Proceedings

Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture

December 1–3, 1997

Research Triangle Park, North Carolina

Sponsored by


IEEE Computer Society

Technical Committee on Microarchitecture


ACM Special Interest Group on Microarchitecture

Los Alamitos, California

Washington • Brussels • Tokyo

Improving Code Density Using Compression Techniques

Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
{lefurgy,pbird,icheng,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, ARM, and i386 instruction sets and achieve an average size reduction of 39%, 34%, and 26%, respectively, for SPEC CINT95 programs.

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. In spite of these impressive statistics, processors for embedded applications have been much less studied than general purpose microprocessors. In many ways they present the designer with a greater challenge because embedded processors are more highly constrained by cost, power, and size. 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, smaller program sizes imply that smaller, cheaper dies can be used in embedded systems. 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 requires twice as much instruction bandwidth as the processor is able to provide due to instruction cache misses. 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 provide higher instruction bandwidth [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, ARM, and i386 instruction sets.

1.1 Repeated instruction encodings

Object code generated by compilers mostly contains instructions from a small, highly used subset of the instruction set. This causes a high degree of repetition 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 repetition of instruction encodings, we profiled the SPEC CINT95 benchmarks [SPEC95]. The benchmarks were compiled for PowerPC with GCC 2.7.2 using -O2 optimization. In Figure 1, the results for the *go* benchmark show that 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. On average, more than 80%

1072-4451/97 $10.00 © 1997 IEEE

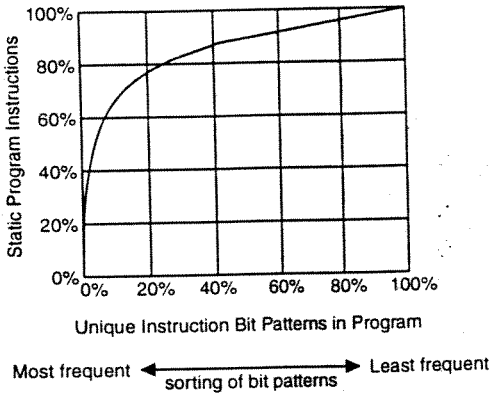

Figure 1: Unique instruction bit patterns in a program as a percentage of static program instructions.

The data is from the *go* benchmark compiled for PowerPC. The x-axis is sorted by the frequency of bit patterns in the static program.

of the instructions in CINT95 have bit pattern encodings which are used multiple times in the program. In addition to the repetition of single instructions, we also observe that programs contain entire repeated sequences of instructions. It is clear that the repetition of instruction encodings provides a great opportunity for reducing program size through compression techniques.

1.2 Overview of compression method

Our compression method finds sequences of instructions 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. Codewords assigned by the compression algorithm are 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.

The organization of this paper is as follows. Section 2 reviews text compression and previous methods of compressing programs. In section 3, we describe our compression method. Our experimental results are presented in section 4. Finally, section 5 contains our conclusions.

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:

