Expression Tree Based Algorithms for Code Compression on Embedded RISC Architectures

Guido Araujo *Paulo Centoducatte*
Rodolfo Azevedo *Ricardo Pannain*

Relatório Técnico IC–00-01

Janeiro de 2000

Expression Tree Based Algorithms for Code Compression on Embedded RISC Architectures

Guido Araujo Paulo Centoducatte Rodolfo Azevedo 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 a number of 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 CodePack PowerPC). 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.

1 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 Thumb [1] and MIPS16 [2] 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. It also means 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, the engine must allow for random codeword decompression. These are the two major features that distinguish *code compression* from other data compression problems.

This paper is divided as follows. Section 2 discusses prior work on the problem of code compression. The experimental framework used in this work is described in Section 3.

All algorithms proposed here use expression trees or parts of expression trees to perform compression. Section 4 explains how expression trees are determined and why we believe they lead to improved compression. Compression algorithms and their corresponding experiments are discussed in Section 5 (Tree Based Compression), Section 6 (Pattern Based Compression), and Section 7 (Instruction Based Compression). The goal of the experiments is to measure the final compression ratio[1], including the decompression engine size overhead. Section 8 summarizes the work and Section 9 proposes new directions and possible extensions.

2 Related Work

This paper deals with the problem of finding code compression techniques that allow efficient implementations of real-time decompression engines. One might be tempted to believe that this problem is a natural extension of the data compression problem, for which there is an extensive literature[4]. Although data compression algorithms form the basis of code compression, they cannot be directly applied. For example, almost all practical dictionary based compression tools of today are based on the work of Lempel and Ziv (LZ)[5] and its variations[6][7][8][9]. In LZ compression, the dictionary is encoded together with the compressed string. Pointers to previously parsed substrings are used to encode the current substring. Decompression is then performed by substituting a pointer by the substring it points to. For the case of real-time decompression, this is a major drawback though. Instructions that are target of branch instructions are reachable from more than one instruction path, and the path is only determined at execution time. Therefore, there is no way to know, at compression time, which instruction path should be compressed. For the rest of this section, we discuss code compression techniques that have been proposed to solve the real-time decompression problem.

The first studies on code compression date back to the 70's, when memory was scarce and instruction sets were designed to minimize memory utilization. In 1972, the designers of the Borroughs B1700 [10] developed an approach, based on instruction utilization, to determine the size of instruction fields. Short (long) instruction fields were assigned to very (un)frequent instructions using Huffman encoding [11]. On a variation of this approach, programs dynamically collected instruction field utilization, such that field sizes could be assigned at execution time.

The first approach for code compression in a RISC architecture was originally proposed by Wolfe and Channin [12]. The processor described in there is called *Code Compression RISC Processor* (CCRP). In the CCRP, code is compressed one cache-line at a time. Compressed cache lines are fetched from main-memory, uncompressed and put into the instruction cache. Instructions in the cache are exactly as in the original uncompressed program. This requires a new design for the instruction cache refill engine, but no modification in the core processor. The main advantage of compressing cache lines is that the latency of the decompression engine is amortized across many cache hits. In CCRP, program target addresses have different values if the line is in main-memory or in cache. The

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

CCRP uses a main-memory based *Line Address Table* (LAT) to map (uncompressed code) addresses in the cache to (compressed code) addresses in main-memory. A *Cache Line Address Lookaside Buffer* (CLB) is used to store sets of recently fetched LAT entries. The compression algorithm for the CCRP is based on encoding byte long symbols using Huffman codewords[11], and results in 73% compression ratio for the MIPS R2000 instruction set [12] and [13]. This compression ratio does not take into consideration the size of the decompression engine.

Lefurgy et al. [3] proposed a code compression technique based on dictionary encoding. In [3] object code is parsed and common sequences of instructions are replaced by a single codeword. Only frequent sequences are compressed. Escape bits are used to distinguish between a codeword and an uncompressed instruction. The instructions corresponding to each codeword are stored into a dictionary in the decompression engine. Codeword bits are used to index the dictionary entries. The decompression engine expands codewords into their original instruction sequences in the dictionary. Since the compressed program is composed of codewords and uncompressed instructions, branch targets are recomputed so as to reflect their new location in the program. The target address bits is divided into two parts: the address of the compressed word and an offset from the beginning of the compressed word. The target address is computed by adding these two, a technique that requires modifications in the control unit of the processor core. Lefurgy et al. studied two compression techniques. The first approach is based on fixed-length codewords. Better compression ratios were achieved by a second approach that uses nibble aligned variable length encoding. In this case, average compression ratios of 61%, 66%, and 74% have been reported for the PowerPC, ARM and i386 processors respectively [3].

Wolf and Lekatsas [14][15] studied two different methods for code compression. The best compression ratio is produced by the SADC method. In SADC, symbols are associated to instruction opcode and operand fields. During compression, instruction sequences are selected and a stream of bits is derived for each sequence of instruction fields. Each stream is then encoded using Huffman codewords. The average compression ratio achieved by this method on a MIPS architecture is 51%. It is not clear from [14] if this number takes into consideration an estimate of the size of the decompression engine.

The CodePack PowerPC processor [16] is an architecture designed to execute compressed code. The compression approach found at CodePack is similar to the one proposed in this paper. They differ on how symbols are selected and encoded though. Huffman codewords are used to encode escape bits, while symbols are selected from sequences of instruction bits in a cache-line. The CodePack approach results in an average 60%-65% compression ratio, not including the size of the decompression engine core.

Liao et al. [17] proposed a compression technique based on dictionaries. The main idea in [17] is the substitution of common instruction sequences by sub-routine calls. A hardware mechanism is proposed to minimize the cost of the sub-routine return instruction. The average compression ratio reported for the TMS320C25 processor was 82%.

3 Experimental Framework

The algorithms we propose have been tested using programs from the SPEC CINT95 benchmark running on the MIPS R4000 processor. The R4000 is a classical RISC architecture, that has most of the features of a modern RISC processor. It is also one of the most used RISC architectures in the embedded systems arena. Benchmark programs were cross-compiled for the R4000 using gcc version 2.8.1 on a Sun Enterprise E450 machine. Object code was generated using compiler options -O2 and -Os, for MIPS instruction sets mips1 and mips2. Option -O2 generates code target to performance and includes all major compiler optimizations. Flag -Os selects from the optimizations available in -O2, only those which do not increase program size. The resulting code size is shown in Table 1. In general, optimizing for performance leads to code that is approximately the same size of those resulting from size optimization. In Table 1, the -mips2 option produces smaller programs than when -mips1 is used. In mips1 architectures (e.g. R2000), delayed branches are handled by the compiler/assembler (resulting in many `nop` instructions). This is not the case of interlocked pipelined architectures like the mips2 (e.g. R4000) [18]. For this work, the benchmark programs were compiled using options -mips2 -Os.

Program	-mips1 -O2	-mips1 -Os	-mips2 -O2	-mips2 -Os
compress	2304	2304	2164	2152
gcc	409204	407636	364524	363560
go	79776	80284	73908	72516
ijpeg	52816	52336	48548	47988
li	20832	20652	18616	18448
perl	80308	79676	70228	69536
vortex	167212	167384	151476	151348

Table 1: Compiler parameters and number of instructions generated.

4 Expression Trees

In compression, *symbol* is the basic unit used to form the *text* to be compressed, and an *alphabet* is the set of all symbols. This work studies three code compression techniques for RISC architectures. The basic idea of all algorithms is the restriction we made that an alphabet must contain only symbols that are expression trees or parts of expression trees. In other words, a sequence of instructions that is not entirely contained in any program tree cannot be considered a symbol. What distinguishes one algorithm from another is the way trees are decomposed into alphabet symbols.

In the first algorithm (Section 5), symbols are whole trees and the alphabet is formed by all distinct trees in the program. In the second algorithm (Section 6), trees are decomposed into smaller distinct parts (i.e. *patterns*), which are then encoded. Finally, in the third algorithm (Section 7), the alphabet is the set of all distinct instructions from all trees in the program. We use expression trees as the basis for compression because compilers tend to generate similar expression trees during the translation of source program statements. This is explained by: (a) the reduced number of instructions in a RISC instruction set; (b) the small size of the majority of the expression trees, and therefore, the small number of possible ways in which instructions can be combined; (c) the deterministic way in which compilers generate code for *abstract syntax tree* constructs, like `if-then-else` and `for` statements .

Expression trees are constructed as in [19]. An instruction is the root of an expression tree [19] if one of the following is true: (a) the instruction stores into memory; (b) the destination operand of the instruction is the source of more than one instruction inside the basic block; (c) the destination operand of the instruction is the source of at least one instruction outside the basic block; (d) the instruction is the first instruction in the basic block; (e) the instruction is a branch. Expression trees do not cross basic block boundaries. Examples of expression trees are listed in Figure 1.

```
addiu  $29, $29, 256
sw     $28, 16($29)

        (a)
```

```
addiu  $2, $2, 60
lw     $4, 0($2)
slti   $4, $4, 17
bne    $4, $0, 16
       (b)
```

Figure 1: Typical expression trees.

