Code compression techniques using operand field remapping

K.Lin and C.-P.Chung

Abstract: Dictionary-based code compression stores the most frequently used instruction sequences in a dictionary, and replaces the occurrences of these sequences in the program with codewords. The large dictionary size is due mainly to many instruction sequences which are different only in operands, but are otherwise the same. The operand factorisation technique divides the expression tree into tree-pattern (opcode sequence) and operand-pattern (operand sequence) to reduce this redundancy; instruction sequences with the same opcodes but different operands may thus share the same tree-pattern dictionary entry. The paper proposes an operand field remapping method to further reduce dictionary size. The key idea is to explore the relations between the current operand to be compressed with those already compressed. The operand-pattern dictionary is therefore divided into an operand remapping dictionary and an operand list dictionary. Each entry in the operand remapping dictionary indicates whether the operand (register or immediate value) to be compressed is the most used operand, the same as the destination register of the previous instructions, or otherwise. With this remapping technique, the operand dictionary size is greatly reduced. An average 46% compression ratio can be achieved where compression ratio = (dictionary size + compressed code size)/(original program size).

1 Introduction

Embedded processors are highly constrained by cost, size, and power. Reducing the program size of the embedded systems is important to reduce system size, cost and power consumption, and to speed-up program execution. Compression methods fall into two categories: statistical [1, 2] and dictionary [3–5]. Statistical compression takes advantage of replacing frequently used instructions with smaller codewords to reduce the code size. Dictionary-based compression methods store frequently used instruction sequences in a dictionary and replace the occurrences with shorter codewords. Classification is made based on compression granularity: *Program*-based methods [6] compress the whole program and expand it back before execution, while *Procedure*-based methods [7] reduce the granularity to a procedure. These two methods avoid the difficulties of branch target re-addressing, and provide the software with an opportunity to help the systems to execute the compressed code directly, at the cost of execution speed and a larger decompression buffer. *Instruction-block*-based [1–4] or *instruction*-based [5–10] methods achieve effective decompression and execution with a much smaller decompression buffer. A front-end decompression engine is used to decompress the instructions and send them to the CPU on the fly, thus speeding up the execution.

IEE Proceedings online no. 20020157

DOI: 10.1049/ip-cdt:20020157

Paper first received 2nd October 2000 and in revised form 2nd October 2001

The authors are with the Department of Computer Science & Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Rd., HsinChu, Taiwan, 300

This paper proposes an instruction-block, dictionary-based compression technique named 'operand field remapping'. This method divides the operand-pattern dictionary into an operand remapping dictionary and an operand list dictionary to reduce its size. The entries in the operand remapping dictionary indicate whether the operand (register or immediate value) to be compressed is the most used operand, the same as the destination register of the previous instruction, or otherwise. With this remapping technique, the operand dictionary size is reduced significantly and the compression ratio is improved.

2 Related code compression work

One way to achieve a reduction in codes is to restrict the size of instructions. This is the approach adopted in the design of the Thumb [9] and MIPS16 [10] for ARM7 [11] and MIPS-III [10] processors, respectively. Shorter instructions are achieved mainly by restricting the number of bits that encode registers and immediate values. This results in 30–40% smaller programs running 15–20% slower than programs using a standard RISC instruction set [10, 12].

Another way to reduce the size of a program is to compress the codes by general compression methods. Lefurgy [3] proposed a dictionary-based compression method named Compressed Program Processor (CPP). This method simply stores one copy of common instruction sequences in the dictionary and replaces the occurrences with codewords. Average compression ratios of 61%, 66% and 74% were reported for the PowerPC, ARM and i386 processors respectively.

Wolfe [2] proposed a statistical compression method in a Compressed Code RISC Processor (CCRP) using Huffman-encoding [13]. A Line Address Table (LAT) is used to map the original program instruction addresses to the compressed code instruction addresses. The size of the LAT is approximately 3% of the original program size. A cache-like hardware called the Cache Line Address Lookaside Buffer (LCB) stores the most recent referenced LAT entries, so the cache refill engine can rapidly translate the addresses and fill the instructions. An average compression ratio of 73% on MIPS R2000 is reported.

Araujo [4] finds that most frequently used instruction sequences are identical with either opcode sequences or operand sequences, but not both, so that he separates the dictionary into a tree-pattern dictionary and an operand-pattern dictionary. The decompression engine reassembles the instruction sequence by combining the entries in both dictionaries indexed by the codeword pair of opcode and operand. The average compression ratio for this scheme is 43% using Huffman [13] and 48% using MPEG-2 VLC [14].