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

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 pro-

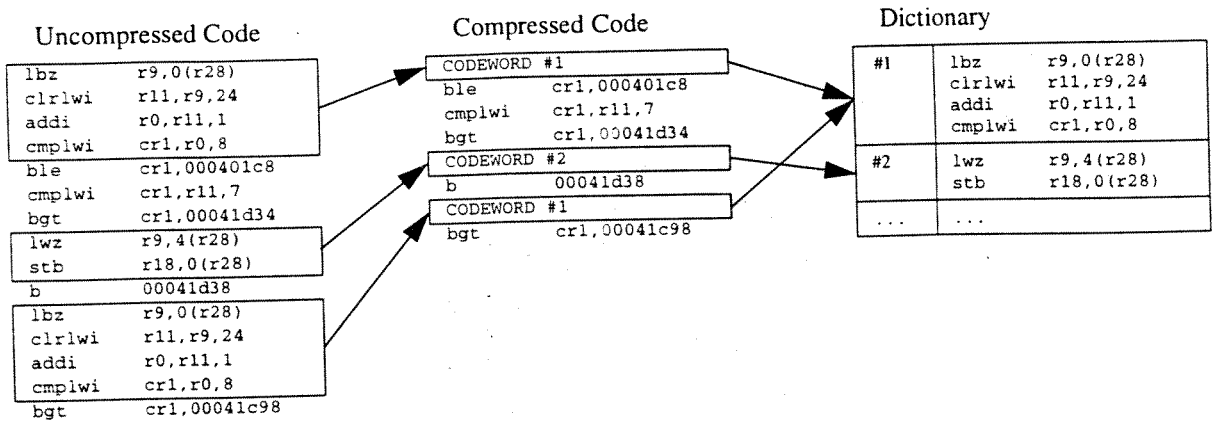

Figure 2: Example of compression.

vides for faster (and simpler) decoding, while statistical compression yields a better compression ratio.

2.2 Thumb and MIPS16

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 and MIPS16 are defined as subsets of the ARM and MIPS-III architectures. A wide range of applications were analyzed to determine the composition of the subsets. The instructions included in the subsets are either frequently used, do not require a full 32-bits, or are important to the compiler for generating small object code. The original 32-bit wide instructions have been re-encoded to be 16-bits wide. Thumb and MIPS16 are reported to achieve code reductions of 30% and 40%, respectively [ARM95][Kissell97].

Thumb and MIPS16 instructions have a one-to-one correspondence to instructions in the base architectures. In each case, a 16-bit instruction is fetched from the instruction memory, decoded to the equivalent 32-bit wide instruction, and passed to the base processor core for execution.

The Thumb and MIPS16 implementations are unable to use the full capabilities of the underlying processor. The instruction widths are shrunk at the expense of reducing the number of bits used to represent register designators and immediate value fields. This confines programs to 8 registers of the base architecture and significantly reduces the range of immediate values. In addition, conditional execution and zero-latency shifts are not available in Thumb and floating-point instructions are not available in MIPS16.

Compression in Thumb and MIPS16 occurs on a per procedure basis. There are special branch instructions to toggle between 32-bit and 16-bit modes.

While Thumb and MIPS16 are effective in reducing code size, they increase the number of static instructions in a program. If the number of instructions did not increase, then it would be possible to achieve a code reduction of 50% by moving from 32-bit instructions to 16-bit instructions. However, Thumb and MIPS16 report code reductions of 30% and 40% respectively. This suggests that the number of static instructions in programs increases by 40% and 20% respectively. This requires a program to execute more instructions which reduces performance. For example, Thumb code runs 15% - 20% slower on systems with ideal instruction memories (32-bit buses and no wait states) [ARM95].

Our method does not cause the number of instructions in a program to increase. Compressed programs are translated back into the instructions of the original uncompressed program and executed, so that the number of instructions executed in a program is not changed. Moreover, a compressed program can access all the registers, operations, and modes available on the underlying processor.

We derive our codewords and dictionary from the specific characteristics of the program under execution. Tuning the compression method to individual programs helps to improve code size.

Our compression method works on the level of individual instructions. There are no special modes for executing compressed programs. This gives the programmer more control over the use of compression.

2.3 Compressed Code RISC Processor

The Compressed Code RISC Processor (CCRP) described in [Wolfe92][Kozuch94] is an interesting

approach that employs 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. Finally, our compression method does not need a LAT mechanism since we patch all branches in the binary executable to use the new instruction addresses in the compressed program.

2.4 Reusing mini-subroutines

Liao proposes a software method for supporting compressed code [Liao96]. He 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.

A hardware modification is proposed 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. The advantage of this method over the purely software approach is that it eliminates the return instruction from the mini-subroutine. However, it also limits the dictionary to sequences of instructions within basic blocks.

While Liao's compression method uses the familiar branch mechanism to uncompress code, it introduces many branch instructions into a program thus reducing overall performance.

