PAPER *Special Section on VLSI Design and CAD Algorithms*

Improving Dictionary-Based Code Compression in VLIW Architectures

Sang-Joon NAM[†], **In-Cheol PARK**[†], *and* **Chong-Min KYUNG**[†], ***Nonmembers***

SUMMARY Reducing code size is crucial in embedded systems as well as in high-performance systems to overcome the communication bottleneck between memory and CPU, especially with VLIW (Very Long Instruction Word) processors that require a high-bandwidth instruction prefetching. This paper presents a new approach for dictionary-based code compression in VLIW processor-based systems using isomorphism among instruction words. After we divide instruction words into two groups, one for opcode group and the other for operand group, the proposed compression algorithm is applied to each group for maximal code compression. Frequently-used instruction words are extracted from the original code to be mapped into two dictionaries, an *opcode dictionary* and an *operand dictionary.* According to the SPEC95 benchmarks, the proposed technique has achieved an average code compression ratio of 63%, 69%, and 71% in a 4-issue, 8-issue, and 12-issue VLIW architecture, respectively.

key words: *code compression, VLIW architecture*

1. Introduction

In VLIW (Very Long Instruction Word) architectures where a high-bandwidth instruction prefetch mechanism is required to supply multiple operations per cycle, reducing the code size is crucial to overcome the communication bottleneck between memory and CPU. Moreover, code size becomes a very important issue in VLIW processors used in embedded systems as well as in high-performance systems [1].

Since the cost of an integrated circuit is more than linearly proportional to the die size, reducing the program size that directly implies cost reduction is very crucial in processor-based embedded systems. As the complexity of embedded systems grows, programming in assembly language followed by optimization by hand is no longer deemed practical, except for time-critical portions of the program. Recent statistics indicate that high-level languages such as C are gradually replacing assembly language as a programming language, because using high-level languages greatly lowers the cost of development and maintenance of embedded system. However, as the programming in high-level language can incur a penalty in code size, securing a means to compress the instruction code becomes extremely important.

Several dictionary-based code compression schemes have been proposed to enhance the code compression ratio in the embedded systems. Liao [2] proposed a software method for supporting code compression using the *mini-subroutine* which is a procedure call representing a common sequence of instructions in the DSP program. Each instance of a mini-subroutine is removed from the program and replaced by a *call-dictionary instruction*, which increases the overall processing latency. Liao's method can reduce the code size by 12% on average.

While Liao used a fixed-length call-dictionary instruction as an index into the dictionary table, Lefurgy [3] has used a variable-lengthcodeword which improves the dictionary-based code density by 33%. However, Lefurgy's method is only applicable to single-issue processors and limited to the exactly identical instructions. Two instruction words are denoted as *identical* if their opcodes and their operands are exactly the same.

In another scheme based on the *object code compression* proposed by Yoshida [4], a lookup method based on ROM table is used to recover the original object code from each compressed object code (pseudocode). The code compression ratio achieved by this approach is 62.5% on average. However, this scheme is only applicable to single-issue processors and suffers from the large object transformation table having thousands of entries which results in long table-lookup latency.

Ishiura [5] has converted the problem of finding a good instruction encoding for code compression to the problem of *instruction field partitioning* in VLIW architectures. Because the instruction encoding depends on the analysis of compiled code and field partitioning, decoder and control unit can be implemented only after the implementation of system software. This scheme reduces the code size to 46 $\sim$ 60%, but the decoding through the table lookup operation is rather complex and slow.

In this paper, we propose a new technique to reduce the code size in VLIW processor-based systems. The proposed approach is based on the dictionary-based code compression but we extend the concept by employing the so-called isomorphism among instruction words. *Isomorphic* instruction words are those having the same opcode with slightly different set of operands, or the same set of operands with different opcodes. Compared to Ishiura's method which is applied to the full set of instructions, our method is applied only to

Manuscript received March 15, 1999.
Manuscript revised May 31, 1999.

[†]The authors are with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, 373–1, Kusong-dong, Yusong-gu, Taejon, 305–701, Korea.

