Space-efficient Executable Program Representations for Embedded Microprocessors

by

Charles Robert Lefurgy

A thesis proposal submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

Doctoral Committee:

Professor Trevor Mudge, Chair
Professor Richard Brown
Assistant Professor Steve Reinhardt
Assistant Professor Gary Tyson

© Charles Robert Lefurgy 1998
All Rights Reserved

Table of Contents

Table of Contents ii

List of Tables iii

List of Figures iv

Chapter 1 Introduction 1

1.1 Data compression 2
1.2 Text compression 3
1.3 Repetition in object code 4
1.4 Organization 5

Chapter 2 Background 6

2.1 Improved encodings for native instructions 7
2.2 Interpreted programs 8
2.3 Frequency-based coding 10
2.4 Procedurization 11
2.5 Load-time representations 12
2.6 Conclusion 13

Chapter 3 Preliminary Work 15

3.1 Introduction 15
3.2 Overview of compression method 15
3.3 Experiments 20
3.4 Discussion 29
3.5 Improvements 30

Chapter 4 Future Work 31

4.1 A template compression instruction set 31
4.2 Why is template-compression a good idea? 34
4.3 A template compression compiler 34
4.4 Template compression experiments 38
4.5 Software compression 38
4.6 Improving execution-time 39
4.7 Conclusion 40

Bibliography 41

List of Tables

Table 3.1: Maximum number of codewords used in baseline compression. 23

List of Figures

Figure 1.1: Unique instruction bit patterns in a program as a percentage of static program instructions. 5

Figure 3.1: Example of compression. 17

Figure 3.2: Compressed program processor. 20

Figure 3.3: Comparison of baseline compression method with 2-byte and 4-byte codewords. 22

Figure 3.4: Analysis of difference in code reduction between 4-byte codewords and 2-byte codewords in baseline compression method. 22

Figure 3.5: Summary of effect of number of dictionary entries and length of dictionary entries in baseline compression method. 23

Figure 3.6: Composition of dictionary for ijpeg. 24

Figure 3.7: Bytes saved in compression of ijpeg according to instruction length of dictionary entry. 25

Figure 3.8: Composition of compressed PowerPC programs. 26

Figure 3.9: Nibble Aligned Encoding. 27

Figure 3.10: Nibble compression for PowerPC, ARM, i386, and MIPS-16 instruction sets. 27

Figure 3.11: Comparison of compression across instruction sets. 28

Figure 3.12: Comparison with MIPS-16. 28

Figure 4.1: Sample PowerPC code from vortex. 32

Figure 4.2: Example of dictionary-based compression. 32

Figure 4.3: Example of template-based compression. 33

Figure 4.4: Relation between intermediate representation and compression dictionary. 36

Figure 4.5: Example of canonical form for IR tree. 37

Chapter 1
Introduction

Embedded microprocessors are highly constrained by cost, power, and size. For control oriented embedded applications, the most common type, a significant portion of the 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. As typical code sizes have grown, high-level languages are being used to control development costs. However, compilers for these languages often produce code that is much larger than hand-optimized assembly code. Thus, the ability to compile programs to a small representation is important to reduce both software development costs and manufacturing costs.

High performance systems are also impacted by program size due to the delays incurred by instruction cache misses. A study at Digital [Perl96] measured the performance of an SQL server on a DEC 21064 Alpha. Due to instruction cache misses, the application could have used twice as much instruction bandwidth as the processor was able to provide. This problem is exacerbated by the growing gap between the cycle time of microprocessors and the access time of commodity DRAM. Reducing program size is one way to reduce instruction cache misses and provide higher instruction bandwidth [Chen97a].

Our contribution

Both low-cost embedded systems and high-performance microprocessors can benefit from small program sizes. This thesis proposal focuses on program representations of embedded applications, where execution speed can be traded for code size. Our prelimi-

nary work borrows concepts from the field of text compression and applies them to the compression of instruction sequences. We present an experiment that examines 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, i386, and MIPS-16 instruction sets. Finally, we use our results to propose another efficient program representation that explicitly communicates the patterns of computation common to each program.

This chapter concludes with an introduction to data compression and its application to program compression.

1.1 Data compression

The goal of data compression is to represent information in the smallest form that still holds the information content. Traditional data compression methods make several assumptions about the data being compressed. First, it is assumed that the compression must be done in a single sequential pass over the data because typical data may be too large to contain in storage (main memory or disk) at one time. One example of such data is a continuous stream of video. Second, this single pass approach takes advantage of history of recent symbols in the data stream. History information allows compressors to utilize repetition in the data and modify the compression technique in response to the changing characteristics of the data stream. This constrains the decompressor to start at the beginning of the data stream. The decompressor cannot begin decompressing at an arbitrary point in the data stream because it will not have the history information that the decompression algorithm depends upon. Third, most data compression methods use bit-aligned output to obtain the smallest possible representations.

In contrast, compression algorithms for computer programs can use a significantly different set of assumptions. First, programs are small enough to contain in storage, so the

compressor can optimize the final compressed representation based on the entire program instead of using only recent history information. Second, if decompression will occur as the program is executing, then it is desirable to begin decompression at arbitrary points in the program. Program execution can be redirected at branch instructions which make it necessary for the decompression to begin at any branch target. The unpredictable nature of the execution path between program runs is likely to constrain the length of history information available to the compressor. Third, most microprocessors have alignment restrictions which imposes a minimum size on instructions.

One advantage that programs have over typical data is that portions of the program (statements, instructions, etc.) can re-arranged to form an equivalent program. This may assist the compressor in finding more compressible patterns.

