O conteúdo do presente relatório é de única responsabilidade do(s) autor(es).
The contents of this report are the sole responsibility of the author(s).

Code Compression Based on Operand Factorization

Guido Araújo *Ricardo Pannain*
Paulo Centoducatte *Mario Côrtes*

Relatório Técnico IC–98-25

Julho de 1998

Code Compression Based on Operand Factorization

Guido Araújo Ricardo Pannain
Paulo Centoducatte Mario Côrtes

July 7, 1998

Abstract

This paper proposes a code compression technique called *operand factorization*. The key idea of operand factorization is the separation of program expression trees into sequences of *tree-patterns* (opcodes) and *operand-patterns* (registers and immediates). Using operand factorization we show that tree and operand patterns have exponential frequency distributions. A set of experiments is performed to determine the best encoding technique that explores this feature. The experimental results show an average compression ratio of 35% for SPEC CINT95 programs, when patterns are encoded using Huffman encoding. Another encoding method, that improves the performance of the decompression engine, results in an average compression ratio of 41%. A decompression engine is proposed which assembles tree and operand patterns into uncompressed instruction sequences, using a combination of dictionaries and state machines.

1 Introduction

The major feature that distinguishes the design of an embedded system from the design of other computing systems are the design constraints. Designing an embedded system usually implies in limiting power consumption, size/weight and cost, while meeting performance goals. Because these systems are designed for high-volume markets, such as consumer electronics, any cost reduction can have a large impact in the final price of the product. Driven by the need to reduce cost, embedded system designers are integrating microprocessors, program/data memories and ASIC modules into a single chip.

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 the ASIC modules. Thus, minimizing program size has become an important part of the design effort (cost) of an embedded system. On the other hand, as the complexity of these systems grows, assembly level programming and debugging are becoming un-practical and time-consuming. The recent trend towards using high-level languages for embedded programs aims at cutting development and maintenance costs. Unfortunately, embedded CISC compilers are still ineffective at optimizing for code size, what again implies in increasing programs size.

Size is also the issue that keeps RISC processors away from embedded system designs. Although modern RISC architectures, combined with optimizing compilers, can result in better performance, the size of RISC programs is usually larger than the size of programs in a CISC architecture. This happens mainly because CISC instruction sets use variable-length instructions, while RISC instructions have fixed-length formats. A way around that is to restrict RISC instruction formats such that shorter instructions are allowed. This is the approach adopted in the design of the Thumb and MIPS16 processors[1]. 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

[1] Thumb and MIPS16 have 16-bit instructions

the same amount of computation. The net result are 30%-40% smaller programs running 15%-20% slower than programs using standard RISC instructions [1]. Another way to reduce the size of RISC 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 also allow for random codeword decompression. These are the two major features which distinguish *code compression* from traditional compression problems.

This paper proposes a code compression technique based on the concept of *operand factorization*. The key idea here 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 in [2, 3]. Our work differs 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 verifies the efficacy of our method. The experimental results show that operand factorization can result in very low compression ratios [2] (average 35% for Huffman encoding). A decompression engine is proposed which assembles the uncompressed instructions from the compressed tree and operand patterns.

This paper is organized 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 resumes the work and proposes two interesting extensions.

2 Related Work

A large number of techniques have been proposed in the literature for the file compression problem [4]. 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.

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

A classical example is Lempel-Zvi (LZ) compression [5]. 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 done by substituting a pointer for the substring it points to. For the case of real-time decompression this implies in a large latency/area overhead, Consider, for example, the case when a branch instruction in the compressed program is taken. This section describes only prior compression techniques that can lead to efficient real-time decompression engines.

The first approach to code compression in a RISC architecture was originally proposed by Wolfe and Channin [6]. The processor described in [6] 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 depending 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 [7], and uncompressed program symbols are one byte long. Using this approach a 73% compression ratio has been reported for the MIPS instruction set [6, 8]. The decompression engine proposed in the CCRP is also the only implemented real-time RISC decompression engine ever reported in the literature [9].

Lefurgy et al. [1] proposed an interesting code compression technique based on dictionary encoding. In [1] 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 key idea there is to divide the target address bits 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. Although this approach restricts the range of branch addresses, few program branches require long range addresses [1], and these can be determined through a small jump table. 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. In the second approach, better compression ratios were achieved through nibble aligned variable length encoding. In this case, average size reductions of 39%, 34%, and 26% have been reported for the PowerPC, ARM and i386 processors [1].

