Using Operand Factorization to Compress DSP Programs

Ricardo Pannain *Paulo Centoducatte*
Guido Araújo

Relatório Técnico IC–99-12

Abril de 1999

Using Operand Factorization to Compress DSP Programs

Ricardo Pannain Paulo Centoducatte Guido Araújo

Abstract

We propose a method for compressing programs running on embedded DSPs. Program expression trees are decomposed into opcode and operand sequences called patterns. We show that DSP program patterns have exponential frequency distributions. Based on that, we encode patterns using a mix of variable-length and fixed-length codewords. A decompression engine is proposed, which converts patterns into uncompressed instruction sequences. The experimental results reveal an average compression ratio of 67% for typical DSP programs running on the TMS320C25 processor. This ratio includes an estimate of the size of the decompression engine.

1 Introduction

Embedded systems are computer systems designed to specific application domains. Because they are aimed at mass production, minimizing the final cost of such systems is a major design goal. As a consequence, embedded system designs are typically constrained by stringent area, power, and performance budgets. In order to reduce the total system cost, designers are integrating memories, microprocessor cores and ASIC modules into a single chip, a methodology known as *System-On-a-Chip* (SOC). Microprocessor cores are selected based on the target application. In areas that require intense arithmetic processing, as in telecommunications, *Digital Signal Processors* (DSPs) have been the processor of choice.

A considerable part of the design effort of an embedded system is devoted to programming the embedded program. Due to performance constraints, this task is predominantly done in assembly. Programming and debugging embedded code is a hard and time-consuming task. With the increase in the size of applications, assembly programming has become unpractical and error-prone. Compilers like SPAM [Sud98], RECORD [Leu97], CodeSyn [PLMS94] and CHESS [LPK+95] have achieved some success in generating quality code from high-level language programs. Unfortunately, compilers can reduce the program size only to some extent. On the other hand, embedded programs are growing considerably large, to the point where the size of the program memory has become the largest share of the final die area (cost). A way to reduce program size is to compress its instructions, using a decompression engine to generate the original code during instruction fetch. This paper proposes a compression algorithm and a decompression engine targeted to DSPs. The experimental work reveals a 67% average compression ratio for a set of typical embedded programs running on the TMS320C25 processor [Tex90].

This paper is divided as follows. Section 2 describes related work in the area of code compression. Section 3 details our basic compression algorithm. The decompression engine

is described in Section 4 and the experimental results are analyzed in Section 5. In Section 6 we wrap up the work and discuss future research.

2 Previous Work

The problem of file compression has been extensively studied [BCW90]. Almost all practical dictionary based compression tools of today are based on the work of Lempel and Zvi (LZ) and its variations [BCW90]. Unfortunately, algorithms derived from LZ are not suitable for real-time code decompression. In LZ, codewords are decompressed sequentially using as dictionary the string formed by all the symbols already decompressed. This is a major drawback if the codeword encodes a forward branch instruction. In the rest of this section we describe only those compression techniques that are suitable to efficient real-time code decompression.

Wolfe and Channin [WC92] proposed the *Compressed Code RISC Processor* (CCRP). Programs are compressed one cache-line at a time using Huffman codewords and byte-long symbols. During a cache miss, compressed cache-lines are fetched from main memory, uncompressed, and stored into the cache. Instructions in the cache and main memory have different addresses. The CCRP uses a main memory table, *Line Address Table* (LAT), to map (compressed) main memory addresses to (uncompressed) cache addresses. In order to reduce the number of accesses to the LAT, a *Cache Lookaside Buffer* (CLB) is provided to store the set of most recently used LAT entries. The advantage of the CCRP approach is that the latency of the decompression engine is amortized across many cache hits. On the other hand, the use of a cache makes it very difficult to estimate the execution time of the embedded program. This is particularly important for embedded systems running time critical applications. Moreover, embedded programs are usually stored into fast on-chip memories. The average compression ratio achieved by CCRP in a MIPS architecture was 73%. This compression ratio does not consider the size of the decompression engine.

Wolf and Lekatsas [LW98] studied two different methods for code compression. The best compression ratio is achieved by their 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 in a MIPS architecture was 51%. Similarly as in the CCRP case, the decompression engine size was not taken into account.

Lefurgy et al [LBCM97] describes a compression technique based on dictionary. Common sequences of instructions are assigned to a codeword. A dictionary in the decompression engine stores the sequence of instructions at the address given by the codeword. The decompression is performed by retrieving the sequence of instructions from the dictionary. Because instructions are compressed, the target address of jump and branch instructions must be recomputed. In order to deal with that, Lefurgy et al divide the target address bits into two parts. The first part stores the address of the word where the compressed target is located. The second part corresponds to the target offset inside the word. The average compression ratio using this technique for the PowerPC, ARM and i386 processors

were 61%, 66% and 74%.