1.2 Text compression

Text compression refers to a class of *reversible* compression methods that allow the compressed text to be decompressed into a message identical to the original. They are particularly tailored to use a linear data stream. These properties make text compression applicable to computer programs. Text compression methods fall into two general categories: statistical and dictionary [Bell90].

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

1.3 Repetition in object code

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 1.1, 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

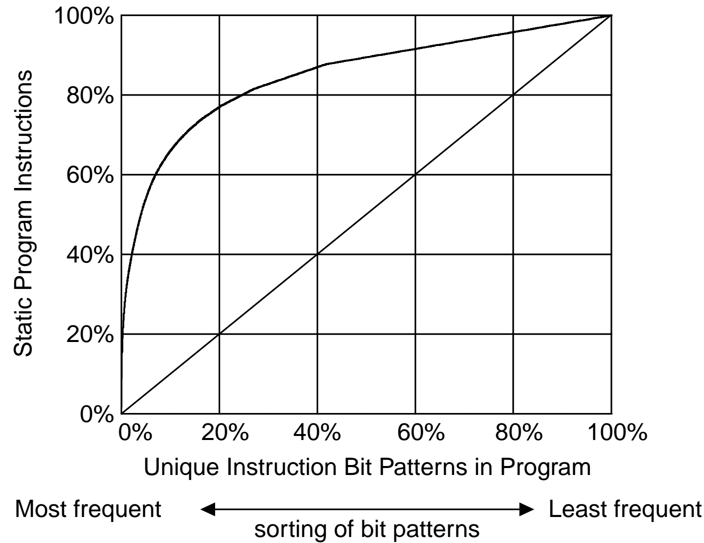

Figure 1.1: Unique instruction bit patterns in a program as a percentage of static program instructions.

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.

than 80% 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 observed that programs contain numerous repeated sequences of instructions. It is clear that the repetition of instruction encodings provides a great opportunity for reducing program size through compression techniques.

1.4 Organization

The organization of this thesis proposal is as follows. Chapter 2 reviews previous work to obtain small program sizes. Chapter 3 contains the results of a preliminary experiment that applies text compression techniques to programs. Finally, Chapter 4 discusses possible directions for future work.

Chapter 2
Background

This chapter provides background information on previous work in the representation of programs for small size. Since there are many techniques used to make code small, we will focus this review on current research trends.

The first technique tries to improve the encoding of native instructions in compiled-program environments. In our experiments, we have observed that the size of programs encoded in conventional instruction sets can differ by a factor of 2. This shows that instruction set design is important to achieve a small program size.

The second technique uses interpreted-program environments. Programs can be translated to a small intermediate form. An interpreter, compiled to native instructions, interprets the intermediate form into native instructions that accomplish the required computation. Because the intermediate code does not need to be concerned with host limitations (instruction word size and alignment), the intermediate instructions can be quite small.

The third technique uses Ziv-Lempel coding to reduce program size. LZ compression cannot be applied to the program as a whole, because it would be necessary to decompress the entire program at once to execute it – invalidating any execution-time size advantage. However, it can be applied to individual procedures and cache lines because when decompressed they are less than the size of the original program.

Finally, the fourth way to make programs small is to use hardware-independent techniques. There are many well known compiler optimizations that produce small code.

A separate line of research focuses on reducing the time to transfer a program over a network or load it from a disk. These techniques are not directly applicable to small executable representations of programs, because at execution time, they are expanded to full

size native instruction programs. Although they do not save space at execution time compared to conventional native instruction programs, the techniques used provide some of the smallest program representations.

2.1 Improved encodings for native instructions

Although a RISC instruction set is easy to decode, its fixed-length instruction formats are wasteful of program memory. Thumb [ARM95, Turley95] and MIPS-16 [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 MIPS-16 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 MIPS-16 are reported to achieve code reductions of 30% and 40%, respectively [ARM95, Kissell97].

Thumb and MIPS-16 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 16-bit instructions retain use of the 32-bit data paths in the base architectures.

The Thumb and MIPS-16 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 is not available in Thumb and floating-point instructions are not available in MIPS-16.

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

Thumb and MIPS-16 instructions are less expressive than their base architectures. Therefore, programs require more instructions to accomplish the same tasks. 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].

2.2 Interpreted programs

2.2.1 Directly Executed Languages

Flynn introduced the notion of Directly Executed Languages (DELs) whose representation could be specifically tailored to a particular application and language [Flynn83]. A DEL is program representation that is between the level of the source language and machine language. DEL programs are executed by a DEL-interpreter which is written in the machine language. The advantage of DELs are that they provide an efficient method to represent programs. The DEL representation is small for several reasons. First, the DEL representation uses the operators of the source language. Assuming that the high level language is an ideal representation of the program, then these are obviously the correct operators to choose. Second, the DEL does not use conventional load/store instructions, but directly refers to objects in the source language. For example, if a program specifies a variable, the DEL-interpreter is responsible for finding the storage location of the variable and loading it into a machine register. Third, all operators and operands are aligned to 1-bit boundaries. The field size of operands changes depending on the number of objects the current scope can reference. Fields are $\log_2 N$ bits in length for a scope with N objects. For example, if a procedure references 8 variables, each variable would be represented as a 3-bit operand. The interpreter tracks scope information to know which set of variables are legal operands at any point in the program.

Flynn measured conventional machine language representations of programs and found them to be between 2.6 to 5.5 times larger than the DEL representation.