The *call-dictionary* instruction is considered to be the size of 1 or 2 instruction words. 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. This method misses an important compression opportunity. We will later show that there is a significant advantage for compressing patterns consisting of one instruction.

Liao does not explore the trade-off of the field widths for the *location* and *length* arguments in the call-dictionary instruction. 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.

2.5 CISC instruction sets

CISC instruction sets are often implemented with micro-code. For example, in the Pentium microprocessor, x86 instructions are expanded into a series of RISC-like micro-instructions. Interestingly, our technique, while analogous to micro-code, can compress x86 programs. We will compare compression of RISC programs to i386 programs and compressed i386 programs.

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 search the program object modules to find common sequences of instructions to place in the dictionary. Our algorithm has 3 parts:

1. Building the dictionary
2. Replacing instruction sequences with codewords
3. Encoding codewords

3.1.1 Building the dictionary

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. 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 Replacing instruction sequences with 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 codewords

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 8-bit, 12-bit, and 16-bit codewords. All instructions (compressed and uncompressed) are aligned on 4-bit boundaries. This achieves better compression than a fixed-length encoding, but complicates decoding.

3.2 Related issues

3.2.1 Branch instructions

One obvious side effect of a 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.

To avoid this problem, 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 again an 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.

3.2.2 Branch targets

Instruction sets 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 these aligned instructions in exchange for more compact code. We use codewords that are smaller than instruction words and align them on 4-bit boundaries. Therefore, we need to specify a method to address branch targets that do not fall at the original 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 codewords. The post-compilation compressor modifies all branch offsets to use this alignment.

One of our compression schemes requires that branch targets align to 4-bit boundaries. In PowerPC and ARM, branch targets align to 32-bit boundaries. Since branches in the compressed program specify a target aligned to a 4-bit boundary, the target could be in any one of 8 positions within the original 32-bit boundary. We use 3 bits in the branch offset to specify the location of the branch target within the usual 32-bit alignment. Overall, the range of the offset is reduced by a factor of 8. In our benchmarks, less than 1% of the branches with offsets had a target outside of this reduced range. Branch targets in x86 align to 8-bit boundaries. We use 1 bit in the offset to specify the 4-bit alignment of the compressed instruction within the usual 8-bit alignment. This reduces the range of branch offsets by a factor of 2. In our benchmarks, less than 2.2% of the branch offsets were outside this reduced range. 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.

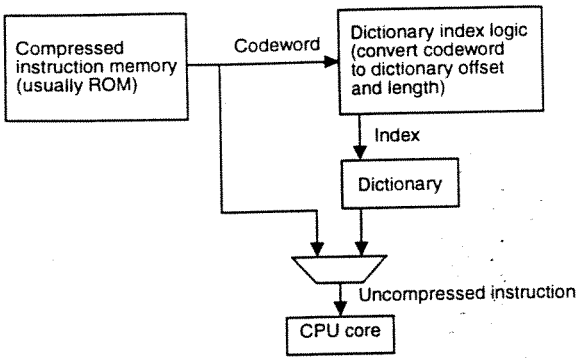

Figure 3: Compressed program processor.

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 instruction 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 simplest implementations, the codewords can be made to index directly into the dictionary. More complex implementations may need to provide a translation from the codeword to an offset and length in the dictionary. Since codewords are groups of sequential values with corresponding sequential dictionary entries, the computation to form the index is usually simple. Since the dictionary index logic is extremely small and is implementation dependent, we do not include it in our results.

4 Experiments

In this section we integrate our compression technique into the PowerPC, ARM, and i386 instruction sets. For PowerPC and i386 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. We compiled SPEC CINT92 and SPEC CINT95 for ARM6 using the Norcroft ARM C compiler v4.30. For all instruction sets, the programs were not linked with libraries to minimize the differences across compiler environments and help improve comparisons across different instruction sets. All compressed program sizes include the overhead of the dictionary.

