Code Compression for RISC Processors with Variable Length Instruction Encoding

S. S. Gupta, D. Das, S.K. Panda, R. Kumar and P. P. Chakrabarty
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur
Kharagpur – 721 302, India
rkumar@cse.iitkgp.ernet.in

Abstract

Most of the work done in the field of code compression pertains to processors with fixed length instruction encodings. In this work use code compression on variable length embedded RISC processors whose encodings are already optimized to a certain extent with respect to their usages. In this paper we present a dictionary based algorithm which addresses issues arising out-of variable lengths. We include results for compression and performance.

1. Introduction

For embedded systems, there has been an increased awareness in recent years on decreasing the memory requirements of applications running in such systems [1-2]. Code compression reduces the code size of the program to be run thus reducing the overall memory requirements along with reducing the cycles needed, system bus activity and power requirements [2-3], [9].

Code compression techniques can be broadly divided into two types i.e. the compiler driven optimization techniques [1] and the post compiler compression techniques [4]. Debray et. al. have presented a survey on the compiler techniques for code compression, and listed various methods used by the compiler to reduce code size, e.g., dead code elimination, strength reduction and code-factoring [1]. The other class of techniques involves object code compression which is done offline, and run-time decompression while actually executing the code. In this paper we focus on the on the later class of techniques.

Compression of object code differs from normal data compression as it should provide for incremental decompression which implies that decompression can start from any point in the code and not necessarily from the start. LZW type of dictionary-based codecs and entropy based Huffman codes can not be applied because they cannot handle incremental decompression. Secondly, such codes are not preferred because it will lose the memory alignment due to the variable length codewords leading to degradation in performance.

Most of the previous work on code compression has been on fixed length embedded RISC processors. In this work, we present a code compression algorithm for variable length RISC processors. The achievable compression for a variable length processor is lesser than that for a fixed length processor as the instructions are already entropy coded to some extent. Thus even in uncompressed instructions-encoding for variable lengths, the more frequently occurring instructions have shorter lengths and less frequent ones have longer lengths thus reducing the overall code size for the variable length processor.

In this paper, first we briefly review, in section 2, the work done for fixed-length instruction sets. Then, we present our algorithm in section 3, followed by the results in section 4. Finally, we conclude the paper along with directions for future work in section 5.

2. A Review

The salient features of a code compression algorithm are the type of encoding, the handling of branch instructions and how decompression is done. There are broadly two types of compression techniques – statistical coding and dictionary coding. In statistical coding individual symbols are replaced by different sized codewords depending on their frequency of occurrence. Huffman coding is an example of such coding. Dictionary coding on the other hand tries to replace a symbol or a series of symbols with a dictionary index. Change of addresses due to compression can be handled for branch instructions by either a translation table containing a translation from the original address to the new address or by the branch offsets to the branch instructions [4].

Lefurgy et. al. [4] described a greedy algorithm which replaces groups of instructions with a dictionary entry which are decompressed back at run time. This places a restriction that the branch target instructions cannot be compresses unless they are at the start of such a group. This is because if the branch target instruction occurs in the middle of such a group it cannot be accessed directly without accessing all the other previous instructions in the group. They have used illegal opcodes as a way to distinguish between ordinary instructions and compressed instructions. A lot of other work is available in literature which we are unable to include here because of space limitation.

One of the most widely used industry standard is the THUMB technique [5]. In this technique which is used in the 32 bit ARM processor, the frequent instructions which use lesser operands/fields are represented using 16 bits getting rid of the redundant information bits and are decompressed back to the 32 bit instruction while execution. In case of a variable length processor such frequent instructions requiring lesser operands/fields already form a part of the main processor instructions with smaller encoding. Thus there is not much of a scope of applying this kind of a technique to variable length processors. Since the THUMB code generation has been made a part of the compilation process the issue of changing of addresses of branch instruction due to compression does not arise.

The processor when running in the ARM mode executes normally whereas in the THUMB mode the 16 bit instruction is decoded to the original 32 bit instruction on-the-fly before decode. Using THUMB instructions instead of ARM instructions gives good compression but leads to performance degradation because of more number of instructions being executed. In case of a code consisting of mixed ARM and THUMB instructions the mode change is done using a mode switch instruction. These switch instructions introduce compression overhead into the code. Krishnaswamy and Gupta [6] have introduced profile guided algorithms for generating mixed ARM and THUMB code to get substantial compression without degradation in performance. The algorithms work on replacing those series of THUMB instructions with ARM instructions for which the compression and performance improve. MIPS16 is a similar extension to the MIPS processor architecture where frequently occurring 32/64 instructions have been represented using 16 bits [7]. These are converted to the original instruction before execution.