A language grammar-based code compression method [15, 16] accepts a grammar for programs written using a simple bytecoded, stack-based instruction set, as well as a training set of sample programs. The system transforms the grammar, creating an expanded grammar that represents the same language as the original grammar. An average compression ratio of about 36% [15] is reported.

3 Operand field remapping

The key idea in this paper involves the transformation of the operands to reduce the dictionary size, and the following subsections describe the detailed operand field remapping method.

3.1 Instruction reformatting

To establish the compression model, consider first the instruction formats (Appendix, Section 8.1) of the embedded processor ARM7TDMI [11]. All fields in the instruction format except opcodes (in conjunction with some fixed bits) are considered operands. The condition field, registers and immediate values are manipulated as multiples of 4 bit operand fields (OF) for simple hardware decompression. For example, Fig. 1*a* shows a shift instruction, which can be reformatted as Fig. 1*b*: a 4 bit conditional field as an OF, the opcode with some fixed-value bits (‘000’and S) as the new opcode, RN, RD and RM as three OFs, and an 8 bit shift value as two OFs.

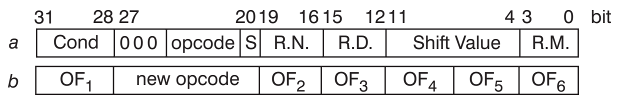

Fig. 1 *Instruction reformatting*

a Original ARM7TDMI instruction format
b New compression/decompression format

3.2 Operand factorisation

The opcode sequences were extracted by factorising all of the OFs from the instruction sequences and storing them into the opcode dictionary (OPD). All the OFs are removed from the instructions sequence to form the operand field dictionary (OFD). For example, the instruction sequence in Fig. 2*a* is factorised as shown in Figs. 2*b* and 2*c*. An occurrence of the instruction sequence is now replaced with a codeword consisting of two parts: Idx_{OPD} and Idx_{OFD} indexing to the OPD and to the OFD, respectively, as shown in Fig. 3.

3.3 Operand field remapping

Note that there are many dependencies among the operands in an instruction sequence, e.g. a destination register may immediately be used as the source operand of the following instructions, or a source operand may be re-used. Therefore, if we could use fewer bits than that of OF to record the dependencies instead of storing the OF itself, further reduction in OFD size may be achieved. To use this advantage, the OFD is transformed into an operand remapping dictionary (ORD) and an operand list dictionary (OLD). The former records the dependency relations and the latter records the modified operand lists after remapping. This method is termed the ‘operand field remapping’ technique.

The Idx_{OFD} in the original codeword is now split into two parts: Idx_{ORD} and Idx_{OLD}, indexing to an ORD entry and an OLD entry, respectively. Each entry in the ORD contains six fields, called *mapping tags*, to specify the six OFs of an instruction. The tag provides two kinds of operation: load and mapping. When a load operation is specified, an operand is loaded from the OLD entry indexed by Idx_{OLD} into the corresponding OF of the instruction. The load operation then also pushes the loaded operand into a mapping queue (MQ) such that other tags can map to this operand. When a mapping operation is specified, an operand in the MQ at a position specified by the value of the tag is loaded. So, if an operand depends on an operand in the previous instruction, which was loaded from the OLD by a previous load tag, then the latter tag can load it from the MQ. This reduces the repetitions in OFD. For a 2 bit mapping tag as an example,

a

		Cond	Opcode	RN	RD	Shift	Imm
cmp	r1, #1	1110	00110101	0001	0000	0000	0000 0001
beq	0xE	0000	10101111	0000	0000	0000	0000 1110
addeq	r3, r1, #0xE0	0000	00101001	0001	0011	0000	1110 0000

b

00110101	10101111	00110101

c

1110	0001	0000	0000	0000	0001
0000	0000	0000	0000	0000	1110
0000	0001	0011	0000	1110	0000

Fig. 2 *Operand factorisation*

a Instruction sequences and their formats
b Opcode sequences
c Factorised operand fields

