Expression Tree Based Algorithms for Code Compression on Embedded RISC Architectures

Guido Araujo, Paulo Centoducatte, Rodolfo Azevedo and Ricardo Pannain

Abstract— **Reducing program size has become an important goal in the design of modern embedded systems target to mass production. This problem has driven efforts aimed at designing processors with shorter instruction formats (e.g. ARM Thumb and MIPS16), or that are able to execute compressed code (e.g. IBM PowerPC 405). This paper proposes three code compression algorithms for embedded RISC architectures. In all algorithms, the encoded symbols are extracted from program expression trees. The algorithms differ on the granularity of the encoded symbol, which are selected from whole trees, parts of trees or single instructions. Dictionary-based decompression engines are proposed for each compression algorithm. Experimental results, based on SPEC CINT95 programs running on the MIPS R4000 processor, reveal an average compression ratio of 53.6% (31.5%) if the area of the decompression engine is (not) considered.**

Keywords— **Code Compression, RISC Architecture.**

I. Introduction

As embedded systems are becoming more complex, the size of embedded programs are growing considerably large. The result are systems in which program memories account for the largest share of the total die area, more than the area of the microprocessor core and other on-chip modules. As a consequence, minimizing program size has become an important part of the design effort (cost) of an embedded system. A way to achieve that is to restrict the size of instructions. This is the approach used in the design of the ARM Thumb and MIPS16 processors. Shorter instructions are obtained mainly by restricting the number of bits that encode registers and immediates. Fewer registers imply in less freedom for the compiler to perform important tasks, like global register allocation, and also more instructions to perform the same amount of computation. The net result are 30%-40% smaller programs running 15%-20% slower than programs using standard RISC instructions [3]. Another way to reduce the size of a program is to design processors that can execute compressed code. In order to do that, the decompression engine has to perform real-time code decompression. Moreover, because programs have branch instructions, decompression should restart from the target of any branch instruction. These are the two major features that distinguish *code compression* from other data compression problems, turning unpractical for this problem well known data compression algorithms like Lempel and Ziv (LZ)[4] and its variations. This paper deals with the problem of finding code compression techniques that allow efficient implementations of real-time decompression engines.

This paper is divided as follows. Section II discusses prior work on the problem of code compression. All algorithms proposed here use expression trees or parts of expression trees to perform compression. Expression trees are constructed as described in [?]. We use expression trees as the basis of our compression algorithms because compilers tend to generate very similar expression trees from program statements. In the first algorithm (Section III), symbols are whole trees and the alphabet is formed by all distinct trees in the program. In the second algorithm (Section IV), trees are decomposed into smaller distinct parts (i.e. *patterns*), which are then encoded. Finally, in the third algorithm (Section V), the alphabet is the set of all distinct instructions from all trees in the program. Using these algorithms, we developed a set of experiments aimed at measuring the final compression ratio[1] which includes the decompression engine size overhead. Section VI compares the resulting compression ratio using each approach. A decompression engine, located between main memory and L1 cache, is described in Section VII. Section VIII summarizes the work and sets new directions.

II. Related Work

Research on code compression has been very active in the compiler community, e.g. [?][?]. Nevertheless, its goal is to find compact program representations rather than enabling real-time code decompression. As a consequence, important issues on decompression engine design, like the engine size and performance are frequently ignored. For example, in Section VI we show that the algorithm which produces the best program compression does not result in the smallest silicon area if the decompression engine area is accounted.

The first approach for code compression in a RISC architecture was originally proposed by Wolfe and Channin [6]. In [6] the compression algorithm is based on encoding byte long symbols in a cache-line using Huffman codewords. The decompression engine discussed by Wolfe and Channin is described in [?]. Lefurgy et al. [3] proposed a code compression technique in which common sequences of instructions are replaced by a single fixed-length codeword. Wolf and Lekatsas [7] associate symbols to instruction fields, and encode field streams using Huffman codewords. The IBM CodePack PowerPC 405 processor is an architecture designed to execute compressed code. In CodePack, Huffman codewords encode sequence of instruction bits within a cache-line.