5 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 satisfying the expression tree definition in Section 4. The set of distinct trees was determined and the results were listed in Table 2. From Table 2, the number of distinct trees in a program is much smaller than

Program	Total Trees	Distinct Trees (%)
compress	1844	832 (45.1)
gcc	291758	51186 (17.5)
go	62423	12460 (20.0)
ijpeg	40621	11264 (27.7)
li	15509	3072 (19.8)
perl	52276	12793 (24.5)
vortex	130336	17463 (13.4)

Table 2: Number of distinct trees in a program. Numbers in parentheses are percentage with respect to the total number of trees.

the total number of trees. On average, distinct expression trees correspond to only 24% of all trees in a program.

5.1 The TBC Algorithm

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 computed. The result is shown in the graph of Figure 2. In the horizontal axis of the graph, trees are ordered in decreasing frequency. Notice from Figure 2, that the frequency distribution of distinct trees in all programs is very non-uniform. Actually, trees have exponential frequency distributions. On average, 80% of all program trees are covered by only 20% of the most frequent ones.

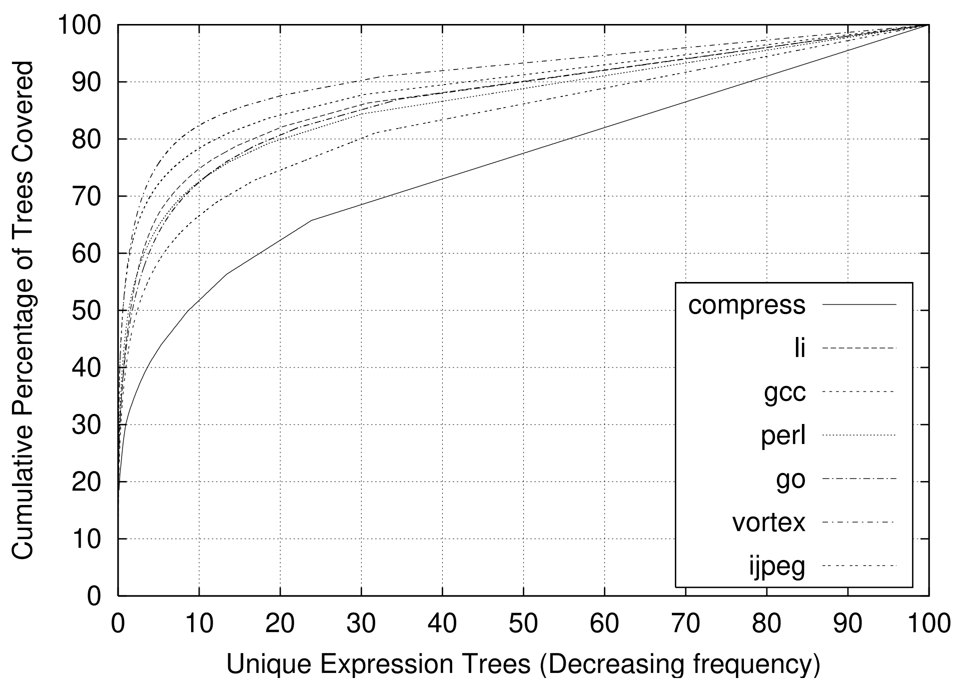

Figure 2: Percentage of program trees covered by distinct trees.

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 [20] we studied four encoding methods based on variations of Huffman. The experimental results reveal an average 37% compression ratio for the same set of programs studied here. Unfortunately, designing fast Huffman decoders is complicated, and it usually results in decoders that are more expensive than if fixed-length codewords had been used [21][22]. 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 smaller compression ratio is then selected as the best partition for the tree set. One can think of this approach as a form of discrete Huffman encoding. From this perspective, the goal of the compression algorithm is to perform a piecewise discretization of the frequency distribution shown in Figure 2, so as to minimize the final compression ratio.

Fixed-length codewords of size $\lceil log_2 n_k \rceil$ are then assigned to the 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 final codeword encoding of a tree is shown in Figure 3. The compression algorithm substitutes each expression tree in the program by its corresponding codeword.

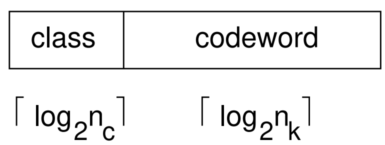

Figure 3: Tree encoding.

Consider, for example, program *li* and a partition of the tree set into four classes. Table 3 shows all possible combinations of codeword sizes using a four classes partition (I-IV). The best compression ratio[2] (23.4%), highlighted in Table 3, 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 4) into four intervals, each interval corresponding to a class. From the discretization perspective, the new value of the cumulative distribution in each interval (i.e. class) is constant and is computed using the average frequency of the trees in that interval.

Once the best compression ratio for a given partition is determined, we repeat the algorithm for other partitions. Figure 5 shows the resulting compression ratio 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 new 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

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

Codeword size				Compr.
I	II	III	IV	Ratio
1	1	1	12	30.3
1	1	2	12	29.2
.	.	.	.	.
1	**5**	**8**	**12**	**23.4**
1	5	9	12	23.5
.	.	.	.	.
9	9	8	12	31.2
9	9	9	12	30.5

Table 3: All possible codeword size combinations for *li* using four classes.

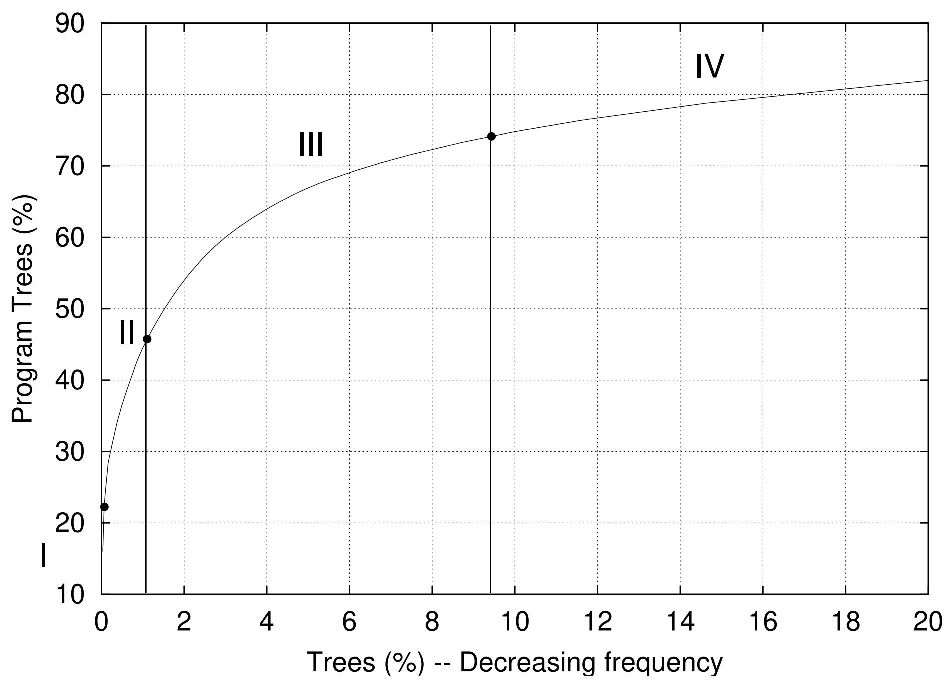

Figure 4: Discretization of the frequency distribution for program `li` after class partitioning. Class I (IV) has $< 1\%$ ($> 9\%$) of all distinct trees.

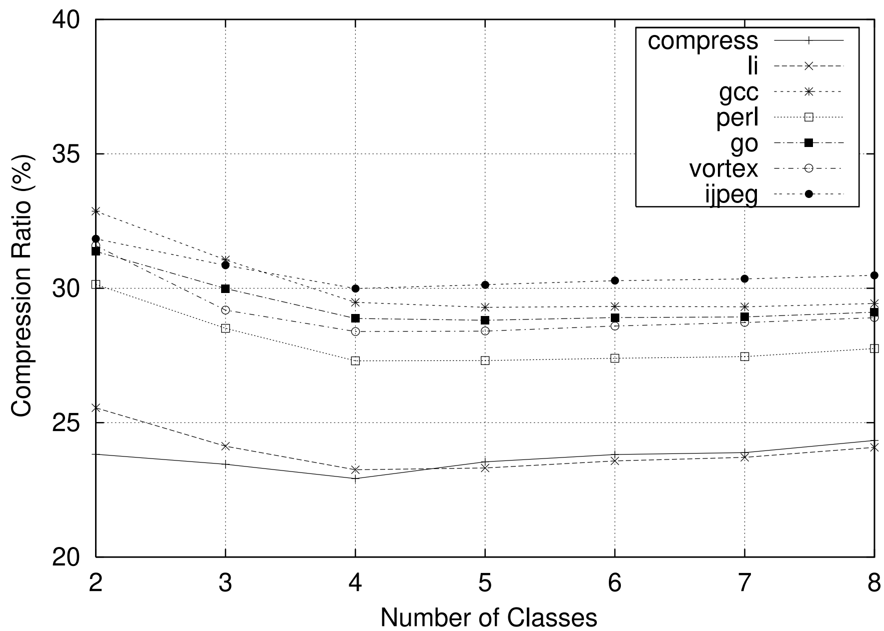

Figure 5: Compression ratio for different partitions.

the compression ratio between classes five and four is only 0.09%. The best compression ratio for each partition is then determined (minima in Figure 5). Table 4 shows, for each program, the best compression ratio using four classes. The average compression ratio for all programs studied was 27.2%.