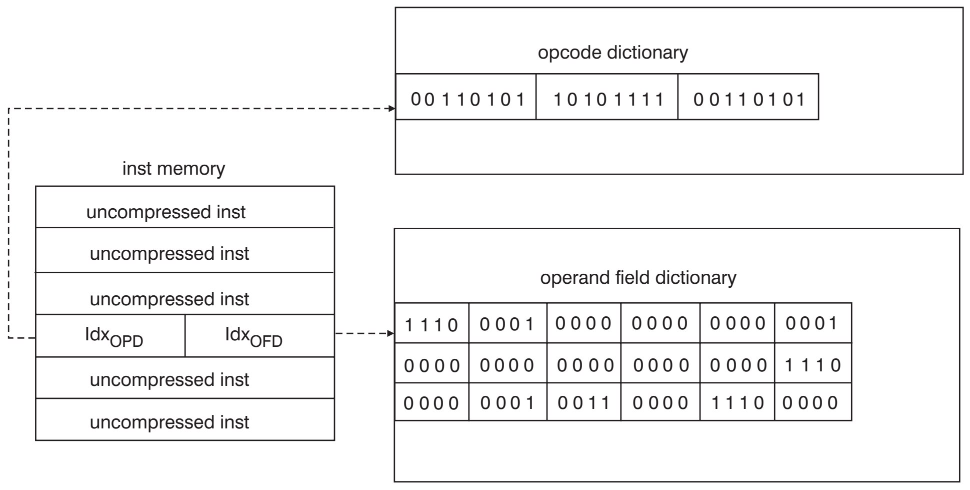

Fig. 3 *Codewords consisting of Idx_{OPD} and Idx_{OFD} indexing to the OPD and OFD*

'00' means a load operation, and '01–11' signify mapping operations that index to the first, second, or third operand in the MQ. When the MQ is full and a further operand is pushed into it, the oldest MQ entry is pushed out and disappears. Further reference to this operand will need a load tag to load it from the OLD again.

Fig. 4 illustrates the operand remapping technique using the example in Fig. 2. First, the three opcodes of the instruction sequence are stored into OPD. Next, notice that the first three operands of the first instruction are different, that three load tags '00' are required to load three operands '1110', '0001' and '0000'. Since the fourth and fifth operands are the same as the third one, these are simply mapped to the third operand. Two mapping tags '11' are used to indicate the operand in the MQ at position 3. The last operand is the same as the second operand, a mapping tag '10' is needed. These six tags form one entry to the ORD. The second and third instructions are coded in the same way.

3.4 Use of operand majority

An operand sequence usually contains some value that appears more frequently. The most frequently used operand is termed the first majority, the second most frequent, the second majority, and so on. Use of the majorities in remapping is extremely efficient, and can reduce the OLD size. These majorities are stored in majority registers (MRs) and include tags to map the OFs to these majorities (refer to the example in Fig. 4). The value '1110' is assumed to be the majority operand in the OLD. Fig. 5 illustrates the remapping technique applied to the original code in Fig. 4. The mapping tag 11 is assigned to the majority. Here the tag 00 still implies a load operation, and tags 01–10 are mapping operations.

3.5 Size of mapping tag

In this Section, OFD size reduction is evaluated when various sizes of mapping tags are applied. The benchmarks tested come from the MediaBench [17]. MediaBench contains applications culled from available image processing, communications and DSP applications. Appendix, Section 8.2 provides a summary of the programs in MediaBench. The benchmark programs are compiled for ARM7TDMI [11] using the ARM's Software Development Toolkit (SDT) ARMCC [18]. All the experimental programs are compiled with '-O2' optimisation.

Figs. 6–8 show the ratios of the size of ORD plus OLD over the size of the original OFD using 3 bit, 2 bit and 1 bit mapping tags, respectively. The x-axis categorises the benchmark programs and the y-axis shows the size reduction ratio. In Fig. 6, each program has eight lines, indicating 0–7 majorities, respectively. It was found that introducing the first majority reduces the OFD most, but the greater the number of majorities, the smaller the reduction in OLD size and the larger the increment in ORD size. The best case is to use one tag 000 for load operation, four tags 001–100 for mapping and three tags 101–111 for majorities. This tag assignment saves about 70% of the OFD size. In Fig. 7, each program has four lines, indicating 0–3 majorities, respectively. The best case