In [3] a technique called *patternization* was proposed to encode bytecode strings. The basic idea behind patternization is to extract all possible instruction patterns from the expression trees in a program. Tree-patterns are compiled into a tree-matching code generator based on *iburg* [10]. Expression trees covering is then performed to determine the best set of patterns that covers the program. The original expression trees in the program are replaced by their patterns and the sequence of patterns is encoded using gzip. The compressed format is named *wire-format*, and results in compression ratios close to 30% [3]. The main goal of patternization is to synthesize an efficient virtual machine that can execute compressed code delivered through a network. The use of gzip makes this technique inappropriate for real-time decompression.

Slim binaries is a compression technique proposed by Frank and Thomas [2] for a platform independent program format. In [2] program *abstract syntax trees* are compressed into a sequence of symbols from an adaptive dictionary. Similar syntax trees are encoded using the same sequence. The compression scheme is based on the LZ algorithm, and results in compression rates of up to 50% [2]. Slim binaries are targeted to source level compression. This technique has been used in MC68020-base Macintosh computers since 1993.

Liao et al. [11] proposed a compression technique exclusively based on software. Their main idea is to substitute common instruction sequences for sub-routine calls. A careful analysis of this method shows that it is effective only if a large number of long instruction

sequences can be found. Otherwise, the overhead due to the introduction of call and return instructions can surpass the gains. In Section 4 we show that, in average, the most common instruction patterns are very small. This was also noticed in [1].

3 Operand Factorization

The central idea in this paper is the separation of each expression tree in the program into two components: a *tree-pattern*, formed by the instructions in the expression tree after removing its operands; and a sequence of operands, called *operand-pattern*, containing the registers and immediates used by the tree-pattern. 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 [12] as an encoding technique for bytecode 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*

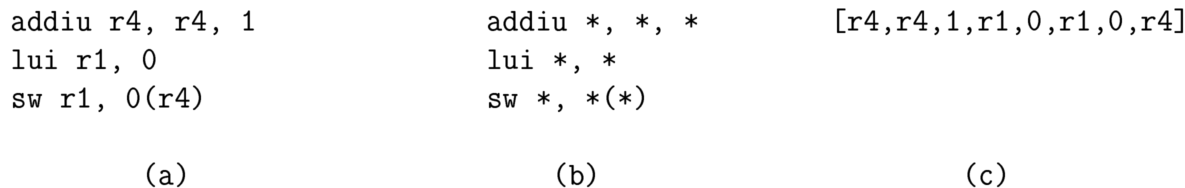

Figure 1: (a) Expression tree; (b) Tree-pattern; (c) Operand-pattern.

(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 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 SPEC CINT95 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. Moreover, 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 much smaller than the total number of expression trees in the same

program. In order to simplify the notation, the term tree (operand) pattern will mean, from now on, *distinct* tree (operand) pattern. Table 1 lists the number of expression trees and patterns for our program set. As shown in 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[3] operand sequences, and these can be represented by 41486 operand-patterns, i.e. 13.3% of all sequences. This is not a surprise, given the large number of possible ways that registers and immediates can be combined in a instruction. Nevertheless, this still shows how

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 number in parenthesis is percentage with respect to the total number of expression trees.

redundant program code can be. Interesting enough, small programs seem to be much less redundant than large programs. In compress (the smaller program studied), tree-patterns correspond to 9% of all possible trees in the program, while operand-patterns are 51% of all operand sequences. We have considered two explanations for that. First, because of their sizes, large programs have a higher probability of using the majority of the combinations of instructions and operands available in the *Instruction Set Architecture* (ISA). Once the

[3]The number of operand sequences is the same as the number of expression trees.

majority of combinations are present, pattern repetition becomes very common. Second, statements like `if-then-else` and `for` are compiled into very similar sets of patterns. As a result, any program compiled by the same compiler, no matter its size, will have a common set of patterns. Large programs use this set many times, small programs do not.

