Optimizing Code Size for Embedded Real-Time Applications

Shao-Yang Wang , Chih-Yuan Chen , and Rong-Guey Chang[*]

Department of Computer Science National Chung Cheng University

Chia-Yi, Taiwan

{wsya92 , chancy, rgchang} @cs.ccu.edu.tw

Abstract. This paper presents an efficient technique for code compression. In our work, a sequence of instructions that occurs repeatedly in an application will be compressed to reduce its code size. During compression, each instruction is first divided into the operation part and the register part, and then only the operation part is compressed. For reducing the run-time overhead, we propose an instruction prefetching mechanism to speed the decompression. Moreover, we devise some optimization techniques to improve the code size reduction and the performance, and show their impacts. The experimental results show that our work can achieve a code size reduction of 33% on average and a low overhead in the process of decompression at run time for these benchmarks.

1 Introduction

Many memories have been incorporated into embedded systems to execute application programs. In some embedded systems, the application programs are expected to completely fit in the memory of the chip. In other embedded systems such as portable devices, the code sizes of application programs must be as small as possible to reduce the cost and decrease the weight. However, the memory required by an application program is determined by the size of its data and instructions. Therefore, how to compress the code size of an application program to shrink the amount of memory required has become a crucial issue in embedded systems.

In this paper, we address the code compression of an application program by compressing its repeated instruction sequences. Here, repeated instruction sequences represent the instruction sequences that occur more than once in an application. In our method, each instruction of an application is divided into the operation part and the register part. Then the operation part is profiled and compressed by reducing the number of its repeated sequences. This process will be applied to the newly operation part recursively until no further compressions can be performed. Note in above process the register part is not compressed. The compression information of operation part and register part is put into the instruction table and the register bank respectively. The

[*] This work was supported in part by National Science Council, Taiwan, under Grant NSC 94-2220-E-194-005-.

information about how to access the items in the instruction table and the register bank is systematically stored in the index table. During execution, the compressed program is decompressed instruction by instruction through accessing these tables. Previous work [8,10] showed that there is a tradeoff between the compression ratio and the performance. Thus, we propose an instruction prefetching mechanism to speed decompression and some optimization techniques to remedy this issue. Our work is performed on the basis of the ARM instructions and the experimental results show that our method is quite effective in achieving a high code size reduction and a low overhead in decompression.

The remainder of this paper is organized as follows. Section 2 describes the related work. We first introduce the compression approach in Section 3 and then present the details of our decompression approach in Section 4. Section 5 shows our experimental results without optimization. In Section 6, we propose our optimization techniques and show their impacts. Finally, we conclude this paper briefly.

2 Related Work

Some researchers addressed this issue by using specific architectural support. ARM Thumb [1,2] provides a compressed 16-bit instruction set to reduce the code size, each of which can be translated to its corresponding 32-bit instruction with a hardware decompressor during execution. In addition, MIPS compresses the code in a similar way by also providing a shorter instruction set called MIPS16 [11]. IBM CodePack uses Huffman encoding [7] for their code compression, partitioning the code word into two parts and applying Huffman encoding to these two parts separately [9]. The code size reductions of these approaches range from 30% to 40%.

Other researchers addressed this issue in terms of software. Evans and Fraser devised an approach for compressing a stack-machine bytecode and achieved a code size reduction of up to 29% [6]. Debray and Evans addressed this issue by proposing a profile-guided code compression on the basis of the 80-20 rule [4]. Bell et al. solved this problem by using a dictionary table, assigning each of the frequently occurring instructions to an index in the dictionary table [3]. Other works provided code compression by improving the dictionary method [12,13]. Ernst et al. applied a hybrid technique to address this issue, presenting two code compressors to handle transmission and memory bottlenecks independently [5]. Keith et al. compressed the code with compilation techniques, but the code size reduction was only 5% on average [10].

3 Compression Approach

In this section we first present the basic idea with an example and we then describe our prefetching mechanism.

