Instruction Re-encoding Facilitating Dense Embedded Code

Talal Bonny and Jörg Henkel
University of Karlsruhe, CES - Chair for Embedded Systems, Karlsruhe, Germany
{bonny, henkel} @ informatik.uni-karlsruhe.de

Abstract

Reducing the code size of embedded applications is one of the important constraint in embedded system design. Code compression can provide substantial savings in terms of size. In this paper, we introduce a novel and efficient hardware-supported approach. Our approach investigates the benefits of re-encoding the unused bits (we call them re-encodable bits*) in the instruction format for a specific application to improve the compression ratio. Re-encoding those bits may reduce the size of decoding table by more than 37%. We achieve compression ratios as low as 44% (including all overhead that incurs). We have conducted evaluations using a representative set of applications and have applied it to two major embedded processors, namely MIPS and ARM.*

1 Introduction

Embedded systems often use a relatively slow processor and small memory size to minimize costs. According to the World Semiconductor Trade Statistics Blue Book, there are an estimated 5 billion embedded microprocessors in use today [1]. The reason for the growing popularity of embedded system-driven devices, such as PDAs (Personal Digital Assistants) and web-enabled cell phones, is the sustainable growth of their application, e.g. the world market for embedded software will grow from about $1.6 billion in 2004 to $3.5 billion by 2009, at an Average Annual Growth Rate (AAGR) of 16% [3]. Since the memory chip of the embedded system must be small according to the demands of the embedded market, different techniques are used to reduce the size of the embedded software by compressing it offline and then de-compressing it online. The idea of using code compression as a tool for chip size reduction in microprocessors has mostly incited interest in the area of single instruction issue (usually RISC) processors. In addition to that, the code compression can be beneficial to energy as well, because it reduces the energy consumed in reading instructions from memory and communicating them to the processor core [15, 22, 5, 18].
The code compression techniques can be used either when the ISA (Instruction Set Architecture) is specified or not. When the ISA is specified, the code compression technique utilizes the information in opcodes or instruction format to build the hardware decoder. In this case, the compression ratio will be improved, since the number and the type of operands in the instruction format can be reduced according to the operation defined by the opcode. When the ISA is not specified, the code compression technique follows the traditional data compression methods, which depend only on the statistics of instruction values or Byte patterns. The decoder in this case is simpler than before because it does not take the instruction format into account. The only disadvantage is that the compressed code is not so efficient as the technique with a specific ISA.
When a code compression technique is used, compressed instructions and a decoding table are generated. The size of the decoding table may be around 40% of the memory requirments for a compressed application whereas the other 60% are occupied by the encoded instructions [7].
In this paper, we use a novel code compression technique (hardware-supported) for reducing the size of the decoding table. Our technique is based on the instruction format and the application itself to achieve high compression ratio.
The crux of our compression technique is to find the position of bits in the instruction format suitable for re-encoding. We call those bits *re-encodable bits* [1]. Re-encoding those bits must have no affect on the functionality of instructions. we re-encode those bits to decrease the number of toggles in each table column (as explained in Section 3) and consequently to decrease the size of the decoding table.
Reducing the size of the decoding table will improve the final compression ratio *CR* according to the equation:

$$CR = \frac{\textit{size(compressed instructions) + size(decoding table)}}{\textit{size(original instructions)}} \tag{1}$$

By analyzing a large set of benchmarks (MiBench), we found that the average size of the *re-encodable bits* can reach up to 26% compared to the original code size, as shown in Fig. 8. Those bits may be discarded from the instruction words or re-encoded depending on the compression algorithm used to achieve better compression ratio. We compare the experimental results with the results achieved in our previous work [8] and show that re-encoding the *re-encodable bits* in instruction format may improve the final compression ratio.
The rest of the paper is organized as follows. In Section 2, we present some previous works with comparison to our work. In Section 3, we present our code compression technique. The hardware decoder is introduced in Section 4. Experimental results and performance are presented in Section 5, and we conclude this paper with Section 6.