An important question for the study of the compression problem is to determine the frequency of each tree and operand pattern in a program. Two experiments have been carried out to answer this question. In the first experiment, the individual frequencies of each unique tree-pattern were determined. Tree-patterns were then ordered in decreasing frequency, and the cumulative percentage of the expression trees covered by these patterns was computed. The results are shown in Figure 2. In Figure 2, the frequency of each tree-pattern is the derivative of the graph. Based on that, we concluded that the frequency of a tree-pattern decreases almost exponentially as the pattern becomes less and less frequent. In other words, frequent tree-patterns cover much more expression trees than infrequent ones. Figure 2 shows that in average 20% of the tree-patterns correspond to almost all trees in a program. This rule works for all programs in Figure 2 but compress. As explained before, the distribution of expression trees in compress is much more uniform. A similar graph was also derived for operand sequences. Figure 3 shows the cumulative number of operand sequences in a program that are covered by distinct operand-patterns. As before, operand-patterns (horizontal axis) are ordered in decreasing frequency. In 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. Very uncommon patterns could contribute with a large number of bits, which could eventually compensate for their low number. The graph in Figure 6 shows the interpolated[4] size distribution of tree-patterns. For the majority of the cases, tree-patterns with medium frequencies have more bits than very uncommon/common patterns. Operand-patterns behavior is similar. The contribution of each pattern, in terms of program bits, was then computed. Figure 4 plots the cumulative percentage of the program bits due to tree-patterns. Tree-patterns in the

[4]Bezier method.

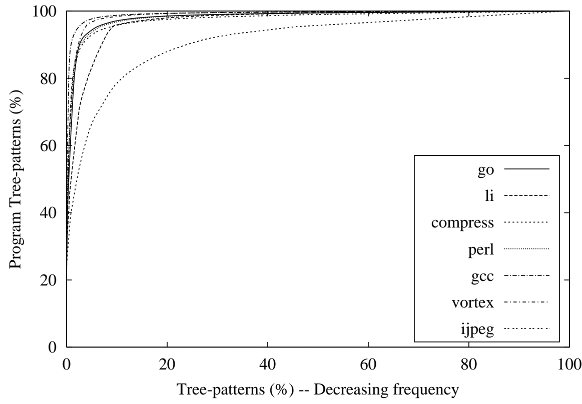

Figure 2: Percentage of expression trees due to tree-patterns.

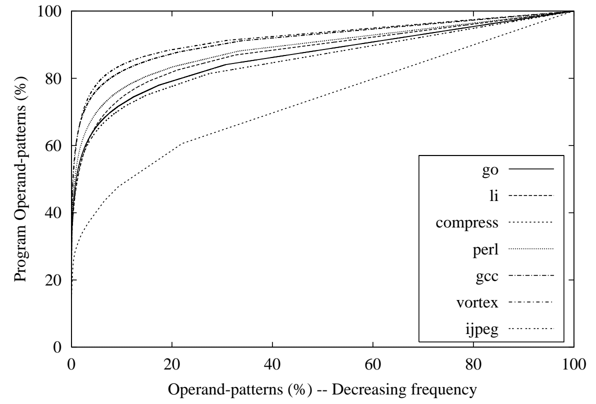

Figure 3: Percentage of operand sequences due to operand-patterns.

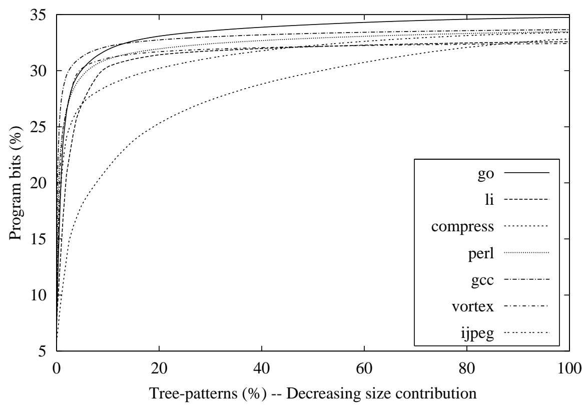

Figure 4: Percentage of program bits due to tree-patterns.

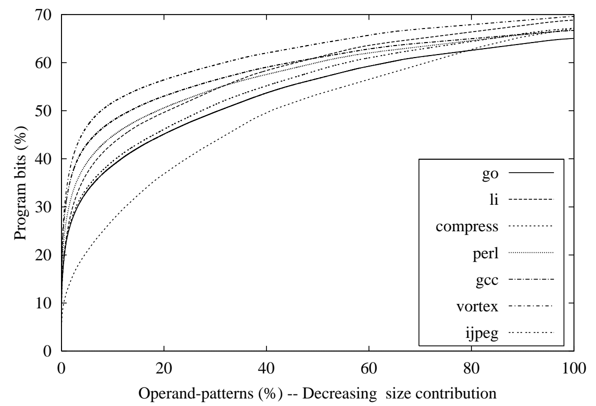

Figure 5: Percentage of program bits due to operand-patterns.