Program Name	Codeword size				Compr. Ratio
	I	II	III	IV	
compress	1	5	8	10	22.9
gcc	2	8	12	16	29.4
go	3	8	11	14	28.8
ijpeg	3	8	11	14	29.9
li	2	6	9	12	23.2
perl	2	7	10	14	27.3
vortex	1	6	10	14	28.4

Table 4: Class partition that results in the best compression ratio for four classes.

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 large compression ratios can only be achieved if we allow this to happen. The reason, also noticed in [3], is that many common trees are originated from a single instruction word (see Section 7 for details). Therefore, constraining codewords to a single word considerably limits the compression

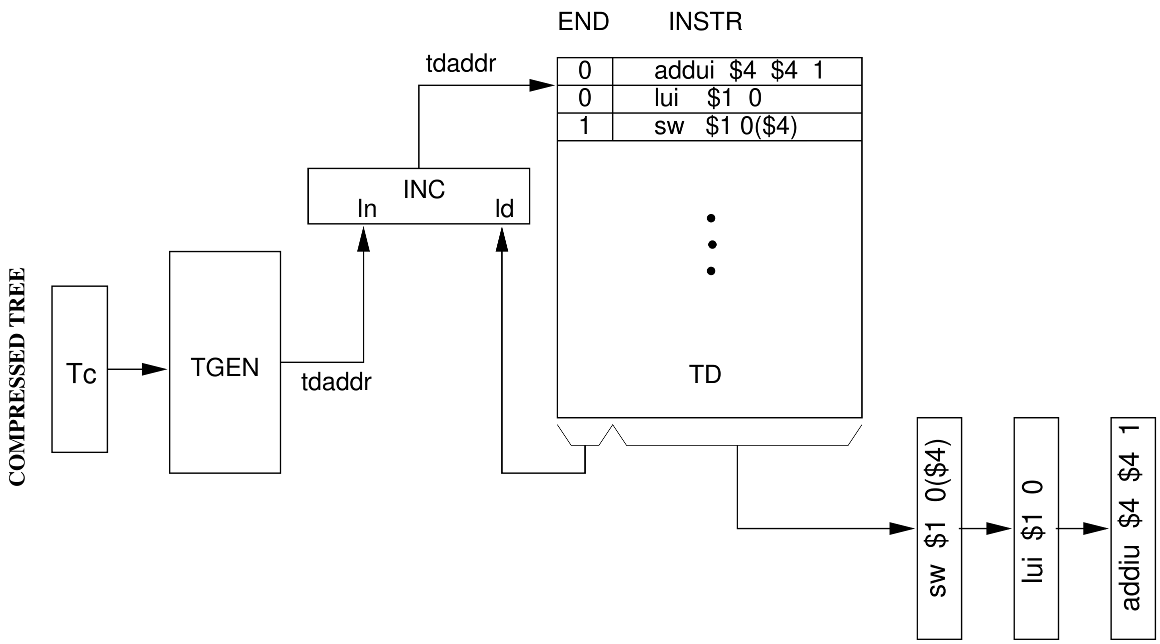

Figure 6: Decompression engine for Tree Based Compression (TBC).

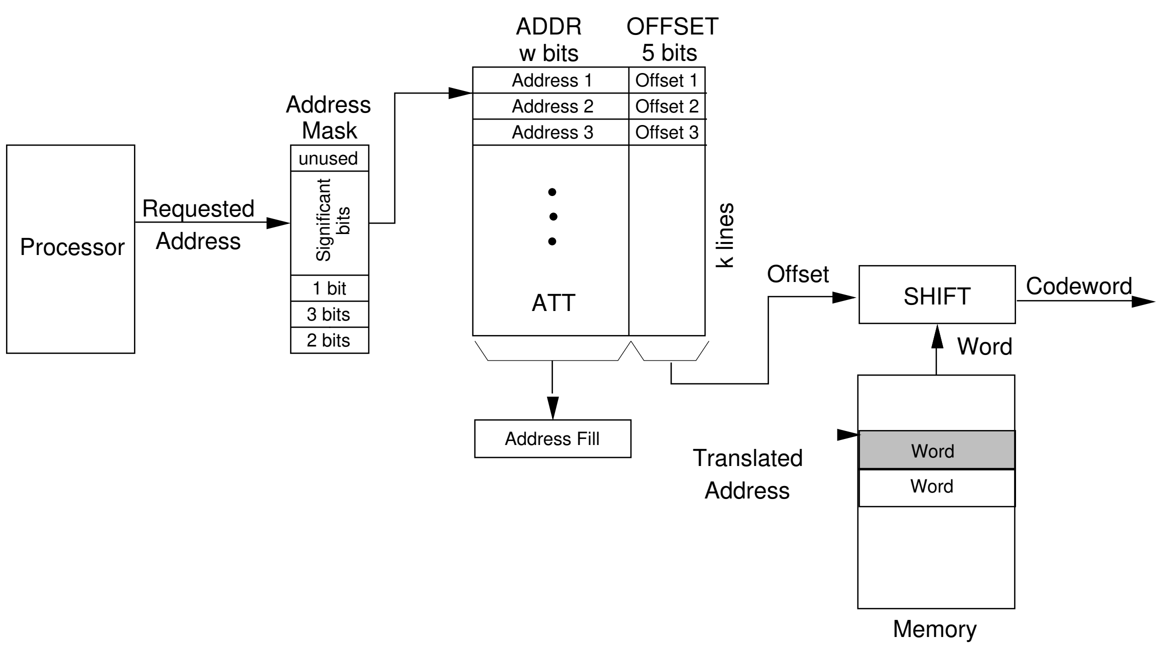

Figure 7: Address Translation Table (ATT).

ratio. 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.

5.2 The TBC Decompression Engine

This section proposes a decoding engine for the TBC compression algorithm (Figure 6). The decompression engine works in two phases. First, tree codewords `Tc` are extracted from a memory word. Second, `Tc` is decoded by logic `TGEN` and converted to address `tdaddr`. Address `tdaddr` points to the entry in the Tree Dictionary (`TD`) that stores the first instruction of the decompressed tree. Each `TD` entry is composed of two fields: `INSTR` and `END`. Field `INSTR` is 32 bits wide and contains one instruction of the decompressed expression tree. Bit-field `END` is used to check for the last instruction in the tree being decompressed. If `END` = 1 (0) the current entry is (not) the last instruction of the current tree. `END` is used as a load input to the Incrementer (`INC`). If the current instruction is not the last instruction of the current tree, `INC` increments and the output of `INC` points to the next tree instruction, otherwise it loads a new `tdaddr` to start decoding the next compressed tree. Address translation for branch and jump instructions is performed by the Address Translation Table (`ATT`) (Figure 7) discussed in Section 5.3 below. A fair assessment of the

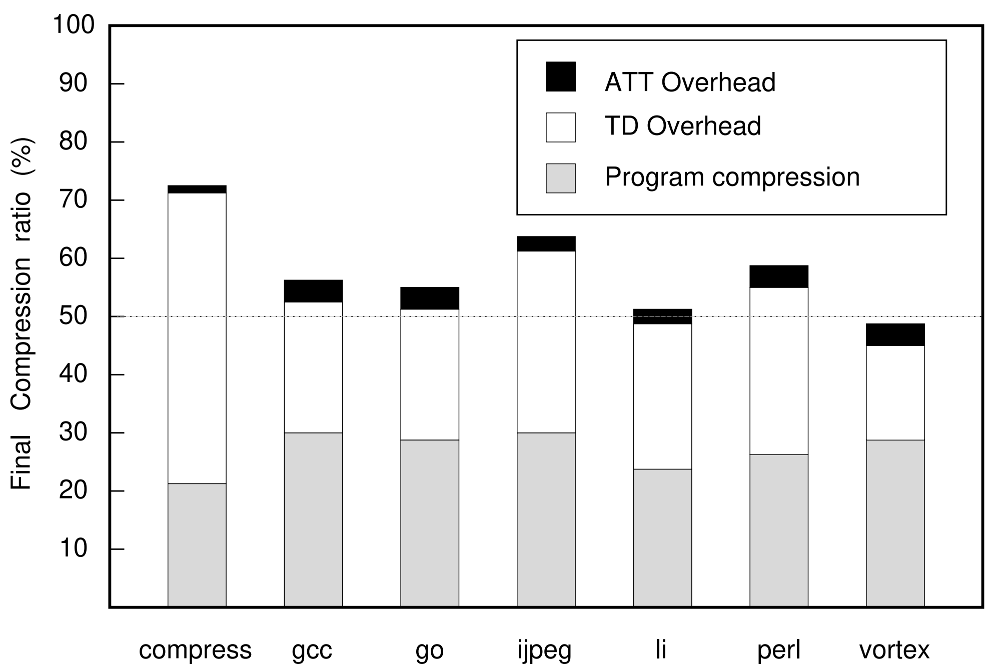

Figure 8: Final compression ratio for TBC.

compression efficiency of the TBC algorithm needs to take into consideration the silicon area of the decompression engine. To determine the size of the decompression engine we assume that the area of the extraction logic and the rest of the decompression engine is much smaller than the size of its tables (`TD` and `ATT`). Figure 8 shows the final compression ratio of the TBC algorithm when the decompression engine modules overhead is considered. The average size of the decompression engine, with respect to the uncompressed program is 33.5%. The final average compression ratio for the TBC algorithm is 60.7%.