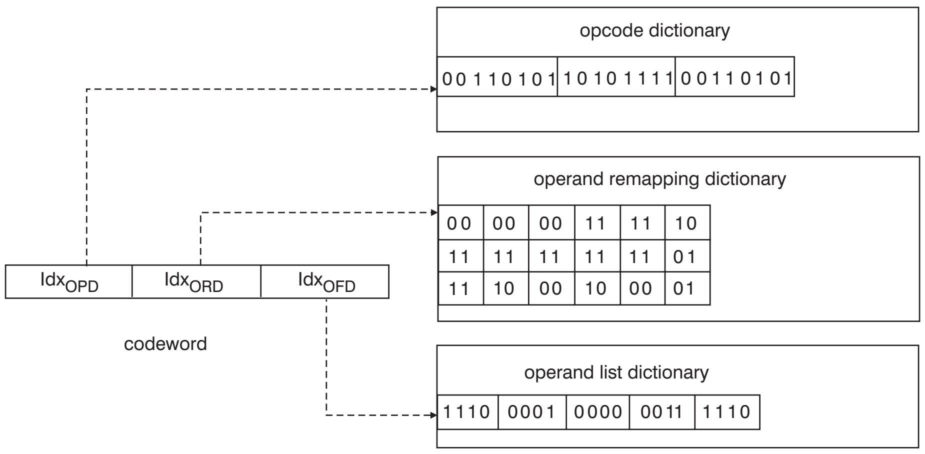

Fig. 4 *Codeword consisting of Idx_{OPD}, Idx_{ORD} and Idx_{OFD} indexing into the OPD, ORD and OLD, respectively*

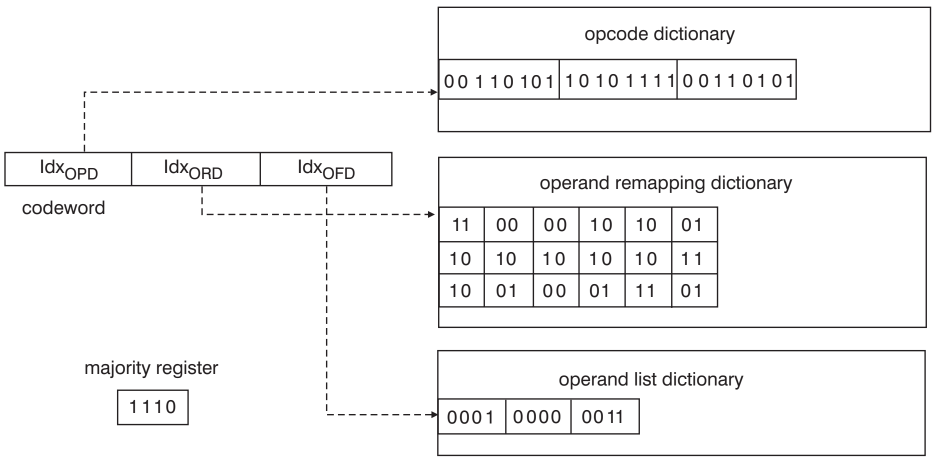

Fig. 5 *Majority remapping method*

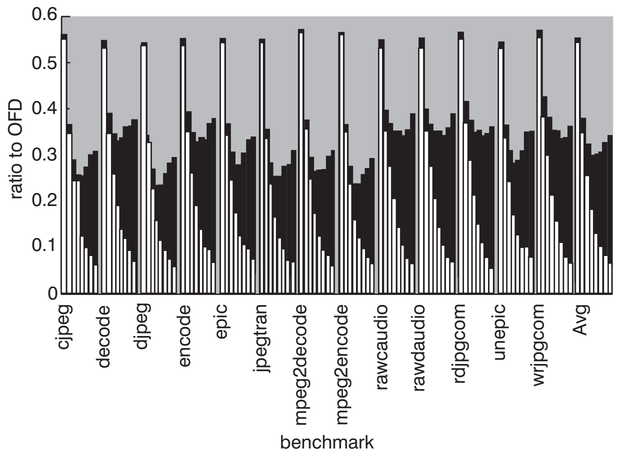

Fig. 6 *Dictionary size reduction using a 3 bit mapping tag*

■ ORD
□ OLD

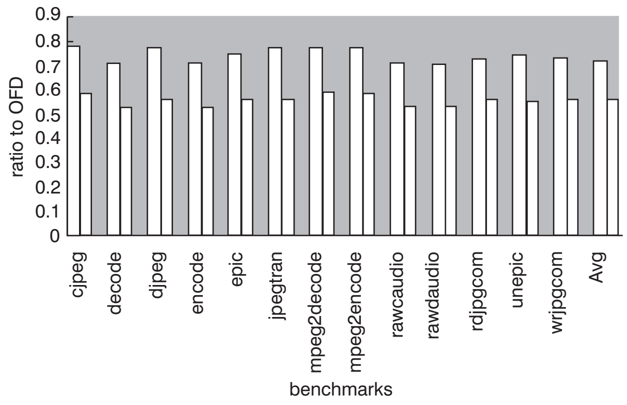

Fig. 8 *Dictionary size reduction using a 1 bit mapping tag*

■ ORD
□ OLD

is one tag 00 for load operation, the other tags 01–11 for majorities. In Fig. 8 the best case is one tag 0 for load operation and one tag 1 for the majority. However, Figs. 7 and 8 show that the size reductions are minor. The reason for this unsatisfactory result is that the mapping tag is too small to take advantage of the remapping. It is concluded that a 3 bit mapping tag is needed to achieve the best compression ratio.