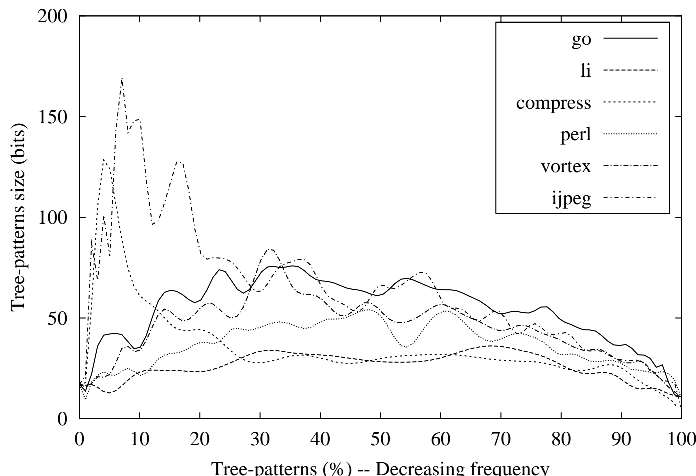

Figure 6: Distribution of the average tree-pattern size.

horizontal axis are ordered based on their contribution to the program size. Tree-patterns can contribute to at most 35% of all program bits, because tree-patterns correspond only to opcode bits. In the R2000 architecture [13] at most 11 bits (i.e. 35.2% of an instruction) are used for opcode. The graph in Figure 4 reveals that 20% of the tree-patterns correspond to approximately 32% of all program bits. A similar graph (Figure 5) was also determined for operand-patterns. In this case, the contribution of operand-patterns is more scattered across different programs. Nevertheless, it is possible to say that 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 [1] 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 [5], is not able

to detect the simple tree-pattern formed by `lw *,*,*` followed by `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 r17,*,*` 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 tree-patterns. For example, in gcc operand-pattern `[r2, r0, r4]` is used in instructions `xor r2, r0, r4` and `sub 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, tree and operand patterns are encoded (Section 4.1). Second, codewords are compacted into a compressed program (Section 4.2).

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. Although these problems can be handled through the design of efficient decoding engines, as in [14], they still represent an overhead to decompression. We studied four different encoding methods to encode tree and operand patterns. The first two methods are fixed-length and Huffman encoding. The other two methods take into consideration the impact they might have in the performance of the decompression engine. They are:

- Bounded-Huffman (BH)

  In Bounded-Huffman an 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 [15] and in CCRP[6]. It is useful as a way to restrict the size

of the Huffman codeword, thus reducing the complexity of the decompression engine. Since Huffman encoding is an optimal technique[4], encoding part of the symbols using fixed-length codewords will result in higher compression ratios. In Section 6 we show that this overhead can be almost completely eliminated.

- VLC Encoding (VLC)

  This is a variation of the MPEG-2 VLC encoding [15]. 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 logic associated to the part of the decoding engine responsible for codeword extraction.

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 2.4% higher than the ratio resulting if the patterns were encoded exclusively using Huffman. The experiments also show that the compression ratio resulting by encoding patterns using VLC is in average 3.3% higher than for Bounded-Huffman.

4.2 Codeword 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 can carry at least one pattern. That is more than enough to achieve very low compression ratios (see Section 6). The possibility of splitting codewords and the use of a variable length code, put a lot of preassure in the design of the decompression engine. On the other hand, we have noticed that lower compression ratios ($< 50\%$) can only be achieved if codewords can split across word boundaries. The reason, also noticed in [1], is related to the fact that many common patterns are originated from a single instruction word. Therefore, constraining codewords to a single word considerably limits the compression ratio. An alternative to that is to combine operand

factorization and cache-line based compression [6, 8]. This is an interesting possibility that may be considered in the future.

5 The Decompression Engine

Despite of its high compression ratios, variable length encoding implies in low decoding efficiency. Nevertheless, we believe that the path to higher compression ratios goes through an encoding scheme which explores the very unbalanced nature of program patterns. This section proposes a decoding engine for our compression method (Figure 7). Our approach is to trade part of the silicon area gained by an agressive compression, by an improved design of the decompression engine. First, fields Tp and Op are extracted from the compressed word. Efficient extraction circuits, like the one proposed in [14], can be used to do that. 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 an Instruction Assembly Buffer (`IAB`) that assembles the decompressed instructions. In the following sections we detail each one of the modules used in 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 the `OPCODE` bits together with the registers (`RS1`, `RS2`, `RD`) and immediate (`IMB`) bits to form an instruction. `TPD` entries are retrieved sequentially starting at `tpaddr`. 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. In average, the `TPD` respresents only 1.7% of the original program size.