2 Related Work

The previous works can be categorized into two categories depending on whether the compression technique is ISA specified or it is orthogonal to any architecture. Several related approaches belong to the ISA dependent category. *Thumb* [2] and *MIPS16* [14] are two processors which

[1]A re-encodable bits are bits in the instruction format that may be re-encoded because they are not used for decoding the instruction but just to maintain the word alignment in the memory

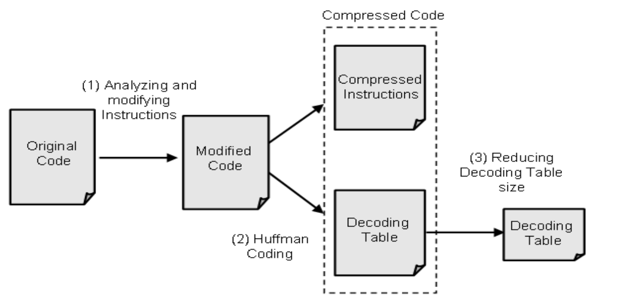


Figure 1: Our code compression technique steps

use this technique. Shorter instructions are achieved mainly by restricting the number of bits that encode registers and immediate values. This results in 30%-40% smaller programs (referring to 70% and 60% compression ratios, respectively) running 15%-20% slower than programs using a standard RISC instruction set. In [16], compiler-driven *Huffman-based* compression with a compressed instruction cache has been proposed for a VLIW architecture. An *Address Translation Table* (ATT) is generated by the compiler, containing the original and compressed addresses of each basic block and its size. An average compression ratio of 65% is reported. the problem in this technique is that the decoder can only decompress full compressed basic blocks in addition to the hardware complexity. In [19], a new dictionary-based compression technique is used which divides the instructions into opcodes and operands to reduce the redundancy and then extracts the sequences and store them in a dictionary. The key idea is to explore the relations between the current operand to be compressed with those already compressed. On an average 46% compression ratio was achieved for the *ARM* processor. The decoder in this technique is complex which can reduce the performance.
There are also several approaches related to the ISA independent category. The IBM CodePack [10] method divides the 32-bit instruction into two 16-bit patterns and encodes both with indices to the dictionaries storing the most frequent patterns. Since high and low halfwords have very different distribution frequencies and values, they are encoded independently. Typically, use of the CodePack compression method results in a 60-65% compression ratio. This technique generated an index table whose overhead is 3% of uncompressed instructions. The worst case for instruction length in compressed code is 38 bits. In [17], the authors extracted common sequences and placed them in a dictionary. Average compression ratios of 61%, 66% and 74% were reported for the *PowerPC*, *ARM* and *i386* processors, respectively. In [6], we introduced a new approach based on *Canonical Huffman Coding* in the dictionary compression scheme. Average compression ratios of 52% and 55% have been achieved for ARM and PowerPC, respectivly.
Our compression technique in this paper belongs to the ISA dependent category. We achieve an average compression ratio of 45% and 48% for MIPS and ARM, respectively. For that, we can obtain not-yet-achieved compression ratios using our approach in comparison to the previous works.

3 Our Code Compression Technique

In our code compression technique, we conduct eventually the following steps (See Fig. 1):
(**1**) We analyze instruction format of the original code to detect the *re-encodable bits* for a specific application. These bits may be re-encoded without effecting on the instruction functionality. We use different techniques to find those bits

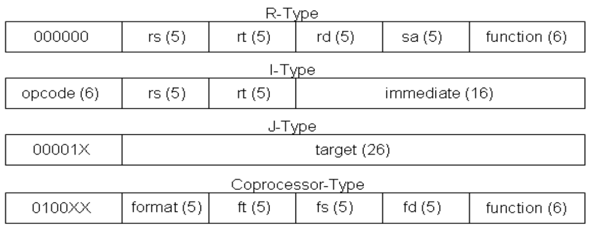
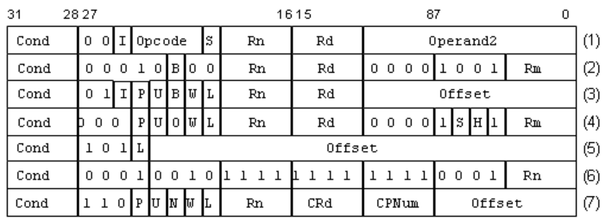


Figure 2: MIPS instruction groups

and to increase their number. The *re-encodable bits* are then replaced with *don't care* symbols 'X'. We call the code in this case "modified code".
(**2**) We compress the modified code using *Huffman Coding* algorithm. The compressed instructions and decoding table are generated.
(**3**) We reduce the size of decoding table by encoding the *don't care* symbols 'X' in each instruction to be identical to the preceding one.
These steps will be explained in details in the following Sections.