2.2.2 Custom instruction sets

Whereas Flynn used the high level language as a basis for the operators in DELs, Fraser [Fraser95] used a bottom-up approach and created macro-instructions from instructions in the compiler intermediate representation (IR). He found repeating patterns in the IR tree and used these as macro-instructions in his compressed code. The code generator emits byte code which is interpreted when executed. The overhead for this interpreter is only 4-8 KB. Fraser showed that this compression method is able to reduce the size of programs by half when compared to SPARC representation. However, the programs execute 20 times slower than the original SPARC representations.

2.2.3 BRISC

Ernst et al. [Ernst97] developed BRISC which is an interpretable compressed program format for the Omniware virtual machine (OmniVM). BRISC adds macro-instructions to the OmniVM RISC instruction set. BRISC achieves small code size by replacing repeated sequences of instructions in the OmniVM RISC code with a byte codeword that refers to a macro-instruction. Macro-instructions that differ slightly may be represented using the same codeword and different arguments. Such macro-instructions are templates that have fields which are supplied by the arguments. The argument values are located in the instruction stream after the codeword. The codewords are encoded using a order-1 Markov scheme. This allows more opcodes to be represented with fewer bits. However, decoding becomes more complicated since decoding the current instruction is now a function of the previous instruction opcode and the current opcode. When BRISC is interpreted, programs run an average of 12.6 times slower than if the program was compiled to native x86 instructions. When BRISC is compiled to native x86 instructions and executed, the program (including time for the compilation) is only 1.08 times slower than executing the original C program which has been compiled to x86 instructions.

Since the compressed program is interpreted, there is a size cost (either hardware or software) for the interpreter. If the size of the interpreter is small enough so that the interpreter and the BRISC program are smaller than a native version of the program, then this system could be useful for achieving small code size in embedded systems.

2.3 Frequency-based coding

2.3.1 Procedure Compression

Kirovski et al. [Kirovski97] describes a compression method that works at the granularity of procedures. Each procedure in the program is compressed using a Ziv-Lempel compression algorithm. A segment of memory is reserved as a *procedure cache* for decompressed procedures. On a procedure call, a directory service locates the procedure in compressed space and decompresses it into the procedure cache. The directory maps procedures between compressed and decompressed address space. For this scheme, a small map with one entry per procedure is sufficient. When there is no room in the procedure cache, a memory management routine evicts procedures to free the resource. Procedures are placed in the procedure cache at an arbitrary address. Intra-procedural PC-relative branches, the most frequent type, will automatically find their branch targets in the usual way. Procedure calls, however, must use the directory service to find their targets since they may be located anywhere in the procedure cache.

The authors obtained a 60% compression ratio on SPARC instructions. However, it is not clear if this compression ratio accounts for the directory overhead, decompression software, procedure cache management software, and the size of the procedure cache. One problem is that procedure calls can become expensive since they may invoke the decompression each time they are used. When using a 64 KB procedure cache, the authors measured an average run time penalty of 166%. When the two programs, *go* and *gcc*, were excluded from the measurement, the average run time penalty was only 11%. One appealing point of this technique is that it can use existing instruction sets and be implemented with minimal hardware support (an on-chip RAM for the procedure cache).

2.3.2 Compressed Code RISC Processor

The Compressed Code RISC Processor (CCRP) [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, decompressed, 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.

The authors report a 73% compression ratio for MIPS instructions. A working demonstration of CCRP has been completed [Benes97]. Implemented in 0.8µ CMOS, it occupies 0.75 mm^2, and can decompress 560 Mbit/s.

2.4 Procedurization

2.4.1 Procedure abstraction

Procedure abstraction [Standish76] is a program optimization for procedure oriented languages that replaces repeated sequences of common code with function calls to a single function that performs the required computation. This is an optimization that the programmer can apply to the source language. Compilers could do this with an intermediate representation or at the level of native instructions. Sequences of code that are identical, except for the values used, can be bound to the same abstracted function and supplied with arguments for the appropriate values.

2.4.2 Mini-subroutines

Liao et al. propose a software method for supporting compressed code [Liao95, Liao96]. They find *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. This method is similar to procedure abstraction at the level of native instructions, but without the use of procedure arguments.

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.

A potential problem with this compression method is that it introduces many branch instructions into a program thus reducing overall performance.

The authors report a 88% compression ratio for the mini-subroutine method and an 84% compression ratio for the call-dictionary method. Their compression results are based on benchmarks compiled for the Texas Instruments TMS320C25 DSP.

2.5 Load-time representations

2.5.1 Slim Binaries

Franz and Kistler developed a machine-independent distribution format called *slim binaries* [Franz94, Franz97]. The slim binary format is a compressed version of the abstract syntax tree (AST) in the compiler. The compression is done by using a dictionary of sub-trees previously seen in the AST. When the program is run, the loader reads the slim binary and generates native code on-the-fly. The benefit of slim binaries is that abstract syntax trees compress well so that the distribution format is very small. This reduces the time to load the program into memory. The time for code-generation is partially hidden because it can be done at the same time that the program is being loaded. Franz and Kistler have reported that loading and generating code for a slim binary is nearly as fast as loading a native binary [Franz97].

Even though the slim binary format represents programs in a very small format (smaller than 1/3 the size of a PowerPC binary), this size does not include the cost of the code generator. Slim binaries may work well to reduce network transmission time of programs, but they are not suitable for embedded systems that typically run a single program because the slim binary format is not directly executable. There is no program size benefit at run-time because a full size native binary must be created to run the program. The only size benefit is during the time the program is stored on disk or being transmitted over a network.