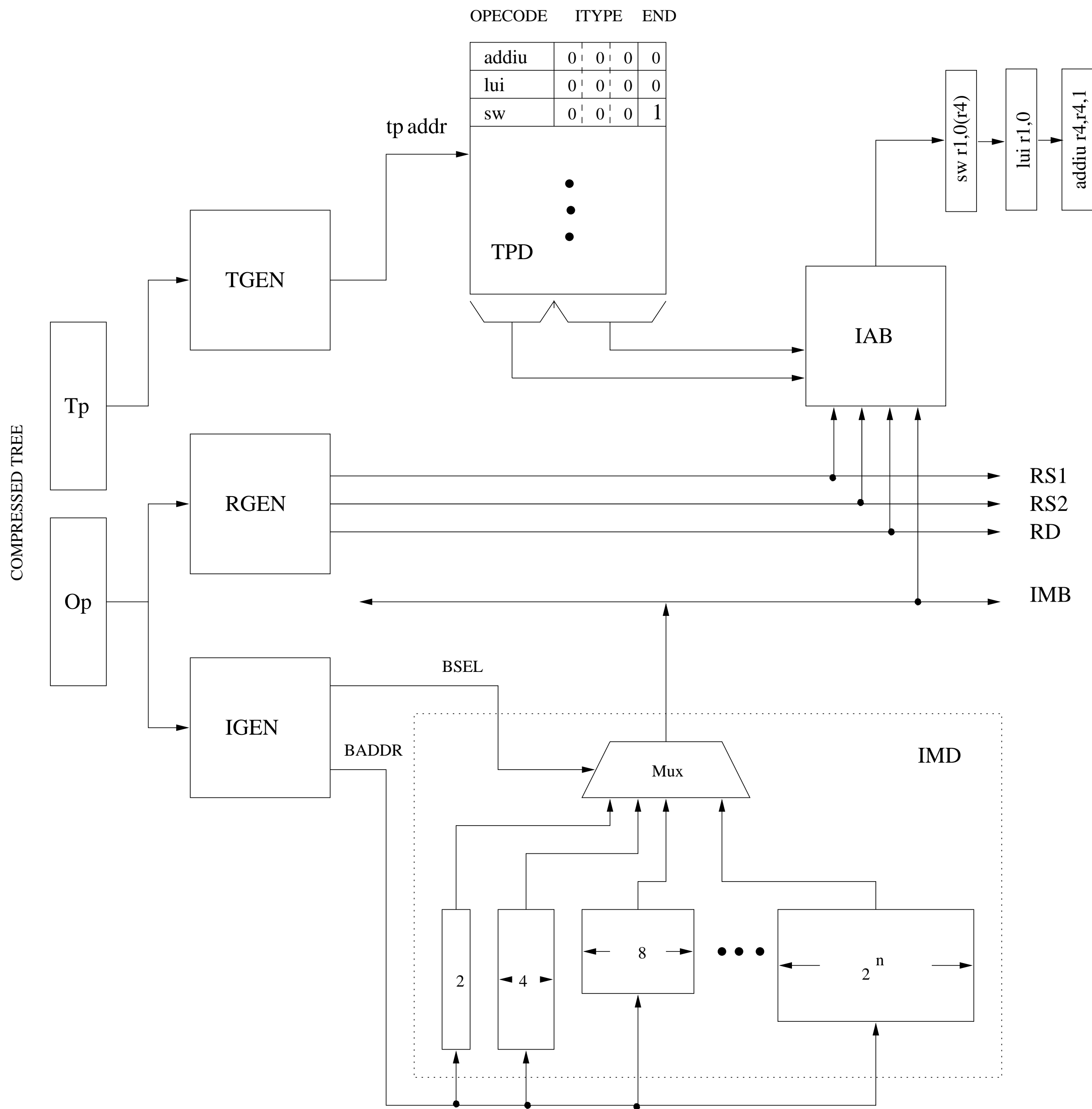

Figure 7: The Decompression Engine.

