Improving Code Density Using Compression Techniques

CSE-TR-342-97

Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge

EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
{lefurgy,icheng,pbird,tnm}@eecs.umich.edu

Abstract

We propose a method for compressing programs in embedded processors where instruction memory size dominates cost. A post-compilation analyzer examines a program and replaces common sequences of instructions with a single instruction codeword. A microprocessor executes the compressed instruction sequences by fetching codewords from the instruction memory, expanding them back to the original sequence of instructions in the decode stage, and issuing them to the execution stages. We apply our technique to the PowerPC instruction set and achieve 30% to 50% reduction in size for SPEC CINT95 programs.

Keywords: Compression, Code Density, Code Space Optimization, Embedded Systems

1 Introduction

According to a recent prediction by In-Stat Inc., the merchant processor market is set to exceed $60 billion by 1999, and nearly half of that will be for embedded processors. However, by unit count, embedded processors will exceed the number of general purpose microprocessors by a factor of 20. Compared to general purpose microprocessors, processors for embedded applications have been much less studied. The figures above suggest that they deserve more attention. Embedded processors are more highly constrained by cost, power, and size than general purpose microprocessors. For control oriented embedded applications, the most common type, a significant portion of the final circuitry is used for instruction memory. Since the cost of an integrated circuit is strongly related to die size, and memory size is proportional to die size, developers want their program to fit in the smallest memory possible. An additional pressure on program memory is the relatively recent adoption of high-level languages for embedded systems because of the need to control development costs. As typical code sizes have grown, these costs have ballooned at rates comparable to those seen in the desktop world. Thus, the ability to compress instruction code is important, even at the cost of execution speed.

High performance systems are also impacted by program size due to the delays incurred by instruction cache misses. A study at Digital [Perl96] showed that an SQL server on a DEC 21064 Alpha, is bandwidth limited by a factor of two on instruction cache misses alone. This problem will only increase as the gap between processor performance and memory performance grows. Reducing program size is one way to reduce instruction cache misses and achieve higher performance [Chen97b].

This paper focuses on compression for embedded applications, where execution speed can be traded for compression. We borrow concepts from the field of text compression and apply them to the compression of instruction sequences. We propose modifications at the microarchitecture level to support compressed programs. A post-compilation analyzer examines a program and replaces common sequences of instructions with a single instruction codeword. A microprocessor executes the compressed instruction sequences by fetching codewords from the instruction memory, expanding them back to the original sequence of instructions in the decode stage, and issuing them to the execution stages. We demonstrate our technique by applying it to the PowerPC instruction set.

1.1 Code generation

Compilers generate code using a *Syntax Directed Translation Scheme* (SDTS) [Aho86]. Syntactic source code patterns are mapped onto templates of instructions which implement the appropriate semantics. Consider, a simple schema to translate a subset of integer arithmetic:

```
expr -> expr '+' expr
{
    emit( add, $1, $1, $3 );
    $$ = $1;
}

expr -> expr '*' expr
{
    emit( mult, $1, $1, $3 );
    $$ = $1;
}
```

These patterns show syntactic fragments on the right hand side of the two productions which are replaced (or reduced) by a simpler syntactic structure. Two expressions which are added (or multiplied) together result in a single, new expression. The register numbers holding the operand expressions ($1 and $3) are encoded into the add (multiplication) operation and emitted into the generated object code. The result register ($1) is passed up the parse tree for use in the parent operation. These two patterns are reused for all arithmetic operations throughout program compilation.

More complex actions (such as translation of control structures) generate more instructions, albeit still driven by the template structure of the SDTS.

In general, the only difference in instruction sequences for given source code fragments at different points in the object module are the register numbers in arithmetic instructions and operand offsets for *load* and *store* instructions. As a consequence, object modules are generated with many common sub-sequences of instructions. There is a high degree of redundancy in the encoding of the instructions in a program. In the programs we examined, only a small number of instructions had bit pattern encodings that were not repeated elsewhere in the same program. Indeed, we found that a small number of instruction encodings are highly reused in most programs.

To illustrate the redundancy of instruction encodings, we profiled the SPEC CINT95 benchmarks [SPEC95]. The benchmarks were compiled for PowerPC with GCC 2.7.2 using -O2 optimization. Figure 1 shows that compiled programs consist of many instructions that have identical encodings. On average, less than 20% of the instructions in the benchmarks have bit pattern encodings which are used exactly once in the program. In the *go* benchmark, for example, 1% of the most frequent instruction words account for 30% of the program size, and 10% of the most frequent instruction words account for 66% of the program size. It is clear that the redundancy of instruction encodings provides a great opportunity for reducing program size through compression techniques.

