Code Compression Based on Operand Factorization

Guido Araujo, Paulo Centoducatte and Mario Cortes
University of Campinas
Institute of Computing
Campinas, SP 13083-970, Brazil
{guido,ducatte,cortes}@dcc.unicamp.br

Ricardo Pannain
PUC Campinas
Institute of Informatics
Campinas, SP 13020-904, Brazil
pannain@zeus.puccamp.br

Abstract

This paper proposes a code compression technique called operand factorization. The central idea of operand factorization is the separation of program expression trees into sequences of tree-patterns (opcodes) and operand-patterns (registers and immediates). Using this technique, we show that tree and operand patterns have exponential frequency distributions. A set of experiments were designed to explore this feature. They reveal an average compression ratio of 43% for SPECInt95 programs. A decompression engine is proposed, which assembles tree and operand patterns into uncompressed instruction sequences. An encoding that improves the design of the decompression engine results in a 48% compression ratio. Compression ratio numbers take into consideration an estimate of the decompression engine size.

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. Thus, 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 adopted in the design of the Thumb and MIPS16 processors. In these processors, shorter instructions are achieved 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 [9]. Another way to reduce the size of a programs is to design a processor which 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 which distinguish *code compression* from other compression problems.

This paper proposes a code compression technique based on the concept of *operand factorization*. 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*. Tree and operand patterns are then encoded separately. Variations of operand factorization have been used before [4, 3]. Our work differ from theirs in the sense that we are mainly interested in finding an encoding which allows efficient implementations of real-time decompression engines. Moreover, we provide a quantitative approach to the problem which validates the efficacy of our algorithm. This is measured by the compression ratio [1]. This paper is divided as follows. Section 2 discusses prior work on the problem of code compression. Section 3 details the concept of operand factorization. The compression algorithm is studied in Section 4, and the decompression engine is described in Section 5. Experimental results are analyzed in Section 6. Section 7 summarizes the work and proposes two extensions.

2. Related Work

A large number of techniques have been proposed in the literature for the file compression problem [2]. Many of these techniques are sequential in nature, in the sense that the decompression of the current codeword requires symbols from the partially decompressed string. A classical example is Lempel-Ziv (LZ) compression [11]. In LZ compression the dictionary is encoded together with the com-

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

pressed string. Pointers to previously parsed substrings are used to encode the current substring. Decompression is done substituting a pointer by the sub-string it points to. For the case of real-time decompression, this implies in a large latency.

The first approach for code compression in a RISC architecture was originally proposed by Wolfe and Channin [14]. The processor described in [14] 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. Program target addresses have different values if the line is in main-memory or in cache. The 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 clever encoding of the LAT entries restricts the size of the table. The CCRP uses a *Cache Line Address Lookaside Buffer* (CLB) to store a set of recently fetched LAT entries. The compression algorithm for the CCRP is based on Huffman encoding [6] of byte long symbols. Using this approach, a 73% compression ratio has been reported for the MIPS instruction set [14, 8].

Lefurgy et al. [9] proposed an interesting code compression technique based on dictionary encoding. In [9] 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 then computed by adding these two. This technique 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 through 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 [9].

Liao et al. [12] proposed a compression technique based on dictionaries. The main idea in [12] is the substitution of common instruction sequences for 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%.

Wolf and Lekatsas [10] proposed a similar approach to operand factorization. Unlike operand factorization, the selection of the instruction combinations is not based on expression trees. The average compression ratio reported for the MIPS architecture was 51%. It is not clear from [10] if this number takes into consideration an estimate of the size of the decompression engine.

3. Operand Factorization

The central idea in this paper is the separation of each expression tree in the program into two components: *tree-pattern*, formed by the instructions in the expression tree after removing its operands; and a sequence of operands, known as *operand-pattern*, containing the registers and immediates used by the tree-pattern. Expression trees are constructed as in [1]. They do not contain branch instructions nor cross basic block boundaries. 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 [13] as an encoding technique for intermediate representation compression. Consider for example, the expression tree in Figure 1(a). Figure 1(b) shows the tree-pattern resulting after the operands have been factored out from the original expression tree. *Stars* (wild-cards) are

```
addiu r4, r4, 1             addiu *, *, *
lui r1, 0                   lui *, *
sw r1, 0(r4)                sw *, *(*)

     (a)                         (b)

          [r4,r4,1,r1,0,r1,0,r4]

                   (c)
```

Figure 1. (a) Expression tree; (b) Tree-pattern; (c) Operand-pattern.