[1] *compression ratio = size of compressed program / size of uncompressed program*

The main contribution of this paper is a practical and effective approach for the code compression problem, supported by extensive experimental data and a number of engine architectures (see [?] for details). The algorithms we propose have been tested using programs from the SPEC CINT95 benchmark running on the MIPS R4000 processor, and compiled with gcc -Os version 2.8.1.

III. Tree Based Compression (TBC)

In Tree Based Compression (TBC) the alphabet is formed by all unique expression trees in the program. Instructions are collapsed into sequences, each forming an expression tree. We have noticed that the number of distinct trees in a program is much smaller than the total number of trees. On average, distinct expression trees correspond to only 24% of all trees in a program. The selection of the best method to encode trees depends on how they contribute to the program size. In order to determine that, we ordered the set of distinct trees based on how frequent they show up in each program. The cumulative distribution of the distinct trees in the programs was then computed. The result is shown in the graph of Figure 1. On

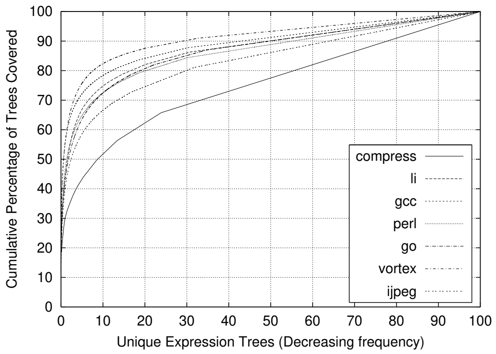

Fig. 1. Percentage of program trees covered by distinct trees.

average, 80% of all program trees are covered by only 20% of the most frequent ones. This suggests that expression trees should be compressed using an encoding that assigns smaller (larger) codewords to (un)frequent trees. Huffman encoding [4] is such an algorithm. In [11] we studied encoding methods based on variations of Huffman codewords for the R4000 processors. Unfortunately, designing fast Huffman decoders is complicated, and usually results in decoders that are more expensive than if fixed-length codewords are used [?].

In order to simplify the design of the decompression engine, we developed a compression algorithm, based on fixed-length codewords, that explores the exponential nature of the tree frequency distribution, while producing very high compression. The algorithm divides the set of distinct trees into n_c classes, each class k having n_k trees. The number of classes (n_c) is determined exhaustively, by exploring all possible partitions from two to eight classes. For each partition of a given number of classes, we determine (again exhaustively) all possible combinations of class sizes and measure their compression ratio. The combination, from all possible partitions, that results in the smallest compression ratio is then selected as the best partition for that program. From this perspective, the goal of the compression algorithm is to perform a piecewise discretization of the frequency distribution shown in Figure 1, so as to minimize the final compression ratio.

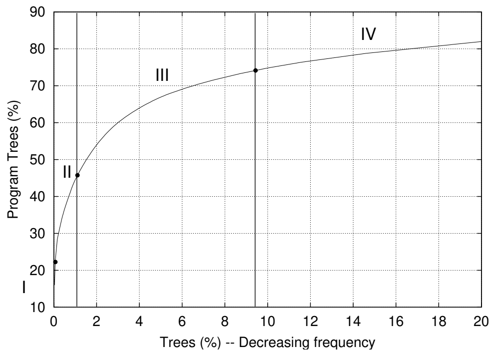

Fig. 2. Discretization of the frequency distribution for program `li` after class partitioning. Class I (IV) has $< 1\%$ ($> 9\%$) of all distinct trees.

Fixed-length codewords of size $\lceil log_2 n_k \rceil$ are then assigned to trees in class k. For each codeword we append a prefix of size $\lceil log_2 n_c \rceil$ bits, that is used by the decoder to identify the class. The compression algorithm substitutes each expression tree in the program by its corresponding prefix and codeword.