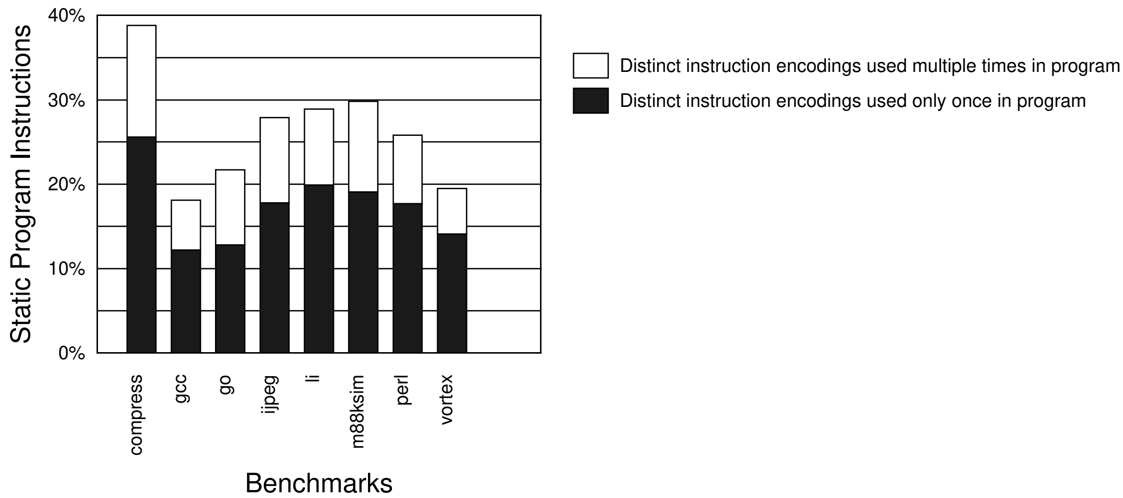

Figure 1: Distinct instruction encodings as a percentage of entire program

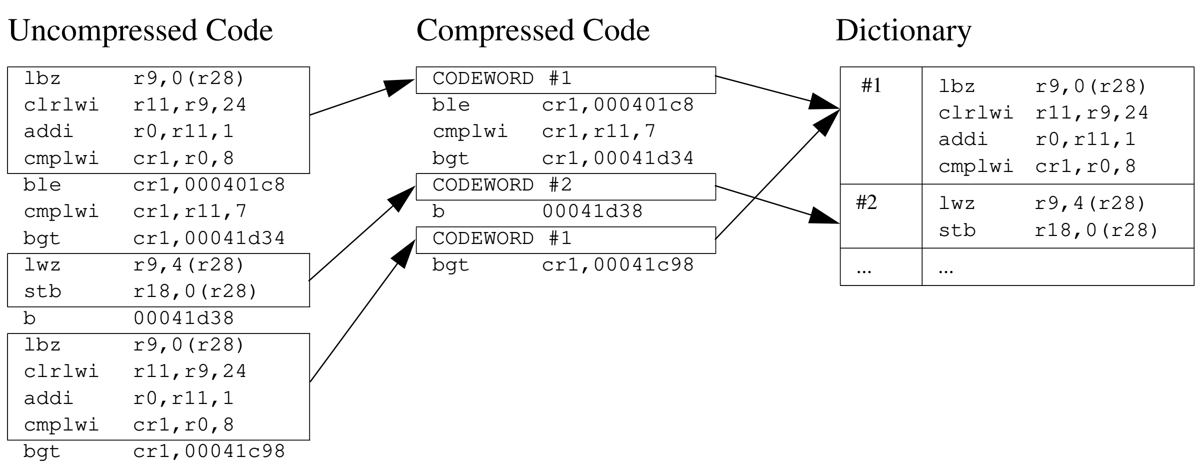

Figure 2: Example of compression

1.2 Overview of compression method

Our compression method finds sequences of instruction bytes that are frequently repeated throughout a single program and replaces the entire sequence with a single codeword. All rewritten (or encoded) sequences of instructions are kept in a dictionary which, in turn, is used at program execution time to expand the singleton codewords in the instruction stream back into the original sequence of instructions. All codewords assigned by the compression algorithm are merely indices into the instruction dictionary.

The final compressed program consists of codewords interspersed with uncompressed instructions. Figure 2 illustrates the relationship between the uncompressed code, the compressed code, and the dictionary. A complete description of our compression method is presented in Section 3.

2 Background and Related Work

In this section we will discuss strategies for text compression, and methods currently employed by microprocessor manufacturers to reduce the impact of RISC instruction sets on program size.

2.1 Text compression

Text compression methods fall into two general categories: *statistical* and *dictionary.*

Statistical compression uses the frequency of singleton characters to choose the size of the codewords that will replace them. Frequent characters are encoded using shorter codewords so that the overall length of the compressed text is minimized. Huffman encoding of text is a well-known example.

Dictionary compression selects entire phrases of common characters and replaces them with a single codeword. The codeword is used as an index into the dictionary entry which contains the original characters. Compression is achieved because the codewords use fewer bits than the characters they replace.

There are several criteria used to select between using dictionary and statistical compression techniques. Two very important factors are the *decode efficiency* and the overall *compression ratio*. The decode efficiency is a measure of the work required to re-expand a compressed text. The compression ratio is defined by the formula:

$$\textit{compression ratio} = \frac{\textit{compressed size}}{\textit{original size}} \qquad \text{(Eq. 1)}$$

Dictionary decompression uses a codeword as an index into the dictionary table, then inserts the dictionary entry into the decompressed text stream. If codewords are aligned with machine words, the dictionary lookup is a constant time operation. Statistical compression, on the other hand, uses codewords that have different bit sizes, so they do not align to machine word boundaries. Since codewords are not aligned, the statistical decompression stage must first establish the range of bits comprising a codeword before text expansion can proceed.