Liao et al [LDK98] were the first to study the code compression problem for a DSP processor. Similarly as in [LBCM97], compressed instructions are stored into a dictionary. Liao's idea is to substitute similar instruction sequences by sub-routine calls. Instructions are represented by boolean variables, and instruction sequences are encoded as minterms. Hence, the problem of compressing the program can be formulated as a set-covering problem. Instruction sequences are then converted into call instructions to sub-routines in the dictionary. An interesting mechanism based on a stack is used to minimize the penalty of the sub-routine return instruction. The average compression ratio achieved by this technique in the TMS320C25 processor was 82%. To the best of our knowledge, apart from the work of Liao et al [LDK98], no other research has addressed the problem of executing compressed DSP code.

In [ACCP98] we proposed a code compression technique for the MIPS architecture. Program expression trees are decomposed into sequences of opcode and operand patterns, a method called *operand factorization*. The goal of this paper is to verify the effectiveness of the operand factorization concept in highly encoded instruction sets, like those found in DSP architectures. We also propose a decompression architecture for these processors.

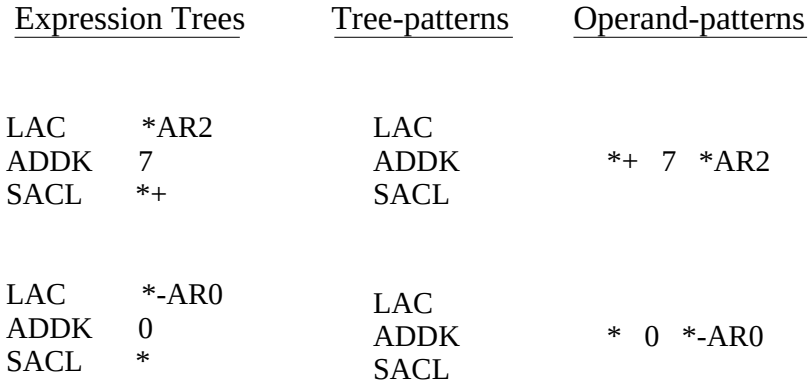

Figure 1: Factored expression trees.

3 The Compression Algorithm

Our compression algorithm is based on the patternization concept [Pro95]. Patternization (we call it *factorization*) has been used by Fraser et al [FHRP93] to compress byte-code for network virtual machines. The basic operation in factorization is the removal of operands from an instruction. First, instruction sequences are selected to correspond to program expression trees. An instruction is the root of an expression tree if: (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. Second, expression tree instructions are decomposed into a list of instruction opcodes and a list of operands. We

use expression trees as the basis for compression because compilers tend to generate similar expression trees during the translation of source statements, like *if-then-else* and *for*. We call each distinct list of opcodes (operands) a *tree-pattern* (*operand-pattern*). Figure 1 shows two expression trees from program *rx* being factored into its tree and operand patterns. Notice that both trees have the same tree-pattern.

The TMS320C25 has 16 bit long instructions that are encoded using 16 different formats. Instruction formats use opcodes of different lengths and three addressing modes (direct, indirect and immediate). In the direct addressing mode the operand address is encoded into an instruction field. In the indirect addressing mode the operand format is `<ind,next>`, where `ind` is a side-effect operation with the current address register `AR`, e.g. auto-increment (`*+`), and `next` is the next current address register. In the immediate mode the value of the operand is encoded into the instruction field.

In order to measure the contribution of tree and operand patterns to the program size we use a set of example programs. These programs are representative of the type of applications running on embedded processors and DSPs. Program *jpeg* is an implementation of the JPEG image compression algorithm, *bench* is a disk cache controller, *gzip* is a compression algorithm and *set* is a set of bit manipulation routines from a DSP application. Programs *hill*, *gnucrypt* are data encryption programs, and *rx* is an embedded state machine controlling routine. We first generate *optimized* code for the example programs using TI's TMS320C25 compiler with optimization flag -O2. Second, programs are analyzed such as to identify the set of tree and operand patterns that cover all program trees. The resulting number of tree and operands patterns for each program is shown in Table 1. On average tree (operand) patterns correspond to only 8.3% (24.1%) of all opcode (operand) sequences in the programs. The lists of tree and operand patterns are then ordered based on their contribution in bits to the program. The cumulative contribution of the patterns to the size of the programs is then computed. The results are shown in Figure 2(a)-(b). Figure 2(a) (Figure 2(b)) shows the percentage of the program bits that are covered by tree (operand) patterns. In the horizontal axis of each graph, patterns are ordered based on their contribution to the program size.