Our compression experiments use two compression schemes. The first scheme uses fixed-length codewords and the second uses variable-length codewords. We implement the fixed-length compression on PowerPC and the variable-length compression on PowerPC, ARM, and i386.

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 Fixed-length codewords

Our baseline compression method, implemented on PowerPC, uses fixed-length 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.

4.2 Compressing patterns of 1 instruction

As outlined above, Liao finds common sequences of instructions and replaces them with a branch (call-dictionary) instruction. The problem with this method is that it is not possible to compress patterns of 1 instruction due to the overhead of the branch instruction. In order to be beneficial, the sequence must have at least two instructions.

Our first experiment measures the benefit of allowing sequences of single instructions to be compressed. Our baseline method allows single instructions to be compressed since the codeword (2 bytes) that is replacing the instruction (4 bytes) is smaller. We compare this against an augmented version of the baseline that uses 4-byte codewords. If we assume that the 4-byte codeword is actually a branch instruction, then we can approximate the effect of the compression used by Liao. This experiment limits compressed instruction sequences to 4 instructions. The largest dictionary generated (for gcc) used only 7577 codewords. Figure 4 shows that the 2-byte compression is a significant improvement over the 4-byte compression. This improvement is mostly due to the smaller codeword size, but a significant portion results from using patterns of 1 instruction. Figure 5 shows the contribution of each of these factors to the total savings. The size reduction due to

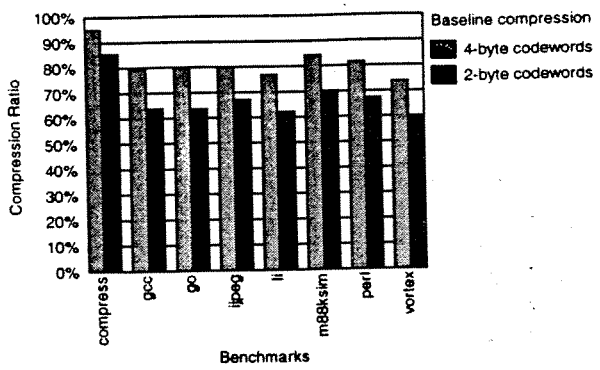

Figure 4: Comparison of baseline compression method with 2-byte and 4-byte codewords.

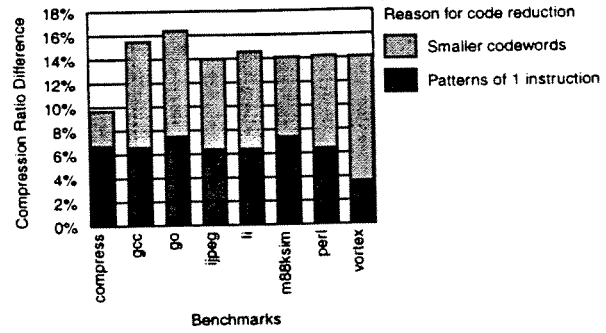

Figure 5: Analysis of difference in code reduction between 4-byte codewords and 2-byte codewords in baseline compression method.

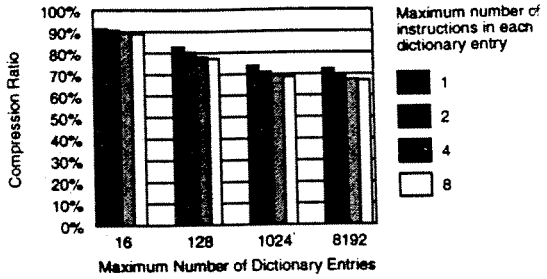

Figure 6: Summary of effect of number of dictionary entries and length of dictionary entries in baseline compression method.

Compression ratio is averaged over the CINT95 benchmarks.

Benchmark	Maximum Number of Codewords Used
compress	72
gcc	7577
go	2674
ijpeg	1616
li	454
m88ksim	1289
perl	2132
vortex	2878

Table 1: Maximum number of codewords used in baseline compression (maximum dictionary entry size is 4 instructions)

using 2-byte codewords was computed using the results of the 4-byte compression and recomputing the savings as if the codewords were only 2 bytes long. This savings was subtracted from the total savings to derive the savings due to using patterns of 1 instruction. For each benchmark, except vortex, using patterns of 1 instruction improved the compression ratio by over 6%.