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 1(c) shows the operand-pattern determined after the expression tree in Figure 1(a) is factored. In order to study operand factorization, we use a set of SPECInt95 programs. The programs were compiled for the MIPS R2000 using gcc -O2 version 2.8.1. Although the R2000 is an old processor, it enabled us to leverage on in-house tools and expertise. On the other hand, the R2000 is a classical RISC architecture that has many of the features of a modern RISC processor.

We have noticed that the number of (distinct) tree and operand patterns in a program is not only small, but also much smaller than the number of expression trees in the same program. Table 1 lists the number of expression trees and patterns for our program set. From Table 1, gcc has 311488 different expression trees, which can be represented by 1547 tree-patterns, i.e. only 0.5% of all trees. This small number can be explained by: (a) the reduced number of instructions in the instruction set of a RISC processor; (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) ways in which compilers generate code for *abstract syntax tree* constructs, like `if-then-else` statements and `for` loops. Similar numbers can also be derived for operand-patterns. Operand-patterns have more uniform distributions though. For example, gcc has 311488 operand sequences and these can be represented by 41486 operand-patterns, i.e. 13.3% of all sequences.

Program	Expression Trees	Tree-Patterns (%)	Operand-Patterns (%)
go	54651	578 (1.1)	12561 (23.0)
li	15761	157 (1.0)	3056 (19.4)
compress	1444	125 (9.0)	731 (51.0)
perl	62915	648 (1.0)	11209 (18.0)
gcc	311488	1547 (0.5)	41486 (13.3)
vortex	128104	471 (0.4)	16143 (13.0)
ijpeg	38426	767 (2.0)	9839 (26.0)

Table 1. Number of tree and operand patterns in a program. The numbers in parentheses are percentage with respect to the total number of expression trees and operand sequences.

Interesting enough, small programs seem to be much less redundant than large programs. In compress (the smallest program studied), tree-patterns correspond to 9% of all possible trees in the program, while operand-patterns are 51% of all operand sequences. This happens because statements like `if-then-else` and `for` are compiled into very similar set of patterns. Large programs use this set many times, while small programs do not. An important issue in our study is the determination of the size contribution of each tree and operand pattern. Two experiments have been carried out to answer that. In the first experiment, the individual frequencies of each unique tree-pattern were determined. Tree-patterns were then ordered in a decreasing order of frequency, and the cumulative percentage of the ex-

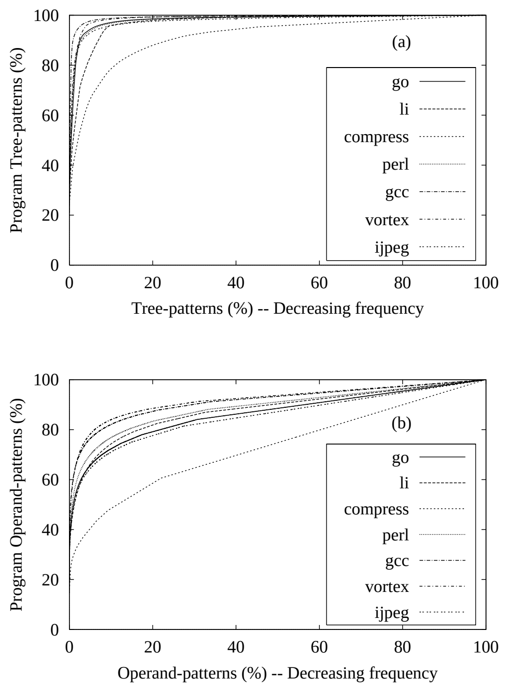


Figure 2. (a) Percentage of covered expression trees; (b) Percentage of covered operand sequences.