2.5.2 Wire Codes

Ernst et al. [Ernst97] also introduced an encoding scheme that is suitable for transmitting programs over networks. The authors compress the abstract syntax tree of the program in the following manner. First, the tree is linearized and split into separate streams of operators and literal operands. The literal operand stream is further separated into streams for each operand type. Second, each stream is move-to-front encoded. Move-to-front coding works by moving symbols to the front of the stream as they are referenced. Assuming that the symbols have temporal-locality, the indices used to address the symbols in the stream will tend to have small values. The indices are coded with a variable-length scheme that assigns short codes to the indices with small values and long codes to the indices with large values. This results in a compact representation for the frequently used symbols at the front of the stream. In the wire code, the move-to-front indices are Huffman-coded. Finally, the results are passed through the *gzip* program. They achieve very small code sizes (1/5 the size of a SPARC executable). When the program is received, it must be uncompressed and compiled before running. Therefore, this is not a representation that can be used at execution-time.

2.6 Conclusion

It is clear that there are many opportunities on several levels to reduce the size of programs. However, it is difficult to compare the results of the research efforts to date. Each study uses a different compiler, instruction set, and compiler optimizations. It is not

meaningful to say that a program was compressed 20% or 50% without knowing what standard this was measured against. Poorly written programs with no optimization may compress very well because they are full of unnecessary repetition while programs that already have an efficient encoding will seem not to compress very well. The value of the techniques presented here is that they represent a range of solutions to program representation. These techniques are not mutually exclusive – it is possible to combine them since they take advantage of different levels of representation.

Flynn assumes that the high level language is an ideal representation of a program. While particular languages are chosen to write particular programs, it is usually the case that languages are not tailored to specific programs. Languages may be good representations for a particular class of programs, but they are not are ideal representations for every program that is written in them.

One way that we approach ideal representations using existing languages is to write procedures for the operations that the language cannot express. The procedures are built using the operators of the language. Procedurization methods [Standish76, Fraser95, Fraser97, Liao95] try to discover the appropriate functions to specify that will yield a space-efficient representation. They are useful in cases where the repetition in a program is not obvious to the programmer. In addition, if the microprocessor has support for compressed code, the compression methods may be able to identify repetition at a level that cannot be represented in the high level language. For example, function prologues and epilogues are not represented in the high level language. There may be fine-grain repetition at the machine instruction level that cannot be captured with the coarse granularity of function calls.

The methods presented here that use LZ coding apply it using algorithms (*gzip* in particular) that analyze data on byte boundaries. These compression algorithms can work quite well with data that is aligned to byte boundaries, such as ASCII text. However, native instruction field boundaries are not usually byte widths. Analyzing bytes in native instructions will mix some bits from different fields and lose the semantic meaning of instruction fields. It is interesting to consider what improvements might be seen if the LZ coding algorithms accounted for the width of fields in instructions.

Chapter 3
Preliminary Work

3.1 Introduction

This chapter details an experiment to analyze one method of compression. We start with a method similar to [Liao95]. We find common sequences of native instructions in object code and replace them with a codeword. We extend this work by considering the advantages from using smaller instruction (codeword) sizes. In [Liao95], 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 show that there is a significant advantage for compressing patterns consisting of one instruction.

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

This chapter is organized as follows. First, we describe the compression method. Second, we present our experimental results. Finally, we draw conclusions about our compression method and propose how it might be improved.

3.2 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 code-

word. 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 3.1 illustrates the relationship between the uncompressed code, the compressed code, and the dictionary. A complete description of our compression method is presented in Figure 3.

3.2.1 Algorithm

Our compression method is based on the technique introduced in [Bird96, Chen97b]. 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

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.

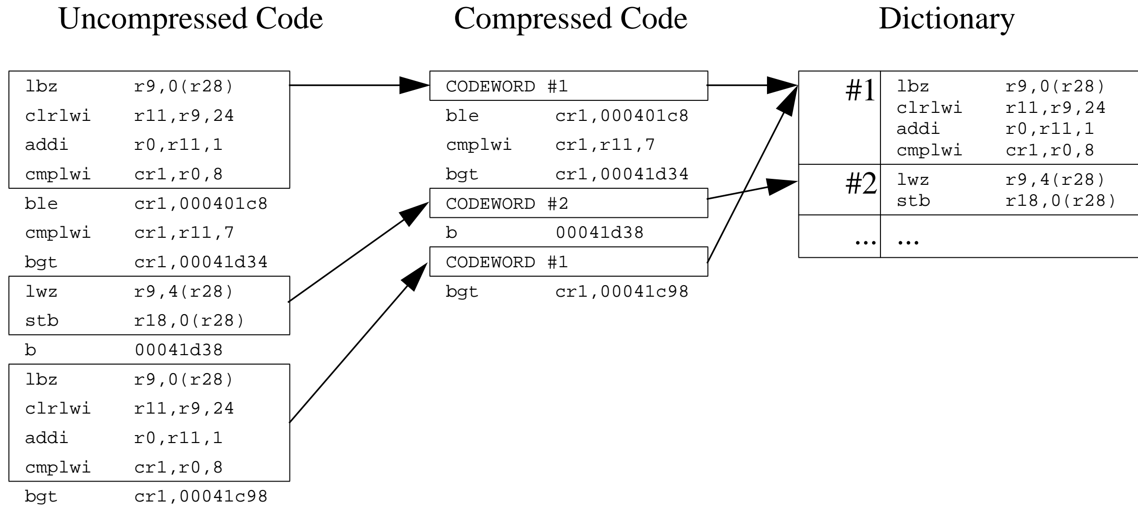

Figure 3.1: Example of compression.