In IBM's PowerPC processor's CodePack technique each instruction is broken into two halves and each of the halves is entropy coded [8]. Due the resulting variable sizes of the encoded instructions an index table is needed to map the old instruction addresses to the new ones. In order to decrease the index table overhead the instructions are compressed and stored in groups so that only one address translation is needed per group. The performance degradation due to the variable lengths is avoided by keeping an output buffer which keeps a group of decompressed instructions available for fast access. This technique can not be directly applied to variable length processors because splitting the instructions will lead to different sized halves and we may not get much repetition of the halves thus effectively resulting in very less compression.

3. Algorithm

The basic idea behind our algorithm is to reduce the size of the frequent instructions while maintaining byte level alignment. We have used a dictionary based method because of the fast decompression thus leading to good performance.

One important issue which does not come up during compression in fixed length processors, is the instruction lengths. However in case of variable length processors we need to know the instruction boundaries in order to know the frequent instructions. This can be done by simply using an object file viewer tool which gives the instruction lengths. Executing the code in a first pass helps to know all the branch targets. This essentially leads to a two pass compression process. We take the most frequent 256 instructions (256 because they can be coded in 8 bits) and form a dictionary of these instructions. In case of a variable length processor the dictionary needs to be arranged in such a way as to keep entries of the same length together so as to be able to index the dictionary properly. This problem does not arise in the case of a fixed length processor as all dictionary entries will be of same length.

We then replace the occurrences of these instructions with the dictionary indexes. This results in a mixed code consisting of instructions and dictionary indexes. In order to be able to distinguish between an instruction and a dictionary index we keep a bit per instruction suggesting whether it has been encoded or not. Attaching this bit with the instruction would in turn make us lose the byte alignment so we keep all these bits together separately in a bit stream stored in memory as bytes.

Since the addresses of the instructions change due to compression, we form a branch address table (BAT) to map the original address to the new address. This table also contains the address and position of the relevant bit for the branch target. One problem with fixed length RISC processors is that the branch targets are aligned to instruction word boundaries. Compressing the instructions leads to loss of alignment. On the other hand in case of variable length instructions the alignment is already at the level of the smallest instruction length and hence compressing the instructions does not pose a major problem.

While running the code, the bit indicators are checked for each instruction. In case it indicates a dictionary entry the instruction is accessed by indexing into the dictionary. In a variable length processor we also need to know the instruction's length that is being accessed from the dictionary. However that is known from the index if the instructions are arranged with same lengths together in the dictionary. If the bit indicator indicates a normal instruction execution occurs normally. In case of a pipelined model getting the decoded instruction from the dictionary would lead to an extra stage in the pipeline which may be called the post-fetch stage. In case of a branch instruction a translation is done by indexing into the BAT. This also gives the address of the corresponding bit indicator to start from.

4. Results

We used 16 and 32 bit variable length RISC processors for our experiments. We developed a pipelined simulator with an additional post-fetch stage for decompression. This simulator was used to run the compressed code for various benchmarks and performance obtained. Figure 1 shows the size of the original code, the compressed code, the dictionary and the branch address table for the various benchmark-applications. The compression ratio is then calculated as the ratio of the total size of the resulting code and the data structures to the size of the original code. This method provides compression up to 33 %.

The second set of results related to the memory access done are shown in Figure 2. In general the memory access for the decompressor system was more than 10% less than those in case of the simple pipeline system. The decompression system however required some extra dictionary accesses which were assumed to be high-speed cache accesses.