5.2 Register Generation

The Register Generator `RGEN` is a state machine that decodes the *Op* field of the incoming compressed word into the 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`. The first two buses (`RS1` and `RS2`) carry the register fields associated to the instruction source registers. Bus `RD` outputs the bits of the instruction destination register. The number of states of `RGEN` is limited by the number of instructions in the largest tree-pattern of the program. Because the size of the majority of expression trees is fairly small, we expect that `RGEN` will not have many states. For those cases when the size of a tree-pattern is smaller than the size of 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 logic in `RGEN`. 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 care"). Observe that patterns which share registers can also be used to minimize `RGEN` logic. For example, pattern `[r2, r0, r5]` results in similar values for register buses `RD` and `RS2` as pattern `[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-patterns. This way of encoding operand patterns naturally translates into a minimization problem for the combinational logic of `RGEN`. Hence, the final size of `RGEN` can directly benefit from the logic minimization tools available today.

5.3 Immediate Generation

The Immediate Dictionary (`IMD`) in Figure 7 stores the immediates used by the program. A single entry in the `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, constant 4 has different meanings for instructions `bgez r5, 4`, `lw r6, 4(r29)`, and `srl r5, r3, 4`. The fact that 4 is an address offset, a stack offset or a shift amount should have no relevance to

Program	TPD Ratio (a)	IMD Ratio (b)	Immediate Compression (c)
go	1.0	1.0	17.5
li	0.6	2.7	32.9
compress	6.0	5.0	32.9
perl	0.9	2.1	28.4
gcc	0.6	1.2	21.6
vortex	0.4	1.2	30.8
ijpeg	2.3	2.0	22.5

Table 2: (a) Percentage of `TPD` size with respect to the uncompressed program; (b) Percentage of `IMD` size with respect to the uncompressed program; (c) Immediate compression ratio.

how it is stored in `IMD`. We use the size variation of the immediates to minimize the number of bits required to store them in the `IMD`. An evaluation of the number of bits required to encode immediates reveals that, in average, more than 70% of the immediates in a program can be encoded into less than 16 bits[5]. 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`. The best number of memory banks depends on how immediates are distributed in the program, and on the area overhead due to the introduction of a new bank. This approach considerably improves the compression ratio for immediates. The immediate compression ratio is computed dividing the number of bits stored in `IMD` by the number of bits in the immediate fields of the instructions in the uncompressed program. For the studied programs it results in 26.7%. The overhead of the `IMD`, with respect to the uncompressed programs, is shown in Table 2. In average it corresponds to only 2.2% of the original code.

5.4 Branch Target Address

We borrow here the ideas developed in [6] and [1]. As mentioned before, our compression method can also be used for cache-line based compression. In fact, depending on the final performance of the decompression engine, a cache-line based approach may be very appropri-

[5] Similar numbers are presented in [16].

ate in our final design. In this case, the LAT/CLB address mapping technique in [6] should be used to compute branch addresses. If that is not the case, the address computation method proposed in [1] will be adopted.

6 Experimental Results

Three sets of experiments have been designed to determine the best encoding for tree and operand patterns. The objective of the first set was to determine the compression ratio using fixed-length and Huffman encoding. The goal of the second experiment set was to determine the best splitting between Huffman and fixed-length encoding when Bounded-Huffman is used. 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) and 50% (0%) encoded using fixed-length. The results of the experiments are plotted in the graphs of Figures 8–9. The horizontal axis of the graphs show the percentage of the patterns which have been encoded using Huffman. The vertical axis is the share of the program compression ratio due to the pattern. On the horizontal axis, 0% corresponds to an encoding using only fixed-length codewords. The points in between 0% and 50% use a mix of fixed-length and Huffman codewords, with an escape bit to distinguish between them. A similar set of experiments was also performed, this time using VLC encoding. The goal was to determine the impact, on the compression ratio, due to reducing the encoding efficiency so that codewords can carry size information. The graphs corresponding to these experiments are shown in Figures 10–11.

6.1 Analysis

From Figures 8–9 it is possible to note 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

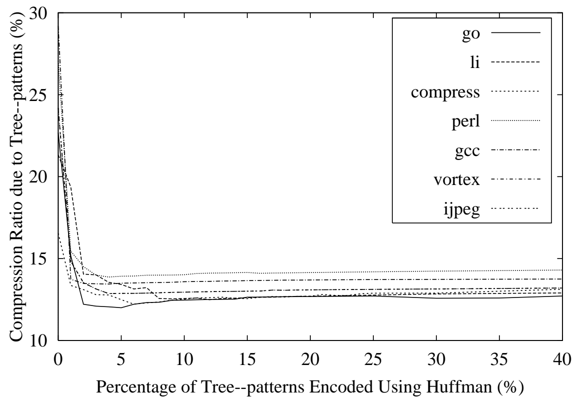

Figure 8: Compression ratio due to encoding tree-patterns using Bounded-Huffman.