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.

Encoding codewords

Encoding refers to the representation of the codewords in the compressed program. As discussed in Section 2.3.2, 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.

3.2.2 Related issues

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.

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 increase program size.

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.

3.2.3 Compressed program processor

The general design for a compressed program processor is given in Figure 3.2. 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.

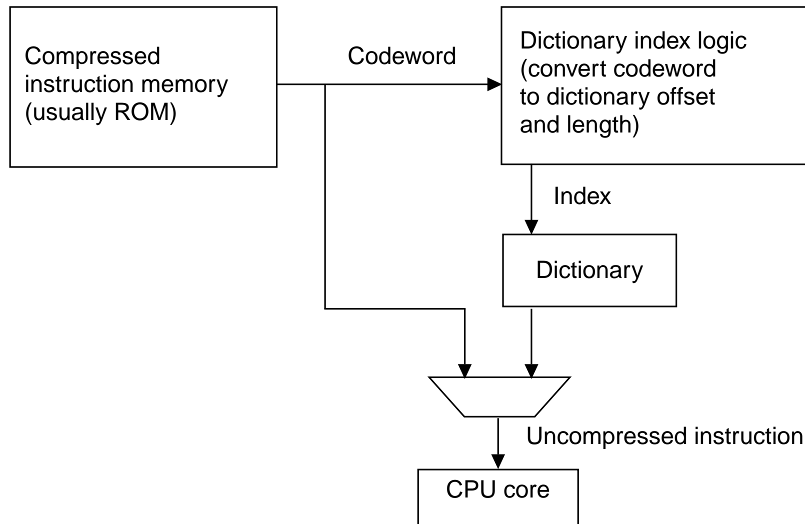

Figure 3.2: Compressed program processor.

3.3 Experiments

In this section we integrate our compression technique into the PowerPC, ARM, i386, and MIPS-16 instruction sets. For PowerPC, i386, and MIPS-16 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, i386, and MIPS-16.

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

3.3.1 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.

3.3.2 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 3.3 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 3.4 shows the contribution of each of these factors to the total savings. The size reduction due to 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 sub-

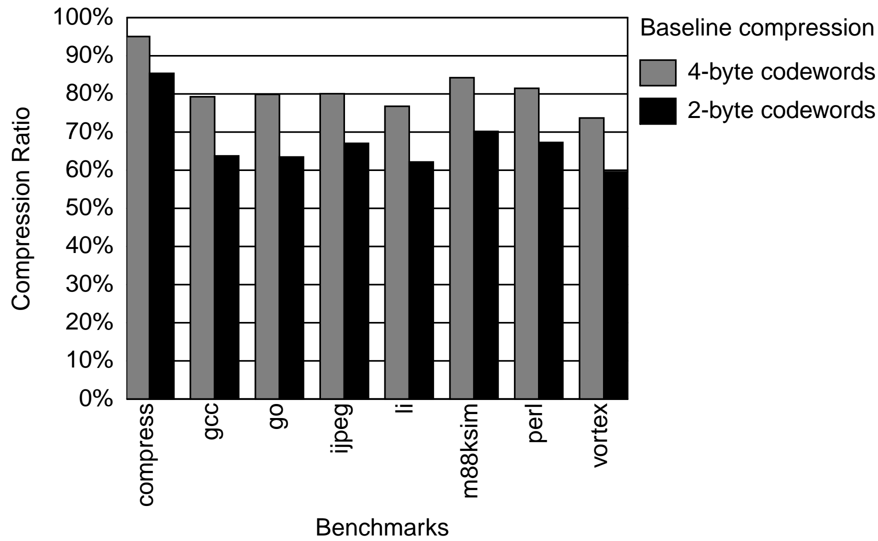

Figure 3.3: Comparison of baseline compression method with 2-byte and 4-byte codewords.

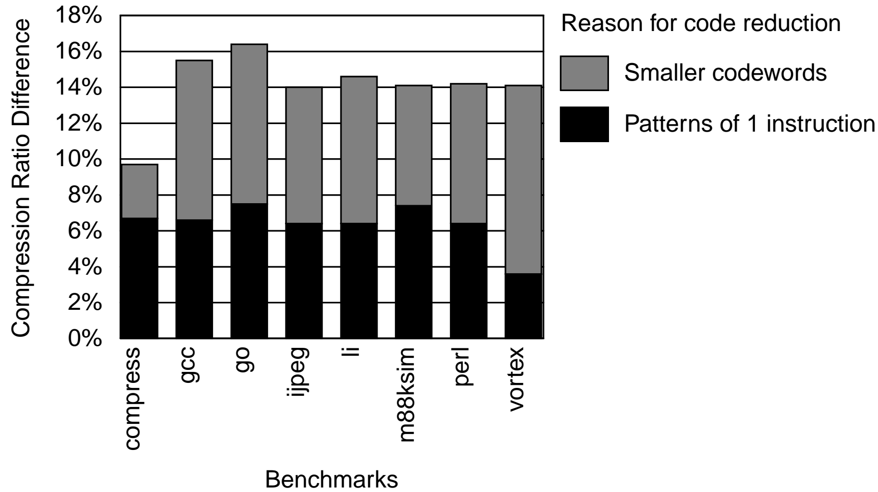

Figure 3.4: Analysis of difference in code reduction between 4-byte codewords and 2-byte codewords in baseline compression method.

tracted 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%.

