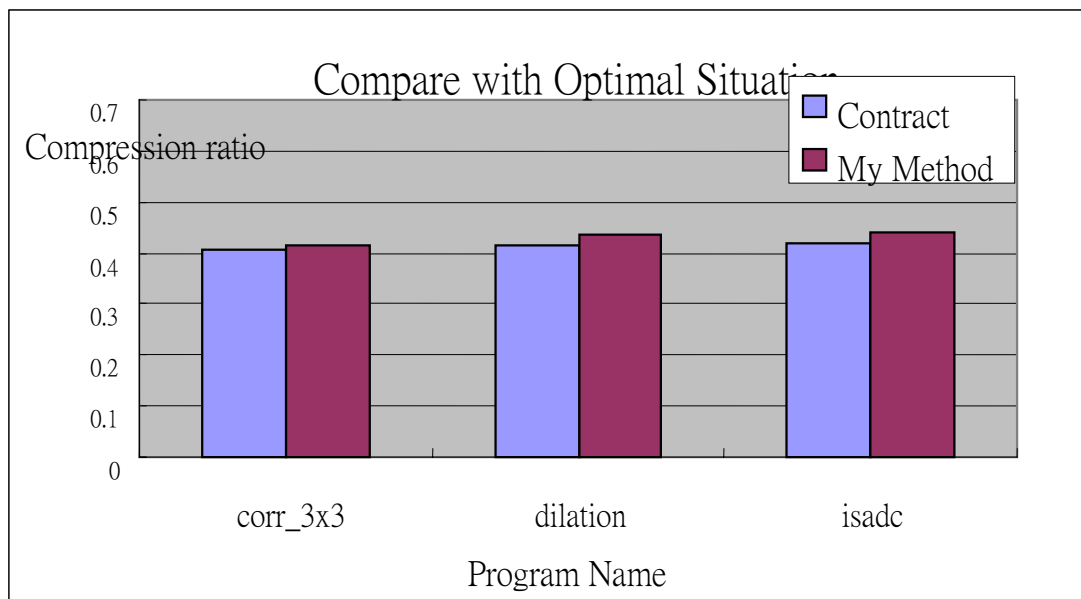
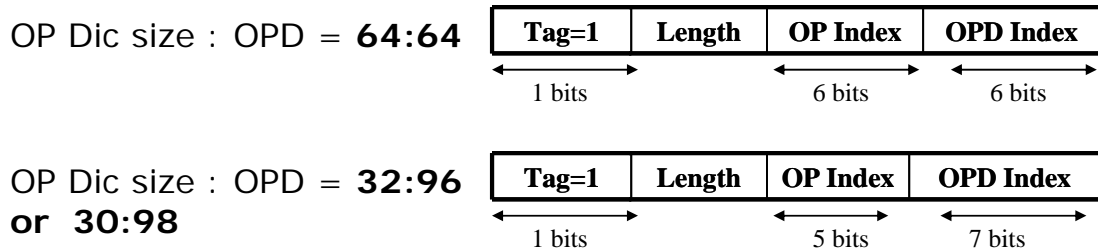


Figure 4-3 Compare with optimal situation

## 4.3 The effect of OP and OPD dictionary with different sizes

When dictionary size increases, compression ratio would be limited by OPD dictionary. We hope to increase OPD dictionary, but wouldn't increase total dictionary size. By half OP dictionary size, we increase OPD dictionary size, and keep the same

total dictionary size and codeword bits.



Ex .Codeword format

We remove 1 codeword bit from OP index to OPD index in example. Original Op index is 6 bits, and we cut 1 bit from OP to OPD. In this case, total codeword bits don't increase and available OPD dictionary size increase. If available OP dictionary size is always the same as OPD, OP index may be not efficient. OP dictionary pressure is always lower than OPD. OPD dictionary has size pressure. So we release OP index to support OPD index and codeword length would not be changed. And OPD dictionary size pressure could be solved.