Consider, for example, program *li* and a partition of its tree set into four classes. The best compression ratio[2] (23.4%) assigns 1/5/8/12 bits to classes I/II/II/IV. The combination of four classes, that minimizes the compression ratio for program *li*, divides the curve of *li* (Figure 2) into four intervals, each interval corresponding to a class. Once the best compression ratio for a given partition is determined, we repeat the algorithm for other partitions. Figure 3 shows the resulting compression ratios when the tree set for each of the programs in the benchmark is partitioned into 2-8 classes. Notice that the compression ratio decreases as the number of classes increases, until it reaches a minimum, after which it starts to increase again. This occurs because the algorithm automatically assigns smaller (larger) codewords to classes for which the trees have a high (low) average frequency distribution. The more classes are added, the lower is the average frequency difference between two neighbor classes, and the larger is the overhead due to the prefix bits required by the new classes. Eventually, the benefit gained by the discretization is offset by the prefix bits overhead, and the compression ratio starts to increase. It is interesting to notice that, for almost all programs, the minimum compression ratio is achieved when the partition is performed using four classes. In some cases (e.g. *go*) the best compression ratio occurs for five classes. Nevertheless, the average difference of the compression ratio between classes five and four is only 0.09%. The average compression ratio for all programs is 27.2%,

[2] All compression ratio numbers take into consideration the prefix size.

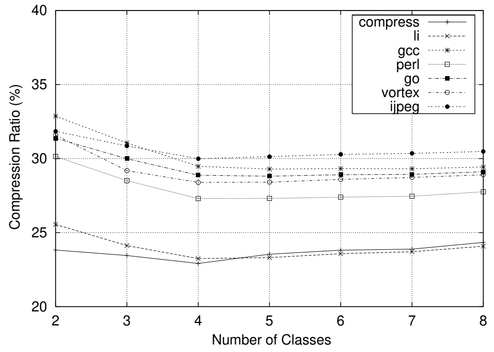

Fig. 3. Compression ratio for different partitions.

and is achieved when four classes are used.

Codewords are allowed to split at the end of each 32-bit words, and bits from split codewords are spilled into the next word. We noticed that small compression ratios can only be achieved if we allow this to happen. This implies that the decompression engine should be able to keep track of codeword boundaries inside the current memory word, and to put together pieces of a split codeword during two consecutive memory fetches. In [?] we propose a variation of TBC for the TMS320C25 DSP.

IV. Pattern-Based Compression (PBC)

The key idea of Pattern-Based Compression (PBC) is an operation that factors out the operands (*operand-patterns*) from the expression trees of a program. The factored expression trees are called *tree-patterns*. We call the task of removing operands from an expression tree *operand factorization* [11]. An operand-pattern is formed by traversing the instruction sequences in the expression tree, listing the operands when they are encountered. The sequence of opcodes without its operands form the tree-pattern. In [11] we have designed a number of experiments to evaluate operand factorization based compression. As before, the individual frequencies of each unique tree and operand patterns were determined, and the cumulative percentage of the expression trees covered by these patterns was computed. Again, the frequency of tree and operand patterns decreases almost exponentially as the pattern becomes less and less frequent. The compression ratio for tree (operand) patterns is on average 13.0% (26.8%), and is achieved when tree (operand) patterns are divided into 4 classes. The final compression ratio, when the codewords for both patterns are combined, is on average 39.8%. We designed a decompression engine for the PBC algorithm [?] and it contributed 21.5% to the final compression ratio (61.3%).

V. Instruction-Based Compression (IBC)

IBC is motivated by the large percentage of expression trees that are composed of single instructions. Our experimental results [?] reveal that, in general, frequent trees have very few instructions, the most frequent of them being single instruction trees. Rare trees are also fairly small, while medium frequency trees are larger (2-4 instructions). On average, all instructions in a program are replica of only 18.3% of its instructions, and a similar exponential behavior was again observed for single instructions [?]. The same compression algorithm employed in Section III and Section IV was used here as well. The resulting final compression ratio was on average 31.5%, and again it is achieved using only four classes. The decompression engine for the IBC algorithm [?] contributed 22.1% to the final compression ratio (53.6%).