For ARM instructions, the register fields of bit 16 to bit 19, bit 12 to bit 15, and bit 0 to bit 3 are used to indicate the register part; the remaining bits are the operation part. In our work, only the operation part is compressed and for simplicity, we use assembly codes to represent binary codes as examples. Our compression concept is shown in

Figure 1. In Figure 1b, we first find that the instruction sequence in Figure 1a, two consecutive add instructions, is the one that occurs most frequently, so these two add instructions are compressed as “add, add”. The process is repeated to check whether other instruction sequences can be compressed. The sequence, the rsb followed by “add, add”, is compressed as the “rsb, add, add” instruction again and the final result is shown in Figure 1c. In contrast with Figure 1a, the length of the code shown in Figure 1c is shorter. The whole process can be represented as the binary tree illustrated in Figure 1d.

```
add  r0, r8, r4
add  r3, r0, r0
rsb  r3, r0, r3
add  r3, r3, r3
add  r0, r0, r3
mul  r1, r7, r10
add  r2, r9, r11
add  r5, r1, r1
add  r6, r2, r2
rsb  r5, r5, r5
add  r6, r2, r6
add  r5, r1, r5
add  r6, r2, r6
```

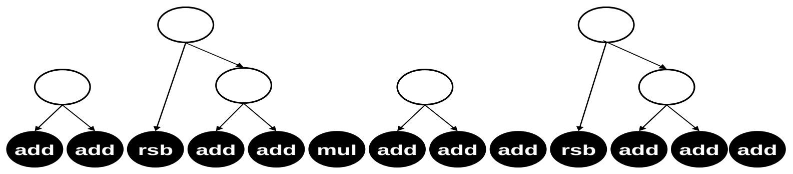
a.

```
add, add    r0, r8, r4, r3, r0, r0
rsb         r3, r0, r3
add, add    r3, r3, r3, r0, r0, r3
mul         r1, r7, r10
add, add    r2, r9, r11, r5, r1, r1
add         r6, r2, r2
rsb         r5, r5, r5
add, add    r6, r2, r6, r5, r1, r5
add         r6, r2, r6
```

b.

```
add, add         r0, r8, r4, r3, r0, r0
rsb, add, add    r3, r0, r3, r3, r3, r3, r0, r0, r3
mul              r1, r7, r10
add, add         r2, r9, r11, r5, r1, r1
add              r6, r2, r2
rsb, add, add    r5, r5, r5, r6, r2, r6, r5, r1, r5
add              r6, r2, r6
```

c.



d.

Figure 1. Motivating example

The compressed code is stored in three parts: instruction table, register bank, and index table, as shown Figure 2.

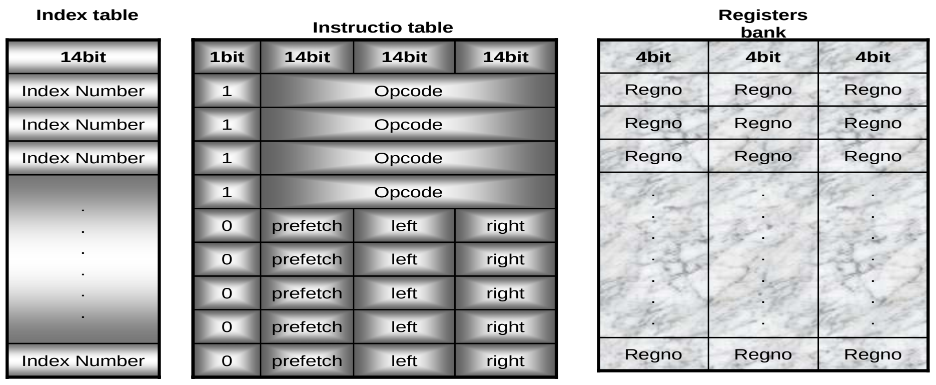


Figure 2. Compression architecture

In compression, each opcode in the operation part is represented as an index number and these opcodes and indices are put into the instruction table and the index table. The instruction table contains two parts, as shown in Figure 3. They can be distinguished from the T bit. The left part contains the opcodes of the instructions used in a program and the right part represents the combination information of the compressed instructions. The sources of compressed instructions may come from the opcodes of the left part, other compressed instructions, or a mixture of both. The Left and Right fields of the case T = 0 represent the sources in the normal case, and the prefetch field is used to optimize the decompression performance. The register operands are stored in the register bank. The entries of the index table point to the addresses of the opcode sources in the instruction table and the addresses of the register operands in the register bank. In decompression, we access the entries in the index table in order to fetch the opcodes and the registers. Thus we can compress a code and then easily decompress it instruction by instruction at run time.