5.3 Address Translation Table

In our architecture model, the processor executes uncompressed instructions that generate uncompressed address requests, while memory stores compressed instructions (i.e. trees). During the execution of branch/jump instructions 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 the address translation module shown in Figure 7, where the mapping is performed using an `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 expression tree requested by the processor. By definition of expression tree, the target of any branch/jump instruction is the root of a tree (Section 4). Notice that the `TBC` algorithm can compress more than one expression tree 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 tree 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 tree, 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. Speed can be improved if the decompressor engine runs faster than memory, allowing the engine to feed more instructions to the processor than the memory system, since codewords are smaller than instructions and there is more than one codeword in each memory word. 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 $w = log_2(Compressed\ Size)$ `ADDR` bits and 5 `OFFSET` bits.

6 Pattern-Based Compression (PBC)

In the previous section, expression trees were divided into subsets and the trees in each set encoded using fixed-length codewords, the size of which is dependent on the average frequency of the trees in the set. In this section, we discuss a compression algorithm for which the alphabet is composed of parts of expression trees.

6.1 The PBC Algorithm

The key idea of this approach 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*. Operand factorization is not a new concept though. It has been proposed in [23] as an encoding technique for intermediate representation in compilers. Variations of operand factorization have been used in [24][25]. Consider, for example, the expression tree

```
addiu $4, $4, 1          addiu *, *, *
lui $1, 0                lui *, *
sw $1, 0($4)             sw *, *(*)
    (a)                      (b)

      [$4,$4,1,$1,0,$1,0,$4]
                (c)
```

Figure 9: (a) Expression tree; (b) Tree-pattern; (c) Operand-pattern.

of Figure 9(a). Figure 9(b) shows the tree-pattern resulting after operand factorization is applied to it. *Stars* (wild-cards) are used in place of the original operands. An operand-pattern is formed by traversing the instruction sequences in the expression tree, listing the operands when they are encountered. Figure 9(c) shows the operand-pattern determined after the expression tree in Figure 9(a) is factored.

Table 5 lists the number of expression trees and patterns for our program set. In Table 5, *gcc* has 291758 different expression trees, that can be represented by only 921 (45469) tree (operand) patterns. In other words, the tree (operand) patterns from only 0.3% (15.6%) of all trees in *gcc* are enough to represent the remaining trees. Interesting enough, small programs seem to be much less redundant than large programs. In *compress* (the smallest program studied), tree-patterns correspond to 5.8% of all possible trees in the program, while operand-patterns are 41.6% of all operand sequences.

At this point, it is interesting to determine what makes expression trees different. Two expression trees are distinct if they have at least one different instruction. Two instructions are different if they have different tree and/or operand patterns. Column (III) of Table 5 shows the number of distinct trees in each program. Notice that, for all programs, there is a strong correlation between the number of distinct trees and operand-patterns (V). This correlation can be measured by the difference (III) - (IV) shown in Column (VI), that results

Program Name (I)	# Trees (II)	# Distinct Trees (III)	Tree-Patterns (IV)	Operand-Patterns (V)	(III) - (IV) (VI)
compress	1844	832	107 (5.8)	767 (41.6)	8.5
gcc	291758	51186	921 (0.3)	45469 (15.6)	12.6
go	62423	12460	256 (0.4)	11373 (18.2)	9.6
ijpeg	40621	11264	348 (0.9)	9907 (24.4)	13.7
li	15509	3072	169 (1.1)	2840 (18.3)	8.2
perl	57276	12793	547 (1.0)	11579 (20.2)	10.5
vortex	130336	17493	324 (0.2)	15592 (12.0)	12.2
Average	85681	15585	382 (1.4)	13932 (17.4)	10.8

Table 5: Number of tree and operand patterns in a program. Numbers in parentheses are percentage with respect to the total number of expression trees.

in an average 10.8%. Hence, for the majority of the programs, given an operand pattern there is usually a single tree-pattern associated to it. In other words, operand-patterns are the main cause for the large diversity of program expression trees. This is not a surprise, given the large number of ways that registers and immediates can be combined, when compared with the number of combinations of instruction opcodes in a RISC architecture. The correlation between distinct trees and operand-patterns is not one-to-one though. In this case, there is still some opportunities for compression, by dividing a tree into its patterns.

In order to determine the form patterns contribute to a program we computed the frequency contribution of all patterns. The individual frequencies of each unique tree-pattern was determined. Tree-patterns were then ordered in a decreasing order of frequency, and the cumulative percentage of the expression trees covered by these patterns was computed. The results are shown in Figure 10(a). The frequency of each tree-pattern is the derivative of the graph in Figure 10(a). Based on that, we reach the conclusion that the frequency of a tree-pattern decreases almost exponentially as the pattern becomes less and less frequent. On average 20% of the tree-patterns correspond to almost all trees in a program. This rule works for all programs in Figure 10(a) but *compress*. The distribution of expression trees in *compress* is smoother. A similar graph was also derived for operand-patterns. Figure 10(b) shows the cumulative number of trees in a program that are covered by distinct operand-patterns. On average, 20% of the operand-patterns account for about 80% of all operand sequences in a program. As before, *compress* numbers differ from the other programs.

Operand factorization recognizes the fact that any encoding technique that intermixes opcode and operand bits during compression misses the opportunity to capture the high correlation exhibited by tree and operand patterns. For example, an algorithm that performs sequential compression, like LZ [5], will not be able to detect the simple tree-pattern `[lw *,*,* :  add *,*,*]`. Any non-sequential algorithm which considers a program as a set of bit strings will also miss that. Consider for example, the tree-pattern `[lw *,*,*]` and a processor that encodes the opcode and the destination register (in this order) using 6 bits each. If a byte is chosen as the encoding symbol, the first byte of instructions `[lw $2,*,*]`

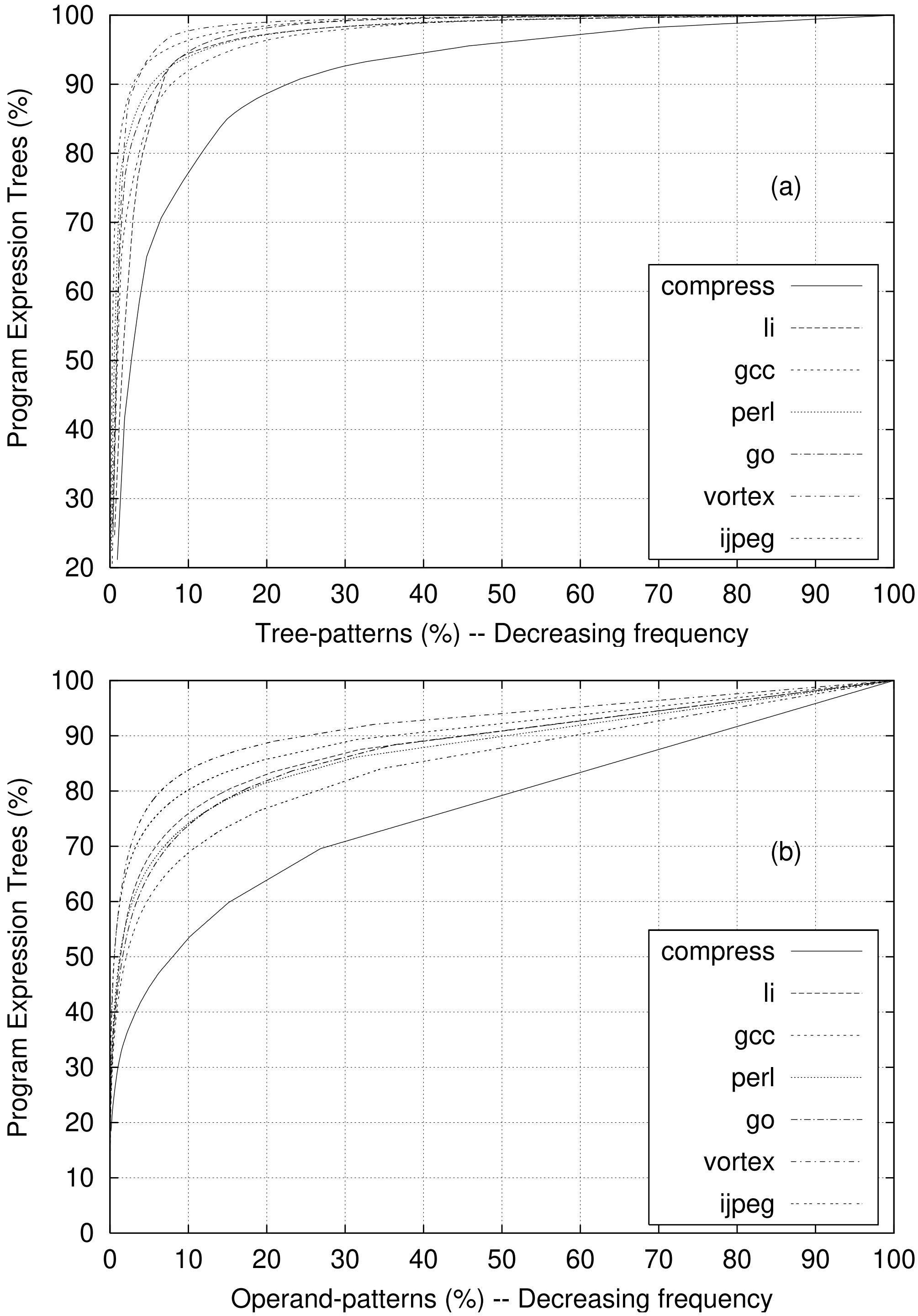

Figure 10: Cumulative percentage of expression trees covered by: (a) tree-patterns; (b) operand-patterns.