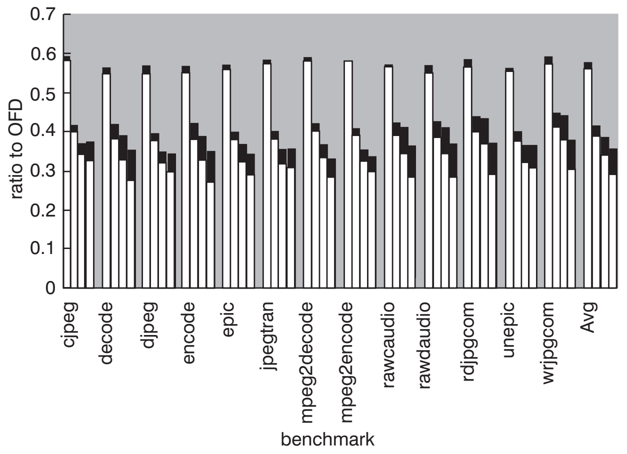

Fig. 7 *Dictionary size reduction using a 2 bit mapping tag*

■ ORD
□ OLD

3.6 Identities of the condition field

All ARM instructions contain a condition field (Cond in Fig. 1) which controls the instruction to be executed depending on the N (Negative), Z (Zero), C (Carry), and V (oVerflow) flags in the current program status register (CPSR) [11]. Experiments reveal that the values in the condition fields of 82% of instructions depend on the previous or the following instructions. Only 18% of instructions have a different condition value from those of their neighbours. Therefore, when building the OPD, a restriction is made that the instruction sequence must have the same condition code. We can then remove the condition field from the OLD. A condition code new stands for all instructions of a sequence and is encoded separately.

4 Compression algorithm

First, the benchmark programs are compiled to the executables using ARMCC [18]. Instruction sequences with the same opcode sequence are extracted from the basic blocks to avoid the branch instructions jumping within them. The opcode sequences are put in the OPD. Each OPD entry has two fields: an opcode field (8 bits) and a boundary field (1 bit). The opcode field contains the opcode of an instruction and the boundary bit indicates whether this instruction is the end of a sequence. The OF sequence is separated into the OF mapping sequence and operand list using 3 bit mapping tags. The OF mapping sequences make up the ORD and the operand lists make up the OLD. It is assumed that the first OPD, ORD and OLD entries of the first instruction of a sequence must be aligned to the byte boundary, while the following entries are arranged side by side.

As all dictionary entries are defined, the occurrences of the instruction sequence are replaced with codewords. A variable-length codeword is used in our compression method to give a better compression ratio. A codeword consists of four parts: a condition code encoding (CC), an index to *OPD* ($\mathrm{Idx}_{\mathrm{OPD}}$), an index to *ORD* ($\mathrm{Idx}_{\mathrm{ORD}}$) and an index to *OLD* ($\mathrm{Idx}_{\mathrm{OLD}}$). All of these four parts are coded separately using Huffman coding [13]. An instruction sequence is coded into the form [$\boldsymbol{CC\ Idx_{OPD}\ Idx_{ORD}\ Idx_{OLD}}$], and the compressed program consists of a list of quadruple: [$CC_1\ Idx_{OPD1}\ Idx_{ORD1}\ Idx_{OLD1}\ CC_2\ Idx_{OPD2}\ Idx_{ORD2}\ Idx_{OLD2}\ \ldots CC_n\ Idx_{OPDn}\ Idx_{ORDn}\ Idx_{OLDn}$]. The codewords are allowed to be split at the byte boundary.