It can be shown that for every dictionary method there is an equivalent statistical method which achieves equal compression and can be improved upon to give better compression [Bell90]. Thus statistical methods can always achieve better compression than dictionary methods albeit at the expense of additional computation requirements for decompression. It should be noted, however, that dictionary compression yields good results in systems with memory and time constraints because one entry expands to several characters. In general, dictionary compression provides for faster (and simpler) decoding, while statistical compression yields a better compression ratio.

2.2 Compression for RISC instruction sets

Although a RISC instruction set is easy to decode, its fixed-length instruction formats are wasteful of program memory. Thumb [ARM95][MPR95] and MIPS16 [Kissell97] are two recently proposed instruction set modifications which define reduced instruction word sizes in an effort to reduce the overall size of compiled programs.

Thumb is a subset of the ARM architecture consisting of 36 ARM 32-bit wide instructions which have been re-encoded to require only 16 bits. The instructions included in Thumb either do not require a full 32-bits, are frequently used, or are important to the compiler for generating small object code. Programs compiled for Thumb achieve 30% smaller code in comparison to the standard ARM instruction set [ARM95].

MIPS16 defines a 16-bit fixed-length instruction set architecture (ISA) that is a subset of MIPS-III. The instructions used in MIPS16 were chosen by statistically analyzing a wide range of application programs for the instructions most frequently generated by compilers. Code written for 32-bit MIPS-III is typically reduced 40% in size when compiled for MIPS16 [Kissell97].

Both Thumb and MIPS16 act as preprocessors for their underlying architectures. In each case, a 16-bit instruction is fetched from the instruction memory, expanded into a 32-bit wide instruction, and passed to the base processor core for execution.

Both the Thumb and MIPS16 shrink their instruction widths at the expense of reducing the number of bits used to represent register designators and immediate value fields. This confines Thumb and MIPS16 programs to 8 registers of the base architecture and significantly reduces the range of immediate values.

As subsets of their base architectures, Thumb and MIPS16 are neither capable of generating complete programs, nor operating the underlying machine. Thumb relies on 32-bit instructions memory management and exception handling while MIPS16 relies on 32-bit instructions for floating-point operations. Moreover, Thumb cannot exploit the conditional execution and zero-latency shifts and rotates of the underlying ARM architecture. Both Thumb and MIPS16 require special branch instructions to toggle between 32-bit and 16-bit modes.

The fixed set of instructions which comprise Thumb and MIPS16 were chosen after an assessment of the instructions used by a range of applications. Neither architecture can access all registers, instructions, or modes of the underlying 32-bit core architecture.

In contrast, we derive our codewords and dictionary from the specific characteristics of the program under execution. Because of this, a compressed program can access all the resources available on the machine, yet can still exploit the compressibility of each individual program.

2.3 CCRP

The Compressed Code RISC Processor (CCRP) described in [Wolfe92][Wolfe94] has an instruction cache that is modified to run compressed programs. At compile-time the cache line bytes are Huffman encoded. At run-time cache lines are fetched from main memory, uncompressed, and put in the instruction cache. Instructions fetched from the cache have the same addresses as in the uncompressed program. Therefore, the core of the processor does not need modification to support compression. However, cache misses are problematic because missed instructions in the cache do not reside at the same address in main memory. CCRP uses a Line Address Table (LAT) to map missed instruction cache addresses to main memory addresses where the compressed code is located. The LAT limits compressed programs to only execute on processors that have the same line size for which they were compiled.

One short-coming of CCRP is that it compresses on the granularity of bytes rather than full instructions. This means that CCRP requires more overhead to encode an instruction than our scheme which encodes groups of instructions. Moreover, our scheme requires less effort to decode a program since a single codeword can encode an entire group of instructions. In addition, our compression method does not need a LAT mechanism since we patch all branches to use the new instruction addresses in the compressed program.

2.4 Liao et al.

A purely software method of supporting compressed code is proposed in [Liao96]. The author finds *mini-subroutines* which are common sequences of instructions in the program. Each instance of a mini-subroutine is removed from the program and replaced with a call instruction. The mini-subroutine is placed once in the text of the program and ends with a return instruction. Mini-subroutines are not constrained to basic blocks and may contain branch instructions under restricted conditions. The prime advantage of this compression method is that it requires no hardware support. However, the subroutine call overhead will slow program execution.

[Liao96] suggests a hardware modification to support code compression consisting primarily of a *call-dictionary* instruction. This instruction takes two arguments: *location* and *length*. Common instruction sequences in the program are saved in a dictionary, and the sequence is replaced in the program with the *call-dictionary* instruction. During execution, the processor jumps to the point in the dictionary indicated by *location* and executes *length* instructions before implicitly returning. [Liao96] limits the dictionary to use sequences of instructions within basic blocks only.

[Liao96] does not explore the trade-off of the field widths for the *location* and *length* arguments in the call-dictionary instruction. Only codewords that are 1 or 2 instruction words in size are considered. This requires the dictionary to contain sequences with at least 2 or 3 instructions, respectively, since shorter sequences would be no bigger than the call-dictionary instruction and no compression would result.

Since single instructions are the most frequently occurring patterns, it is important to use a scheme that can compress them. In this paper we vary the parameters of *dictionary size* (the number of entries in the dictionary) and the *dictionary entry length* (the number of instructions at each dictionary entry) thus allowing us to examine the efficacy of compressing instruction sequences of any length.