and `[lw $15,*,*]` are encoded as two different codewords, even if pattern `[lw *,*,*]` accounts for a considerable share of the program bits. Moreover, operand factorization can identify operand-patterns that are shared by two different instructions. For example, in *gcc* operand-pattern `[$2, $0, $4]` is used by expression trees `[subu $2, $0, $4]` and `[nor $2, $0, $4]`.

Tree and operand patterns are encoded separately using the same algorithm discussed in Section 5.1. Expression trees are encoded as codeword pairs $[Tp, Op]$, where Tp (Op) is the codeword for a tree (operand) pattern. Pairs $[Tp, Op]$ are then appended sequentially to form a list of codeword pairs that results in the compressed program. As before, Figure 11(a-b) shows how the compression ratio varies according to the number of classes used to divide each pattern set. Notice that for tree-patterns the best encoding is achieved for 3 classes. This certainly has to do with the sharp exponential distribution for three-patterns in Figure 10(a). Few tree-patterns cover the majority of the trees in a program and these are assigned to a single class. The rest of the trees have similar (small) contributions, and therefore there is no point in assigning them to more than a couple of classes, since this only increases the overhead due to the extra prefix bits required to encode the additional classes. On the other hand, the compression ratio difference when 4 classes are used instead of 3 is only 0.49%. The frequency distribution for operand-patterns (Figure 10(b)) is smoother than for tree-patterns, resulting in a better compression if the set of operand-patterns is divided into more classes. In general this number is four, but for some programs (e.g. *gcc*) the compression using five classes is better. Here again, the compression ratio difference between both partitions is only 0.1%.

Similarly for the case of TBC (Section 5), patterns that do not fit into a single word are spilled to the next word. This is an additional problem for the decompression engine, given that it now should be able to realign two codewords, one for each pattern. The compression

Program Name	Tp Compr Ratio	Op Compr. Ratio	Composed Ratio
compress	22.8	13.2	35.9
gcc	29.1	13.3	42.4
go	28.6	12.5	41.1
ijpeg	29.5	14.1	43.6
li	22.9	12.1	35.0
perl	27.1	13.3	40.4
vortex	27.5	12.7	40.2

Table 6: Composed compression ratio when Tree-pattern (Tp) and Operand-pattern (Op) codewords are combined.

ratio for tree (operand) patterns is on average 13.0% (26.8%), and is achieved when tree (operand) patterns are divided into 4 classes. Table 6 shows the final compression ratio (average 39.8%), when the codewords for both patterns are combined.

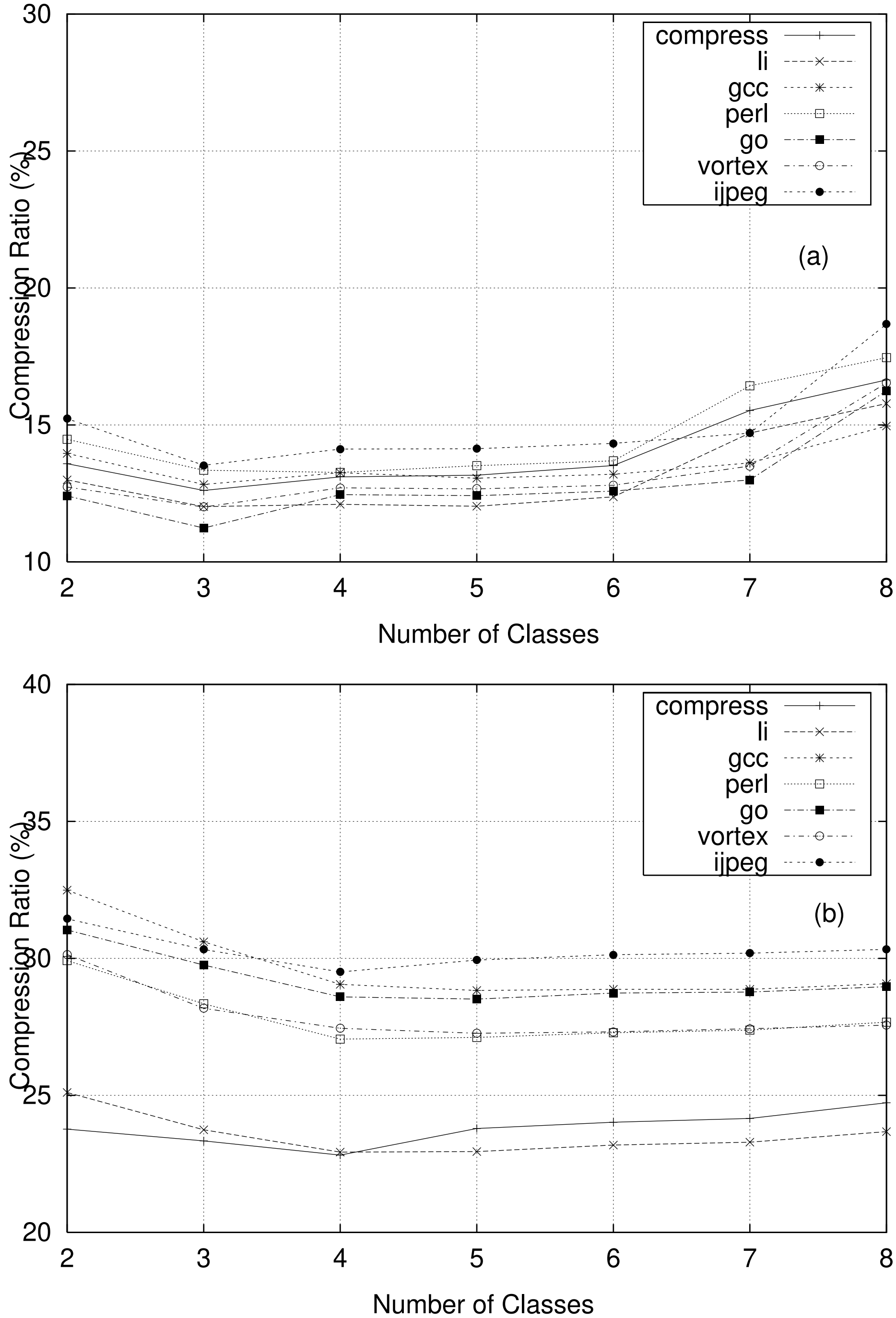

Figure 11: (a) Tree-pattern compression ratio for different partitions; (b) Operand-pattern compression ratio for different partitions.

6.2 The PBC Decompression Engine

This section proposes a decoding engine for the PBC Algorithm (Figure 12). As before the decompression engine works in two phases. First, fields Tp and Op are extracted from the compressed word. Second, Tp is mapped into a sequence of uncompressed instructions, and Op is used to generate registers and immediate bits for them. This information is fed into the Instruction Assembly Buffer (`IAB`) that assembles the decompressed instructions. In the following sections we describe each module of the decompression engine.