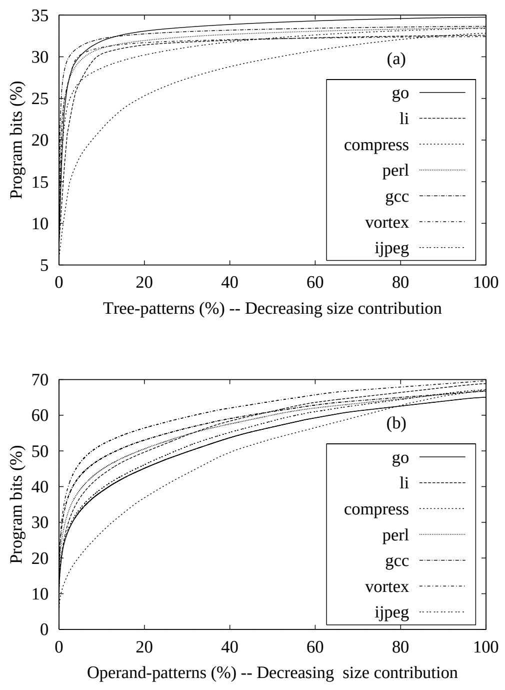


Figure 3. Percentage of program bits due to: (a) tree-patterns; (b) operand-patterns.

pression trees covered by these patterns was computed. The results are shown in Figure 2(a). The frequency of each tree-pattern is the derivative of the graph in Figure 2(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. Figure 2(a) shows that on average 20% of the tree-patterns correspond to almost all trees in a program. This rule works for all programs in Figure 2(a) but compress. The distribution of expression trees in compress is much more uniform. A similar graph was also derived for operand sequences. Figure 2(b) shows the cumulative number of operand sequences in a program that are covered by distinct operand-patterns. On average, 20% of the operand-patterns account for about 80% of the operand sequences in a program. As before compress numbers differ from the other programs. The fact that frequent patterns correspond to a large share of the total number of bits in a program cannot be implied only based on the frequency of the patterns. The size of each pattern has to be considered. We have noticed (Figure 4) that for the majority of the cases, patterns with medium frequencies are larger than very uncommon/common patterns. The contribution of each pattern, in terms of program bits, was then computed. Figure 3(a) plots the cumulative percentage of the program bits due to tree-patterns. Tree-patterns in the horizontal axis are ordered based on its size contribution to the program. Tree-patterns can contribute to at most 35% of all program bits, because tree-patterns correspond only to opcode bits. In the R2000 architecture [7] at most 11 bits (i.e. 35.2% of an instruction) are used for opcode. The graph in Figure 3(a) reveals that 20% of the tree-patterns correspond to approximately 32% of all program bits. A similar graph (Figure 3(b)) was also determined for operand-patterns. Although the contribution of operand-patterns is more scattered across different programs, still 20% of the operand-patterns correspond to 50% $\pm$ 5% of all program bits.

4. Compression Algorithm

The experiments in Section 3 reveal that a small percentage of small tree and operand patterns account for the large majority of the bits in a program. This confirms, at an instruction level, the observation made in [9] about the role played by small bit strings in program code. On the other hand, it also brings to light a feature of programs that cannot be captured by other compression methods. Operand factorization recognizes the fact that any encoding technique which 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 which performs sequential compression, like LZ [11], will not be able to detect the simple tree-pattern

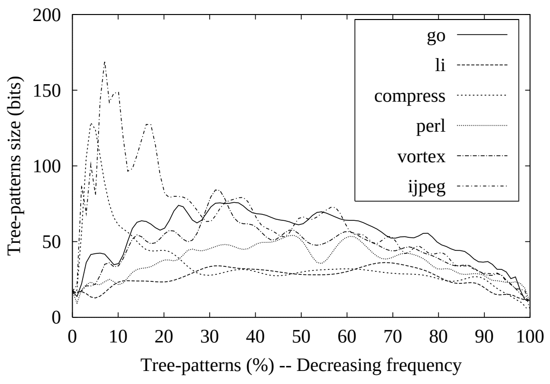


Figure 4. Distribution of the average tree-pattern size.

`[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 which encodes the opcode and the destination register (in this order) using 6 bits each. If a byte is chosen as the size of the encoding symbol, then the first byte of instructions `[lw r2,*,*]` and `[lw r15,*,*]` 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 which are shared by two different tree-patterns. For example, in gcc operand-pattern `[r2, r0, r4]` is used by expression trees `[ori r2, r0, r4]` and `[addu r2, r0, r4]`. Based on that, we encode tree and operand patterns separately. Expression trees are encoded as codeword pairs $[Tp, Op]$, where Tp (Op) is the codeword for a tree (operand) pattern. The encoding algorithm is divided in two phases. First (Section 4.1), tree and operand patterns are encoded. Second (Section 4.2), codewords are compacted into a compressed program.