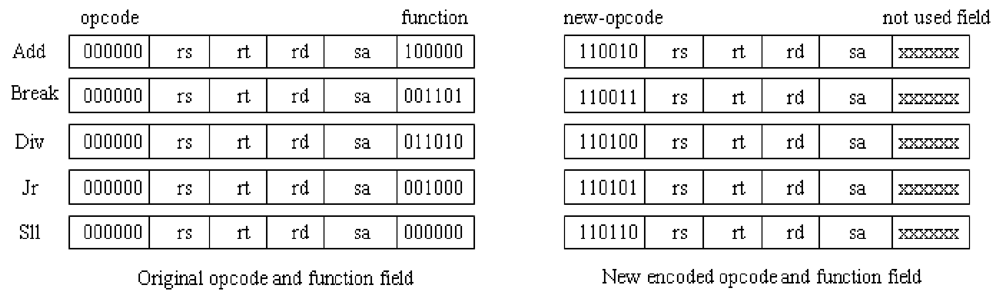
3.1 Analyzing the Instruction Format

The instruction set of any architecture is classified into different groups according to their coding formats. One group may contain instructions that have three register fields. Another one may have instructions with two register fields and immediate operand. Some instructions which have only one register and one target address fields may belong to another group, etc.
The first step in our code compression is to analyze the instruction format for a specific processor architecture and for a specific application. The purpose of that is to detect the *re-encodable bits* in instruction format for that application and then replace them with *don't care* symbols 'X'.
We use three different techniques to detect and increase the number of the *re-encodable bits* for a specific application:
1- Optimizing the opcode size: In this technique, we explore the opcode of each instruction group for a specific application. Most of applications do not utilize all the available opcodes of instruction group. For that, we re-encode the opcodes in this group to have less number of bits than the original ones. The new opcodes cover only the opcodes needed for that application. We replace the remaining unused bits of the opcodes with *don't care* symbols 'X'.
The evaluations has been conducted for two processor architectures MIPS and ARM on a large set of benchmarks (MiBench).
MIPS instructions are classified into different groups according to their coding formats [21]. The opcode can differ


31 28 27 16 15 8 7 0
Cond 0 0 I Opcode S Rn Rd Operand2 (1)
Cond 0 0 0 1 0 B 0 0 Rn Rd 0 0 0 0 1 0 0 1 Rm (2)
Cond 0 1 I P U B W L Rn Rd Offset (3)
Cond 0 0 0 P U 0 W L Rn Rd 0 0 0 0 1 S H 1 Rm (4)
Cond 1 0 1 L Offset (5)
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 (6)
Cond 1 1 0 P U N W L Rn CRd CPNum Offset (7)

(1) Data Processing Type
(2) Swap Type
(3) Load/Store Type
(4) Halfword Data Transfer Type
(5) Branch Type
(6) Branch Exchange Type
(7) Coprocessor Data Transfer Type

Figure 3: Example for ARM instruction groups



Original opcode and function field

	opcode					function
Add	000000	rs	rt	rd	sa	100000
Break	000000	rs	rt	rd	sa	001101
Div	000000	rs	rt	rd	sa	011010
Jr	000000	rs	rt	rd	sa	001000
Sll	000000	rs	rt	rd	sa	000000

New encoded opcode and function field

new-opcode					not used field
110010	rs	rt	rd	sa	xxxxxx
110011	rs	rt	rd	sa	xxxxxx
110100	rs	rt	rd	sa	xxxxxx
110101	rs	rt	rd	sa	xxxxxx
110110	rs	rt	rd	sa	xxxxxx