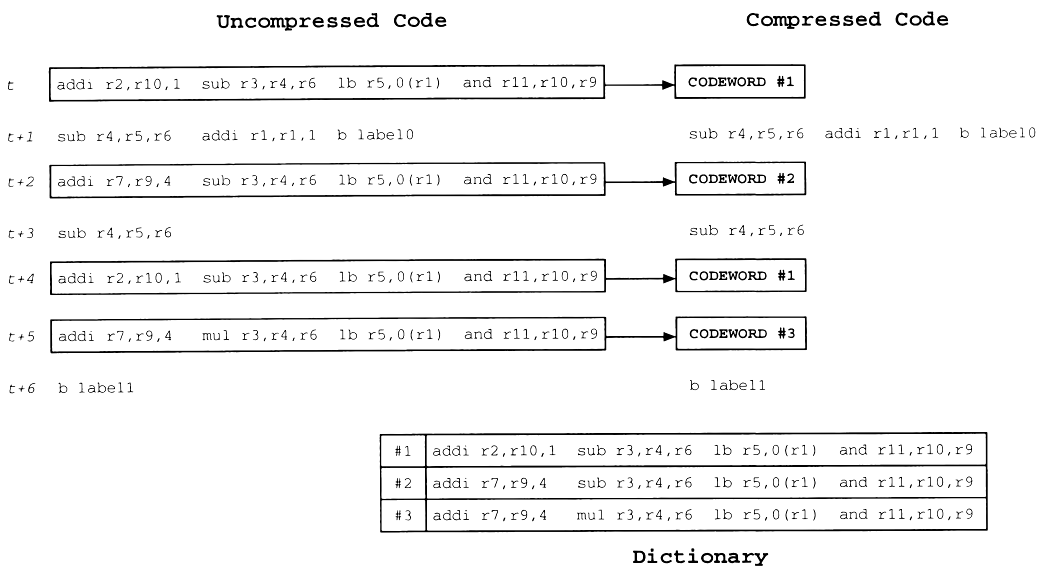

Uncompressed Code
Compressed Code
t addi r2,r10,1 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
CODEWORD #1
t+1 sub r4,r5,r6 addi r1,r1,1 b label0
sub r4,r5,r6 addi r1,r1,1 b label0
t+2 addi r7,r9,4 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
CODEWORD #2
t+3 sub r4,r5,r6
sub r4,r5,r6
t+4 addi r2,r10,1 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
CODEWORD #1
t+5 addi r7,r9,4 mul r3,r4,r6 lb r5,0(r1) and r11,r10,r9
CODEWORD #3
t+6 b label1
b label1
#1 addi r2,r10,1 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
#2 addi r7,r9,4 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
#3 addi r7,r9,4 mul r3,r4,r6 lb r5,0(r1) and r11,r10,r9
Dictionary

Fig. 1 Compression using identical instruction words.

frequently-used instructions. We extract isomorphic instruction words as well as identical instruction words that are frequently used in the program and store their opcodes and operands into two dictionaries as a compressed form. They are determined after the analysis of compiled code, but do not change the instruction encoding which results in simple decoding and table lookup operation. As the compressed instruction words are fetched, they are dynamically decompressed by referring to the two dictionaries. We present a technique using the dictionary-based code compression as a good trade-off between code compression ratio and decoding delay.

The organization of this paper is as follows. Section 2 explains the code compression schemes using identical or isomorphic instruction words. In Sect. 3, we describe the proposed compression algorithm using isomorphism among instruction words. Experimental results using various machine configurations are shown in Sect. 4.

2. Code Compression Using Instruction Isomorphism

A compressed form of an original code can be represented by a *dictionary* and a *codeword*. The dictionary contains instruction words frequently used in the original code and is accessed at the time of program execution to expand the accessed codeword into the corresponding uncompressed instruction word. The codeword is a symbol pointing to an entry of the dictionary, and replaces the instruction word to be compressed. In other words, the instruction words frequently used are extracted and stored in the dictionary and the occurrence of these instruction words is replaced by a codeword.

The instruction words frequently used can be classified into two groups: *identical instruction words* and *isomorphic instruction words*. But, identical instruction words are rarely found in VLIW processors, because an instruction word consists of multiple operations. Figure 1 illustrates the relationship between the uncompressed code, the compressed code, and the dictionary with assuming a 4-issue processor. Instruction word $t+2$ is the same as instruction word t except for the operands in the first operation. So, instruction word $t+2$ is represented by another "CODEWORD #2" pointing to the second entry in the dictionary, resulting in an increase of the dictionary size. Again, instruction word $t+5$ is represented by "CODEWORD #3", although it is similar to instruction word $t+2$ except the "mul" operation.

Therefore, to increase the access frequency of the dictionary and enhance the code compression ratio, we use the isomorphism among instruction words. To support isomorphism among instruction words, we classify instruction words into two groups, an *opcode group* and an *operand group*. Figure 2 shows how the isomorphism is applied to instruction words. In this scheme, the isomorphism among instruction word t, $t+2$, and $t+5$ is detected and represented by a codeword. As we find more isomorphism relations, the total code size becomes smaller than that in Fig. 1.