4.3 Dictionary parameters

Our next experiments vary the parameters of the baseline method. Figure 6 shows the effect of varying the dictionary entry length and number of codewords (entries in the dictionary). The results are averaged over the CINT95 benchmarks. In general, dictionary entry sizes above 4 instructions do not improve compression noticeably. Table 1 lists the maximum number of codewords for each program under the baseline compression method, which is representative of the size of the dictionary.

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 CINT95 programs.

4.3.1 Usage of the dictionary

Our experiments reveal that dictionary usage is similar across all the benchmarks, thus we illustrate our results using *ijpeg* as a representative benchmark. We extend the baseline compression method to use dictionary entries with up to 8 instructions. Figure 7 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 from 50% to 80%. Surprisingly, the larger the dictionary, the higher the proportion of short dictionary entries. Figure 8 shows why

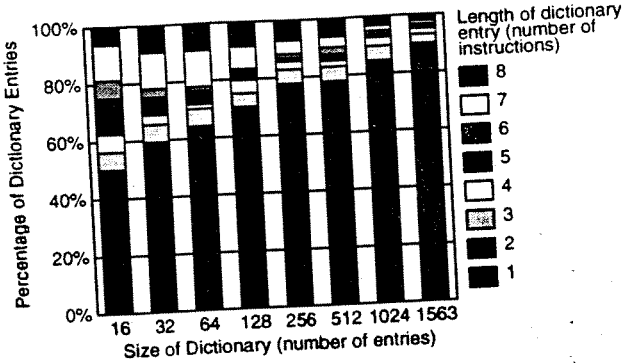

Figure 7: Composition of dictionary for *ijpeg* (longest dictionary entry is 8 instructions).

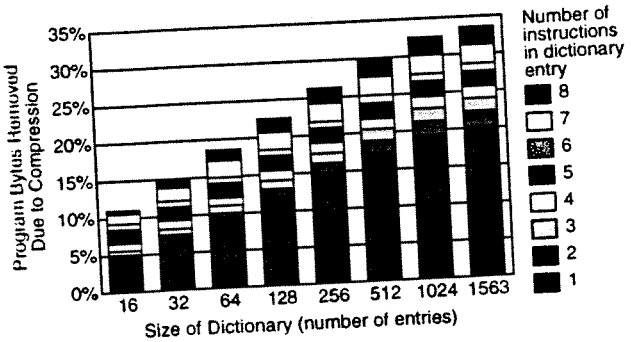

Figure 8: Bytes saved in compression of *ijpeg* according to instruction length of dictionary entry.

dictionary entries contribute the most to compression. Dictionary entries with 1 instruction achieve between 46% 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 of this since the codewords are the size of single instructions, so single instructions are not compressed.

4.4 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 in the baseline compression, 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 aligned to 4 bits (nibbles). Although there is a higher decode penalty for using variable-length codewords, they make possible better compression. By restricting the codewords to integer multiples of 4 bits, we still retain some of the decoding process regularity that the 1-bit aligned Huffman encoding in [Kozuch94] lacks.

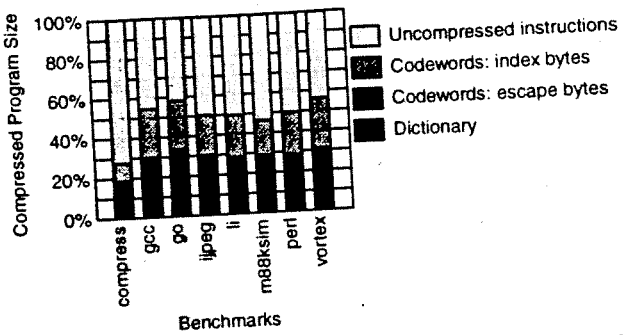

Figure 9: Composition of compressed PowerPC programs (maximum of 8192 2-byte codewords, longest dictionary entry is 4 instructions).