3 Compression Method

3.1 Algorithm

Our compression method is based on the technique introduced in [Bird96][Chen97a]. A dictionary compression algorithm is applied after the compiler has generated the program. We take advantage of SDTS and find common sequences of instructions to place in the dictionary. Our algorithm is divided into 3 steps:

1. Build the dictionary
2. Replace instruction sequences with codewords

3. Encode the codewords

3.1.1 Dictionary content

For an arbitrary text, choosing those entries of a dictionary that achieve maximum compression is NP-complete in the size of the text [Storer77]. As with most dictionary methods, we use a greedy algorithm to quickly determine the dictionary entries[1]. On every iteration of the algorithm, we examine each potential dictionary entry and find the one that results in the largest immediate savings. The algorithm continues to pick dictionary entries until some termination criteria has been reached; this is usually the exhaustion of the codeword space. The maximum number of dictionary entries is determined by the choice of the encoding scheme for the codewords. Obviously, codewords with more bits can index a larger range of dictionary entries. We limit the dictionary entries to sequences of instructions within a basic block. We allow branch instructions to branch to codewords, but they may not branch within encoded sequences. We also do not compress branches with offset fields. These restrictions simplify code generation.

3.1.2 Replacement of instructions by codewords

Our greedy algorithm combines the step of building the dictionary with the step of replacing instruction sequences. As each dictionary entry is defined, all of its instances in the program are replaced with a token. This token is replaced with an efficient encoding in the encoding step.

3.1.3 Encoding

Encoding refers to the representation of the codewords in the compressed program. As discussed in Section 2.1, variable-length codewords, (such as those used in the Huffman encoding in [Wolfe92]) are expensive to decode. A fixed-length codeword, on the other hand, can be used directly as an index into the dictionary making decoding a simple table lookup operation.

Our baseline compression method uses a fixed-length codeword to enable fast decoding. We also investigate a variable-length scheme. However, we restrict the variable-length codewords to be a multiple of some basic unit. For example, we present a compression scheme with codewords that are 4 bits, 8 bits, 12 bits, and 16 bits. All instructions (compressed and uncompressed) are aligned to the size of the smallest codeword. The shortest codewords encode the most frequent dictionary entries to maximize the savings. This achieves better compression than a fixed-length encoding, but complicates decoding.

3.2 Related Issues

3.2.1 Branch instructions

One side effect of any compression scheme is that it alters the locations of instructions in the program. This presents a special problem for branch instructions, since branch targets change as a result of program compression.

1. Greedy algorithms are often near-optimal in practice.

For this study, we do not compress relative branch instructions (i.e. those containing an offset field used to compute a branch target). This makes it easy for us to patch the offset fields of the branch instruction after compression. If we allowed compression of relative branches, we might need to rewrite codewords representing relative branches after a compression pass; but this would affect relative branch targets thus requiring a rewrite of codewords, etc. The result is a NP-complete problem [Szymanski78].

Indirect branches are compressed in our study. Since these branches take their target from a register, the branch instruction itself does not need to be patched after compression, so it cannot create the codeword rewriting problem outlined above. However, jump tables (containing program addresses) need to be patched to reflect any address changes due to compression. GCC puts jump table data in the `.text` section immediately following the branch instruction. We assume that this table could be relocated to the `.data` section and patched with the post-compression branch target addresses.

3.2.2 Branch targets in fixed-length instruction sets

Fixed-length instruction sets typically restrict branches to use targets that are aligned to instruction word boundaries. Since our primary concern is code size, we trade-off the performance advantages of aligned fixed-length instructions in exchange for more compact code. We use codewords that are smaller than instruction word boundaries and align them to the size of the smallest codeword (4 bits in this study). Therefore, we need to specify a method to address branch targets that do not fall at instruction word boundaries.

One solution is to pad the compressed program so that all branch targets are aligned as defined by the original ISA. The obvious disadvantage of this solution is that it will decrease the compression ratio.

A more complex solution (the one we have adopted for our experiments) is to modify the control unit of the processor to treat the branch offsets as aligned to the size of the smallest codeword. For example, if the size of a codeword is 8 bits, then a 32-bit aligned instruction set would have its branch offset range reduced by a factor of 4. Table 1 shows that most branches in the benchmarks do not use the entire range of their offset fields. The post-compilation compressor modifies all branch offsets to use the alignment of the codewords. Branches requiring larger ranges are modified to load their targets through jump tables. Of course, this will result in a slight increase in the code size for these branch sequences.

Table 1: Usage of bits in branch offset field

Bench	Static number of PC-relative Branches	Branch offsets not wide enough to provide 2-byte resolution to branch targets		Branch offsets not wide enough to provide 1-byte resolution to branch targets		Branch offsets not wide enough to provide 4-bit resolution to branch targets	
		Number	Percent	Number	Percent	Number	Percent
compress	2,047	0	0.00%	0	0.00%	5	0.24%
gcc	56,750	15	0.03%	146	0.26%	378	0.67%
go	9,719	0	0.00%	0	0.00%	5	0.05%
ijpeg	6,147	0	0.00%	0	0.00%	5	0.08%
li	4,806	0	0.00%	0	0.00%	40	0.83%
m88ksim	6,346	0	0.00%	0	0.00%	3	0.05%
perl	14,578	15	0.10%	74	0.51%	191	1.31%
vortex	22,658	0	0.00%	2	0.01%	54	0.24%

3.3 Compressed program processor

The general design for a compressed program processor is given in Figure 3. We assume that all levels of the memory hierarchy will contain compressed instructions to conserve memory. Since the compressed program may contain both compressed and uncompressed instructions, there are two paths from the program memory to the processor core. Uncompressed instructions proceed directly to the normal instruction decoder. Compressed instructions must first be translated using the dictionary before being decoded and executed in the usual manner. The dictionary could be loaded in a variety of ways. If the dictionary is small, one possibility is to place it in a permanent on-chip memory. Alternatively, if the dictionary is larger, it might be kept as a data segment of the compressed program and each dictionary entry could be loaded as needed.