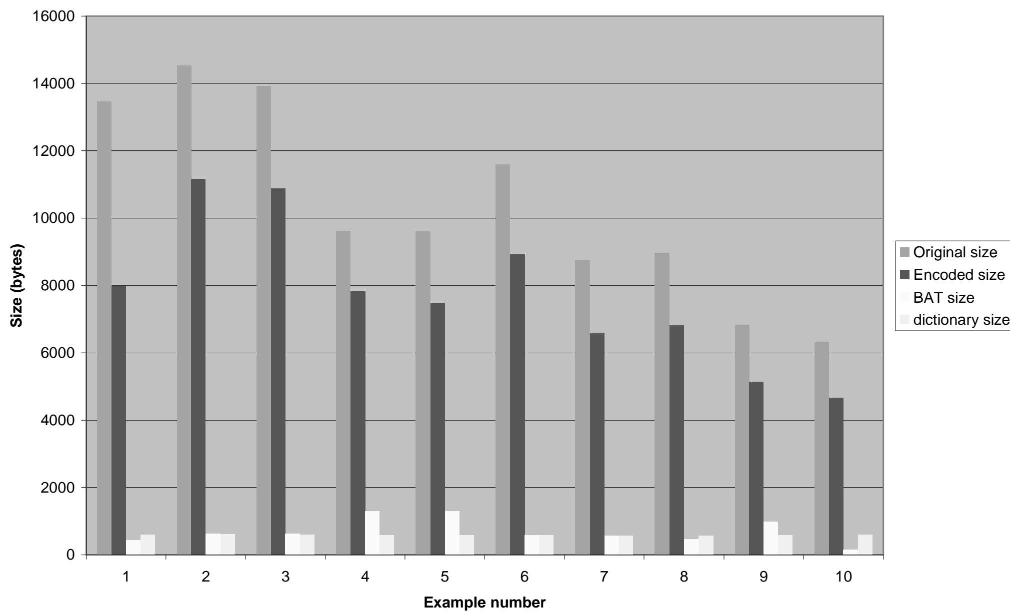

Figure 1: Sizes of the original and compressed codes along with the additional space needed for BAT and dictionary. (Examples used are numbered as: 1. sha 2.Basicmath_large 3.Basicmath_small 4.Bitcnts_large 5. Bitcnts_small 6. qsort_large 7. qsort_small 8. dijkstra_large 9. search_large 10. crc)

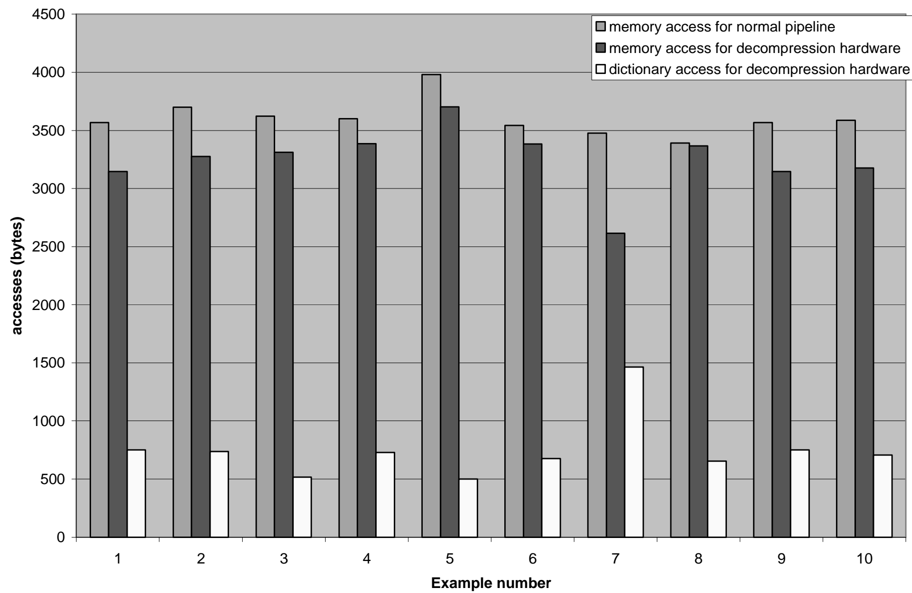

Figure 2. The memory/cache accesses for the normal processor and the one with the decompression system. (Numbers on abscissa are the same as in Figure 1.)

5. Conclusions and Future work

In this paper, we presented an algorithm for code compression suitable for variable length instructions. The algorithm addresses the various issues arising while trying to compress code in variable length processors. For example, we compress instructions individually so the problem of branch targets occurring in between dictionary entries does not arise. We do not need any free processor instruction as we make use of separate bit indicators to mark the status of an instruction. We have gotten rid of the use of switch instructions with the help of bit indicators. Compressed code is byte-aligned. We have included compression as well as the performance results.

Future scope for work lies in getting more performance metrics and some modifications to the algorithm to improve on the compression figures. Also the algorithm can be applied to various processor architectures to get some more insight into the compression.

References

[1] S. Debray, W. Evans, R. Muth and B. D. Jutter. Compiler techniques for code compression. ACM Trans. Programming Language & Systems 22(2) : 378 –415, March 2000.

[2] H. Lekatsas and Wayne Wolf. SAMC: A code compression algorithm for embedded Processors. ACM Trans. Computer-Aided Design, December 1999.

[3] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. Proc. 37th Design Automation Conf., June 2000.

[4] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge. Improving code density using compression techniques. Proc. 30th Annual Int. Symp. Micro-architecture. December 1997.

[5] An Introduction to Thumb, Version 2.0. Advanced RISC Machines Ltd.1995.

[6] A. Krishnaswamy and R. Gupta**.** Profile Guided selection of ARM and Thumb instructions. ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems, June 2002.

[7] MIPS32 Architecture for Programmers Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS32 Architecture.

[8] Mark Game and Alan Booker, CodePack: Code Compression for PowerPC Processors. Version 1.0. IBM Corp., USA.

[9] Guido Araujo, Paulo Centoducatte, Rodolfo Azevedo and Ricardo Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. IEEE Trans. VLSI Systems, October 2000.