4.1. Pattern Encoding

The analysis above shows that tree and operand patterns have very non-uniform distributions. This suggests that a variable-length encoding algorithm can result in improved compression ratios. On the other hand, variable length encoding implies in low decoding efficiency. The main issues involved here are: detecting the length of a codeword, extracting and aligning codeword bits. We studied four different encoding methods to encode patterns. The first two methods are fixed-length (i.e. lossy) and Huffman encoding. The other two methods (described below) take into consideration the impact of encoding in the performance of the decompression engine.

- Bounded-Huffman (BH)
  In Bounded-Huffman a escape bit is appended to the beginning of the codeword so as to identify if the codeword uses Huffman or fixed-length encoding. Bounded-Huffman is also used in MPEG-2 [5] as a way of limiting the size of the Huffman codeword.

- VLC Encoding (VLC)
  This is a variation of the MPEG-2 VLC encoding [5]. In this method, Bounded-Huffman codewords are selected so that the codeword leading zeroes encode the size of the codeword. The goal of this method is to simplify the codeword extraction logic.

In Section 6, two sets of experiments were performed in order to determine the best encoding technique for program patterns. The experiments show that using Bounded-Huffman encoding for both tree and operand patterns results in an average 37% compression ratio. Moreover, this ratio is only 2% higher than the ratio resulting if patterns were encoded exclusively using Huffman. The experiments also show that the compression ratio resulting by encoding patterns using VLC is on average 3% higher than that of Bounded-Huffman.

4.2. Codewords Compaction

After tree and operand patterns are encoded, they are appended sequentially to form a list of codeword pairs: $[Tp_1, Op_1 | Tp_2, Op_2 | \ldots | Tp_n, Op_n]$. Bars are used here to mark the end of a codeword pair, and the beginning of another. Codewords are allowed to split at the end of each 32-bit words. Bits from splitted codewords are spilled into the next word. The size of a codeword is limited to 16 bits, so that in the worst case, a word will carry at least one pattern. The possibility of splitting codewords and the use of a variable length code puts a lot of pressure in the design of the decompression engine. On the other hand, we have noticed that large compression ratios can only be achieved if codewords can split across word boundaries. The reason, also noticed in [9], is that many common patterns are originated from a single instruction word. Therefore, constraining codewords to a single word considerably limits the compression ratio.

5. Decompression Engine

This section proposes a decoding engine for our compression method (Figure 5). Our approach is to trade part of the silicon area gained by an aggressive compression, for an improved design of the decompression engine. 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.

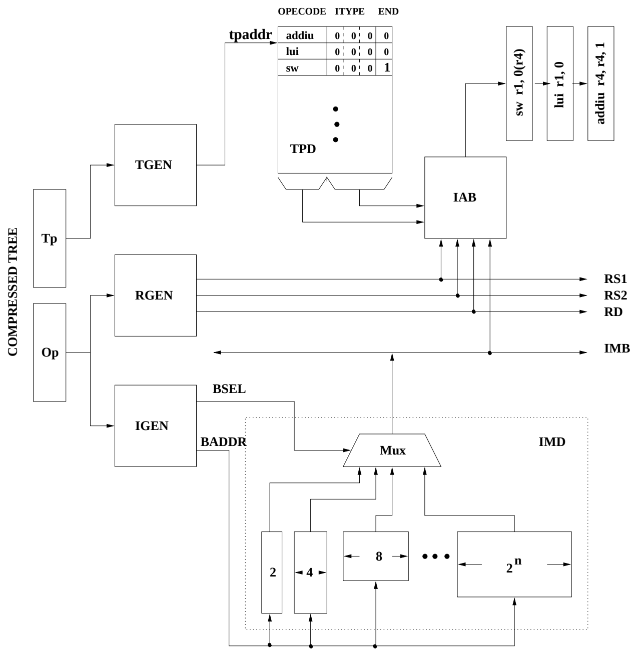


Figure 5. The Decompression Engine.

5.1. Tree-pattern Generation

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 the `IAB` to decide how to assemble a decompressed instruction. The `IAB` puts together `OPCODE`, register (`RS1`, `RS2`, `RS2`) and immediate (`IMB`) bits to form an instruction. Bit-field `END` is used to check for the last instruction in the tree-pattern. The overhead of the `TPD`, with respect to the uncompressed programs, is shown in Table 2. On average, the `TPD` is only 1.7% of the original program size.