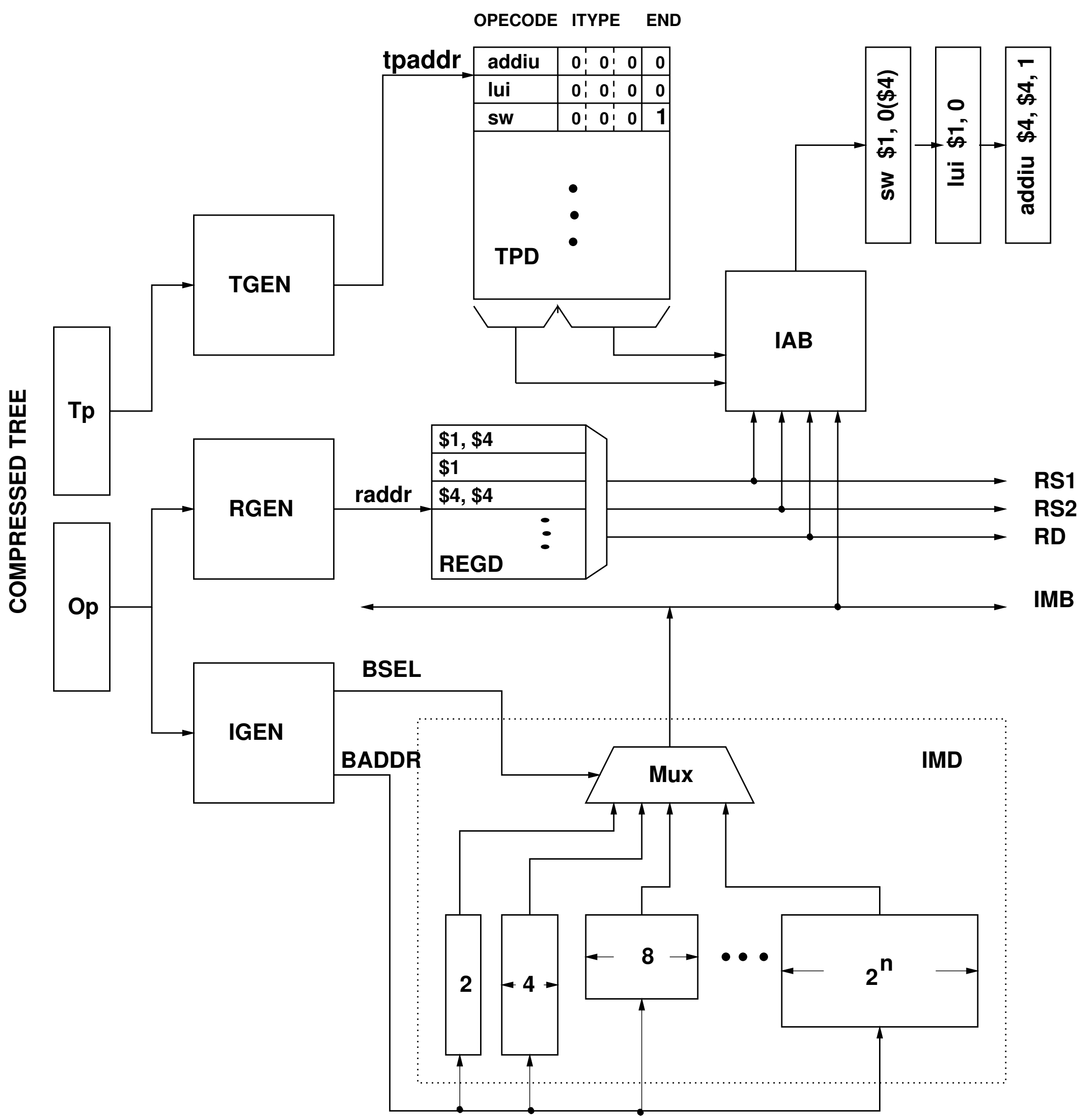

Figure 12: Decompression engine for Pattern Based Compression (PBC).

6.2.1 Tree-pattern Dictionary

The Tree-pattern Dictionary (`TPD`) stores the opcodes encoded by each tree-pattern codeword. Tp is decoded by the Tree-pattern Generator (`TGEN`) into a `TPD` address `tpaddr`. The opcode fields encoded by Tp are then fetched from a sequence of `TPD` entries starting at `tpaddr`. Each `TPD` entry is composed of three fields: `OPCODE`, `ITYPE`, and `END`. Field `OPCODE` carries the opcode bits of an instruction in the tree-pattern. Field `ITYPE` encodes the type (i.e. format) of the instruction. The information stored in `ITYPE` is used by `IAB` to decide how to assemble a decompressed instruction. The `IAB` puts together `OPCODE`, register (`RS1`, `RS2`, `RD`) and immediate (`IMB`) bits to form an instruction. Bit-field `END` is used to check for the last instruction in a tree-pattern. On average, `TPD` area is only 0.9% of the original program size.

6.2.2 Register Dictionary

The Register Dictionary (`REGD`) decodes the Op field of the incoming compressed word, into a sequence of operand registers required by the instructions in the tree-pattern. The output of `REGD` is formed by three (register) buses: `RS1`, `RS2` and `RD`, that generate the bits corresponding to the instruction source and destination registers. An estimate of the size of `REGD` was determined by adding up the size (in bits) of all register fields in the operand-patterns. In this case, the average size of `REGD`, with respect to the uncompressed program, is 3.8%.

6.2.3 Immediate Dictionary

The `IMD` module in Figure 12 stores the immediates used by the program. A single entry in `IMD` is created for each distinct immediate in the program, no matter which instruction uses it, or how many times it shows up. For example, a single constant 4 is stored for instructions `[bgez $5, 4]`, `[lw $6, 4($29)]`, and `[srl $5, $3, 4]`. We use the variation on the size of immediates to minimize the number of bits stored in `IMD`. An evaluation of the size of the immediates reveals that, on average, more than 70% of the immediates in a program can be encoded into less than 16 bits. Immediates are clustered into memory banks according to the number of bits they use. Memory bank address `BADDR` and bank selection `BSEL` are generated by the `IGEN` module from codeword Op. This approach considerably reduces the average number of distinct immediates in a program (26.7%). As a result, the average share of the compression ratio due to IMD is 13%.

The sizes of the immediate (`IMD`), tree-pattern (`TPD`) and register dictionaries (`REGD`) are easy to estimate. As before we assume that the size of the extraction logic and the rest of the decompressor is much smaller than the size of the dictionaries. The size of `ATT` is computed as described in Section 5.3. The final compression ratio, including the overhead of the decompression engine modules is shown in Figure 13. The total area overhead of the decompression engine contributes on average 21.5% to the final compression ratio. The final compression ratio using PBC is 61.3%.

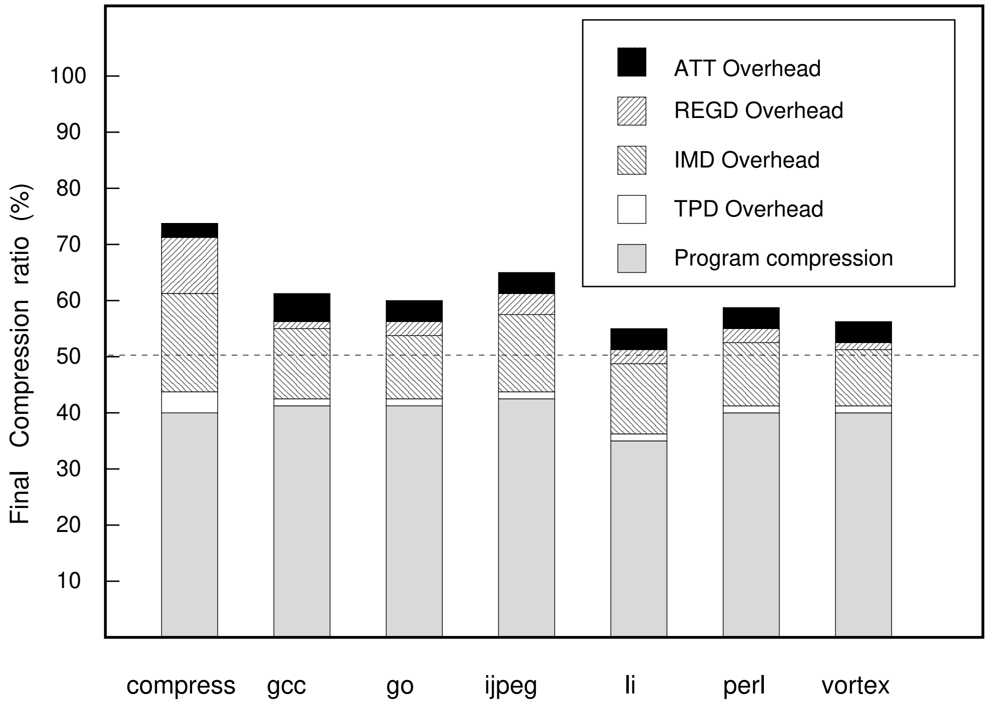

Figure 13: Final compression ratio for the PBC algorithm.

7 Instruction-Based Compression (IBC)

In the two previous sections, compression was performed using whole expression trees (Section 5) and parts of expression trees (Section 6) for symbols. In this section, the alphabet used in compression is formed by instructions. This approach is motivated by the large percentage of expression trees that are composed of single instructions, as shown in Figure 14. The horizontal axis of Figure 14 lists the set of distinct expression trees, ordered by their frequency in the program, and the vertical axis the size of the trees in instructions. The 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). This confirms, at an instruction level, the observation made in [3] about the role played by small bit strings in program code.