While one dictionary is enough for compressing identical instruction words, two dictionaries (*opcode dictionary* and *operand dictionary*) are used in this scheme as shown in Fig. 3. Opcode_words and operand_words assigned by the compression algorithm are used as the indices of the opcode dictionary and operand dictionary, respectively. The decoder checks the incoming instruction words to determine whether they are compressed or not. For an uncompressed in-

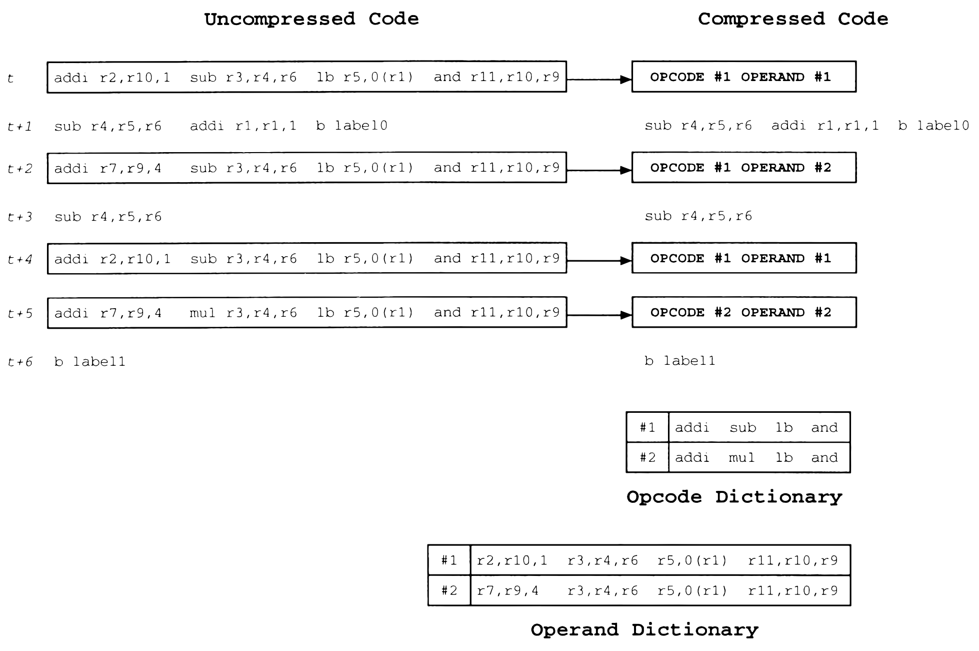

Uncompressed Code
Compressed Code
t
addi r2,r10,1 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
OPCODE #1 OPERAND #1
t+1
sub r4,r5,r6 addi r1,r1,1 b label0
sub r4,r5,r6 addi r1,r1,1 b label0
t+2
addi r7,r9,4 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
OPCODE #1 OPERAND #2
t+3
sub r4,r5,r6
sub r4,r5,r6
t+4
addi r2,r10,1 sub r3,r4,r6 lb r5,0(r1) and r11,r10,r9
OPCODE #1 OPERAND #1
t+5
addi r7,r9,4 mul r3,r4,r6 lb r5,0(r1) and r11,r10,r9
OPCODE #2 OPERAND #2
t+6
b label1
b label1
#1 addi sub lb and
#2 addi mul lb and
Opcode Dictionary
#1 r2,r10,1 r3,r4,r6 r5,0(r1) r11,r10,r9
#2 r7,r9,4 r3,r4,r6 r5,0(r1) r11,r10,r9
Operand Dictionary

Fig. 2 Compression using isomorphic instruction words.

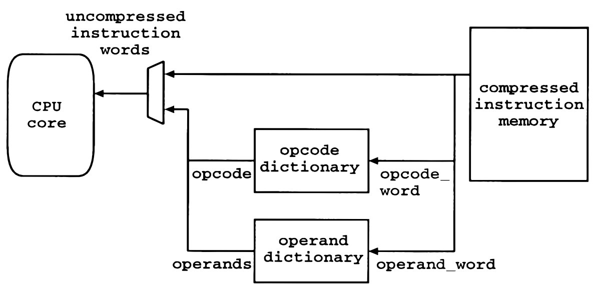


Fig. 3 Instruction fetch path in the proposed code compression scheme for VLIW processor-based systems.

struction word, it proceeds in a conventional fashion through the upper path. When the decoder encounters a compressed instruction word, it is recovered to the uncompressed one through the lower path by retrieving the original opcodes and operands concurrently from the corresponding entries of the dictionaries pointed to by the opcode_word and the operand_word. Then the uncompressed instruction word is issued to the functional units.