5.2. Register Generation

The Register Generator (`RGEN`) is a state machine that 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 RGEN is formed by three (register) buses: RS1, RS2 and RD, that generate the bits corresponding to the instruction source and destination registers. The number of states of RGEN is bounded by the number of instructions in the largest tree-pattern of the program. In those cases when a tree-pattern is smaller than the largest tree-pattern in the program, the unused state variables can be made don't care for the RGEN combinational logic. Operand-patterns that have immediates can be used to simplify the RGEN logic. When the operand-pattern encoded by Op contains an immediate, the register bus associated to the immediate operand is not needed and can be made don't care. For example, the operand-pattern `[r2, r5, 4]` will result in `RD = 00010`, `RS1 = 00101` and `RS2 = xxxxx` (i.e. don't cares). Notice that patterns which share registers can also be used to minimize the RGEN logic. For example, sequence `[r2, r0, r5]` results in similar values for register buses RSD and RS2 as sequence `[r2, r3, r5]`. In other words, the encoding of the product terms in the combinational logic of the RGEN machine reflects the sharing of common registers in the operand sequences. This encoding of operand-patterns

Program	TPD Ratio (a)	IMD Ratio (b)	RGEN Ratio (c)
go	1.0	1.0	5.5
li	0.6	2.7	2.4
compress	6.0	5.0	12.6
perl	0.9	2.1	3.0
gcc	0.6	1.2	2.4
vortex	0.4	1.2	1.7
ijpeg	2.3	2.0	6.6

Table 2. Percentage with respect to the uncompressed program of: (a) TPD Ratio; (b) IMD Ratio; (c) RGEN Ratio.

can be naturally translated into a minimization problem for the RGEN logic. An upper bound on the size of RGEN was determined by adding up the size (in bits) of all operand-patterns. This is equivalent to have a dictionary implementation for the RGEN module. In this case, the average RGEN size with respect to the uncompressed program (RGEN Ratio in Table 2) is 4.9%. Notice that 4.9% is a very loose bound for the size of RGEN, given that it does not consider the minimization resulting from a state machine implementation.

5.3. Immediate Generation

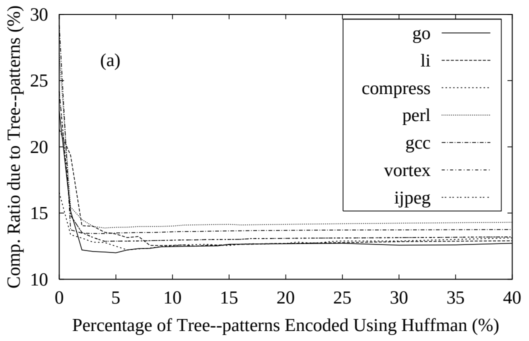
The IMD module in Figure 5 stores the immediates used by the program. A single entry in IMD is provided to each distinct immediate in the program, no matter which instruction uses it, or how many times it shows up. For example, a single constant 4 is stored for instructions `[bgez r5, 4]`, `[lw r6, 4(r29)]`, and `[srl r5, r3, 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. IGEN is a state machine that works in parallel with RGEN and TGEN. 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 the IMD (Table 2) is only 2.2%.