Program	Expression Trees	Tree-Patterns (%)	Operand-Patterns (%)
aipint2	1043	90 (8.6)	285 (27.3)
bench	9483	572 (6.0)	2263 (23.9)
gnucrypt	3682	263 (7.1)	778 (21.1)
gzip	10835	582 (5.4)	2354 (21.7)
hill	920	121 (13.2)	279 (30.3)
jpeg	2305	190 (8.2)	563 (24.4)
rx	563	61 (10.8)	114 (20.3)
set	4565	319 (7.0)	1084 (23.7)

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

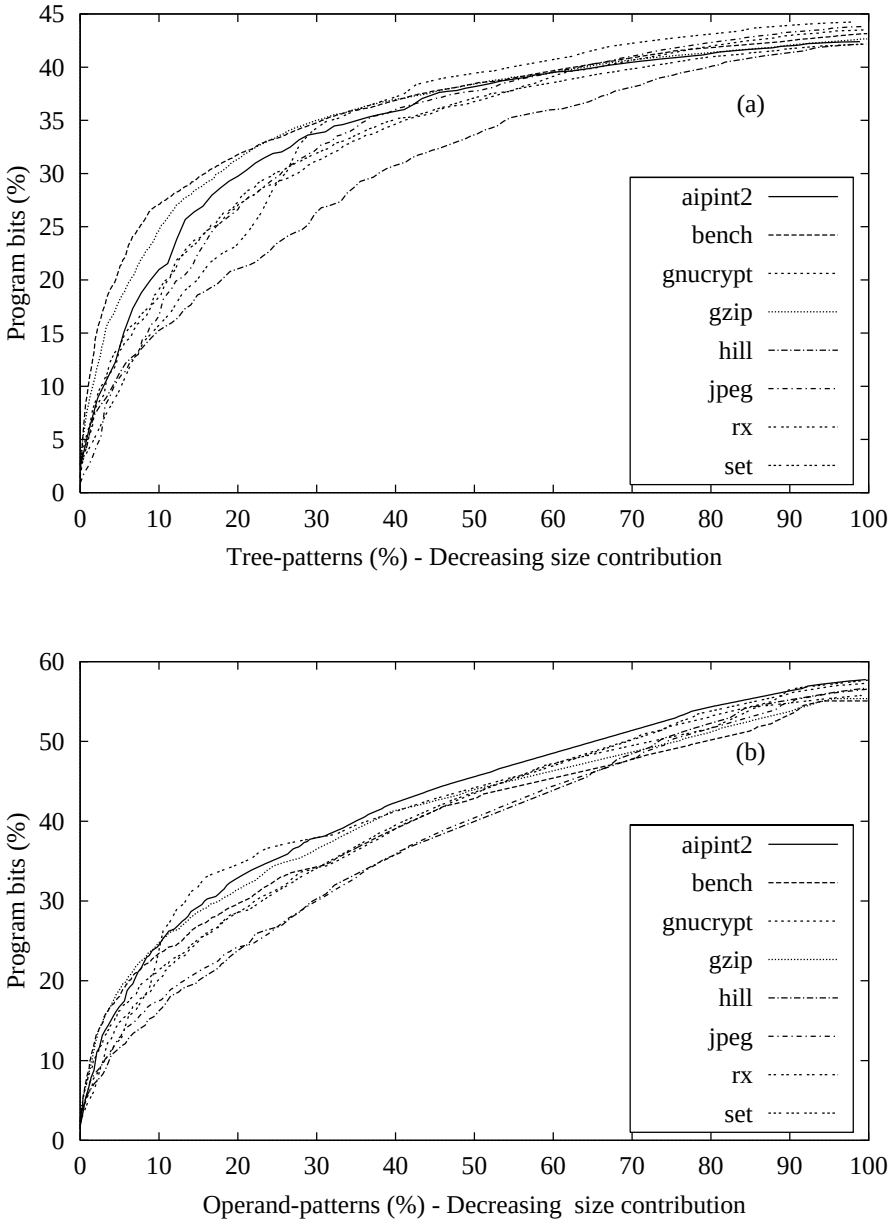

Figure 2: Percentage of program bits due to: (a) tree-patterns; (b) operand-patterns.

Notice that the contribution of tree and operand patterns in DSP programs is non-uniform. Actually, tree and operand patterns have exponential frequency distributions. In other words, a small set of patterns covers a large number of trees in the program. Almost 20% of the program bits are covered by only 10% of the tree-patterns. A similar behavior was also observed for operand-patterns. In this case, 10% of the most frequent operand-patterns account for 19% of the program bits.