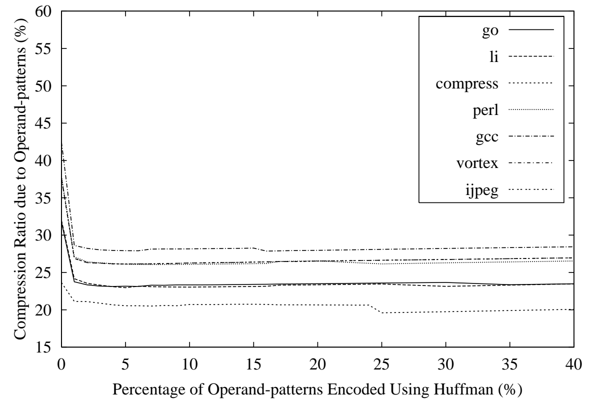

Figure 9: Compression ratio due to encoding operand-patterns using Bounded-Huffman.

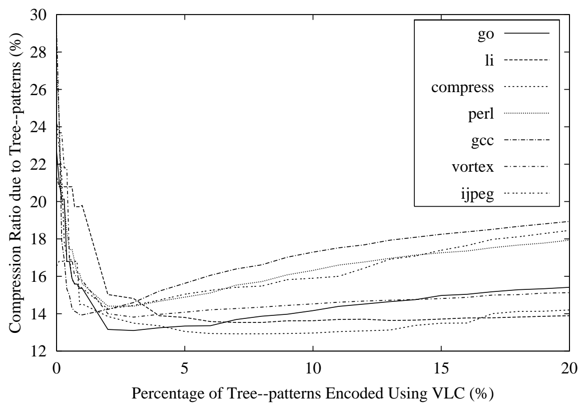

Figure 10: Compression ratio due to encoding tree-patterns using VLC.

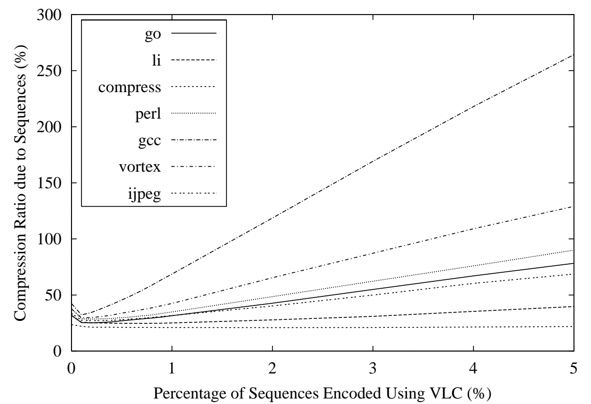

Figure 11: Compression ratio due to encoding operand-patterns using VLC.

3. Patterns which have small contributions do not change much the program entropy[6]. Lower entropy results in small redundancy what causes Huffman compression to approach fixed-length compression [4]. By switching from Huffman to fixed-length encoding, at a point when the pattern distribution becomes more uniform, one can minimize the impact of not using Huffman. Consider, for example, program go in Figure 8. The contribution to the compression ratio when 5% of the tree-patterns are encoded using Bounded-Huffman is 12.0%. This is only 1.4% higher than if all tree-patterns were optimally encoded using Huffman[7]. A similar number can also be determined for operand-patterns (Figure 9). The contribution to the go compression ratio due to encoding 4% of the operand-patterns using Bounded-Huffman is 23.1%, very close to the compression ratio achieved only using Huffman (22.6%). The graphs in Figures 10–11 show the compression ratio when VLC codewords are used. In this case, the global minima occur when a smaller percentage of patterns are encoded using VLC. For example, the minimun for tree (operand) patterns occur around 2.5% (0.2%) of the patterns. This can be explained because VLC codewords are larger than Huffman codewords (they have to encode size information). This overhead becomes more evident as more codewords are encoded using VLC, resulting in a rapid increase of the compression ratio. 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 for

		Operand-patterns			
	Encoding Methods	Fixed-Length	Huffman	Bounded-Huffman	VLC
Tree-Patterns	Fixed-length	57.7	46.2	48.1	50.5
	Huffman	45.4	35.0	35.8	38.2
	Bounded-Huffman	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 and operand patterns.