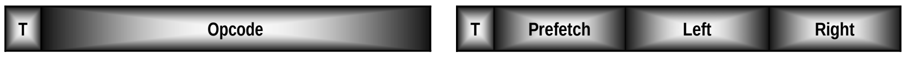


Figure 3. Format of instruction table when *T*=1 (left) and *T*=0 (right)

Now we propose an prefetching mechanism in compression to speed the decompression. Consider the example shown in Figure 4. In Figure 4a, three accesses are required to fetch instruction "a" from the root. In this case, the internal nodes contained in the path are the decompression overheads. Our prefetching scheme keeps the index of the left-most subtree or a leaf (that will be executed first) by moving the leaf or the tree to the internal nodes in advance. Figure 4b is the tree after the instruction prefetching is applied to Figure 4a. In fact, Figure 4b can be further optimized to be Figure 4c by applying the prefetching scheme again. In this paper, the letters in black circles in the tree represent the instructions, while the letters in white circles mean the prefetching information that points to an instruction. In addition, we use a dashed line to indicate the target when the prefetching is applied to a tree, numbers to represent the indices in instruction table, dot nodes to indicate a tree is "don't care", and slash nodes to mean a tree including NULL is "don't care".

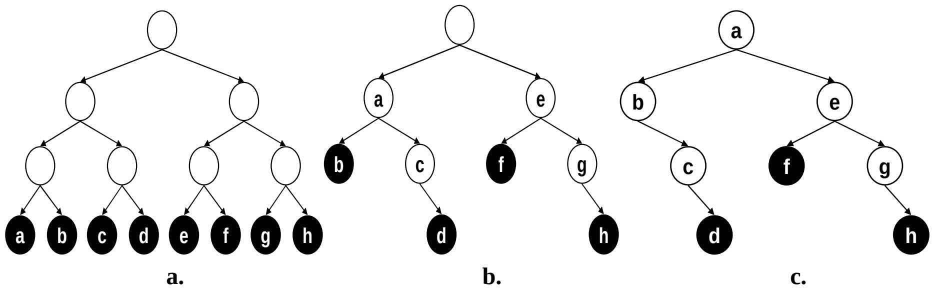
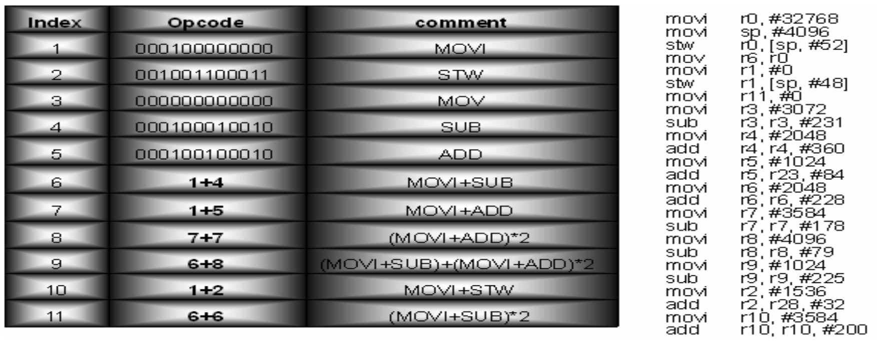


Figure 4. Instruction prefetching scheme

Consider the example shown in Figure 5. The operation part and the register part are first put in the instruction table and the register bank. Next the repeated instruction sequence "MOVI, SUB" has been compressed with index 6, and T bit is set to false. The T bit of index 1 is set true, and prefetch = 1, left = NULL, right = 4. The combination (1+4) is inserted into entry 6 in the instruction table. These steps will be applied repeatedly and 1+4, 1+5, 7+7, 6+8, 6+6, 1+2 is inserted sequentially. In prefetching, four cases must be handled to optimize the compression. First, the left subtree is a leaf, thus it will be prefetched. Second, the left is NULL and the right is a leaf, then the prefetching information is stored in the root. If the right is an internal node, the prefetching information must be kept for further executions. Third, the prefetch is a leaf, the prefetching information is not kept and the prefetch points to a tree. Finally, the prefetch is the same as the prefetch of left subtree since the information kept in prefetch is a left-most tree. In our approach, if a node and its prefetch are not leaves, then it is called a *Redundant Node*, abbreviated R-node. In this case, we are not able to acquire any leaf information during decompression.