The number of distinct instructions in a program is very small, as shown in Table 7. On average, all instructions in a program are replica of only 18.3% of its instructions. This is mainly due to the regularity and orthogonality of the instructions in a RISC architecture. This level of code redundancy cannot be found, for example, in irregular instructions sets, like those found in DSP architectures [26]. It remains to be shown that instructions preserve the exponential frequency property shown by whole trees and patterns. Similarly as before, the graph of Figure 15 shows the cumulative frequency distribution of the instructions in the benchmark programs. On average 20% of the most frequent instructions cover almost 80% of all instruction of a program. It is exactly this exponential behavior, at the instruction

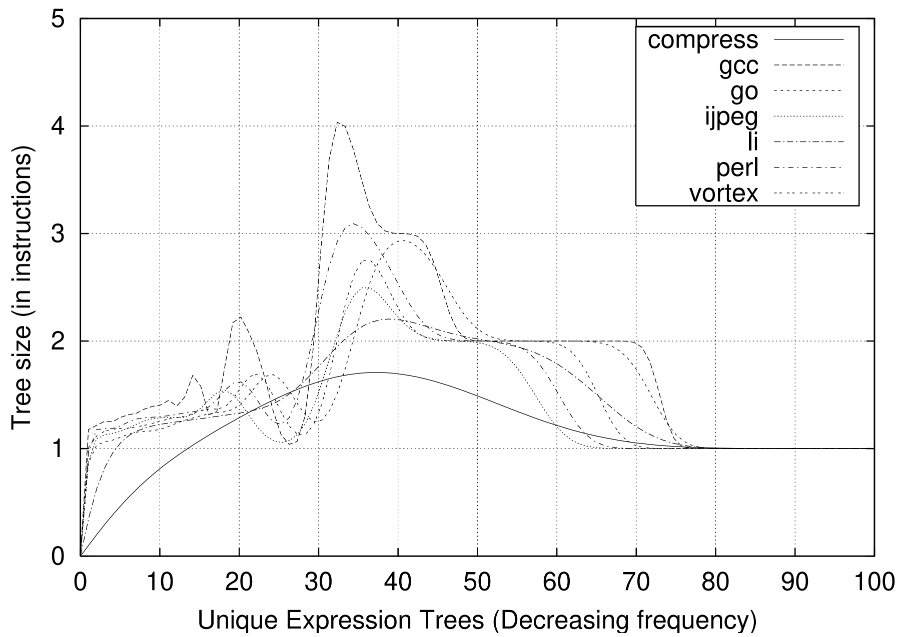

Figure 14: Distribution of the average tree size (Bezier approximation).

Program Name	Program Size	Unique Instructions (%)
compress	2152	846 (39.3)
gcc	363560	38600 (10.6)
go	73908	10267 (13.9)
ijpeg	47988	10536 (22.0)
li	18448	2959 (16.0)
perl	69536	11178 (16.1)
vortex	151348	15200 (10.0)

Table 7: Number of distinct instructions in a program. Numbers in parentheses are percentage with respect to the total number of instructions.

level, that explains the similar behavior for larger symbols like whole trees and patterns. The combination of small instruction sets with a deterministic compiler does result in very redundant code.

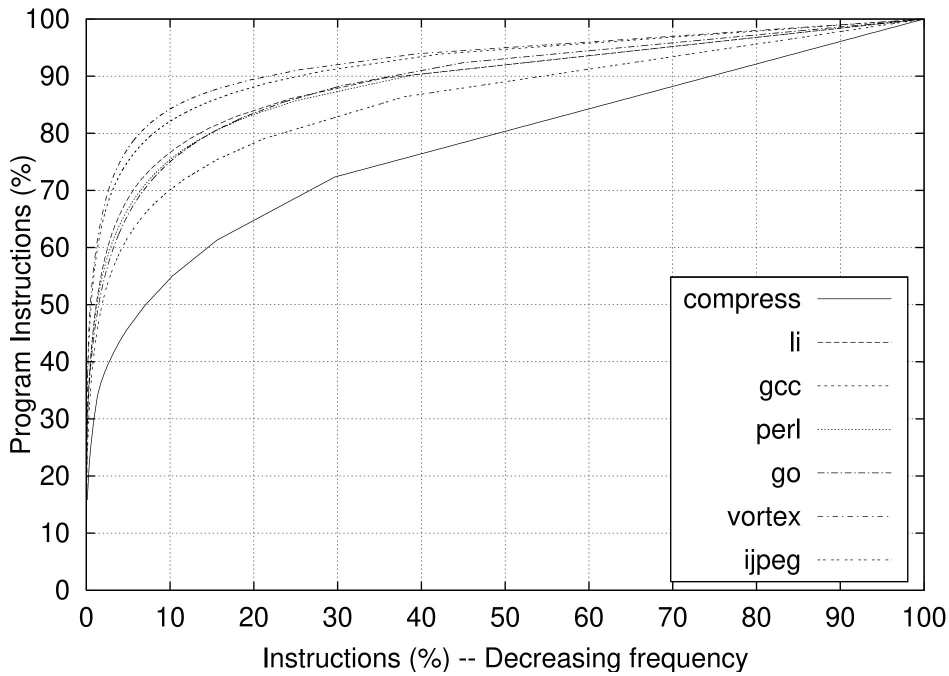

Figure 15: Cumulative instruction frequency distribution.

7.1 The IBC Algorithm

The same compression algorithm employed in Section 5 and Section 6 is used here to encode instructions. The final compression ratio, for all possible instruction partitions from 2-8 classes is shown in Figure 16. The compression ratio is on average 31.5%, and again it is achieved using only four classes. For some programs, the best compression is achieved for five classes, but the difference with respect to four classes is insignificant (0.1%).

7.2 The IBC Decompression Engine

The compression algorithm proposed in this section associates an instruction codeword to a single uncompressed instruction. The decompression engine design (Figure 17) in this case is very simple. 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` (average 3.8%), computed as in Section 5.3. The final compression ratio, if the size of the decompression engine is considered, is shown in

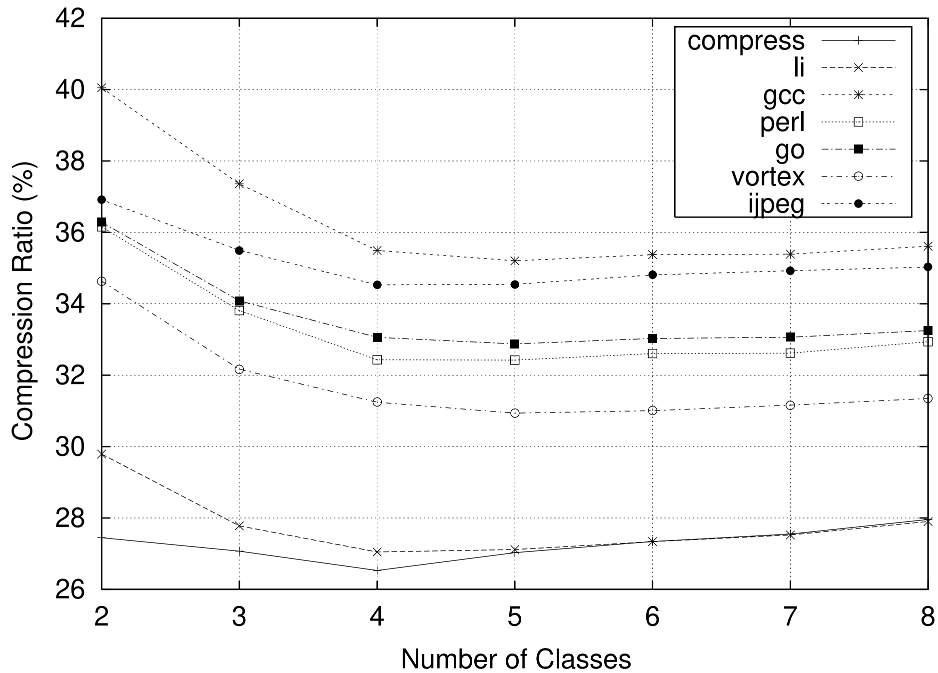

Figure 16: Compression ratio for various partitions.

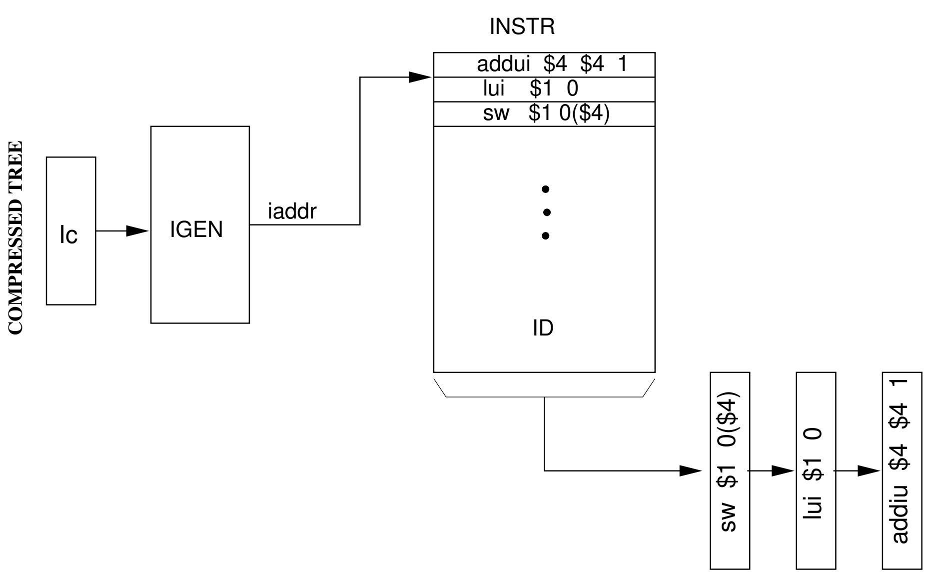

Figure 17: Decompression engine for Instruction Based Compression (IBC).