3.1 Expression Tree Compression

The experimental results in the previous section show that the contribution of tree and operand patterns to the program size is non-uniform. This suggests that patterns should be encoded using variable-length codewords [BCW90]. Patterns which have large (small) contributions are encoded using small (large) codewords. The drawback of this approach is that it can result in very large codewords for those patterns which do not contribute much. Decoding very large patterns is a difficult task for the decompression engine, given that a codeword can spill across many memory words. In order to avoid that, we use a mixed encoding method. Patterns are encoded using a mix of variable-length and fixed-length (lossy) encoding. An escape bit is appended to the beginning of the codeword to indicate the encoding method (see Figure 3). We used two variable length encoding

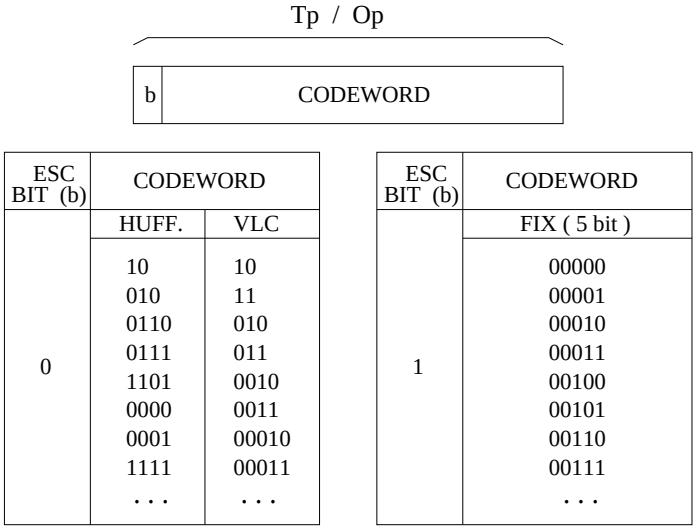

Figure 3: Pattern Encoding

methods: Huffman and an adaptation of the VLC encoding method adopted in MPEG-2 [HPN91]. The main motivation to use VLC is that, unlike Huffman, it enables codewords to carry size information. In VLC codewords the number of leading zeroes encodes the size of the codeword. This makes it easier to design the decompression engine logic that extracts codewords from a memory word. A compressed expression tree is constructed putting together the codewords for the tree and operand patterns that form it. The compressed

program is then assembled into a sequence: $Tp_1Op_1|Tp_2Op_2|\ldots|T_nOp_n$, where Tp_i (Op_i) is a tree (operand) pattern codeword.

4 The Decompression Engine

Figure 4 shows the decompression engine for our compression algorithm. It consists of a set of state machines and dictionaries that generate the various fields of the decompressed instruction. An extraction logic extracts tree-pattern `Tp` from a memory word and decodes it using state machine `OPGEN`. `OPGEN` generates signals `OPSEL` and `OPADDR` to select the appropriate opcode bank. Opcodes that have similar lengths are stored into the same bank. Only opcodes of the instructions in the program are stored. After operand-pattern `Op` is extracted, machines `ARGEN`, `INGEN` and `IMGEN` decode the instruction operands. `ARGEN` selects the field of the `AR` register being used by the instruction and `INGEN` generates the 7 bits that encode instruction fields `<ind,next>`. Instruction immediates are generated using the `Immediate Dictionary` (`IMD`). `IMD` also stores the compressed and uncompressed forms of the branch target addresses. Immediates are stored into banks such that each bank contains immediates of similar sizes. An extra bit is used to enable automatic sign extension when immediates are retrieved from `IMD`. State machine `IMGEN` generates the signals (`BADDR` and `BSEL`) required to address the immediate banks. The various fields of the instructions merge into the *Instruction Assembly Buffer* (`IAB`), where they are assembled and shipped to the CPU.

Machines `OPGEN`, `ARGEN`, `INGEN` and `IMGEN` work in parallel. In each cycle they select the appropriate instruction fields that are then assembled by `IAB` into a decompressed instruction. The process continues until all the instructions of the current expression tree are decompressed (i.e. `OPEND = 1`). The number of state variables is bounded by the size of the largest tree. When smaller trees are decompressed the non-used state variables become don't cares.

4.1 Branch Addresses

During a branch instruction (see Figure 5) the following is retrieved from `IMD`: the target address of the original uncompressed tree (16 bits) and the address of the compressed tree (20 bits). Because variable length codewords are used to encode patterns, the address of a compressed tree can start at any position inside a memory word. Hence, the engine must keep track of: (a) the memory address of the word where the current compressed tree is (16 bits) and (b) the offset, with respect to the beginning of that word, where the tree starts (4 bits). The computation of the compressed tree location is performed by the `EXTRACTION` module. The *Memory Data Register* (`MDR`) is used to store the memory word that contains the current compressed tree (remember that one memory word can have many compressed trees). The *Memory Address Register* (`MAR`) stores the address of the next word in memory.