Figure 4: Example for re-encoding the opcode in the “R-Type” group

from one group to other (see Fig. 2). For example, the opcode of the instruction in “R-Type” group is “000000” and the instruction is specified by the function field which needs also 6 bits. This will reserve 12 bits in the instruction format to decode the instruction. Instead, we can re-encode the opcode field with a new code (which is not used by the instructions in the other MIPS instruction groups) and replace the 6-function-field bits with *don’t care* symbols ’X’. If the application use only 32 different “R-Type” instructions (or less), then only 5 bits can be used for re-encoding the opcode field and one bit can be replaced with ’X’. Fig. 4 shows an example for different instructions in the “R-Type” group before and after re-encoding the opcode. The new opcode is selected to be unique and not used by the other MIPS instruction groups.
In “J-Type” group, all instructions use the opcodes “00001X” (see Fig. 2).
In “Coprocessor-Type” group, all instructions use the opcodes “0100XX”. The floating point instructions use opcode “010001”. The format field (5 bits) and the function field (6 bits) in the instruction format are used to specify the instructions in this group (see Fig. 2). Actually, we do not need all these bits for specifying the instruction. For that, we can use the function field to specify the opcode (which can accept up to 64 different floating point instructions) and replace the bits in the format field with *don’t care* symbols.
ARM instructions are also classified into different groups according to their coding formats [9]. All instructions are conditionally executed depending on the instructions condition field (Fig. 3). This field (bits 31:28) determines the circumstances under which an instruction is to be executed. ARM instructions contain primary opcode and secondary opcodes. For example, the instruction in “SWAP” group has primary opcode “00010” (bits 27:23) and three secondary opcodes “00” (bits 21:20), “0000” (bits 11:8) and “1001” (bits 7:4). We investigated the opcodes in all groups and found that the secondary opcode in “SWAP” group “0000” (bits 11:8) may be replaced with symbols “XXXX” without collisions. Another example is the instructions in “Halfword Data Transfer” group. Their secondary opcode “0000” (bits 11:8) may be also replaced with symbols “XXXX” without collisions. The same scenario can be applied to the “Branch Exchange” group. Its opcode has ’24’ bits which can be shortened with a new one. The new opcode must be unique for a specific application and the remaining bits may be replaced with symbols ’X’.
2- Finding the unused register fields: In this technique, we explore the unused register fields of each instruction in a group and then replace them with *don’t care* symbols ’X’. In **MIPS** architecture, for example, some instructions in “R-Type” group utilize the ’rs’, ’rt’ and ’rd’ register fields and leave the ’sa’ field unused. Other instructions use two registers, one register or even do not use any register field, like “Break”. All the unused register fields may be replaced with ’X’. The same thing can be applied to the other MIPS groups. Fig. 5 shows an example for some instructions with unused register fields after replacing them with ’X’.
In **ARM** architecture, the instructions in “Data Processing” group can reach more than 50% compared to instrcutions in all groups for a specific application. “MOV” and “MVN” instructions, which their frequencies can reach more than 50% compared to other instructions in this group, utilize the ’Rd’ register field (bits 15:12) and “Operand2” (bits 11:0), leaving the ’Rn’ register field (bits 19:16) unused (Fig. 3). This register field may be replaced with symbols ’X’. The same scenario can be done in all groups.
3- Reducing the size of immediate or target offset fields: This technique can detect high number of unused bits compared to the other previous techniques. Normally, the immediate or target offset values occupy the least significant bits in their fields, leaving the most significant bits in the field either unused or less frequently used. For that, we search in these fields the most frequent sequence of bits through all instructions starting from the most significant bits side toward the least significant side and for a specific application. The most frequent sequence of bits (we call them pattern) may be replaced with symbols ’X’. Of course, we leave some bits to distinguish between those instructions who have the symbols ’X’ and the instructions who have not. Algorithm 1 shows pseudo code for re-encoding the immediate or target offset field.
In **MIPS** architecture, this technique can be applied to all instructions in “I-Type” and “J-Type” groups. In addition to that, in “J-Type” group we can replace the last two least significant bits (bits 1:0) with symbols ’X’ because we know that they are always ’0’ since the instructions are word-aligned. When we decode these instructions, we have to replace these bits again with zeros.
In **ARM** architecture, this technique can be applied to all instructions in “Load/Store”, “Branch” and “Coprocessor Data Transfer” groups and most of the instructions (which need Immediate or target offset fields) in “Data Processing” group (Fig. 3).

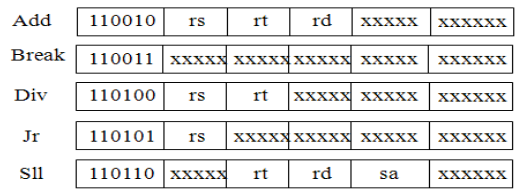
3.2 Huffman Coding Algorithm

The second step in our compression technique is to use the *Huffman Coding* algorithm (Fig. 1). *Huffman Coding* [4] is an entropy encoding algorithm based on the estimated probability of occurrence for a block of code (which can be one or sequence of instructions). The most frequently occurring blocks are encoded with short codewords, whereas the less frequently occurring ones are encoded with large