One obvious side effect of code compression is that it alters the addresses of the instructions. To overcome this problem, all branch targets must be a codeword. The branch instruction (B) and the branch and link instruction (BL) use the offset addressing that could be simply corrected according to the compressed memory locations

Table 1: Detail compression ratio for all benchmarks

Program name	Size, bytes	Compression method	Dic/OPD/ OPD (%)	RegDic/ ORD (%)	Uncomp/ ImmDic/ OLD (%)	Comp. code (%)	Comp. ratio (%)
cjpeg	44471 * 4	Traditional	22.08		16.34	23.28	61.7
		Od Fact	1.41	12.5	7.48	23.03	44.42
		Of Remap	1.24	2.08	4.63	35.05	43
decoder	9485 * 4	Traditional	21.92		22.98	23.08	67.98
		Od Fact	2.82	14.86	9.55	22.29	49.52
		Of Remap	2.65	5.16	7.33	32.93	48.07
djpeg	50252 * 4	Traditional	21.59		16.37	23.44	61.4
		Od Fact	1.52	12.53	7.74	22.93	44.72
		Of Remap	1.21	1.95	4.27	34.91	42.34
encoder	9473 * 4	Traditional	21.72		23.33	22.92	67.97
		Od Fact	2.82	14.85	9.5	22.29	49.46
		Of Remap	2.63	5.15	7.25	32.99	48.02
epic	25579 * 4	Traditional	20.77		19.64	23.83	64.24
		Od Fact	1.7	12.7	10.44	23.42	48.26
		Of Remap	1.31	2.83	5.05	36.1	45.29
jpegtran	42555 * 4	Traditional	21.85		16.65	23.21	61.70
		Od Fact	1.11	11.93	7.24	23.61	43.44
		Of Remap	1.22	2.02	4.19	33.91	41.34
mpeg2dec	32225 * 4	Traditional	21.57		20.02	23.51	65.11
		Od Fact	2.1	13.07	12	23.36	50.53
		Of Remap	1.28	2.83	5.73	36.03	45.87
mpeg2enc	46415 * 4	Traditional	22.25		18.66	23.45	64.37
		Od Fact	1.74	13.3	10.05	24.2	49.29
		Of Remap	1.2	2.2	4.86	35.15	43.41
rawcaudio	6979 * 4	Traditional	19.77		25.4	22.7	67.87
		Od Fact	2.71	15.27	11.51	21.59	51.08
		Of Remap	1.75	5.64	8.23	32.94	48.56
rawdaudio	6975 * 4	Traditional	19.81		25.32	22.74	67.85
		Od Fact	2.69	15.25	11.47	21.57	50.98
		Of Remap	1.78	5.65	8.24	32.95	48.62
rdjpgcom	6995 * 4	Traditional	20.1		23.32	22.99	67.01
		Od Fact	2.41	14.19	10.3	21.8	48.7
		Of Remap	1.79	5.6	8.78	33.25	49.42
unepic	22557 * 4	Traditional	22.73		19.98	23.84	66.55
		Od Fact	2.05	13.53	9.05	22.88	47.51
		Of Remap	1.34	3.21	5.15	34.23	43.93
wrjpgcom	7603 * 4	Traditional	20.82		22.01	22.85	65.68
		Od Fact	2.26	13.47	9.43	21.94	47.1
		Of Remap	1.73	5.43	8.9	33.62	49.68
Average	23504 * 4	Traditional	21.35		20.77	23.22	65.34
		Od Fact	2.1	13.65	9.67	22.65	48.08
		Of Remap	1.62	3.83	6.35	34.16	45.97

of targets. The 24 bit offset of B (BL) instruction is divided into two parts: the first 21 bits indicate byte offset, and the last three bits are treated as bit offset. The call-return instructions need not be patched because these instructions load the contents of the link register, which contains the compressed address during execution. Note that the indirect branch is a branch and exchange instruction (BX). The BX instruction uses an address register to identify the boundary location between the ARM and Thumb codes. When modifying this type of instruction, we must backtrack the contents of the address register and modify the contents before storing the address into the register.

5 Experiment results

Compression ratios using the traditional [3], operand factorisation [4] and operand field remapping compression methods are compared. In the traditional compression method, the undefined instruction (refer to Appendix, Section 8.1) is used to store the codewords. Since the undefined instruction occupies only 4 bits (bit 27 to 24 and bit 4), at most two codewords can be packed into an undefined instruction if there are consecutive codewords. The maximal number of dictionary entries is 2^{14}. In the operand factorisation method, expression trees [19] are factorised into opcode sequences and operand sequences. The condition field is treated as the immediate value and the instruction sequence is factorised into new opcode (refer to Fig. 1) sequence and the operand field sequence, then the occurrences are encoded with codeword pairs using Huffman encoding. The first entries of both the opcode sequence and the operand sequence are also aligned to the byte boundary. Figure 9 shows the final compression ratios. The x-axis categorises the benchmarks tested and the y-axis shows the final compression ratio. Each benchmark has three lines, indicating the compression ratios using traditional, operand factorisation and operand field remapping methods, respectively. The average compression ratio using the traditional compression method is 65%, ranging from 61% to 67%. The compression ratio is constrained since there are still 16% to 25% of instructions uncompressed. The reason is that these instructions are unique and are not selected into the dictionary, and there are not enough dictionary entries to include them.