3.3.3 Dictionary parameters

Our next experiments vary the parameters of the baseline method. Figure 3.5 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, dic-

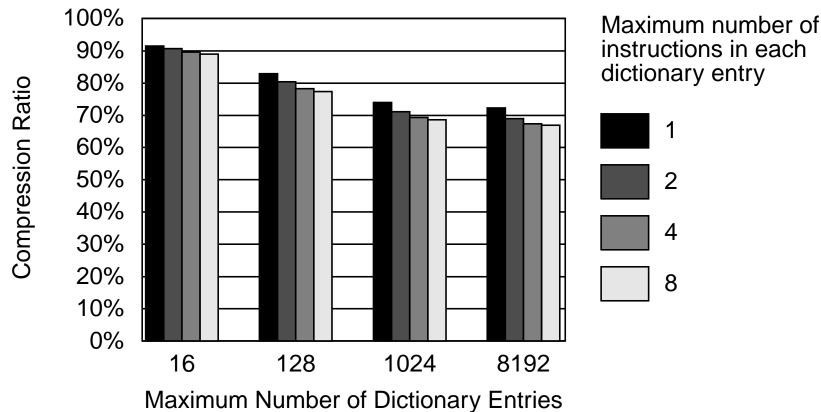

Figure 3.5: Summary of effect of number of dictionary entries and length of dictionary entries in baseline compression method.

Compression ratio is averaged over the CINT95 benchmarks.

Table 3.1: Maximum number of codewords used in baseline compression.

Maximum dictionary entry size is 4 instructions.

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

tionary entry sizes above 4 instructions do not improve compression noticeably. Table 3.1 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 num-

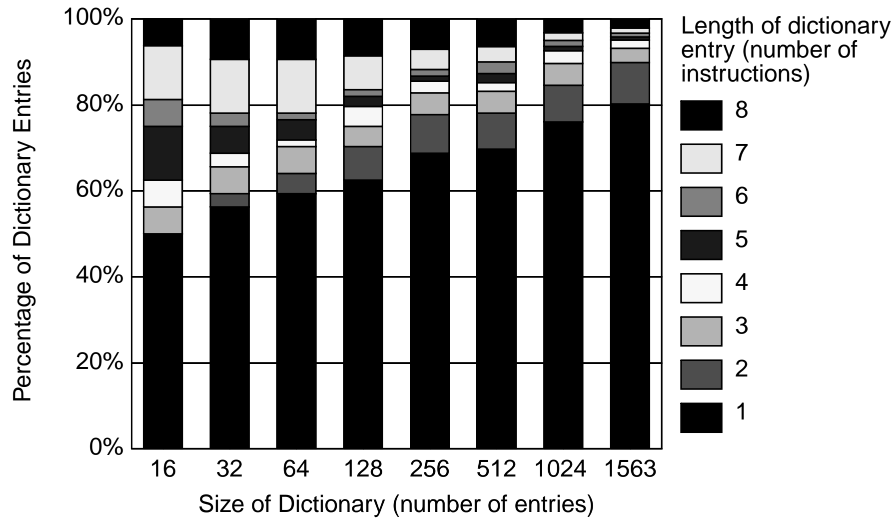

Figure 3.6: Composition of dictionary for *ijpeg.*
Longest dictionary entry is 8 instructions.

ber of codewords in the dictionary rather than increase the length of the dictionary entries. A few thousand codewords is enough for most CINT95 programs.

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 3.6 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 from 50% to 80%. The greedy algorithm tends to pick smaller, highly used sequences of instructions. This has the effect of breaking apart larger patterns that contain these smaller patterns. This results in even less opportunity to use the larger patterns. Therefore, the larger the dictionary grows, the higher the proportion of short dictionary entries it contains. Figure 3.7 shows which 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.

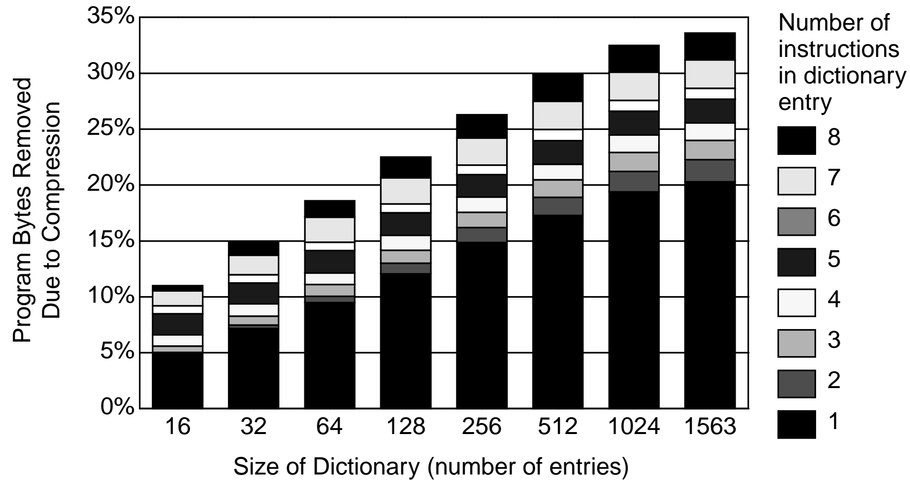

Figure 3.7: Bytes saved in compression of *ijpeg* according to instruction length of dictionary entry.

3.3.4 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 3.8 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.