Our choice of encoding is based on CINT95 benchmarks. We present only the best encoding choice we have discovered. We use codewords that are 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 the large number of 12-bit and 16-bit codewords we use could be replaced with fewer (shorter) 4-bit and 8-bit codewords 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 contains more codewords than uncompressed instructions. 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 instructions. 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, such as ARM and i386.

Our results for PowerPC, ARM, and i386 using the 4-bit aligned compression are presented in Figure 11. We allowed the dictionaries to contain a maximum of 16 bytes per entry. We obtained average code reductions of 39%,

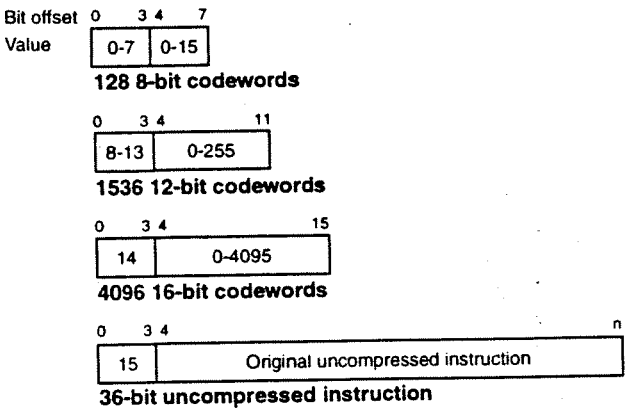

Figure 10: Nibble Aligned Encoding.

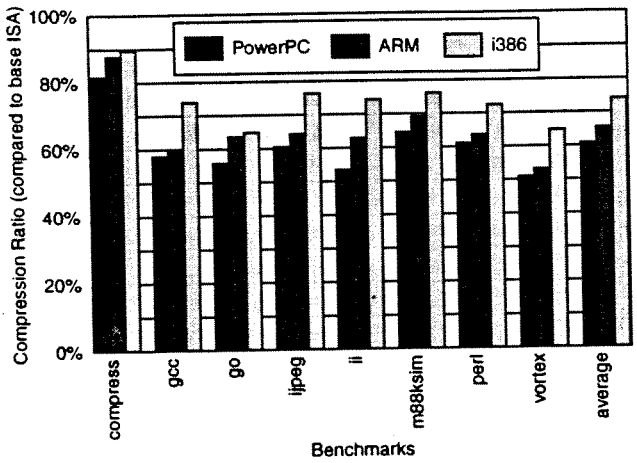

Figure 11: Nibble compression for PowerPC, ARM, and i386 instruction sets.

34%, and 26% for PowerPC, ARM, and i386, respectively. Figure 12 shows the uncompressed size and the compressed size of each benchmark for all instruction sets. The data is normalized to the size of the original uncompressed PowerPC program. One clear observation is that compressing PowerPC or ARM programs saves more memory than recompiling to the i386 instruction set. Compression of PowerPC programs resulted in a 39% size reduction, while using the i386 instruction set only provided a 29% size reduction over PowerPC. Compression of ARM programs yielded a 34% size reduction, but using i386 only gave a 18% size reduction over ARM. Overall, we were able to produce the smallest programs by compressing i386 and ARM programs.