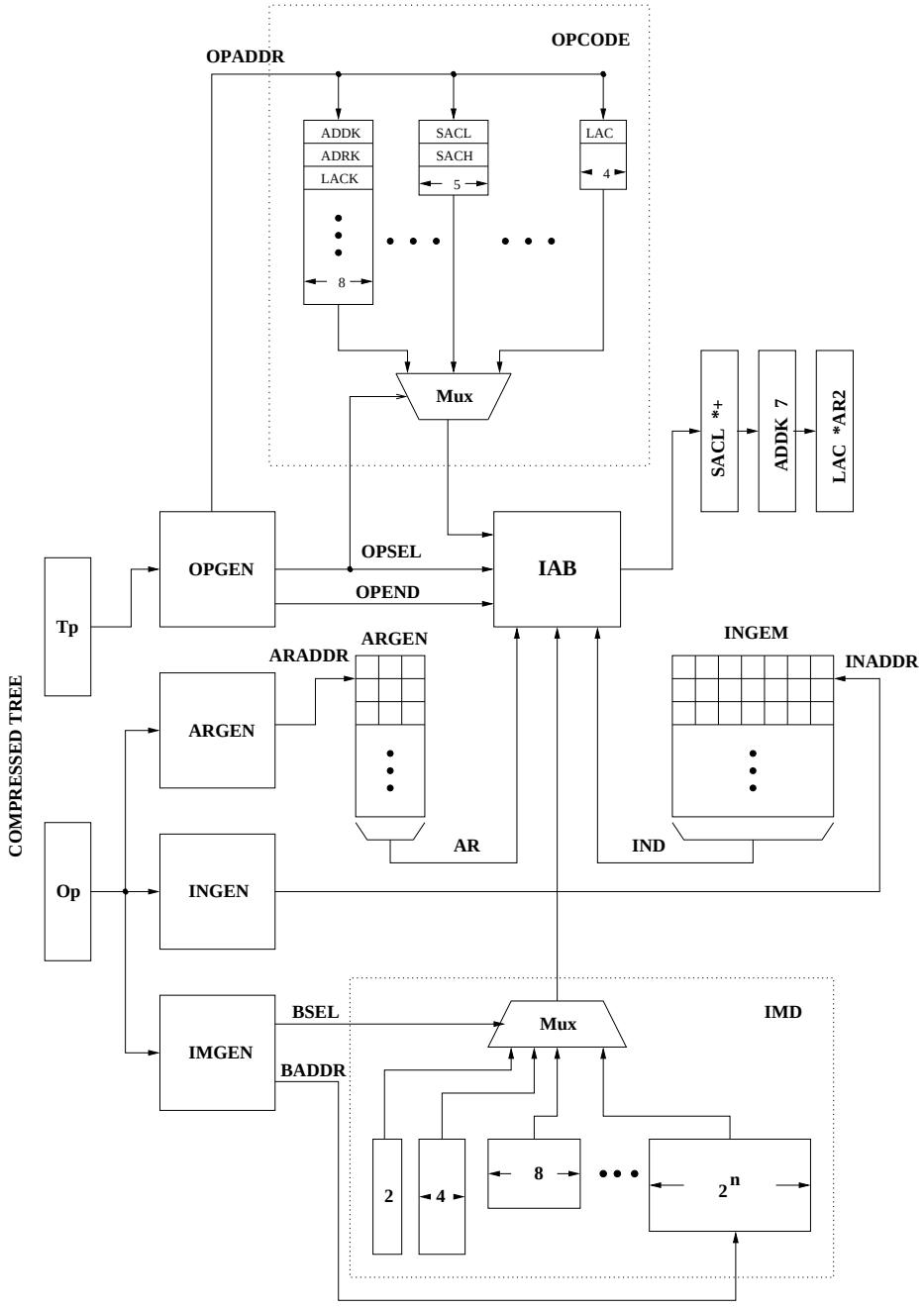

Figure 4: The Decompression Engine.

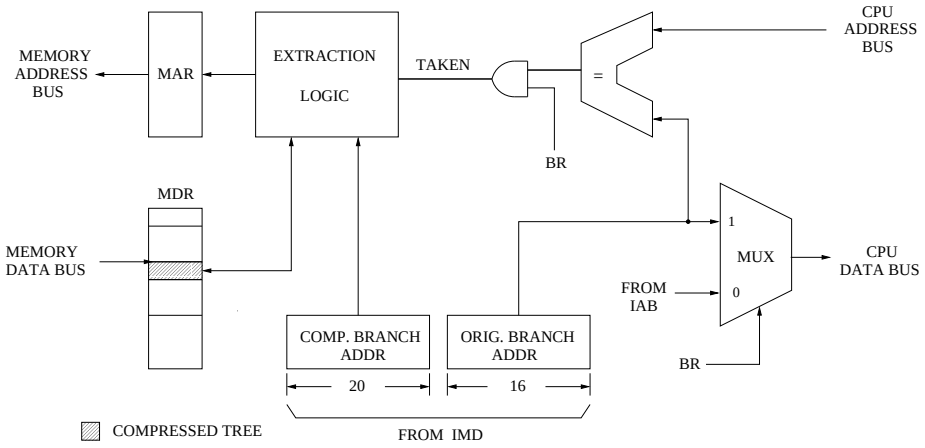

Figure 5: The Branch Address Generator.