Algorithm 1 : Re-encoding immediate field

```
n: # of instructions which has immediate value
imm_len: length of immediate value in bits
j: length of selected pattern in bits

1. for (each instruction  i  of  n) {
2.         j = 1
3.         while  (j  <  imm_len){
4.            Freq = frequency of j for all instructions
5.            Gain = Freq x j
6.            j = j + 1
7.         }
8.         Find the biggest Gain for pattern j in instruction i
9.  }
10. Find the biggest Gain for pattern j in all instructions
11. Replace the pattern j with symbols 'X'
```



Add	110010	rs	rt	rd	xxxxx	xxxxxx
Break	110011	xxxxx	xxxxx	xxxxx	xxxxx	xxxxxx
Div	110100	rs	rt	xxxxx	xxxxx	xxxxxx
Jr	110101	rs	xxxxx	xxxxx	xxxxx	xxxxxx
Sll	110110	xxxxx	rt	rd	sa	xxxxxx

Figure 5: New instruction format with don't care codewords. We call these codewords "compressed instructions". In this way, the average codeword length is minimized. It is obvious however that, if all distinct blocks in a code appear with the same (or nearly the same) frequency, then no compression can be achieved. We selected *Huffman Coding* as a compression technique to encode the instruction object code. In *Huffman Coding*, the compressed instructions are used as indices to the original unique instructions which are stored in a decoding table. The original instructions are retrieved by decoding the compressed instructions using the decoding table.

The factor of measuring the compression efficiency is called "compression ratio" *CR* (Eq. 1). This factor is affected by the size of the compressed instructions and the decoding table. The size of the compressed instructions (in Byte) depends on the frequency of these instructions, and consequently is determined by the average number of bits assigned to the instructions for compression. The size of the decoding table (in Byte) is determined by the number of unique instructions extracted from the application, and by the size of the original instruction word (i.e. 4 Bytes in MIPS or ARM instruction word).

To reduce the size of the decoding table, we re-encode the *re-encodable bits* in the instruction format (which we extracted and created in section 3.1) as a primary step to further achieve reduction in the decoding table when we use table compression technique.

3.3 Reducing the Size of Decoding Table

As explained in Section 3.2, the *Huffman Coding* algorithm generates variable length compressed instructions and decoding table which is used to retrieve the original instructions. The decoding table contains the original unique instructions. The compressed instructions are used as indices to the decoding table. As the compressed instructions have variable length, they can not be used as indices to only one decoding table. For that, we divide the decoding table into different decoding tables as many as we have different compressed instruction lengths. In this case, the compressed instructions which have the same code length are used as indices to the same decoding table.

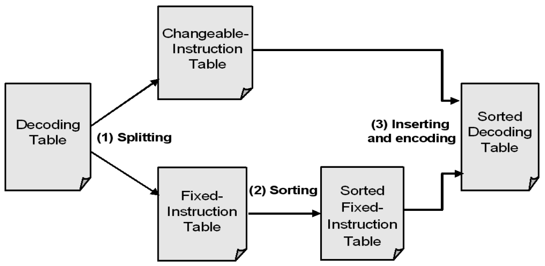

Changeable-Instruction Table
Decoding Table
(1) Splitting
Fixed-Instruction Table
(2) Sorting
Sorted Fixed-Instruction Table
(3) Inserting and encoding
Sorted Decoding Table

Figure 6: Steps for reducing the size of the decoding table

The compressed instructions are stored non-contiguously in each decoding table. This will take space in memory and diminish the benefits which may be achieved using code compression. To overcome this problem we use *Canonical Huffman Coding*. This coding technique is also prefix-free-code like *Huffman Coding* algorithm. It re-encodes the compressed instructions in such way that the instructions with the same length are binary representations of consecutive integers [20]. Reducing the size of any decoding table may be achieved by reducing the size of its columns. For that, we use our Look-Up Table compression method which we used in [8]. In this method we compress each decoding table column by storing the address where the transition (0—>1 or 1—>0) in each column happens in the decoding table instead of storing the complete column. If the size needed to store these addresses is less than the size of the complete column, the column can be compressed. Otherwise, we leave the column without compressing. The crux of our compression technique is to reduce the number of transitions happening in each decoding table column. This can be achieved by **(1)** selecting good sorting algorithm to sort the decoding table entries. **(2)** encoding the *don't care* symbols in the instruction format to be identical to the preceding instruction in the decoding table.

Fig. 6 shows the three steps we use to reduce the size of the decoding table. These steps need to be applied to each decoding table. They are performed off-line (at the design time). Hence, it is no matter for how long time these steps will take.