Index	Opcode	comment
1	000100000000	MOVI
2	001001100011	STW
3	000000000000	MOV
4	000100010010	SUB
5	000100100010	ADD
6	**1+4**	MOVI+SUB
7	**1+5**	MOVI+ADD
8	**7+7**	(MOVI+ADD)*2
9	**6+8**	(MOVI+SUB)+(MOVI+ADD)*2
10	**1+2**	MOVI+STW
11	**6+6**	(MOVI+SUB)*2

```
movi    r0, #32768
movi    sp, #4096
stw     r0, [sp, #52]
mov     r6, r0
movi    r1, #0
stw     r1, [sp, #48]
movi    r11, #0
movi    r3, #3072
sub     r3, r3, #231
movi    r4, #2048
add     r4, r4, #360
movi    r5, #1024
add     r5, r23, #84
movi    r6, #2048
add     r6, r6, #228
movi    r7, #3584
sub     r7, r7, #178
movi    r8, #4096
sub     r8, r8, #79
movi    r9, #1024
sub     r9, r9, #225
movi    r2, #1536
add     r2, r28, #32
movi    r10, #3584
add     r10, r10, #200
```

a. **b.**

Figure 5. Example for ARM instructions

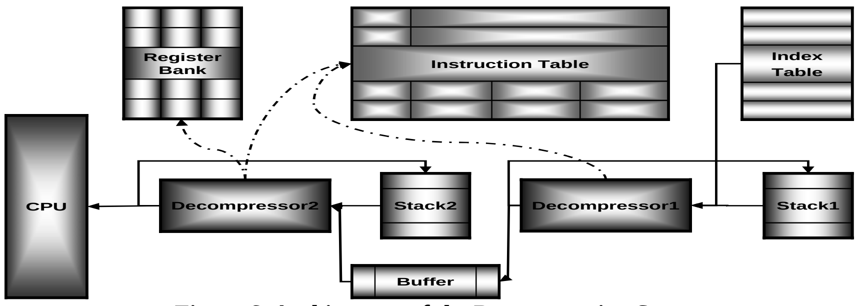


Figure 6. Architecture of the Decompression System

4 Decompression Approach

In decompression, the prefetching is performed only in the root and the right subtree. We use the two decompressors synchronously shown in Figure 6 to reduce the decompression overhead. One decompresses the indices from the instruction table into buffer and the other decompresses the data from buffer into the CPU. Figure 7 shows an example of our decompression scheme. For decompressor 1, the index 9 is first acquired from the index table and its root is set to true. As the T bit of the index 9 is false, index 8 is pushed into the stack and the root of its right side is set to true. Then the index 4, to the left of index 9, is pushed into the stack and its root is set to false. Next, the prefetch of index 9, index 1, is put in the buffer and the root is set to true since the root of index 9 is true. Then the stack is not empty and the process will return to the beginning of this step and acquire an index from the stack. Now we acquire the index 4 and its T bit is true, thus it is put in the buffer and the root is set to true. The above steps will be repeated and the content of buffer will be (1,4,1,5,1,5). Now we use the above buffer to explain the action of decompressor 2. First, we acquire the item index = 1 from the buffer. Second, we can access the registers based on the program counter and combine them together into one item because the T bit of index = 1 is true. Then the new item is delivered to the CPU and the process is rpeated because the stack is still empty. These steps will be performed recursively and the instructions (movi, sub, movi, add, movi, add) will be delivered to the CPU in order.

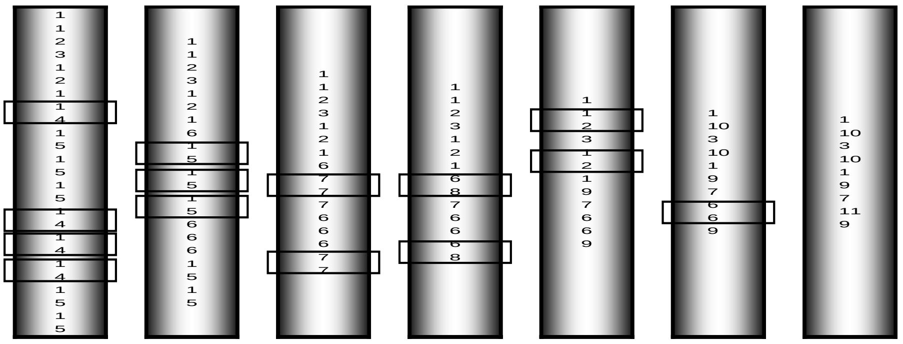


Figure 7. Insertions of compressed instructions

The overhead of decompression arises from R-nodes or when the buffer shown in Figure 6 is empty. For the first case, R-nodes will not exist in the subtrees indicated by prefetches. With the help of this architecture and the optimization schemes described in Section 5, our method can fetch an instruction in each access. For the second case, each index in the buffer is a prefetch or a leaf. Thus, we can fetch an instruction from decompressor2 whenever the buffer is not empty.