The location of the next compressed tree depends on the value of `TAKEN` (see Figure 5). `TAKEN = 1` if the last decompressed instruction passed to the CPU was a branch and the branch was taken. Bit `BR` from `OPGEN` is active during the decompression of a branch instruction. `TAKEN = 0` if the last instruction was not a branch or if it was a branch and the branch was not taken. Signal `TAKEN` is determined by monitoring the progression of the CPU address bus. If during the next (program) memory read cycle the content of the CPU address bus equals the last branch target passed to the CPU, `TAKEN = 0` and the next compressed tree is in `MDR`, or in the first tree of the next memory word. In the first case, the `EXTRACTION` logic removes the tree from `MDR` and passes it to the decoding logic (Figure 4). In the second case, `MAR` is incremented, the next memory word is fetched and its first tree is extracted and decoded. This approach allows the CPU to see the same addresses as those in the original uncompressed program, while the access to memory is performed using compressed addresses. Unlike the approach in [LBCM97] ours do not require a modification of the processor address generation unit.

5 Experimental Results

We use the compression algorithm described in Section 3 to compress the programs in our program set. Tree and operand patterns are encoded separately using the encoding methods discussed in Section 3.1. Figure 6 and Figure 7 shows the compression ratio resulting when patterns are encoded using Huffman (Figure 6(a)-(b)) and VLC (Figure 7(a)-(b)) codewords. In order to limit the size of the codewords, we encode only part of the patterns using a variable-length code. The horizontal axes of Figure 6(a)-(b) and Figure 7(a)-(b) correspond to the number of patterns that are encoded using variable-length codewords. Horizontal axis value 0% (100%) is an encoding where all patterns are encoded using fixed-length (Huffman/VLC) codewords and the rest is encoded using Huffman/VLC (fixed-length). Since pattern distribution is close to exponential (see Figure 2), the more

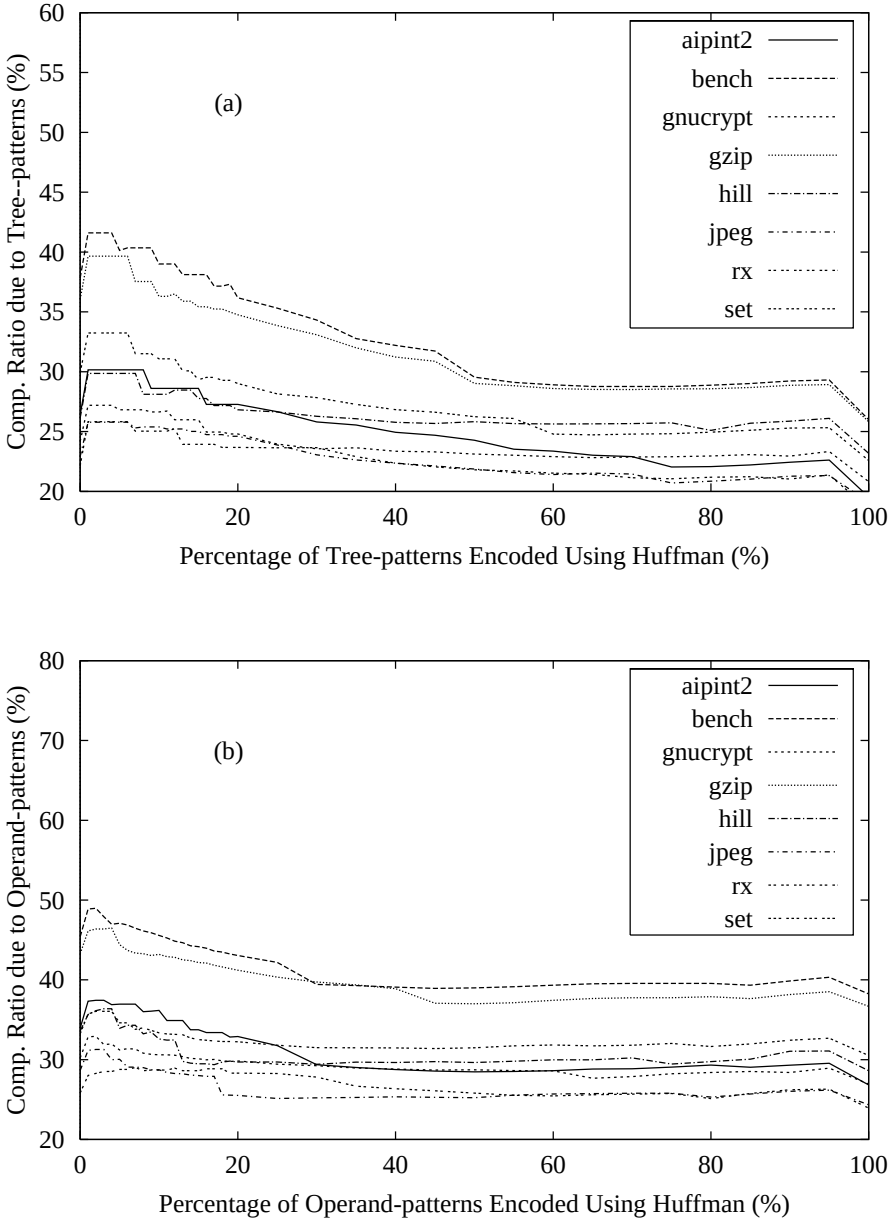

Figure 6: Compression ratios for: (a) tree-patterns (Huffman); (b) operand-patterns (Huffman)