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 3.9. This scheme is predicated on the observation that when an unlimited number of codewords are used, the

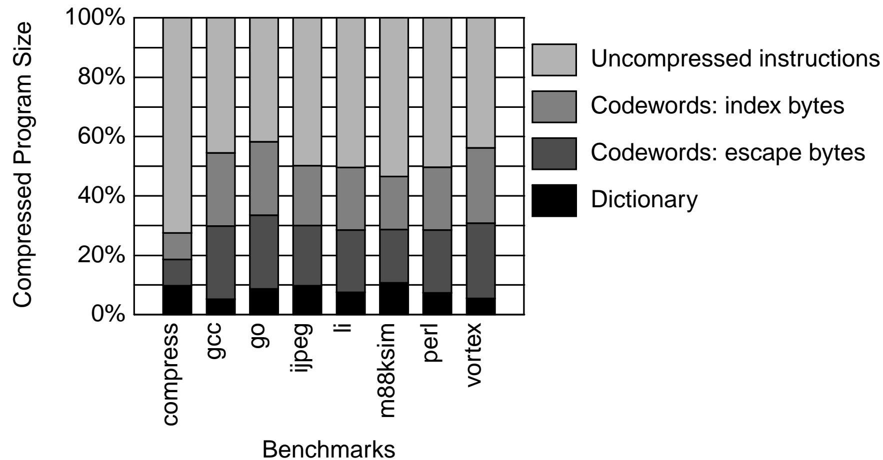

Figure 3.8: Composition of compressed PowerPC programs.
Maximum of 8192 2-byte codewords. Longest dictionary entry is 4 instructions.

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, i386, and MIPS-16 using the 4-bit aligned compression are presented in Figure 3.10. We allowed the dictionaries to contain a maximum of 16 bytes per entry. We obtained average code reductions of 39%, 34%, 26%, and 16% for PowerPC, ARM, i386, and MIPS-16, respectively. Figure 3.11 shows the average original size and the average compressed size of the benchmarks for all instruction sets. The data is normalized to the size of the original uncompressed PowerPC programs. 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 reduc-

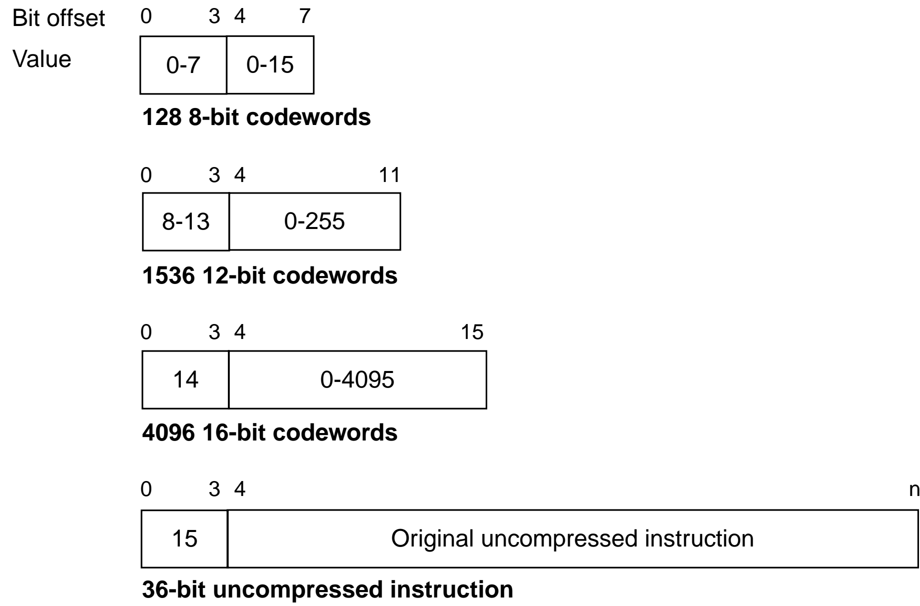

Figure 3.9: Nibble Aligned Encoding.

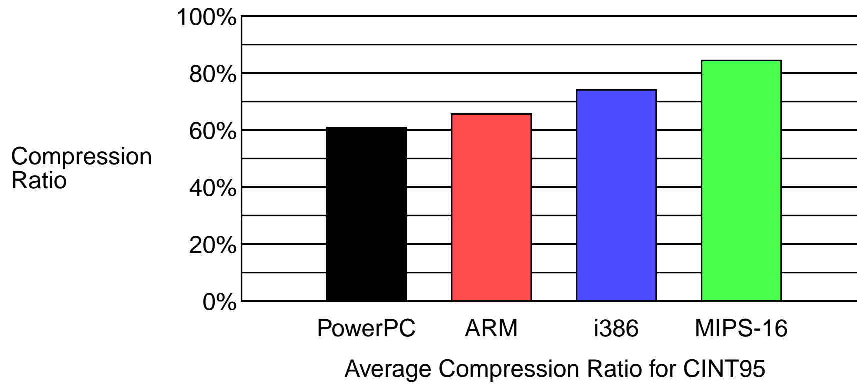

Figure 3.10: Nibble compression for PowerPC, ARM, i386, and MIPS-16 instruction sets.

tion 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 MIPS-16 programs.

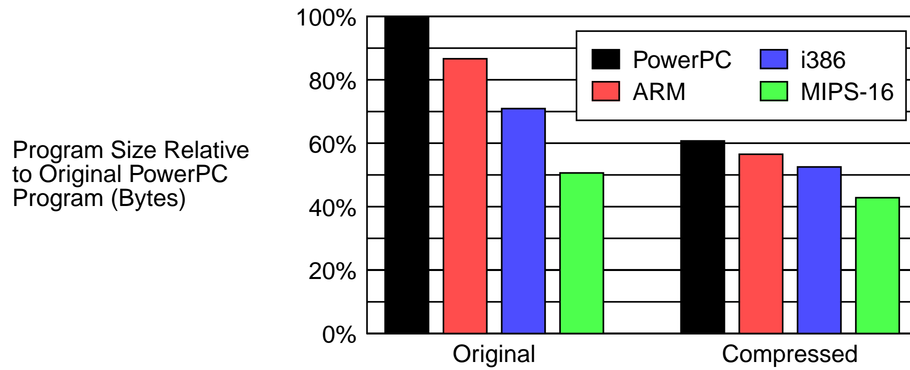

Figure 3.11: Comparison of compression across instruction sets.
All program sizes are normalized to the size of the original PowerPC programs.