In the first step, we separate the decoding table into two tables: the fixed-instruction table and the changeable-instruction table. The fixed-instruction table contains the instructions which can not be changed, i.e. the instructions which do not have any *don't care* symbol 'X'. The changeable-instruction table contains the instructions which has at least one 'X' symbol.

In the second step, we sort the entries of the fixed-instruction table to minimize the number of transitions in each column. This will compress more table columns and achieve better table compression. We sort the table entries using the *Lin-Kernighan* sorting algorithm [12] (as we did in [8]).

In the third step, we insert the instructions of the changeable-instruction table in the sorted fixed-instruction table in the position where each inserted instruction must be identical in the most bits to the preceding one. The 'X' field of the inserted instruction must be encoded to be the same as that field of its former instruction.

Another technique may be used to reduce the number of bit transitions in a table columns. This can be done by swapping the position of register fields in instruction format (if this will not change the instruction functionality). For example, in MIPS architecture, some instructions use the register fields 'rs', 'rt' and 'rd'. Those instructions seem to be identical if the positions of the register fields 'rs' and 'rt' are reversed. Hence, we can swap the position of these two registers to make these instructions completely identical. This will have no effect on the instruction functionality, but on the other hand, will decrease the number of the unique instructions and consequently will reduce the size of the compressed instructions as well. The following addition instructions, for example, are identical in the functionality but they have different opcode:

Add r5, r3, r4 , *Add r5, r4, r3*

Swapping the registers 'r3' and 'r4' in any of the previous instruction will make them identical.
The previous steps reduce the number of bit transitions in each column. Therefore, we can compress more number of columns in the decoding table by storing in each column only the address where the bit transition happens instead of storing the complete column. This will reduce the size of the decoding table.

4 Hardware Decoder

The hardware decoder decodes the compressed instructions in two stages. In the first one, the length of the compressed code is computed (Fig. 7). This can be done by using number of comparators as much as there are different compressed code length, i.e. one comparator for each length. The incoming 32-bit from the memory is stored in a 32-bit register and shifted to a shift register whose length is equal to the longest table index 'L'. These 'L' bits code is compared with the minimum index of each decoding table simultaneously and the corresponding comparator refers to the actual code length. The task of the 32-bit register is to keep the shift register filled each time its content is reduced. The second stage in the decoding is to retrieve the original instruction from the specified decoding table. This can be done by finding out the number of bit transitions in each compressed column for the incoming compressed code. If the number of bit transitions is even, the bit in the corresponding column is '0'. Otherwise, it is '1'.
When the instruction is decoded, it needs a slight modification to match the original instruction format. For example, in "R-Type" group of MIPS architecture, the "000000" is assigned to the opcode field and the correct instruction function is assigned to the function field.
The decoder has been described in VHDL, synthesized using Xilinx ISE8.1 for VirtexII and implemented on a scalable FPGA platform *"Platinum"* from *Pro-Design* [13]. An average access time of 4 *ns* was achieved and around 1200 slices were used.

5 Experimental Results

In this Section, we present the experimental results of our code compression technique. We conducted the results for two embedded processor architectures, MIPS (4KC) and ARM (SA-110). For both architectures the *MiBench* [11] benchmark suite is considered as a representative (in terms of application domains and size) set of applications. We have compiled the benchmarks using cross-platform compiler for MIPS and ARM target architectures, and kept the default flags as they provided by the *MiBench* package.
The experimental results are presented in Figures 8 - 11. In each diagram, the bar labeled "Average" shows the average across all benchmarks in that diagram.