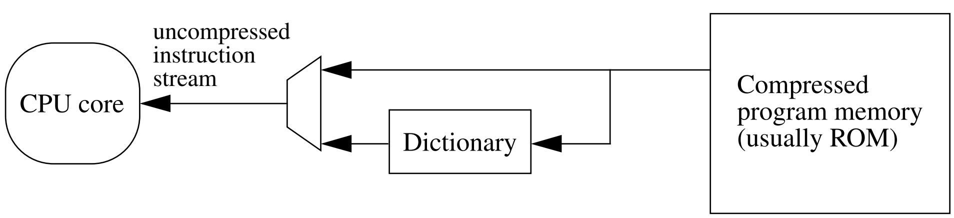

Figure 3: Overview of compressed program processor

4 Experiments

In this section we integrate our compression technique into the PowerPC instruction set. We compiled the SPEC CINT95 benchmarks with GCC 2.7.2 using -O2 optimization. The optimizations include common sub-expression elimination. They do not include function in-lining and loop unrolling since these optimizations tend to increase code size. Linking was done statically so

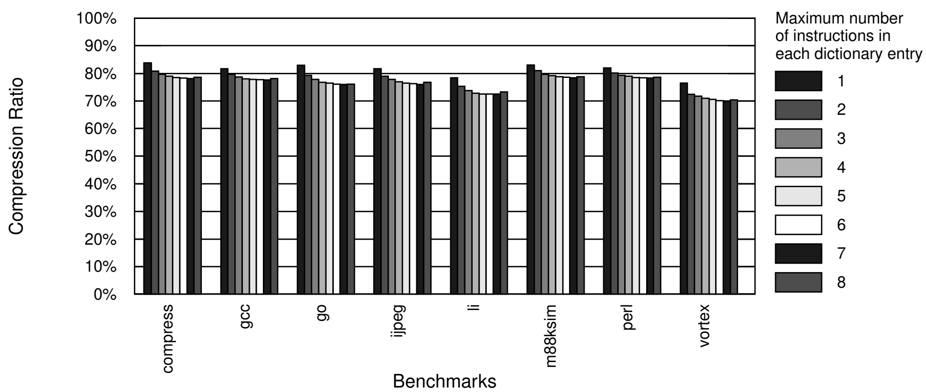

Figure 4: Effect of dictionary entry size on compression ratio

that the libraries are included in the results. All compressed program sizes include the overhead of the dictionary.

Recall that we are interested in the *dictionary size* (number of codewords) and *dictionary entry length* (number of instructions at each dictionary entry).

4.1 Baseline compression method

In our baseline compression method, we use codewords of 2-bytes. The first byte is an escape byte that has an illegal PowerPC opcode value. This allows us to distinguish between normal instructions and compressed instructions. The second byte selects one of 256 dictionary entries. Dictionary entries are limited to a length of 16 bytes (4 PowerPC instructions). PowerPC has 8 illegal 6-bit opcodes. By using all 8 illegal opcodes and all possible patterns of the remaining 2 bits in the byte, we can have up to 32 different escape bytes. Combining this with the second byte of the codeword, we can specify up to 8192 different codewords. Since compressed instructions use only illegal opcodes, any processor designed to execute programs compressed with the baseline method will be able to execute the original programs as well.

Our first experiments vary the parameters of the baseline method. Figure 4 shows the effect of varying the dictionary entry length. Interestingly, when dictionary entries are allowed to contain 8 instructions, the overall compression begins to decline. This can be attributed to our greedy selection algorithm for generating the dictionary. Selecting large dictionary entries removes some opportunities for the formation of smaller entries. The large entries are chosen because they result in an immediate reduction in the program size. However, this does not guarantee that they are the best entries to use for achieving good compression. When a large sequence is replaced, it destroys the small sequences that partially overlapped with it. It may be that the savings of using the multiple smaller sequences would be greater than the savings of the single large sequence. However, our greedy algorithm does not detect this case and some potential savings is lost. In general, dictionary entry sizes above 4 instructions do not improve compression noticeably. Figure 5 illustrates what happens when the number of codewords (entries in the dictionary) increases. The compression ratio for each program continues to improve until a maximum amount of codewords is reached, after which only unique, single use encodings remain uncompressed. Table 2 lists the maximum number of codewords for each program under the baseline compression method, representing an upper bound on the size of the dictionary.