5 Experimental Result

The experiments are performed on the ARM simulator running the RedHat9.0 operating system and compiled with a GNU compiler with default settings. The testing sets consist of SPEC2000, MediaBench, DSPstone benchmarks.

5.1 Compression Ratio Evaluation

The bars in Figure 8 show the compression ratio of four benchmarks calculated by the following equation.

$$\left(\frac{(compressed\_instruction \times compressed\_width) + (instruction\_counts \times 12) + instruction\_table\_size}{instruction\_counts \times 32}\right)$$

The compression ratio ranges between 54% and 81%. With our experiences, the width is limited to 14 bits. In addition, the instruction savings of applications in SPEC2000 and MediaBench are around 30% and 36%, respectively. However, in DSPstone, the instruction saving is low because its applications are quite small. For MPEG4, the instruction saving is between 28% and 68%. In particular, the "bitstream" has the best result because there are many macros used in it. Consider ghostscript in MediaBench as an example, which is the largest program in all benchmarks. It has 304124 instructions and its compressed code only contains 5415 opcodes. Therefore, the width and the length of the index table are 13 bits and 189545, the width and the length of the instruction table are 40 bits and 8192, and the width and the length of the register bank are 12 bits and 304124. The compression ratio is $(13 \times 189545 + 40 \times 8192 + 12 \times 304124) / 32 \times 304124=0.66$.

5.2 Performance Evaluation

The experiments are performed in the ARM simulator running on RedHat9.0 to count the cycles needed to decompress the compressed codes. In the experiments, the sizes of buffer and the two stacks are limited to 16 slots. The curves in Figure 14 show the performance degradations. The ratio of performance degradation is calculated from the number of cycles during decompression to those executed in the normal cases. The range of the performance loss is below 25%. For DSPstone, the average performance levels are worse because the average heights of the compression trees in it are higher than those of other benchmarks. The average performance in Xvid is better except the bitstream. Its performance is bad because it use many macros, which increases the height of the tree.

6 Optimizations

In this section, we propose some optimizations to enhance our method in compression ratio and performance. They are described in the following and the results are shown in Figure 9 and Figure 10.

6.1 Performance Enhancement

There are some optimizations performed on the instruction table to improve the performance without increasing the size. First, in the first part of the instruction table (T = 1), the opcode is shorter than the length of entries. For a large application, it is very possible to utilize the remaining space of entries to occupy another opcode. In this way, a single leaf can contain two opcodes and we can fetch two instructions each time to improve the performance. Second, if the width of the left field and the prefetch field is longer than that of opcode, the left must be NULL, and the prefetch must be a leaf, we can put the opcode of prefetch in the left field and the prefetch field. We can use one bit to specify that the content is an index or an opcode. Therefore, the performance will be improved. The result is shown in Figure 9 with the line 'Opcode in prefetch and left'. Moreover, the case in Figure 10b can be further optimized to be that in Figure 10c. Its performance can be improved by applying the above techniques to it. We can ensure that the prefetch will point to a leaf or a tree. In this case, as the left of a tree is NULL, the decompressor2 can deliver more than one instruction each fetch. The defect of this way might cause the increase of R-nodes and degrade the performance loss. The curve 'Optimization on instruction table' in Figure 9 shows the performance after all methods described in this section are applied.