To reduce the latency due to the extra table lookup, we can store predecoded control information in the dictionaries rather than the plain uncompressed opcodes and operands. Using this predecoded information, we can save one decoding cycle for compressed instruction words and make up for the cycle needed to lookup the dictionaries, but this method requires a large ROM table. The predecoded information is similar to microcode and hence its length is determined by implementation details. In our implementation, to make the table lookup operation simple and to minimize the additional latency due to this lookup, the length of opcode_word and the operand_word is fixed to enable them to be directly used as the indices of the dictionaries.

3. Compression Algorithm for Instruction Isomorphism

A dictionary-based compression algorithm is applied after a compiler has generated a program. We search the program to find the isomorphic instruction words frequently used, and their opcodes and operands are placed in the opcode dictionary and operand dictionary, respectively. Our algorithm has 2 steps as follows.

1. Building entries of two dictionaries
2. Replacing instruction words with the opcode_words and operand_words

3.1 Building Entries of Two Dictionaries

Building a dictionary that can achieve maximum compression is known as an NP-complete problem [6]. Most previous dictionary methods for compressing instruction sequences use greedy algorithms to quickly determine the dictionary entries. After the usage frequency of each instruction word is examined, an entry in the dictionary is allocated for the instruction word if the instruction sequence has a high probability of occurrence. The allocation is governed by the constraints such as the maximum number of entries in each dictionary, threshold of frequency and the encoding scheme for codewords. However, these methods do not allow branching into the interior of an instruction sequence pointed to by a codeword.

But, our code compression scheme replaces an instruction word by an opcode_wordand an operand_word, and limits their total bit-width to be the same as that ofan normal operation. Thus, the maximum compression problem is changed from an NP-complete problem to a simple greedy one. First, we determine whether two instruction words are isomorphic. If they are isomorphic, we regard the corresponding opcode_word and operand_word as candidate entries for two dictionaries. In this procedure, if the opcode group or the operand group of the isomorphic instruction words already has the candidate entry, its frequency is incremented. After making the candidate entry set, the gain of each element in candidate opcode set is calculated and elements are sorted in an increasing order. Finally, the number of dictionary entries and total code size are constrained by the given maximum number of dictionary entries. The whole selection algorithm for isomorphic instruction words in the proposed scheme is presented as a pseudo code in Fig. 4.

Moreover, since operations stored in an entry of a dictionary are for one instruction word and should be issued concurrently, there is no problem on the branching in our code compression scheme.

3.2 Replacing Instruction Words With the opcode_words and operand_words

The occurrence of each opcode and operand in the entry of two dictionaries is simply represented by a fixed-length opcode_word and a fixed-length operand_word. Specifically, the total bit width as required for the opcode_word and operand_word is made equal to that of an uncompressed operation in order to align the compressed instruction words with the cache boundary. This can result in worse compression than a variable-length opcode_word and operand_word encoding, but makes instruction-fetching and decoding mechanism simple and fast. In general, variable-length encodings such as Huffman encoding are expensive to decode [7]. A fixed-length encoding, on the other hand, can be used directly as an index into the dictionary, making the table lookup operation simple.

```
candidate opcode set(CL) = NULL;
candidate operand set(OL) = NULL;
foreach i in instruction words {
   foreach j in instruction words {
      if (i and j are isomorphic) {
         if (opcode group(i) or operand group(j) does not exist
          in CL or OL) {
            Insert opcode group(i) or operand group(i) into CL or OL;
         } else {
            frequency(opcode group(i))++;
            frequency(operand group(i))++;
         }
         if (opcode group(j) or operand group(j) does not exist
          in CL or OL) {
            Insert opcode group(j) or operand group(j) into CL or OL;
         } else {
            frequency(opcode group(j))++;
            frequency(operand group(j))++;
         }
      } /* end of isomorphic */
   } /* end of j */
} /* end of i */

foreach i in CL {
   gain(i) = (opcode size(i) - 1) * frequency(opcode group(i));
}

Sort elements of CL in increasing order of their gain;
count = 0;
opcode dictionary = NULL;
operand dictionary = NULL;
while (count < MAX_ENTRIES) {
   Insert {count}'th opcode group of CL into opcode dictionary;
   Insert operand group whose opcode group is {count}'th opcode group
     into operand dictionary;
   count++;
}
```

Fig. 4 Algorithm for selecting isomorphic instruction words.