The compression ratio using operand factorisation is about 48%, ranging from 43% to 51%. It was found that the RGEN (operand sequence dictionary) and the

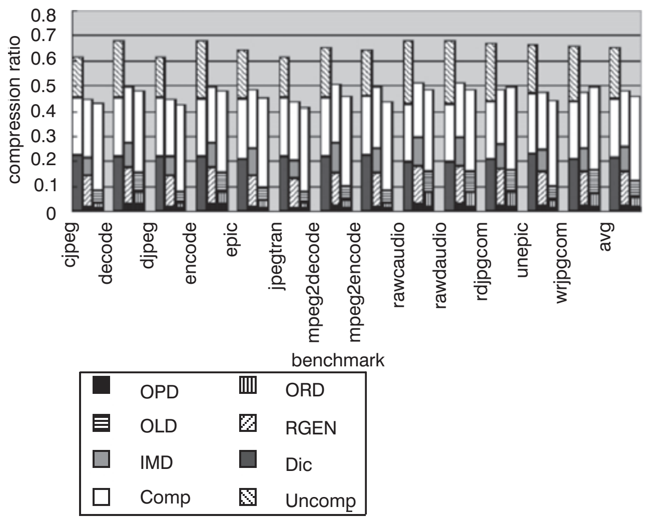

Fig. 9 *Final compression ratios using traditional, operand factorisation and operand field remapping methods*

compressed code are the major two parts that contribute to the compression ratio. The average compression ratio using the operand field remapping compression method is 46%, ranging from 41% to 50%. This result is better than that of operand factorisation, if the dictionaries of both methods are aligned to the byte boundary. The better result is because the mapping sequence can exploit more repetitions than the operand sequence does, and omitting the conditional codes can reduce the dictionary size in advance. Table 1 shows the percentages of all components of the compressed program. Table 1 indicates that the average reduction of RGEN and IMD to ORD and OLD is about 13%, although the compressed code increases by about 12%. This is the main advantage over the operand remapping method.

6 Conclusions

The paper proposes an operand remapping compression method to compress embedded system programs for an ARM processor. The key idea of this method is to map later occurrences of an operand to the previous same one in the same operand field sequence. The best compression ratio obtained using this method was 41.34%, with an average of 45.9%. This can be further improved in several ways. First, compressing the OPD, ORD and OLD by utilising recursive pointers [20] to reuse the dictionary entries can further reduce dictionary sizes. Pointers in the dictionaries will cause the decompression engine to be much more complicated, but will produce a denser code. Second, the compiler could attempt to produce identical instruction sequences for the same expression tree [19] so that the more common instruction sequences become more compressible [21]. One way to accomplish this is to allocate the same registers [21, 22] for the same expression tree. Finally, the compression algorithm could be improved by finding more relations between operands, such as the register allocation rules.