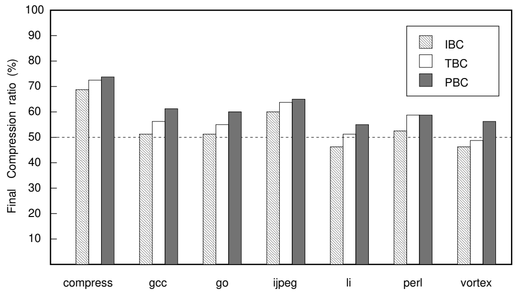

Fig. 4. Final compression ratio for TBC/PBC/IBC.

VI. Algorithms Comparison

Figure 4 shows the final compression ratios for all programs, using all three algorithms discussed above. The average compression ratio for TBC/PBC/IBC are respectively 60.7%/61.3%/53.6% (27.2%/39.8%/31.5%) if the decompression engine overhead is (not) included. Although the use of operand factorization in PBC results in a small compression ratio, the improvement is jeopardized by the presence of two sets of prefix bits (one for each pattern) instead of one. Notice that IBC (TBC) produces the best ratio if the engine overhead is (not) included. The reason is that although two entries in the TBC dictionary store distinct trees, the trees can have similar instructions, while entries in the IBC dictionary are unique instructions. Hence, the best compression ratio for program compression does not lead to the smallest silicon area if the engine size is considered.

VII. The IBC Decompression Engine

Because IBC results in the best compression ratio we show here its corresponding decompression engine. TBC and PBC decompression engines are described in details in [?]. The decompression engine for IBC is shown in Figure VI. A codeword `Ic` is extracted from a memory word, and decoded by the Instruction Generator (`IGEN`) that outputs a pointer (`iaddr`) to the Instruction Dictionary (`ID`). Each entry in `ID` stores a single uncompressed instruction, that is passed to the processor. The area used by the decompression engine is basically the size of dictionary `ID` (average 18.3%) plus the size of the `ATT` module (average 3.8%), which is responsible for mapping uncompressed addresses to compressed addresses.

In our architectural model, the processor executes uncompressed instructions that generate uncompressed address requests, while memory stores compressed instructions. During the execution of branch/jump instructions,

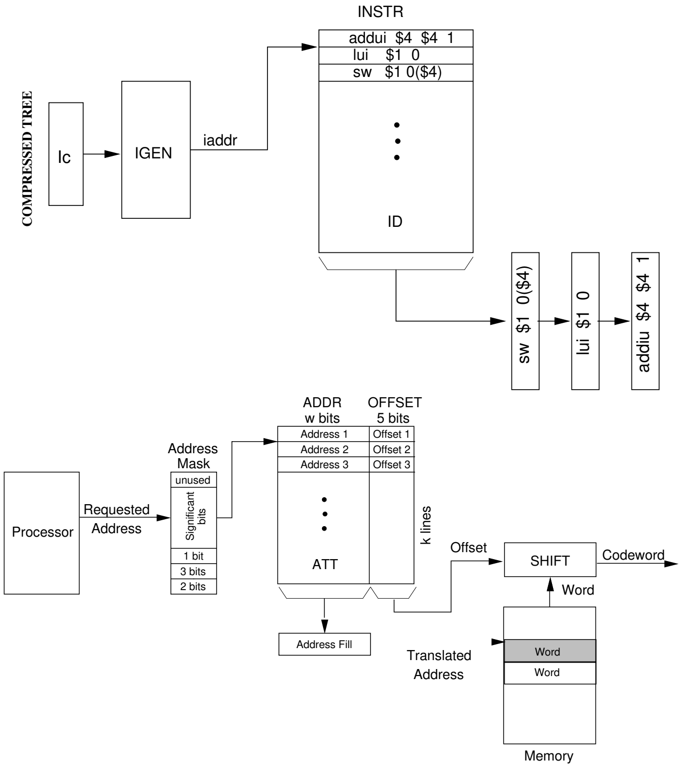

Fig. 5. Decompression engine for the IBC algorithm.

the address requested to memory by the processor changes from the address of the next instruction to some arbitrary (uncompressed) address. In order to satisfy this request, the decompression engine should be able to map (uncompressed) processor addresses to (compressed) memory addresses. To make this possible, we propose an address translation module (Figure VI), where the mapping is performed using the `Address Translation Table` (`ATT`).