4. Experimental Results

The experimental environment used in our work consists of the SHADE tool [8], instruction scheduling tool and the trace-driven simulator for VLIW processor based on the SPARC architecture [9], allowing various machine parameters and machine configurations as shown in Fig. 5. Three versions of the machine configuration classified by the issue rate and the number of functional units are summarized in Table 1. Eight SPEC95 benchmarks (*099.go*, *132.ijpeg*, *134.perl*, and *147.vortex* from SPECint95, and *103.su2cor*, *104.hydro2d*, *107.mgrid*, and *110.applu* from SPECfp95) are used to evaluate the effect of each machine configuration on the code compression ratio.

4.1 Comparison of Compression Ratio

For each machine configuration, Figs. 6–13 compare the

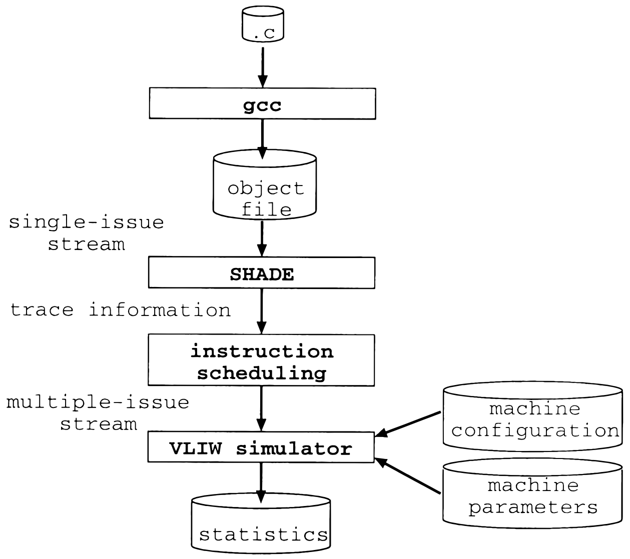


Fig. 5 Experimental environment consisting of the SHADE tool, instruction scheduling tool and VLIW trace-driven simulator.

Table 1 Issue rate and number of functional units in machine configuration: PI4, PI8, and PI12.

Feature	PI4 model	PI8 model	PI12 model
Issue rate	4	8	12
Fixed-point units	4	8	12
Floating-point units	2	3	4

code compression ratio of two schemes, (a) identical instruction words and (b) isomorphic instruction words. In the comparison, we have assumed for fair comparison that NOP's (No OPerations) which are usually padded for fixed-length instruction words in the conventional VLIW architectures are removed from the original programs and that there is no limitation to the size of dictionaries. The code compression ratio is defined as the total size of memory and the dictionaries divided by the size of the original memory. Simulation results show that the average code compression ratio of the scheme based on isomorphic instruction words is 63%, 69%, and 71% in PI4, PI8, and PI12, respectively and our scheme is a good trade-off between code compression ratio and decoding delay. For each benchmarks, isomorphic instruction words improve the code compression ratio significantly over identical instruction words by over 17%.

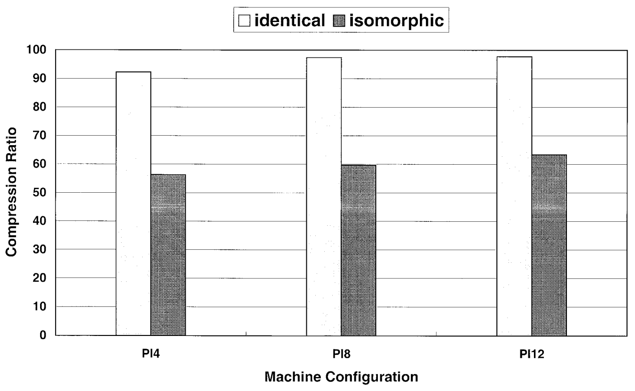


Fig. 6 Comparison of code compression ratio in *go*.