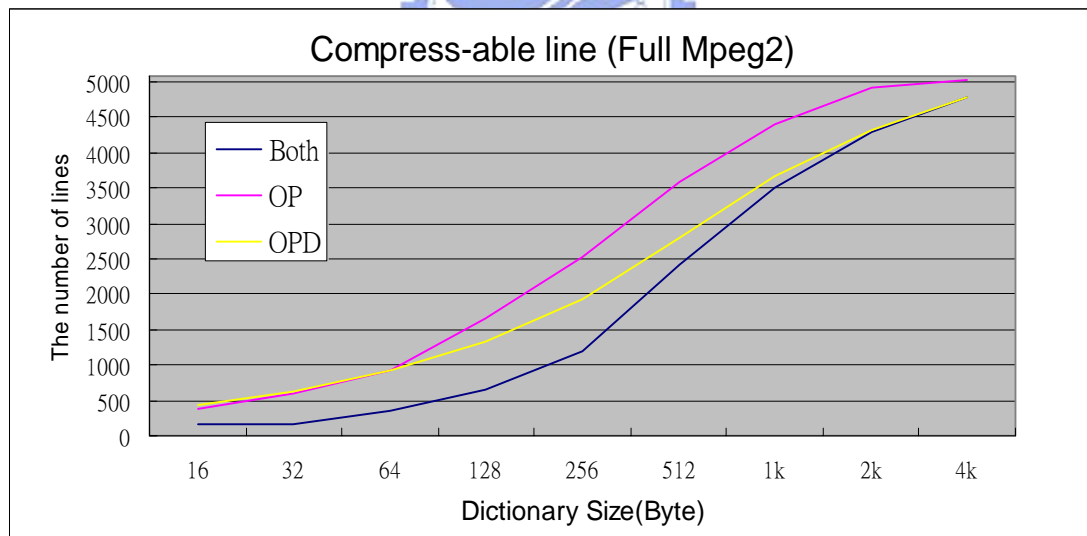


Figure 4-4 OP and OPD bit pattern program cover ratio

About 6.7~10.5% OP bit pattern dominate all program Ops. About 15.2~21.7% OPD bit pattern dominate all program OPDs. Considered that just program which could be inquired by both dictionaries can be compressed, it is better to balance percentage of OP and OPD inquired program. In figure4-4, the ratio of program which

is dominated by 1k OP dictionary is similar as 2k OPD dictionary.

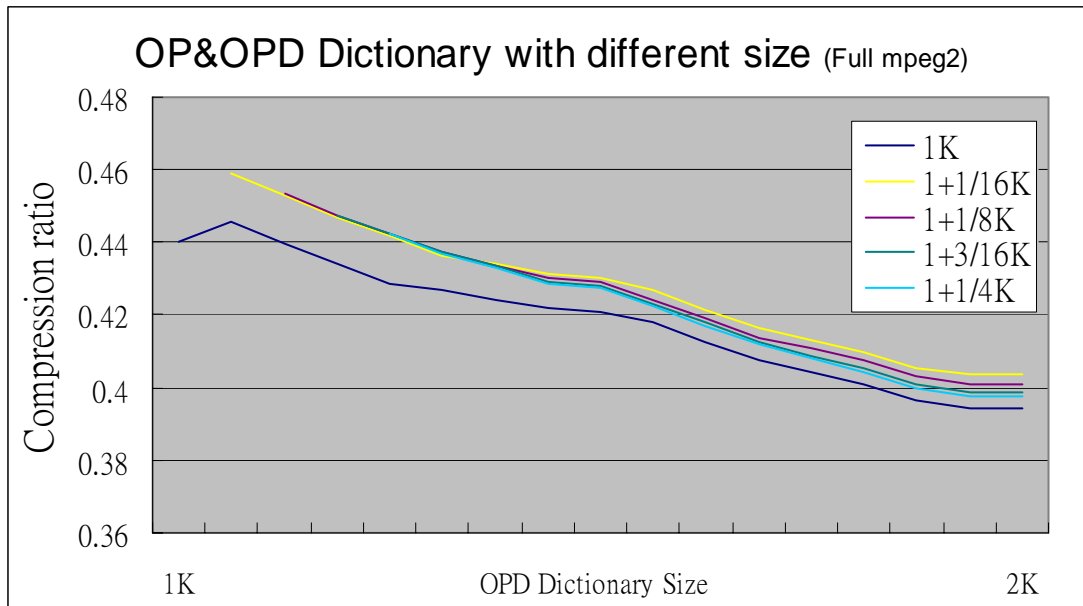


Figure 4-5 Compared with giving enough dictionary size, it has better cut some dictionary size to reduce codeword length.