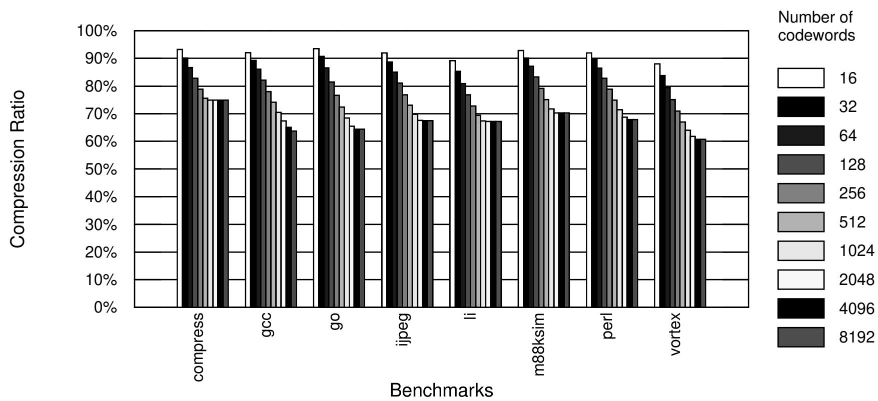

Figure 5: Effect of number of codewords on compression ratio

Table 2: Maximum number of codewords used in baseline compression (max. dictionary entry size = 4)

Bench	Maximum Number of Codewords Used
compress	647
gcc	7927
go	3123
ijpeg	2107
li	1104
m88ksim	1729
perl	2970
vortex	3545

The benchmarks contain numerous instructions that occur only a few times. As the dictionary becomes large, there are more codewords available to replace the numerous instruction encodings that occur infrequently. The savings of compressing an individual instruction is tiny, but when it is multiplied over the length of the program, the compression is noticeable. To achieve good compression, it is more important to increase the number of codewords in the dictionary rather than increase the length of the dictionary entries. A few thousand codewords is enough for most SPEC CINT95 programs.

4.1.1 Usage of the dictionary

Since the usage of the dictionary is similar across all the benchmarks, we show results using ijpeg as a representative benchmark. We extend the baseline compression method to use dictionary entries with up to 8 instructions. Figure 6 shows the composition of the dictionary by the number of instructions the dictionary entries contain. The number of dictionary entries with only a single instruction ranges between 48% and 80%. Not surprisingly, the larger the dictionary, the higher the proportion of short dictionary entries. Figure 7 shows which dictionary entries contribute the most to compression. Dictionary entries with 1 instruction achieve between 48% and 60% of the compression savings. The short entries contribute to a larger portion of the savings as the size of the dictionary increases. The compression method in [Liao96] cannot take advantage

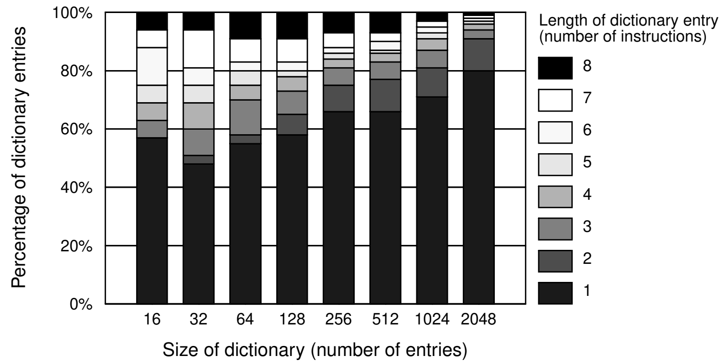

Figure 6: Composition of dictionary for ijpeg (max. dictionary entry = 8 instructions)

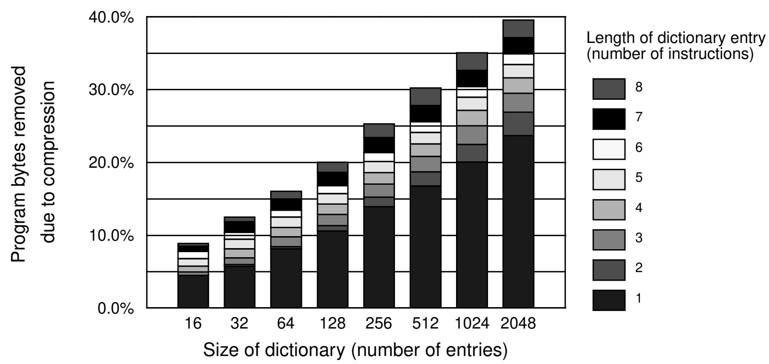

Figure 7: Bytes saved in compression of ijpeg according to instruction length of dictionary entry

of this since the codewords are the size of single instructions, so single instructions are not compressed.

4.1.2 Compression using small dictionaries

Some implementations of a compressed code processor may be constrained to use small dictionaries. We investigated compression with dictionaries ranging from 128 bytes to 512 bytes in size. We present one compression scheme to demonstrate that compression can be beneficial even for small dictionaries. Our compression scheme for small dictionaries uses 1-byte codewords and dictionary entries of up to 4 instructions in size. Figure 8 shows results for dictionaries with 8, 16, and 32 entries. On average, a dictionary size of 512 bytes is sufficient to get a code reduction of 15%.