7 References

1 KOZUCH, M., and WOLFE, A.: 'Compression of embedded system programs'. IEEE international conference on *Computer design*, 1994
2 WOLFE, A., and CHANIN, A.: 'Executing compressed programs on an embedded RISC architecture'. Proceedings of the 25th annual international symposium on *Microarchitecture*, December 1992
3 LEFURGY, C., BIRD, P., CHEN, I.-C., and MUDGE, T.: 'Improving code density using compression techniques'. Proceedings of the 30th annual international symposium on *Microarchitecture*, December 1997
4 ARAUJO, G., CENTODUCATTE, P., CORTES, M., and PANNAIN, R.: 'Code compression based on operand factorization'. 31st Annual ACM/IEEE international symposium on *Microarchitecture*, 1998
5 YOSHIDA, Y., SONG, B.-Y., OKUHATA, H., ONOYE, T., and SHIRAKAWA, I.: 'An object code compression approach to embedded processors'. Proceedings of international symposium on *Low power electronics and design*, 1997, pp. 265–268
6 CATE, V., and GROSS, T.: 'Combining the concepts of compression and caching for a two-level filesystem' Architectural Support for Programming Languages and Operating Systems, 1991
7 KIROVSKI, D., KIN, J., and MANGIONE-SMITH, W.H.: 'Procedure based program compression'. Proceedings of Microarchitecture, 1997
8 IBM: 'CodePack: PowerPC code compression utility user's manual'. Version 3.0, International Business Machines (IBM) Corporation, 1998
9 Advanced RISC Machines Ltd., 'An introduction to Thumb', 1995
10 KISSELL, K.: 'MIPS16: high-density MIPS for the embedded market' (Silicon Graphics MIPS Group, 1997)
11 Advanced RISC Machines Ltd.: 'ARM architecture reference manual', 1996
12 'Thumb squeezes ARM code size', *Microprocessor Report*, 1995, **9**, (4)
13 HUFFMAN, D.A.: 'A method for the construction of minimum redundancy codes', *Proc. IEEE*, 1952, **40**, pp. 1089–1101
14 HASKELL, B.G., PURI, A., and NETRAVALI, A.N.: 'Digital video: an introduction to MPEG-2' (Chapman & Hall)
15 FRASER, C.W., and EVANS, W.: 'Bytecode compression via profiled grammar rewriting'. Proceedings of international conference on *Programming languages design and implementation*, June 2001

16 HOOGERBRUGGE, J., AUGUSTEIJN, L., TRUM, J., and VAN DE WIEL, R.: 'A code compression system based on pipelined interpreters', *Softw.–Pract. Exp.*, 1999, **29**, (11), pp. 1005–1023
17 LEE, C., POTKONJAK, M., and SMITH, W.H.M.: 'MediaBench: a tool for evaluating and synthesizing multimedia and communications systems'. 30th Annual ACM/IEEE international symposium on *Micro-architecture*, 1997
18 Advanced RISC Machines Ltd., 'ARM software development toolkit version 2.50 user guide', 1998
19 AHO, A., SETHI, R., and ULLMAN, J.: 'Compilers, principles, techniques and tools' (Addison Wesley, Boston, 1988)
20 STORER, J.A.: 'Data compression: methods and theory' (Computer Science Press, 1988)
21 DEBRAY, S., EVANS, W., and MUTH, R.: 'Compiler techniques for code compression'. Workshop on Compiler Support for System Software, May 1999
22 COOPER, K.D., and MCINTOSH, N.: 'Enhanced code compression for embedded RISC processors'. Proceedings of conference on *Programming languages design and implementation*, May 1999

8 Appendix

8.1 ARM7TDMI instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
Cond				0	0	1	Op				S	Rn				Rd				Operand 2												**Data Processing/ PSR Transfer**
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				**Multiply**
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				**Multiply Long**
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				**Single Data Swap**
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				**Branch and Exchange**
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				**Halfword Data Transfer: Register offset**
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				**Halfword Data Transfer: Immediate offset**
Cond				0	1	1	P	U	B	W	L	Rn				Rd				Offset												**Single Data Transfer**
Cond				0	1	1																					1					**Undefined**
Cond				1	0	0	P	U	S	W	L	Rn				Register List																**Block Data Transfer**
Cond				1	0	1	L	Offset																								**Branch**
Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset								**Coprocessor Data Transfer**
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				**Coprocessor Data Operation**
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				**Coprocessor Register Transfer**
Cond				1	1	1	1	Ignored by processor																								**Software interrupt**
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Bit**

8.2 The MediaBench benchmarks

Component name	Description
JPEG	JPEG is a standardised compression method for full-colour and grey-scale images. Two applications are derived from the JPEG source code; cjpeg does image compression and djpeg, which does decompression
MPEG	MPEG2 is the current dominant standard for high-quality digital video transmission. Two applications used are mpeg2enc and mpeg2dec for encoding and decoding respectively
GSM	European GSM 06.10 provisional standard for full-rate speech transcoding, prl-ETS 300 036, which uses residual pulse excitation/long term prediction coding at 13 Kbit/s
G.721 Voice Compression	Reference implementations of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721, and G.723 voice compressions
PEGWIT	A program for public key encryption and authentication. It uses an elliptic curve over GF(2^{555}), SHA1 for hashing, and the symmetric block cipher square
RASTA	A program for speech recognition that supports the following techniques: PLP, RASTA, and Jah-RASTA
EPIC	An experimental image compression utility
ADPCM	Adaptive differential pulse code modulation is one of the simplest and oldest forms of audio coding