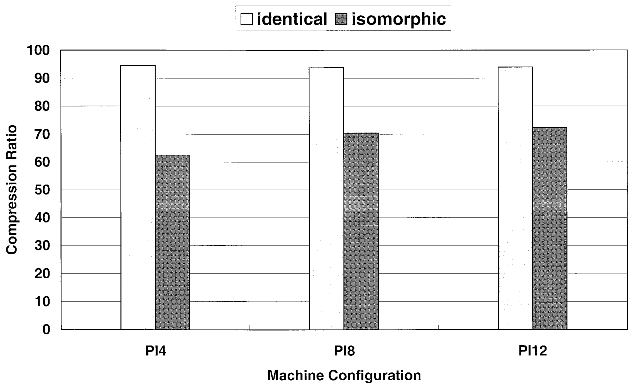


Fig. 7 Comparison of code compression ratio in *ijpeg*.

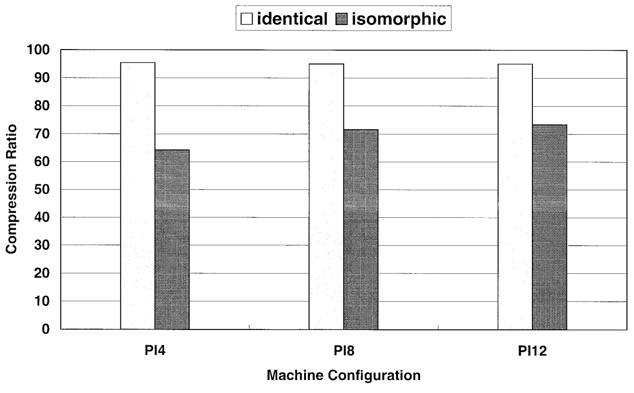


Fig. 8 Comparison of code compression ratio in *perl*.

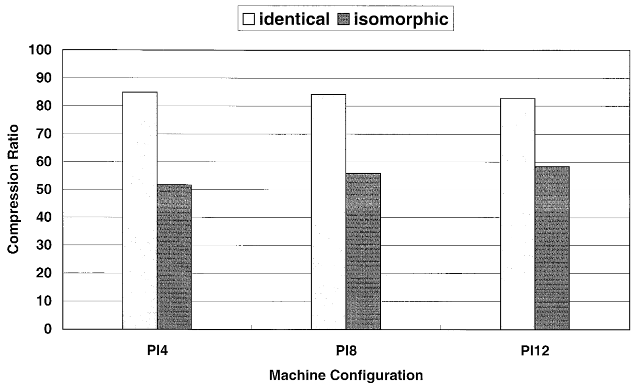


Fig. 9 Comparison of code compression ratio in *vortex*.

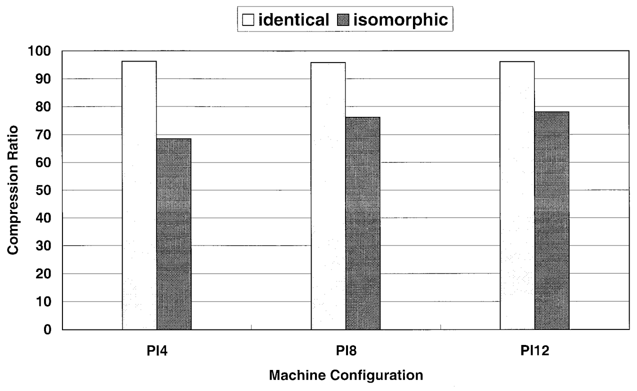


Fig. 10 Comparison of code compression ratio in *su2cor*.

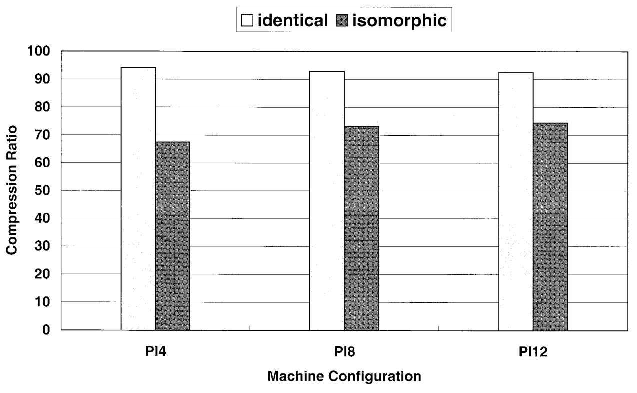


Fig. 11 Comparison of code compression ratio in *hydro2d*.

Table 2 Comparison of maximum number of entries and total size in each dictionary when isomorphism method is used.