4.1.3 Variable-length codewords

In the baseline method, we used 2-byte codewords. We can improve our compression ratio by using smaller encodings for the codewords. Figure 9 shows that when the baseline compression uses 8192 codewords, 40% of the compressed program bytes are codewords. Since the baseline compression uses 2-byte codewords, this means 20% of the final compressed program size is due to escape bytes. We investigated several compression schemes using variable-length codewords

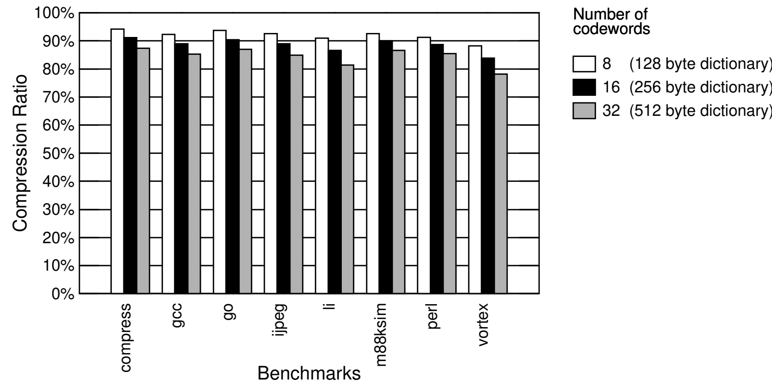

Figure 8: Compression Ratio for 1-byte codewords with up to 4 instructions/entry

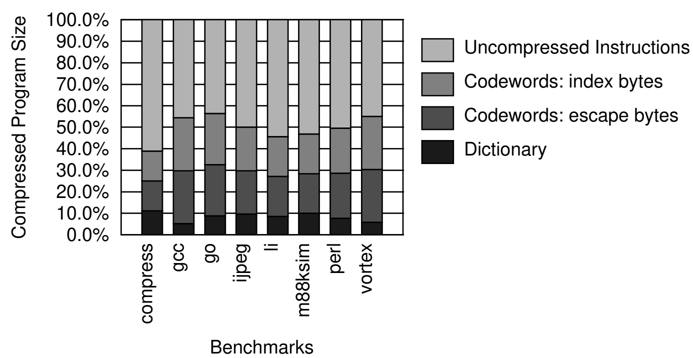

Figure 9: Composition of Compressed Program (8192 2-byte codewords, 4 instructions/entry)

aligned to 4-bits (nibbles). Although there is a higher decode penalty for using variable-length codewords, we are able to achieve better compression. By restricting the codewords to integer multiples of 4-bits, we have given the decoding process regularity that the 1-bit aligned Huffman encoding in [Wolfe94] lacks.

Our choice of encoding is based on SPEC CINT95 benchmarks. We present only the best encoding choice we have discovered. We use codewords that are 4-bits, 8-bits, 12-bits, and 16-bits in length. Other programs may benefit from different encodings. For example, if many codewords are not necessary for good compression, then more 4-bit and 8-bit code words could be used to further reduce the codeword overhead.

A diagram of the nibble aligned encoding is shown in Figure 10. This scheme is predicated on the observation that when an unlimited number of codewords are used, the final compressed program size is dominated by codeword bytes. Therefore, we use the escape code to indicate (less frequent) uncompressed instructions rather than codewords. The first 4-bits of the codeword determine the length of the codeword. With this scheme, we can provide 128 8-bit codewords, and a few thousand 12-bit and 16-bit codewords. This offers the flexibility of having many short codewords (thus minimizing the impact of the frequently used instructions), while allowing for a large overall number of codewords. One nibble is reserved as an escape code for uncompressed instruc-

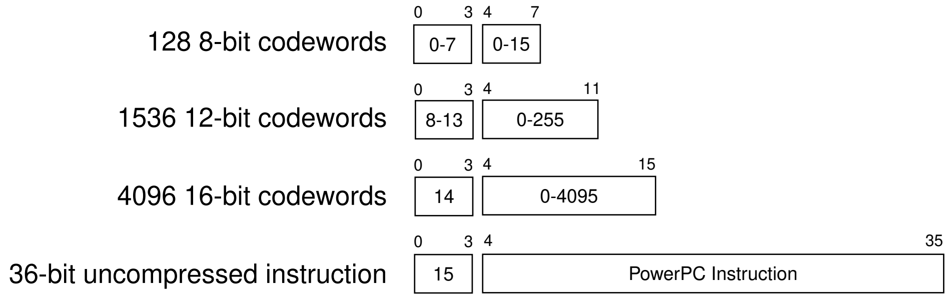

Figure 10: Nibble Aligned Encoding

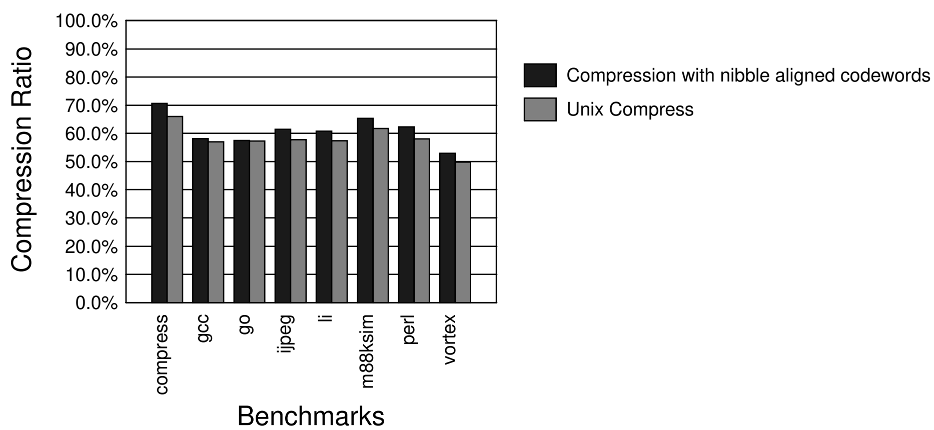

Figure 11: Comparison of nibble aligned compression with Unix Compress

tions. We reduce the codeword overhead by encoding the most frequent sequences of instructions with the shortest codewords.