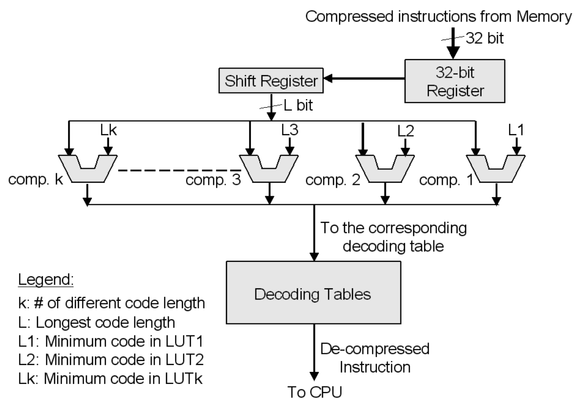


Figure 7: Hardware Decoder

From the experimental results we can observe and conclude the following:
1- Our compression technique analyzes the instructions to find out the *re-encodable bits* and to increase the number of these bits. Fig. 8 shows how large the number of these bits can be for different applications compiled for MIPS and ARM processors. The size of the *re-encodable bits* in the instructions can reach up to 26% and 22% of the size of the whole instructions, for MIPS and ARM processors, respectively. This ratio differs depending on the instruction format and the number of instructions in the application.
2- Encoding the *re-encodable bits* in instruction format to be identical to the preceding instruction reduces the size of decoding table (as declared in section 3.3). To show the efficiency of this technique, we compare the size of decoding table before and after encoding the *re-encodable bits* in the instruction format. The results are presented in Fig. 9. This figure shows that encoding those bits may reduce the size of decoding table on an average by 37% and 35% for MIPS and ARM processors, respectively.
3- The compressed code contains compressed instructions and compressed decoding tables. Hence, encoding the *re-encodable bits* reduces the size of compressed code because the size of decoding table is reduced. Consequently, the compression ratio is improved when the *re-encodable bits* are re-encoded (Fig. 10). The compression ratios achieved differ between 44% and 47% for MIPS processor (on an average 45%) and between 45% and 50% for ARM processor (on an average 48%), depending on the size of the application and the instruction format of the processor. For the large applications, our compression technique gives better results. This has been expected since large number of instructions imply more *don't care* fields and this gives more reduction in the compressed instruction size. In addition to that, the large number of instructions result in large decoding tables and this gives more chances to re-order their entries and to achieve better table compression.
4- Finally, code compression does not entirely come for free. On the plus side, it does reduce the code size and therefore, minimizes memory requirements even when factoring in the hardware for decompression and decoding tables: a large net gain remains. However, a performance loss is the price to pay for (see Fig. 11). This figure shows the time taken by the original and the compressed code (in Million of cycles) for MIPS and ARM processors. In our case performance loss is due to the time needed to fill the shift register (see Fig. 7) with the incoming compressed instructions, every time a branch instruction occurs. The 4 *ns* latency of the decoding hardware we reported earlier could

be further reduced if the decoding hardware would be built into the CPU itself as part of the instruction decode phase. This is a point for future work.