Program	Model	Opcode		Operand		Total
		entry	size (bit)	entry	size (bit)	size (bit)
go	PI4	1179	39797	171	11499	51296
	PI8	1576	64565	208	16405	80970
	PI12	1654	72096	238	19691	91787
ijpeg	PI4	764	24437	108	6954	31392
	PI8	754	27147	100	6933	34080
	PI12	763	28779	102	7360	36139
perl	PI4	926	29515	134	8576	38091
	PI8	869	30293	127	8555	38848
	PI12	839	30123	122	8661	38784
vortex	PI4	958	32064	131	8725	40789
	PI8	1131	48213	163	13760	61973
	PI12	1157	53312	160	14891	68203
su2cor	PI4	595	18037	69	4096	22133
	PI8	517	16971	60	3904	20875
	PI12	514	17067	63	4117	21184
hydro2d	PI4	529	16000	59	3435	19435
	PI8	464	15467	59	4011	19478
	PI12	453	15541	60	3947	19488
mgrid	PI4	485	14773	56	3307	18080
	PI8	458	15531	54	3584	19115
	PI12	451	15403	60	3819	19222
applu	PI4	826	26272	93	5995	32267
	PI8	858	31680	93	6656	38336
	PI12	866	33323	100	7509	40832

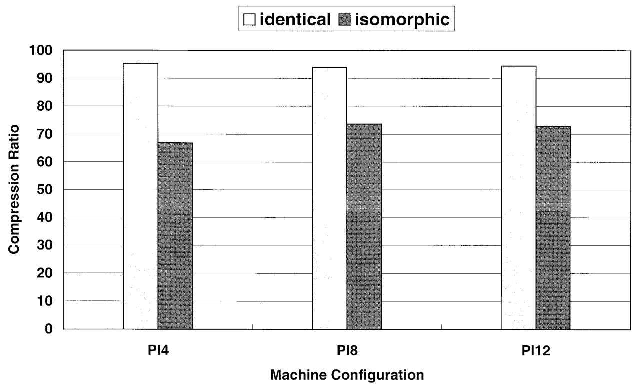


Fig. 12 Comparison of code compression ratio in *mgrid*.

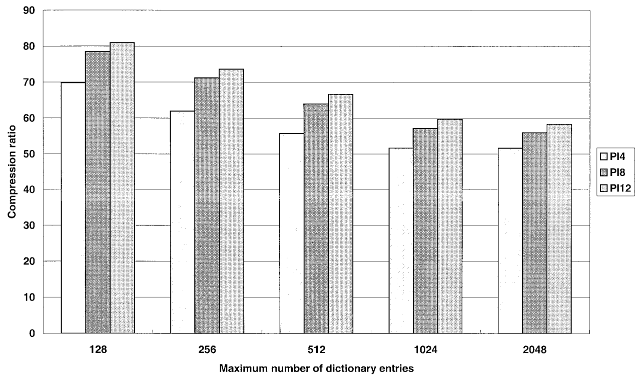


Fig. 14 Effect of number of opcode dictionary entries on compression ratio in *vortex*.

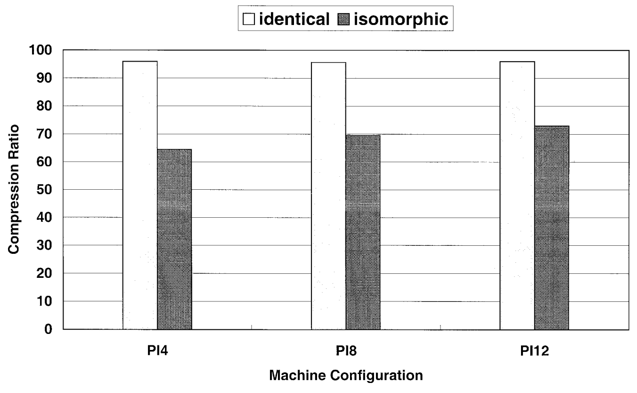


Fig. 13 Comparison of code compression ratio in *applu*.

4.2 Size of Dictionaries and Access Frequency

Actually, the code compression ratio is greatly affected by the number of dictionary entries available and the frequencies of appearance of each instruction word. Our next experiments vary the number of dictionary entries available. Figure 14 shows the effect of varying the number of opcode dictionary entries on compression ratio of *147.vortex* benchmark using isomorphic instruction words in PI4, PI8, and PI12, respectively. The compression ratio at 1024 entries of opcode dictionary is similar to that at infinite entries. We can see the same phenomena from results in other benchmarks.

Since the simulation results show that the maximum number of dictionary entries is smaller than that of [4], we assumed all entries needed to accommodate all isomorphic instruction words are available in the dictionary as shown in Table 2.