Using this encoding technique effectively redefines the entire instruction set encoding, so this method of compression can be used in existing instruction sets that have no available escape bytes. Unfortunately, this also means that the original programs will no longer execute unmodified on processors that execute compressed programs without mode switching.

Our results for the 4-bit aligned compression are presented in Figure 11. We obtain a code reduction of between 30% and 50% depending on the benchmark. For comparison, we extracted the instruction bytes from the benchmarks and compressed them with Unix Compress. Compress uses an adaptive dictionary technique (based on Ziv-Lempel coding) which can modify the dictionary in response to changes in the characteristics of the text. In addition, it also uses Huffman encoding on its codewords, and thus should be able to achieve better compression than our method. Figure 11 shows that Compress does indeed do better, but our compression ratio is still within 5% for all benchmarks.

5 Conclusions and Future Work

We have proposed a method of compressing programs for embedded microprocessors where program size is limited. Our approach combines elements of two previous proposals. First we use a dictionary compression method (as in [Liao96]) that allows codewords to expand to several instructions. Second, we allow the codewords to be smaller than a single instruction (as in [Wolfe94]). We find that the size of the dictionary is the single most important parameter in attaining a better compression ratio. The second most important factor is reducing the codeword size below the size of a single instruction. We find that much of our savings comes from compressing patterns of single instructions. Our most aggressive compression for SPEC CINT95 achieves a 30% to 50% code reduction.

Our compression ratio is similar to that achieved by Thumb and MIPS16. While Thumb and MIPS16 designed a completely new instruction set, compiler, and instruction decoder, we achieved our results only by processing compiled object code and slightly modifying the instruction fetch mechanism.

There are several ways that our compression method can be improved. First, the compiler could attempt to produce instructions with similar byte sequences so they could be more easily compressed. One way to accomplish this is by allocating registers so that common sequences of instructions use the same registers. Another way is to generate more generalized STDS code sequences. These would be less efficient, but would be semantically correct in a larger variety of circumstances. For example, in most optimizing compilers, the function prologue sequence might save only those registers which are modified within the body of the function. If the prologue sequence were standardized to always save all registers, then all instructions of the sequence could be compressed to a single codeword. This space saving optimization would decrease code size at the expense of execution time. Table 3 shows that the prologue and epilogue combined typically account for 12% of the program size, so this type of compression would provide significant size reduction.

Table 3: Prologue and epilogue code in benchmarks

Bench	Static prologue instructions (percentage of entire program)	Static epilogue instructions (percentage of entire program)
compress	5.3%	6.2%
gcc	4.2%	4.9%
go	6.2%	6.8%
ijpeg	6.9%	9.4%
li	8.1%	9.9%
m88ksim	5.5%	6.4%
perl	3.7%	4.3%
vortex	6.3%	7.1%

We also plan to explore the performance aspects of our compression and examine the tradeoffs in partitioning the on-chip memory for the dictionary and program.

6 References

[Aho86] A. Aho, R. Sethi and J. Ullman, *Compiler: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[ARM95] Advanced RISC Machines Ltd., *An Introduction to Thumb*, March 1995.

[Bell90] T. Bell, J. Cleary, I. Witten, *Text Compression*, Prentice Hall, 1990.

[Bird96] P. Bird and T. Mudge, *An Instruction Stream Compression Technique*, CSE-TR-319-96, EECS Department, University of Michigan, November 1996.

[Chen97a] I. Chen, P. Bird, and T. Mudge, *The Impact of Instruction Compression on I-cache Performance*, CSE-TR-330-97, EECS Department, University of Michigan, 1997.

[Chen97b] I. Chen, *Enhancing Instruction Fetching Mechanism Using Data Compression*, Dissertation, University of Michigan, 1997.

[Kissell97] K. Kissell, *MIPS16: High-density MIPS for the Embedded Market*, Silicon Graphics MIPS Group, 1997.

[Liao96] S. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Dissertation, Massachusetts Institute of Technology, June 1996.

[MPR95] "Thumb Squeezes ARM Code Size", Microprocessor Report 9(4), 27 March 1995.

[Perl96] S. Perl and R. Sites, *Studies of Windows NT performance using dynamic execution traces*, Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation, October 1996.

[SPEC95] SPEC CPU'95, Technical Manual, August 1995.

[Storer77] J. Storer, "NP-completeness results concerning data compression," Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

[Szymanski78] T. G. Szymanski, "Assembling code for machines with span-dependent instructions", *Communications of the ACM 21:4*, pp. 300-308, April 1978.

[Wolfe92] A. Wolfe and A. Chanin, *Executing Compressed Programs on an Embedded RISC Architecture*, Proceedings of the 25th Annual International Symposium on Microarchitecture, December 1992.

[Wolfe94] M. Kozuch and A. Wolfe, *Compression of Embedded System Programs*, IEEE International Conference on Computer Design, 1994.