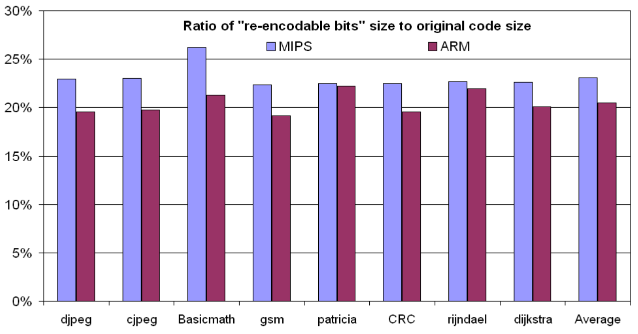


Figure 8: Ratio of "re-encodable bits" size

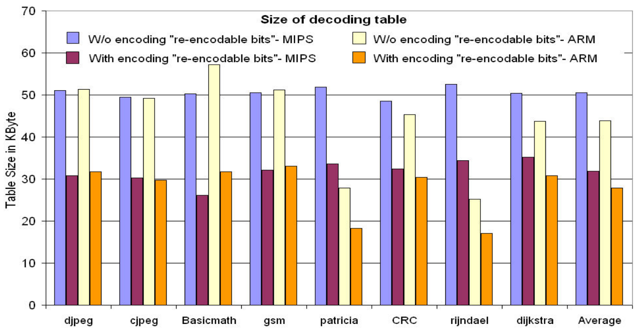


Figure 9: Size of decoding table for MIPS and ARM

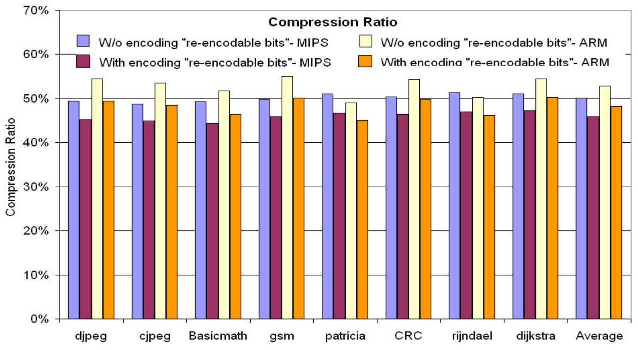


Figure 10: Compression ratios for MIPS and ARM

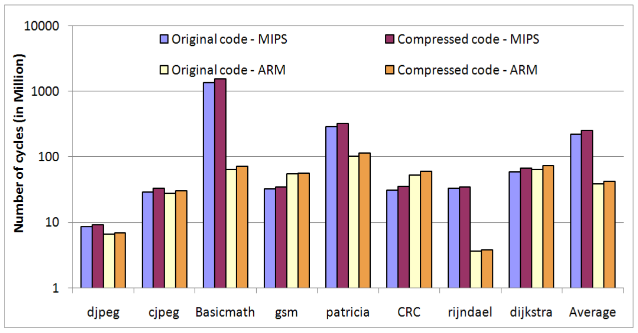


Figure 11: Time taken by the original and the compressed code for MIPS and ARM processors (in Million of cycles)

6 Conclusion

We have presented a new ISA dependent approach for embedded system code compression. Unused fields of the instruction format are extracted and then encoded to reduce the size of the decoding table. Our approach can be applied to any processor architecture if the ISA is known. We achieve an average compression ratio of 45% and 48% for MIPS and ARM processors, respectively, with a little impact on performance. Hence, our scheme is the basis for not-yet-achieved compression ratios in hardware-based schemes.

References

[1] *http://www.wsts.org*.
[2] Thumb squeezes ARM code size. *Microprocessor Report*, 1995.
[3] Report RG-229R Future of Embedded Systems Technology from Business Communications Company. 2005.
[4] T. Bell, J. Cleary, and I. Witten. Text Compression. *Prentice-Hall, Englewood Cliffs*, 1990.
[5] Y. Benini, A. Macii, E. Macii, and M. Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. *ISLPED-99*, pp. 206-211, 1999.
[6] T. Bonny and J. Henkel. Using Lin-Kernighan Algorithm for Look-Up Table Compression to Improve Code Density. *Proc. of the 16h Great Lakes Symposium on VLSI-(GLSVLSI'06)*, pp. 259-265, 2006.
[7] T. Bonny and J. Henkel. Instruction Splitting for Efficient Code Compression. *Design Automation Conference (DAC'07)*, pp. 646-651, 2007.
[8] T. Bonny and J. Henkel. Efficient Code Density Through Look-up Table Compression. *Proc. of IEEE/ACM Design Automation and Test in Europe Conference (DATE'07)*, pp. 809-814, 2007.
[9] S. Furber. ARM System-on-Chip Architecture (2nd Edition). *Addison Wesley Trade Computer Publ.*, 2000.
[10] M. Game and A. Booker. Code Pack code Compression for PowerPC Processors. *PowerPC Embedded Processor Solutions, IBM*, 2000.
[11] M. Guthaus and J. R. et al. MiBench: a free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, pp. 3-14, 2002.
[12] K. Helsgaun. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *European Journal of Operational Research*, pages Vol. 126, Issue 1, pp. 106–130, 2000.
[13] http://www.prodesigncad.com.
[14] K. Kissell. MIPS16: high-density MIPS for the embedded market. *Silicon Graphics MIPS Group*, 1997.
[15] A. M. L. Benini and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. *International Symposium on Low Power Electronics and Design*, pp. 322-327, 2001.
[16] S. Larin and T. Conte. Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors. *Proc. of the Annual International Symposium on Microarchitecture*, pp. 82-92, 1999.
[17] C. Lefurgy, P.Bird, I.Chen, and T.Mudge. Improving Code Density Using Compression Techniques. *Micro-30*, pp. 194-203, 1997.
[18] H. Lekatsas, J. Henkel, and W. Wolf. Code Compression for Low Power Embedded Systems Design. *Design Automation Conference DAC-00*, pp. 294-299, 2000.
[19] C. Lin and C. Chung. Code Compression Techniques Using Operand Field Remapping. *IEE Proc. in Computer and Digital Tech., Vol. 149, pp. 25-31*, 2002.
[20] Y. Nekritch. Decoding of Canonical Huffman Codes with Look-Up Tables. *Proceedings of the Conference on Data Compression, pp. 566,*, 2000.
[21] D. Sweetman. See MIPS Run. *Morgan Kaufmann, ISBN 1558604103*, 1999.
[22] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An Object Code Compression Approach to Embedded Processors. *ISLPED-97*, pp. 265-268, 1997.