[6]Entropy is a measure of the amount of redundant information in a set of symbols [4]

[7]Not shown in in the graphs.

all programs, when the encoding methods for tree and operand patterns are combined. The Bounded-Huffman and VLC compression ratios were determined from the graphs of Figures 8–11 by taking the average of the global minima of all programs. Table 3 provides a solution space for chosing an encoding method. Based on it, one can trade compression efficiency for decompression speed. The worst compression ratio (57%) is obtained when tree and operand patterns are encoded using fixed-length codewords. As expected, the best compresion 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 both patterns using VLC results in a 41% compression ratio. This ratio is only 6% higher than optimal Huffman encoding. This is the price that has to be paid in order to minimize the latency of the length detection logic in the decompression engine.

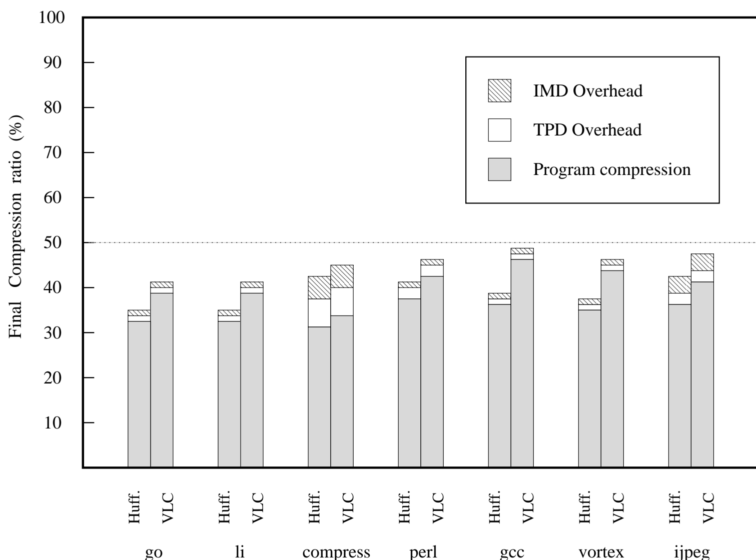

Figure 12: Final compression ratio using Huffman and VLC.

The compression ratios in Table 3 are not a fair assessment of the compression efficiency of the operand factorization technique. To do that, the silicon area of the decompression engine has been taken into consideration. The size of the immediate and tree-pattern dictionaries can be estimated based on the number of bits they use. Figure 12 shows the final average compression ratio for Huffman (39%) and VLC (44%) encoding once the overhead due to the dictionaries is considered. As shown, the overhead is very small (in average 2%). An accurate estimate of the size and performance of modules `TGEN`, `RGEN` and `IGEN` can be done

only through implementation. TGEN is the decoder of a small dictionary (i.e. TPD). Thus, it should account for a small share of the total silicon area. On the other hand, RGEN and IGEN encode a large number of operand-patterns and should contribute more for the engine size.

7 Conclusions and Future Work

This paper proposes a code compression technique called operand factorization. The average compression ratio using this technique and Huffman (VLC) encoding is 35% (41%). A decompression engine is proposed to assemble tree and operand patterns into uncompressed instructions. This work can be improved in a number of ways. First, operand-patterns encode the same temporary register twice, when the register is used as the destination of an instruction, and when it is a source in the instruction that reads it. If the compiler schedules expression trees in a pre-order schedule[8], then it is possible to eliminate the second reference to the temporary, thus minimizing the size of the operand-pattern. In order to do that, 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. For example, it is very common in a program to find the `lw` opcode occuring with register `r29` (stack-pointer). Thus, this pattern can be encoded separately yielding a lower compression ratio.

References

[1] 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*, pages 194–203, December 1997.

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

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

[8]This is very common in code generation.

[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*, IT–22(1):75–81, January 1976.

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

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

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

[9] 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.

[10] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator. *Journal of the ACM*, 22(12):248–262, March 1993.

[11] Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proceedings of 16th Conference on Advanced Research in VLSI*, pages 272–285, Los Alamitos, CA, 1995. IEEE Society Press.

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

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

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

[15] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video: an Introduction to MPEG–2*. Chapman & Hall.

[16] J.L. Hennessy and D.A. Patterson. *Computer Architectures: A Quantitative Approach*. Morgan Kaufmann, 1996.