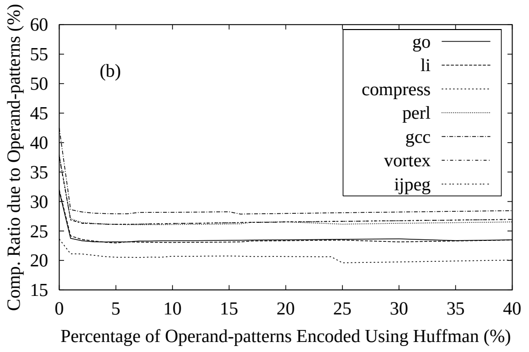
5.4. Branch Target Address

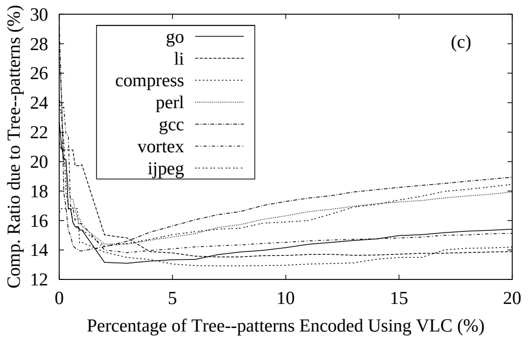
We borrow here the ideas proposed in [9]. The branch target is divided into a target address `addr` (21 bits) and `offset` (5 bits). Unlike the approach in [9], branch instructions are compressed. During decompression the values of `addr` and `offset` are retrieved from the IMD dictionary and the branch instruction is assembled. We assume that the control unit of the processor treats branch offsets as aligned to codeword boundaries (i.e. bit aligned). During a fetch operation, the word at address `addr` is fetched from memory and the compressed instruction is extracted starting at bit `offset`. The overall reduction in the branch range is 32. Nevertheless, for the programs analyzed, only a small percentage of targets required more than 21 bits. For those branches, a jump table is provided which stores the addresses of the targets. Similarly to [9], jump table addresses are patched up to reflect the new compressed addresses.

6. Experimental Results

Three sets of experiments have been designed to determine the best encoding for tree and operand patterns. The purpose of the first set was to determine the compression ratio using fixed-length and Huffman encoding. The goal of the second set was to determine the best splitting between Huffman and fixed-length encoding in Bounded-Huffman. Tree and operand patterns were ordered into two separate lists according to their contribution (in bits) to the program. Patterns were then encoded using combinations of fixed-length and Huffman codewords. In the first (last) combination, 0% (50%) of the patterns were encoded using Huffman (fixed-length), while the rest was encoded using fixed-length (Huffman). The results of the experiments are plot-







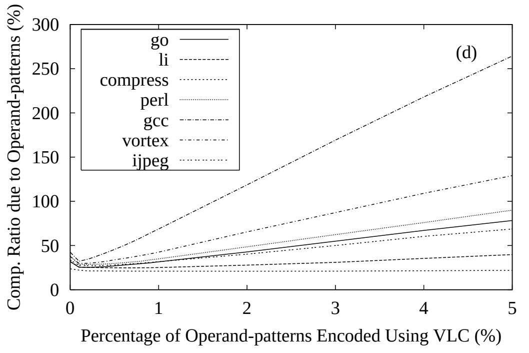


Figure 6. Compression ratio for: (a) tree-patterns using BH; (b) operand-patterns using BH; (c) tree-patterns using VLC; (d) operand-patterns using VLC.

ted in the graphs of Figure 6(a)-(b). The graphs' horizontal axes show the percentage of the patterns which have been encoded using Huffman. The vertical axes are the share of the program compression ratio due only to that encoding. The points between 0% and 50% use a mix of fixed-length and Huffman codewords, with an escape bit to distinguish them. From Figure 6(a)-(b) one can notice that the compression ratio tends to saturate. The more patterns are encoded using Huffman, the less is their contribution to the compression ratio. This reflects the exponential distribution of program patterns studied in Section 3. Patterns that have small contributions do not change much the program entropy. Lower entropy results in a Huffman encoding that approaches fixed-length encoding [2]. By switching from Huffman to fixed-length encoding, at a point where the pattern distribution becomes more uniform, one can minimize the impact of not using Huffman. Consider, for example, program go in Figure 6(a). The contribution to the compression ratio when 5% of the tree-patterns are encoded using Bounded-Huffman is 12.0%, only 1.4% higher than if all tree-patterns were encoded using Huffman. A similar number can also be determined for operand-patterns (Figure 6(b)).

Encoding Methods	Fixed-Length	Huffman	Bounded-Huffman	VLC
Fixed-length	57.7	46.2	48.1	50.5
Huffman	45.4	35.0	35.8	38.2
BH	46.0	36.6	37.4	39.8
VLC	47.9	37.5	38.3	40.7

Table 3. Average compression ratio after combining encoding methods for tree (rows) and operand (columns) patterns.

The third set of experiment uses VLC codewords. The graphs in Figure 6(c)-(d) show the resulting compression ratio. In this case, fewer patterns are encoded using VLC than using Huffman in Figure 6(a)-(b). The minima for tree (operand) patterns occur around 2.5% (0.2%) of the patterns. The reason comes from the fact that VLC codewords are larger than Huffman codewords (they have to encode size information). This becomes evident as more patterns are encoded using VLC, causing a rapid increase of the compression ratio that sometimes can surpass 100%. The average VLC compression ratio contribution for tree (operand) patterns in all programs was 13.7% (26.9%).