When the processor fetches an instruction, it first looks for the requested word in the instruction cache. If there is a cache miss, the processor requests one cache-line from memory. The processor `Requested Address` is then used to generate an address to `ATT`. The address of `ATT` is computed from `Requested Address` by masking out 6 bits: 2 bits that are used for byte-offset, 3 bits to address the word in the cache-line (assuming 8 word cache-lines), and one extra bit to reduce the number of entries (size) of `ATT`. As a consequence of this extra bit, `ATT` can only address one every two consecutive compressed cache-lines in memory, increasing the response time of the engine to a memory request. Therefore, there is a trade-off, that can be explored by the designer, between the size of `ATT` and the latency of the decompression engine. After the mask operation is finished, `Significant` bits are used to point to `ATT`. Each `ATT` entry has two fields: `ADDR` and `OFFSET`. The `ADDR` field is the address of the memory word (`Word`) that contains the compressed target requested by the processor. Notice that the `IBC` algorithm can compress more than one instruction into a single memory word, and these can start at any one of its 32-bit positions. Field `OFFSET` (5 bits) is used by the `SHIFT` module to determine the position of the requested compressed instruction in `Word`.

The latency of this address translation approach is mainly a result of the time required to fetch, from memory, the sequence of words up to the requested instruction, plus the time to decode it. After that, the decompression engine fetches, using an internal counter, the remaining words that are required by the processor to complete the compressed cache-line. The size of the address translation engine is basically the size of `ATT`. There are $k = Program\ Size/(4+8+2)$ lines in `ATT`, each line containing $w = log_2(Compressed\ Size)$ `INSTR` bits and 5 `OFFSET` bits.

VIII. Conclusions and Future Work

This paper proposes a set of three code compression algorithms for programs running on RISC architectures. All algorithms are based on expression trees. Symbols used for compression are whole expression trees, parts of expression trees (patterns) or single instructions. A preliminary design of the IBC decompression engine is underway. The design is based on a synthesizable VHDL model of the decompression engine using Leonardo Spectrum Tools from MGC/Exemplar.

IX. Acknowledgments

This research was supported in part by a grant from CNPq under contract 300156/97-9, a FAPESP research fellowship (98/13728-0), and a grant from the CNPq/NSF 1998 Collaborative Research Project.

References

[1] A. van Someren and A. Atack, *The ARM RISC Chip: A Programmer's Guide*, Addison-Wesley, 1994.
[2] K. D. Kissell, "MIPS16: High–density MIPS for the embedded market,," in *Real Time System'97*, 1997.
[3] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge, "Improving code density using compression techniques," in *Proceedings of MICRO–30: The 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 194–203.
[4] Timothy C. Bell, Jhon G. Cleary, and Ian H. Witten, *Text Compression*, Advanced Reference Series. Prentice Hall, New Jersey, 1990.
[5] W. T. Wilner, "Burroughs B1700 memory utilization," in *Fall Joint Computer Conference*, 1972, pp. 579–586.
[6] Andrew Wolfe and Alex Channin, "Executing compressed programs on an embedded RISC architecture," in *Proceedings of MICRO–25: The 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 81–91.
[7] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *Proc. of 35th ACM Design Automation Conference*, 1998.
[8] IBM Corporarion, *CodePack PowerPC Code Compression Utility User's Manual*, 1998, Version 3.0.
[9] S. Liao, S. Devadas, and K. Keutzer, "A text-compression-based method for code size minimization in embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 1, pp. 12–38, 1998.
[10] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley, Boston, 1988.
[11] Guido Araujo, Paulo Centoducatte, Mario Cortes, and Ricardo Pannain, "Code compression based on operand factorization," in *Proceedings of MICRO–31: The 31th Annual International Symposium on Microarchitecture*, December 1998, pp. 194–201.
[12] Martin Beneš, Steven M. Nowick, and Andrew Wolfe, "A fast asynchronous huffman decoder for decompressed–code embedded processors," in *Async98*. 1998, ACM.