5. Conclusions

We have proposed a new code compression method in VLIW processor-based systems. The method is based on the isomorphism between instruction words in the dictionary-based code compression. After the analysis of a given instruction word stream, frequently-used isomorphic instruction words are extracted. Each occurrence of these extracted instruction words is replaced by an operand_word and an operand_word, which are used as the indices of the opcode dictionary and operand dictionary, respectively.

For the SPEC95 benchmarks, the simulation results show that the code compression ratio is 63%, 69%, and 71% on the average in PI4, PI8, and PI12, respectively and our scheme is a good trade-off between code compression ratio and decoding delay. Moreover, isomorphic instruction words improve the code compression ratio significantly over identical instruction words by over 17%.

References

[1] "TMS320C62xx CPU and Instruction Set: Reference Guide," Texas Instruments, Jan. 1997.

[2] S. Liao, S. Devadas, and K. Keutzer, "Code density optimization for embedded DSP processors using data compression techniques," Advanced Research in VLSI, 1995.

[3] C. Lefurgy, P. Bird, I.C. Chen, and T. Mudge, "Improving code density using compression techniques," Proc. the 30th Annual International Symposium on Microarchitecture, pp.194–203, Dec. 1997.

[4] Y. Yoshida, B.Y. Song, H. Okuhara, T. Onoye, and I. Shirakawa, "An object code compression approach to embedded processors," International Symposium on Low Power Electronics and Design, pp.265–268, Aug. 1997.

[5] N. Ishiura and M. Yamaguchi, "Instruction code compression for application specific VLIW processors based on automatic Field Partitioning," Proc. the Workshop on Synthesis and System Integration of Mixed Technologies, pp.105–109, Dec. 1997.

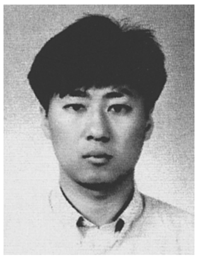
[6] J. Storer, "NP-completeness results concerning data compression," Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

[7] D.A. Huffman, "A method for the construction of minimum-redundancy codes," Proc. IRE, vol.4D, pp.1091–1101, Sept. 1952.

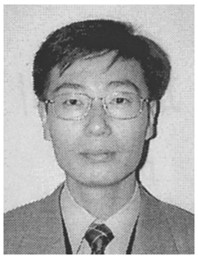
[8] Sun Microsystems Laboratories, SHADE User's Manual, Feb. 1993.

[9] D.L. Weaver and T. Germond, The SPARC Architecture Manual, Prentice-Hall, Inc., 1994.

[10] D. Liu and C. Svensson, "Power consumption estimation in CMOS VLSI chips," IEEE J. Solid-State Circuits, pp.663–670, June 1994.



Sang-Joon Nam received the B.S. and M.S. degrees in Electrical Engineering from KAIST (Korea Advanced Institude of Science and Technology), Korea in 1993 and 1995, respectively. He is currently pursuing the Ph.D. degree in Electrical Engineering in KAIST. His current research interests include multiple-issue microprocessor design, multimedia VLSI design and verification methodology.



In-Cheol Park received the B.S. degree in Electrical Engineering from Seoul National University in 1986, the M.S. and Ph.D. degrees in Electrical Engineering from KAIST (Korea Advanced Institude of Science and Technology), in 1988 and 1992, respectively. From May 1995 to May 1996, he worked at IBM T.J. Watson Research Center, Yorktown, New York as a postdoctoral member of the technical staff in the area of circuit design. He joined KAIST in June 1996 as an Assistant Professor in the Department of Electrical Engineering. His current research interests include CAD algorithms for high-level synthesis and VLSI architectures for general-purpose microprocessors.

Chong-Min Kyung received the B.S. degree in Electronic Engineering from Seoul National University, Korea in 1975, and the M.S. and Ph.D. degree in electrical engineering from KAIST (Korea Advanced Institude of Science and Technology), Korea in 1977 and 1981, respectively. After graduation from KAIST, he worked at AT&T Bell Laboratories, Murray Hill, NJ, from April 1981 to January 1983 in the area of semiconductor device and process simulation. In February 1983, he joined the Department of Electrical Engineering at KAIST, where he is now a Professor. His current research interests include microprocessor/DSP architecture, chip design and verification methodology. He is Director of the IDEC (Integrated Circuit Design Education Center) established to promote the VLSI design education in Korean universities through CAD environment setup, chip fabrication services, and providing various educational materials and media related with integrated circuits and systems design.