4.5 Comparison to Thumb

In this section we compare Thumb against the nibble compression method. In Figure 13 we show the results

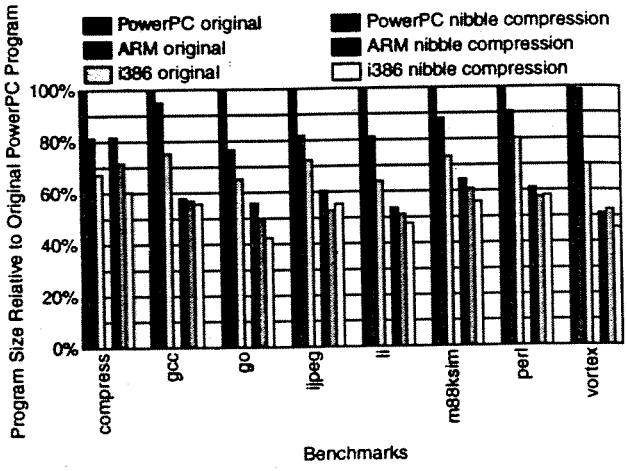

Figure 12: Comparison of compression across instruction sets.

All program sizes are normalized to the size of the original PowerPC programs.

published by ARM along with the results of nibble compression.

It is impossible to directly compare the compression ratios because we did not have the same binary executables as ARM. However the graph indicates an interesting trend. For the smaller programs (under 40K-bytes), Thumb compression is better, while for large programs, nibble compression is better. For the large programs, we started with an executable slightly larger than the ARM Ltd. version and compressed it to slightly smaller than the Thumb version.

The reason for this is that in small programs there are fewer repeated instructions, and this causes compressible sequences to be less frequent. Thumb is able to compress single instances of 32-bit instructions down to 16-bits, but our nibble compression requires at least 2 instances of the same instructions to compress it (due to dictionary overhead). Therefore on the small benchmarks where there are fewer repeated instructions, Thumb has the advantage. When programs are larger (over 40K-bytes) then there are enough repeated instructions and the nibble compression can overcome the dictionary overhead to beat the Thumb compression.

5 Discussion

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 [Kozuch94]). We

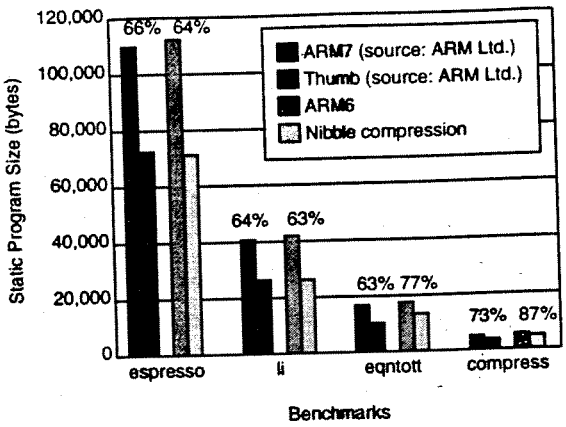

Figure 13: Comparison with Thumb.

The numbers over the bars give the compression ratio.

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. To obtain good compression we find it is crucial to have an encoding scheme that is capable of compressing patterns of single instructions. Our most aggressive compression for SPEC CINT95 achieves an average size reduction of 39%, 34%, and 26% for PowerPC, ARM, and i386, respectively.

Our compression ratio is similar to that achieved by Thumb and MIPS16. Our advantage over these instruction sets is that we do not increase the number of executed instructions and retain full access to the resources of the underlying processor. Compression is available on a per instruction basis without introducing any overhead to switch between compressed and non-compressed code.

There are several ways that our compression method could be improved. First, the compiler should minimize producing instructions with encodings that are used only once. In our PowerPC benchmarks, we found that between 6% - 15% of the instructions (not including branches) were not compressible by our method because they had instruction encodings that were only used once in the program. Second, the compiler could attempt to produce instructions with identical byte sequences so they become more compressible. One way to accomplish this is by allocating registers so that common sequences of instructions use the same registers. Finally, we could improve the selection of codewords in the dictionary by using covering algorithms instead of our greedy algorithm.

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

Acknowledgments

This work was supported by ARPA grants DAAH04-94-G-0327 and DABT63-97-C-0047.

References

[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*, Ph.D. Dissertation, University of Michigan, 1997.

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

[Kozuch94] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," *IEEE International Conference on Computer Design*, 1994.

[Liao96] S. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. 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.