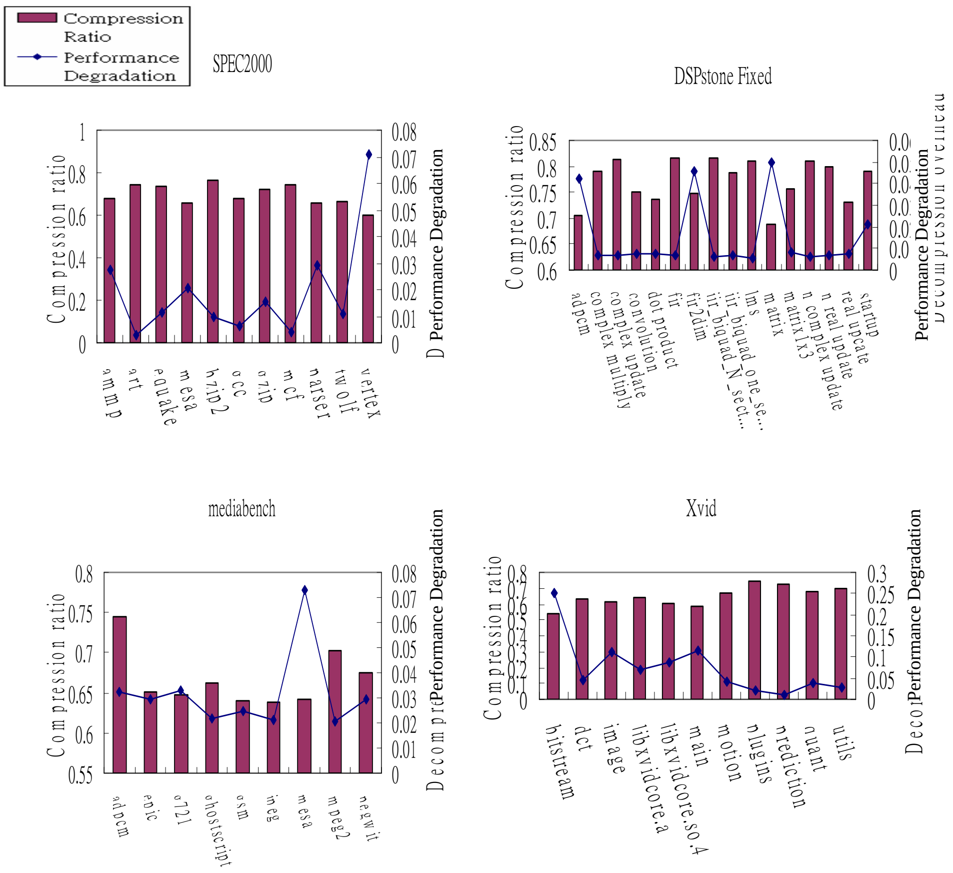


Fig. 8. Results of compression ratio and performance degradation

If the width of the prefetch in the instruction table can be extended to contain an opcode, then the performance will be almost the same as that of the original program. This is because it is unnecessary to look up the prefetches. In addition, we still need one bit to specify that the content of the prefetch is an opcode or an index, but it will increase the size of instruction table. For example, in the case of application 'mesa' in the MediaBench, the size of the instruction table will be increased from 40KB to 47KB.

In the experiment, we found that the number of R-nodes has a close relationship with the depth of a tree. Therefore, in the process of compressing an instruction sequence, we can check first if the depth of the new tree is higher to stop the merging. In this way, we can improve the performance by reducing the R-nodes. We limit the depth of the tree to 4 and show the performance result in Figure 9 and the compression result in Figure 10 with the line ‘Depth = 4’.

6.2 Compression Optimization

In our work, we use T-bit in the instruction table to tell whether the entry is an opcode or not. Indeed, we can use one register or one counter to specify that entries belong to which part instead of using the T-bit for each entry. Thus, the size of the instruction table can be smaller. The width of the first part of the instruction table is always shorter than that of the second part. Therefore, the instruction table can be split as two individual parts so that the size of the first part can be reduced greatly. The result is shown in Figure 10.

6.3 Core Size Reduction

To reduce the hardware area, we can use only one decompressor during compression, but it will lead to a performance penalty. The right axis in Figure 17 shows the results performed with using only one decompressor.

In the experiment, the case that the buffer is empty occurs seldom. It means that the decompressor2 always decompress more instructions than those produced from the decompressor1. If the tree in the buffer contains more than one instruction, that is, there are not too many R-nodes in it, the size of the buffer can be reduced to optimize the memory required. The experimental results show that the performances are the same no matter the size of the buffer is 4, 8 or 16. Thus, the buffer may be not very large.