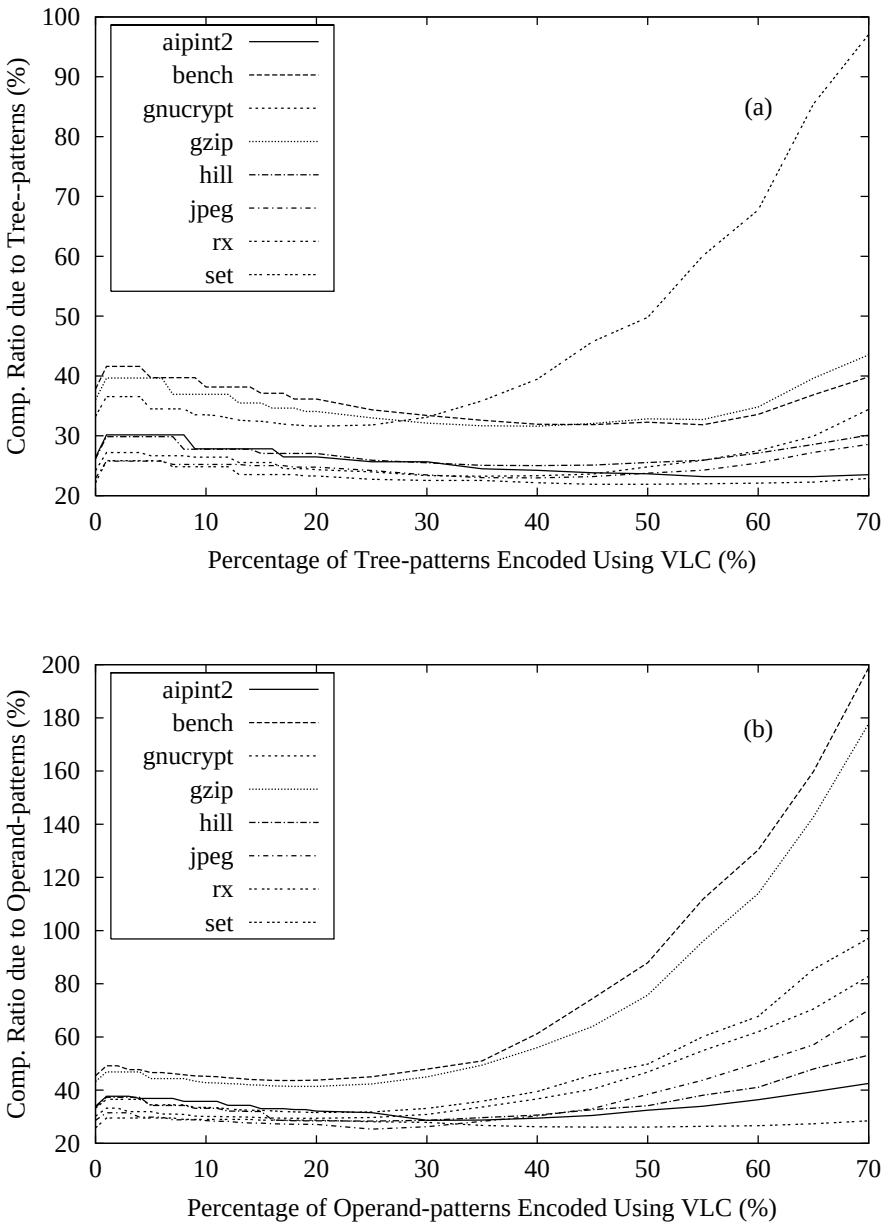

Figure 7: Compression ratios for: (a) tree-patterns (VLC); (b) operand-patterns (VLC).

small contribution patterns are encoded using variable-length codewords, the more the size of the variable-length codeword approaches that of a fixed-length codeword [BCW90]. This is reflected by the saturation behavior of the compression ratio in Figure 6(a)-(b). Thus, switching from variable-length to fixed-length encoding does not have a large impact on the final compression ratio. In Figure 7(a)-(b) the compression ratio can grow larger than 100% when VLC encoding is used. This happens because of the presence of leading zeroes in the VLC codewords, which carry only size information. The best split between variable-length