In figure4-5, different curves mean different OP dictionary size. Because small dictionary size has no use value, we test size from 1k OP and OPD size. Lower compression ratio is better. We can find that best compression ratio is appear at 1K OP dictionary size and 2K OPD dictionary size. Different sizes of OP and OPD are useful and needed.

# Chapter 5 Conclusion

In this thesis, we proposed a heuristic algorithm to compress VLIW programs by partitioned dictionary and codeword with length slot. The key idea of this method is to separate the original instruction sequences into the opcode sequences and the operand list sequences, and then more repetitions of these sequences in the programs can be exploited. A benefit equation is proposed to illustrate how to build a partitioned dictionary and two related issues, replacements of common sequences and collaboration between opcode and operand dictionaries.

There are several directions that our compression method could be further improved. First, we can reduce bit toggles for power efficiency. This method is off-line, and has no overhead on hardware. By some profiles, we can rearrange the placement of codeword and dictionary to reduce bit toggles during fetching codeword and decompression. When fetching codeword, we can place frequent-used continuous codeword with fewer bit toggles. As inquiring dictionary, continuous index's bit toggles could be reduced too.

Second, when dictionary-based code compression is adopted on VLIS (various length instruction set), how dictionary we design to suit it. Because VLIW line length could be one to eight instructions, dictionary entry which size is eight instructions may be not suitable. Multiple ports and variable dictionary entry size are feasible on VLIS. Multiple output ports provide the ability to inquire two dictionary entries at the same time but it will lead to increase dictionary size greatly. Variable dictionary entry



has ability to change basic entry size by hardware logic. Depending on program character, we can take two instructions or four instructions as our basic dictionary entry size. If one VLIW line's length is over dictionary basic size, it may need special method to handle like adding output ports, changing codeword format or using two codeword.



# REFERENCES

- [1] Intel Web Site, <http://www.intel.com>
- [2] Jim Turley, “Code compression under the microscope”, Embedded Systems Programming, February, 2004.
- [3] M. Kozuch and A. Wolfe, “Compression of Embedded System Programs”, IEEE International Conference on Computer Design, 1994.
- [4] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, “Code Compression Based on Operand Factorization”, 31<sup>st</sup> Annual ACM/IEEE International Symposium on Microarchitecture, 1998.
- [5] A. Wolfe and A. Chanin, “Executing Compressed Programs on an Embedded RISC Architecture”, Proceedings of the 25<sup>th</sup> Annual International Symposium on Microarchitecture, December 1992.
- [6] Yuan Xie, Princeton University, Princeton, NJ, USA, Wayne Wolf Princeton University, Princeton, NJ, USA, Haris Lekatsas, NEC USA, Princeton, NJ, USA, “Code compression for VLIW processors using variable-to-fixed coding”, ACM Special Interest Group on Ada Programming Language IEEE-CS/DATC : IEEE Computer Society, 2002
- [7] Sang-Joon NAM, In-Cheol PARK, and Chong-Min KYUNG, “Improving Dictionary-Based Code Compression in VLIW Architectures”, Special Section on VLSI Design and CAD Algorithms, NOVEMBER 1999.
- [8] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, “Improving Code Density Using Compression Techniques”, Proceedings of the 30<sup>th</sup> Annual International

Symposium on Microarchitecture, December 1997.

- [9] Chang Hong Lin, Yuan Xie, Wayne Wolf, “LZW-Based Code Compression for VLIW Embedded Systems”, 2003.
- [10] M. Benes, A. Wolfe, S. M. Nowick, “A High-Speed Asynchronous Decompression Circuit for Embedded Processors”, Proceedings of the 17<sup>th</sup> Conference on Advanced Research in VLSI, September 1997.
- [11] S. S. Gupta, D. Das, S.K. Panda, R. Kumar and P. P. Chakrabarty, “Code Compression for RISC Processors with Variable Length Instruction Encoding”, hipc2003, 2003
- [12] Guido Araujo, Paulo Centoducatte, Rodolfo Azevedo and Ricardo Pannain., “Expression tree based algorithms for **code compression** on embedded RISC architectures”, IEEE Trans. VLSI Systems, October 2000.