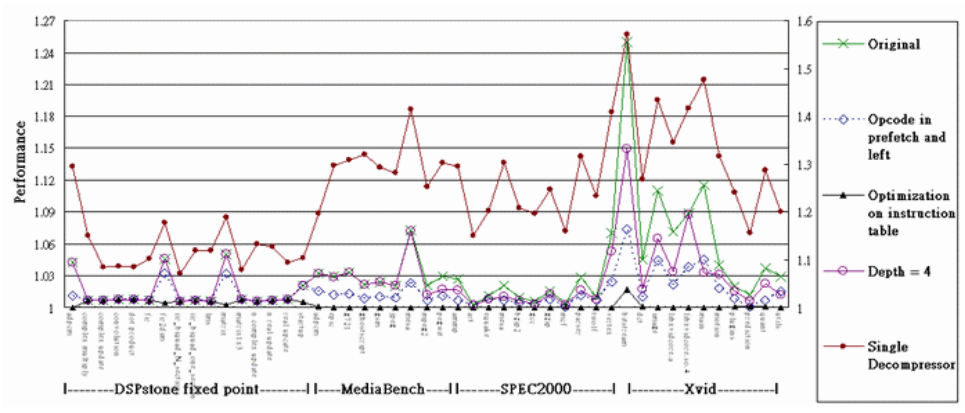


Figure 9. Performance with optimizations

7 Conclusions

In this paper, for an application, we introduce a code compression approach to effectively reduce its code size without causing a great performance loss. In compression, applications will be compressed as a binary tree. To speed the performance of decompression, we also propose an instruction prefetching mechanism and some optimization techniques for improving compression ratio and performance. The experimental results show that our work can achieve a code size reduction of 33%

on average and a performance degradation of 3% measured by the number of instruction fetches.

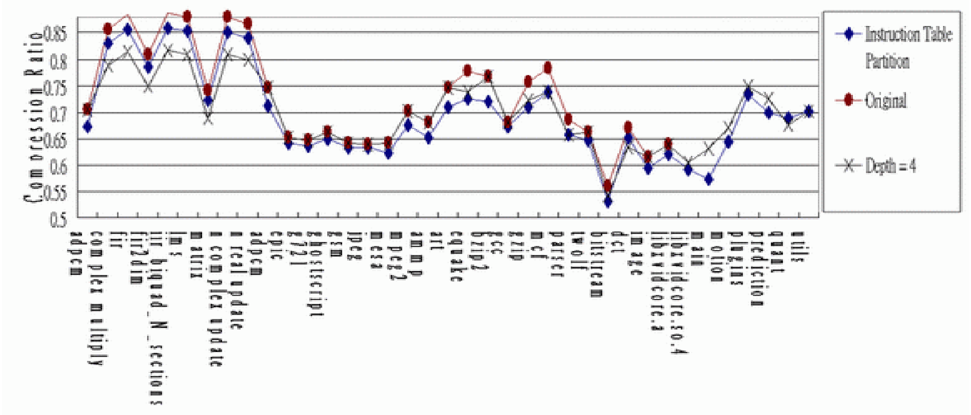


Figure 10. Compression ratio with optimizations

References

1. ARM. ARM7TDMI (Rev4) Technical Reference Manual. Advanced RISC Machines Ltd., (15 May 2003).
2. ARM. An Introduction to Thumb. Advanced RISC Machines Ltd., (March 1995).
3. T. Bell, J. Cleary, and I. Witten. Text Compression. Prentice Hall, (1990).
4. Saumya Debray, and William Evans: Profile-Guided Code Compression, Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).
5. Jens Ernst, William Evans, Christopher W. Fraser, Steven Luco, and Todd A. Proebsting: Code Compression, Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).
6. William S. Evans and Christopher W. Fraser: Bytecode Compression via Profiled Grammar Rewriting, Proceedings of the 2001 International Conference on Programming Language Design and Implementation (PLDI).
7. R.M. Fano: Transmission of Information. Cambridge, MA: MIT Press 1961.C.
8. W. Fraser, T. A. Proebsting: Custom Instruction Sets for Code Compression. Unpublished. Available at http://www.cs. arizona.edu/people/todd/papers/pldi2.ps, Oct. 1995.
9. IBM. CodePack PowerPC Code Compression Utility User's Manually Version 3.0. 1998.
10. Keith, D. Cooper and Nathaniel McIntosh. Enhanced Code Compression for Embedded RISC Processors. Proceedings of the 1999 International Conference on Programming Language Design and Implementation (PLDI).
11. K. Kissell. MIPS16: High-Density MIPS for the Embedded Market. Silicon Graphics MIPS Group, (1997).
12. C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge: Improving Code Density Using Compression Techniques. Proceedings of the 30th Annual International Symposium on Microarchitecture, (December 1997).
13. S. Liao, S. Devadas, and K. Keutzer: Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. Proceeding of the 15th Conference on Advanced Research in VLSI.(March 1995).