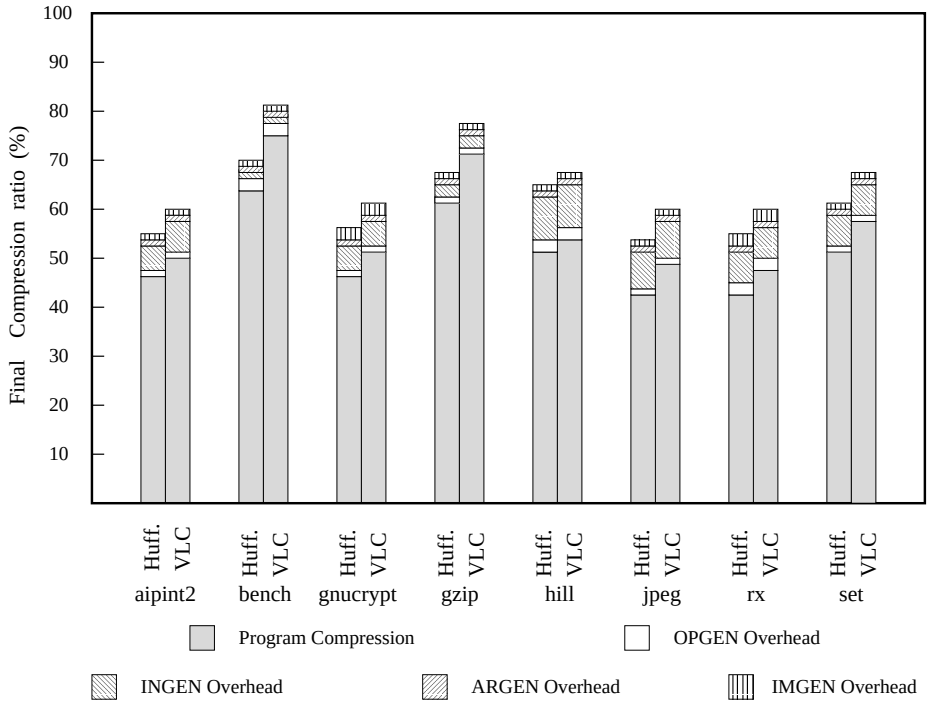

Figure 8: Final compression ratio considering overheads.

and fixed-length encoding in a program corresponds to the minimum in the curve of that program for the selected pattern. For example, when Huffman is used to encode gzip's operand-patterns, the best compression ratio (37%) is achieved when approximately 45% of the operand-patterns are encoded using Huffman. Figure 8 shows the final compression ratio for the programs in our program set, once the size of the engine is taken into consideration. The average compression ratios achieved using Huffman (VLC) was 60% (67%). The program compression ratio was determined from the minima of the graphs in Figure 6(a)-(b) and Figure 7(a)-(b). Figure 8 takes into consideration an estimate of the engine dictionaries overhead with respect to the uncompressed program. A precise estimate of the overhead will only be possible through synthesis of the state machines. We do not expect that these machines will result in a large overhead though. The reason is that many distinct patterns have similar parts which will eventually translate into shared logic for the state machines. For example, state machine `ARGEN` will produce the same dictionary address for

distinct operand-patterns `AR3,*+` and `AR3,*-,AR4`, since both use the same address register.

6 Conclusions

This paper proposes a code compression technique for DSP programs. This approach is based on decomposing the expression trees into tree and operands patterns. We show that patterns have exponential distributions. We also propose a decompression engine that assembles opcode and operand sequences into uncompressed instructions.

7 Acknowledgements

The authors are grateful to Stan Liao for providing the test programs. We also thank the anonymous referees for their comments. This work was supported by a CNPq-ProTeM-CC/NSF collaborative research grant and by CNPq Research Fellowship 300156/97-9.

References

[ACCP98] 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.

[BCW90] Timothy C. Bell, Jhon G. Cleary, and Ian H. Witten. *Text Compression.* Advanced Reference Series. Prentice Hall, New Jersey, 1990.

[FHRP93] C. W. Fraser, Hanson, D. R., and T. A. Proebsting. Engineering a simple, efficient code generator. *Journal of the ACM*, 22(12):248–262, March 1993.

[HPN91] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video: an Introduction to MPEG–2.* Chapman & Hall, 1991.

[LBCM97] 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.

[LDK98] 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.

[Leu97] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, June 1997.

[LPK+95] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embed-*

ded Processors, chapter 5, pages 85–102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[LW98] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. of 35th ACM Design Automation Conference*, 1998.

[PLMS94] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. CodeSyn: A Retargetable Code Synthesis System. In *Proceedings of the 7th International High-Level Synthesis Workshop*, Spring 1994.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *ACM Conference on Principles of Programming Languages*, pages 322–332, January 1995.

[Sud98] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* PhD thesis, Princeton University, May 1998.

[Tex90] Texas Instruments. *TMS320C2x User's Guide*, 1990.

[WC92] 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*, pages 81–91, December 1992.