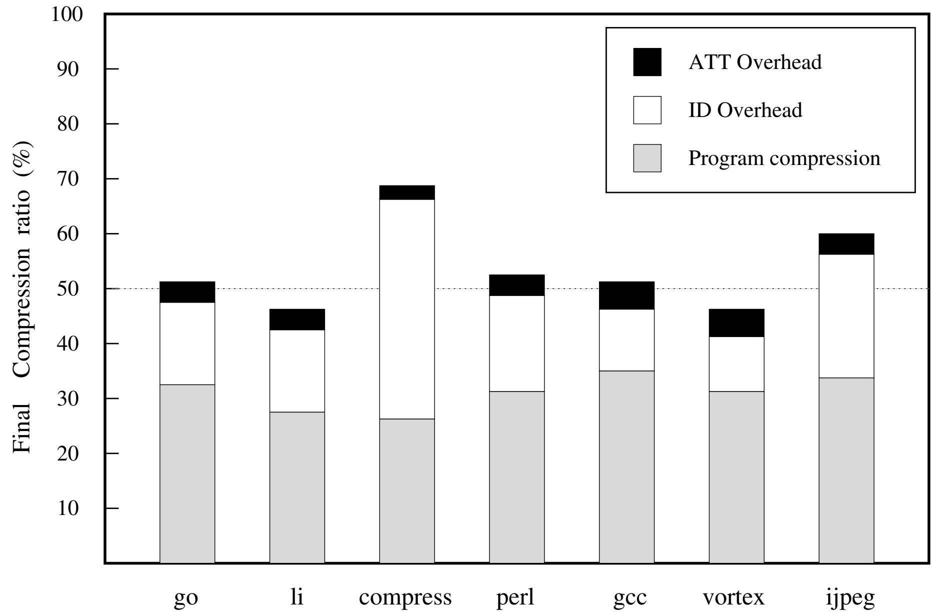

Figure 18: Final compression ratio for IBC.

Figure 18. The average overhead of the decompression engine for all programs is 22.1%, and the final compression ratio for the IBC algorithm is 53.6%.

8 Conclusions

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. We show that, no matter the granularity of the symbol used for compression, symbol distribution is exponential. The central idea, in all three approaches, is the partition of the symbols into classes, based on the average frequency distribution of the symbols in each class. Symbols in each class are assigned codewords of same length. We show that, in general, the best compression ratio is achieved when symbols are divided into four classes. Figure 19 shows the final compression ratios for all programs, using all three algorithms. 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. IBC produces the best ratio. There is not much difference between TBC and IBC ratios if the engine overhead is not included. The reason is that a large number of trees in a program have one single instruction. The difference between TBC and IBC grows when the engine overhead is added. This is basically caused by two reasons. First, each entry in `TD` (Figure 6) has an the extra `END` bit required to mark the end of a tree. Second, entries in a `TD` dictionary (Figure 6) store whole trees, which can have repeated instructions, while entries in `ID` are single instructions. This result in `TD` dictionaries that are larger than `IBC` dictionaries. The reason PBC is outperformed by IBC/TBC is much more subtle. Although separating

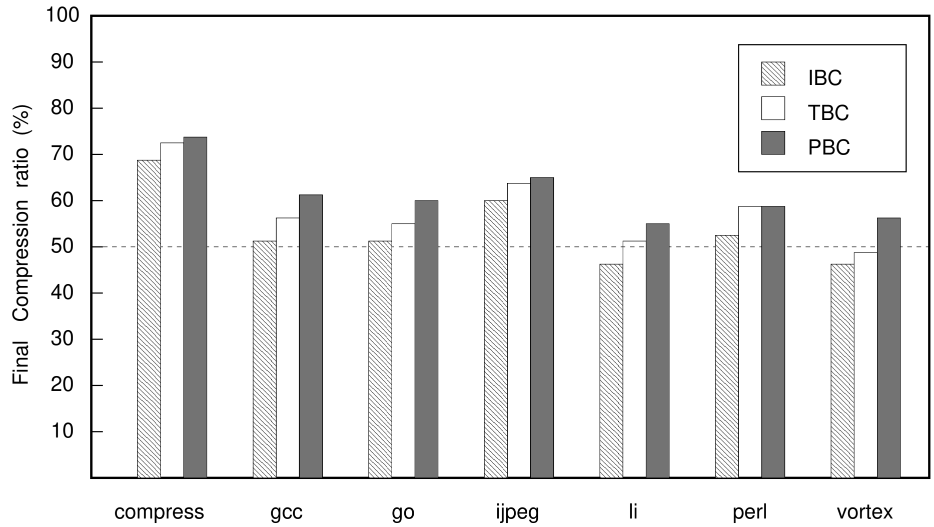

Figure 19: Final compression ratio for TBC/PBC/IBC.

expression trees into tree and operand patterns might result in some gain, the improvement is offset by the overhead resulting from using two sets of prefix bits, one for each pattern codeword. Moreover, as discussed in Section 6, there is a strong correlation between the number of distinct trees in a program and the number of distinct operand-patterns.

A preliminary design of the decompression engine is under way. The design is based on a synthesizable VHDL model of the decompression engine proposed for the IBC algorithm (Section 7.2). Preliminary performance tests have been carried out, using Exemplar/MGC Leonardo Compiler, and the results reveal minimum operation frequencies of 120MHz across all programs.

9 Future Work

This work can be improved in two ways. First, trees encode the same temporary register twice. If the compiler schedules expression trees in a pre-order schedule, it is possible to eliminate the second reference to a temporary, minimizing the size of the tree dictionary in `TBC`. In this case, the decompression engine should be able to keep track of the temporary registers, and to perform local register allocation at decompression time. There is evidence that other forms of dividing expression trees into symbols can result in better compression. For example, instruction `lw` usually appears together with operand `$29` (*stack-pointer*). A thorough analysis of this kind of correlation might reveal new patterns, that could eventually result in better compression.

10 Acknowledgments

This research was supported in part by a grant from CNPq under contract 300156/97-9, a grant from FAPESP under contract 1997/10982-0, a FAPESP research fellowship (98/13728-0), and a grant from CNPq/NSF 1998 Collaborative Research Project. This work would not be possible without the tools and resources provided by Mentor Graphics Corporation, through their Educational Program.

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] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Transaction on Information Theory*, vol. IT–22, no. 1, pp. 75–81, January 1976.

[6] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Transactions on Information Theory*, vol. 23, pp. 337–343, 1977.

[7] J. Ziv and Lempel A., "A universal algorithm for sequential data compression," *IEEE Transaction on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[8] J. Ziv and Lempel A., "Compression of individual sequences via variable–rate coding," *IEEE Transaction on Information Theory*, vol. 24, no. 5, pp. 337–343, September 1978.

[9] Welch, "A technique for high-performance data compression," *IEE Computer*, vol. 17, no. 6, pp. 8–19, July 1984.

[10] W. T. Wilner, "Burroughs B1700 memory utilization," in *Fall Joint Computer Conference*, 1972, pp. 579–586.

[11] D. A. Huffman, "A method for the construction of minimum–redundancy codes," *Proceedings of the IRE*, vol. 40(9), pp. 1098–1101, September 1952.

[12] 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.

[13] Michael Kozuch and Andrew Wolfe, "Compression of embedded system programs," in *Proceedings of the IEEE International Conference on Computer Design*, October 1994, pp. 270–277.

[14] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *Proc. of 35th ACM Design Automation Conference*, 1998.

[15] H. Lekatsas and W. Wolf, "Random access decompression using arithmetic coding," in *Proc. of the Data Compression Conference*, March 1999.

[16] IBM Corporarion, *CodePack PowerPC Code Compression Utility User's Manual*, 1998, Version 3.0.

[17] 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.

[18] Gerry Kane and Joe Heinrich, *MIPS RISC Architecture*, Prentice Hall, New Jersey, 1992.

[19] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley, Boston, 1988.

[20] 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.

[21] Martin Beneš, Andrew Wolfe, and Steven M. Nowick, "A high–speed asynchronous decompression circuit for embedded processors," in *Proceedings of 17th Conference on Advanced Research in VLSI*, Los Alamitos, CA, 1997, IEEE Society Press.

[22] Martin Beneš, Steven M. Nowick, and Andrew Wolfe, "A fast asynchronous huffman decoder for decompressed–code embedded processors," in *Async98*. 1998, ACM.

[23] Todd A. Proebsting, "Optimizing an ANSI C interpreter with superoperators," in *ACM Conference on Principles of Programming Languages*, January 1995, pp. 322–332.

[24] Michael Franz and Kistler Thomas, "Slim binaries," *Communication of the ACM*, vol. 40, no. 12, pp. 87–94, december 1997.

[25] Jean Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting, "Code compression," in *SIGPLAN Programming Languages Design and Implementation*, 1997.

[26] P. Centoducatte, G. Araujo, and R. Pannain, "Compressed code execution on dsp architecture," in *To appear in Proc. of 12th International Symposium on System Synthesis*, November 1999.