Table 3 shows the average compression ratio, when the encoding methods for tree and operand patterns are combined. The Bounded-Huffman and VLC compression ratios were determined from the graphs of Figure 6(a)-(d) by taking the average of the global minima of all programs.

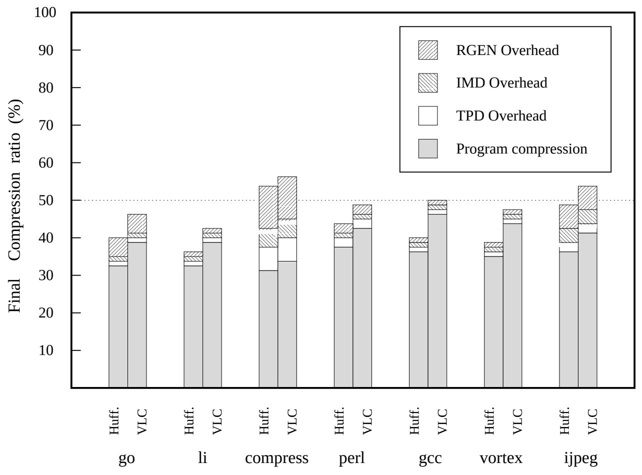


Figure 7. Final compression ratio using Huffman and VLC.

Table 3 provides a solution space for choosing an encoding method. Based on that, one can trade compression efficiency for decompression speed. The worst compression ratio (57%) is achieved when tree and operand patterns are encoded using fixed-length codewords. As expected, the best compression ratio (35%) results when both patterns are encoded using Huffman. The drawback of this encoding is that it results in unbounded codewords, which carry no size information. On the other hand, encoding patterns using VLC results in a 41% compression ratio, This ratio is only 5.7% higher than Huffman encoding. This is the price one has to pay in order to minimize the latency of the size detection logic in the decompression engine.

A fair assessment of the compression efficiency of operand factorization needs to take into consideration the silicon area of the decompression engine. The size of the immediate (Section 5.3) and tree-pattern (Section 5.1) dictionaries can be estimated based on the number of bits they use. An upper bound on the size of RGEN was determined in Section 5.2. Figure 7 shows the final average compression ratio for Huffman (43%) and VLC (48%) encoding, once the overhead due to the dictionaries and RGEN is considered. As shown, the engine overhead is fairly small (on average 8%).

7 Conclusions

This paper proposes a code compression technique called operand factorization. The best compression ratio using this technique results in a 35% compression ratio. A decompression engine is proposed to assemble tree and operand patterns into uncompressed instructions. This work can be improved in two ways. First, operand-patterns encode the same temporary register twice. If the compiler schedules expression trees in a pre-order schedule, then it is possible to eliminate the second reference to a temporary, thus minimizing the size of operand-patterns. In this case, the decompression engine should be able to keep track of the temporary registers. The second improvement can be done by analyzing the correlation between tree and operand patterns.

8 Acknowledgments

This research was supported in part by CNPq under contract 300156/97-9.

References

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

[2] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Advanced Reference Series. Prentice Hall, New Jersey, 1990.

[3] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *SIGPLAN Programming Languages Design and Implementation*, 1997.

[4] M. Franz and K. Thomas. Slim binaries. *Communication of the ACM*, 40(12):87–94, december 1997.

[5] B. G. Haskell, A. Puri, and A. N. Netravali. *Digital Video: an Introduction to MPEG–2*. Chapman & Hall.

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

[7] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, New Jersey, 1992.

[8] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proceedings of the IEEE International Conference on Computer Design*, pages 270–277, October.

[9] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proceedings of MICRO–30: The 30th Annual International Symposium on Microarchitecture*, pages 194–203, December 1997.

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

[11] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transaction on Information Theory*, IT–22(1):75–81, January 1976.

[12] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *To appear in ACM Transactions on Design Automation of Electronic Systems*, 4(1), 1998.

[13] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *ACM Conference on Principles of Programming Languages*, pages 322–332, January 1995.

[14] A. Wolfe and A. Channin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of MICRO–25: The 25th Annual International Symposium on Microarchitecture*, pages 81–91, December 1992.