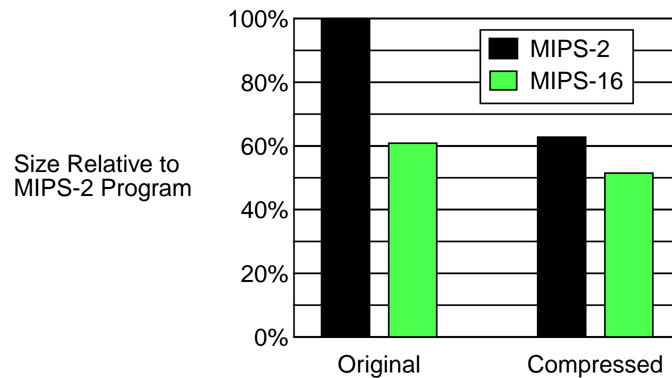

Figure 3.12: Comparison with MIPS-16.

3.3.5 Comparison to MIPS-16

In this section we compare the size improvement that MIPS-16 and nibble compression have over MIPS-2. In Figure 3.12 we show the original and compressed average sizes of the benchmarks for both MIPS-2 and MIPS-16.

For the smaller programs, MIPS-16 compression is better, while for large programs, nibble compression is better. For the large programs, nibble compression does significantly better than MIPS-16.

The reason for this is that in small programs there are fewer repeated instructions, and this causes compressible sequences to be less frequent. MIPS-16 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, MIPS-16 has the advantage. When programs are larger then there are enough repeated instructions so that the nibble compression can overcome the dictionary overhead to beat the MIPS-16 compression. Since MIPS-16 is just another instruction set, we can apply nibble compression to it. Therefore, we can always obtain a program smaller than the MIPS-16 version.

3.4 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 [Liao95]) that allows codewords to expand to several instructions. Second, we allow the codewords to be smaller than a single instruction (as in [Kozuch94]). 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. 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%, 26%, and 16% for PowerPC, ARM, i386, MIPS-16 respectively.

Our compression ratio is similar to those achieved by Thumb and MIPS-16. While Thumb and MIPS-16 are effective in reducing code size, they increase the number of static instructions in a program. We compared the CINT95 benchmarks compiled for MIPS-16 and MIPS-II using GCC. We found that overall, the number of instructions increased 6.7% when using MIPS-16. In the worst case, for the *compress* benchmark, the number of instructions increased by 15.5%. On the contrary, 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 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.

Compression is available on a per instruction basis without introducing any special instructions to switch between compressed and non-compressed code.

3.5 Improvements

There are several ways that our compression method could be improved. First, the compiler should avoid producing instructions with encodings that are used only once. In our PowerPC benchmarks, we found that 8% 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, we need an effective method to compress branch instructions. In the PowerPC benchmarks, 18% of the instructions were branches with PC-relative offsets. We did not compress these instructions in order to simplify the compression mechanism. These instructions offer an opportunity to improve compression significantly. Third, 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.

Chapter 4
Future Work

In our preliminary study, the compression took advantage of the patterns that existing compilers produce. We often found code sequences that had identical patterns of dependencies, but the values in the code were given different register names. The use of different register names caused these patterns to be mapped to separate dictionary entries. This increased the size of the dictionary and number of index bits in the codewords. We believe that we can achieve even better code sizes by modifying the compiler to generate code that is more suitable for compression.

One method to improve the problem of different register names is to guide the compiler in register allocation so that the most frequent patterns use the same register names. Another method is to "templatize" the patterns so that the registers they use can be specified in the codewords. This makes the codewords longer, but reduces the overall number of patterns in the dictionary. These two approaches can be used together. In fact, common instruction sequences which use the same register names can be viewed as matching a template with no arguments.

4.1 A template compression instruction set

Our preliminary work defines a dictionary-based compression for identical instruction sequences. The encoding allows regular instructions to mix freely with codewords. The codewords represent sequences of instructions. These sequences contain only instructions of the underlying instruction set and can be decoded in the usual fashion by the processor. Figure 4.1 shows a sample of PowerPC code from the benchmark *vortex*. The code patterns labeled A and B each occur twice and pattern C occurs only once in the program.

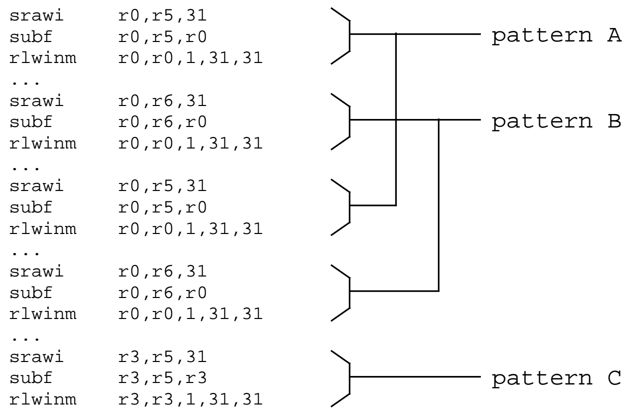

Figure 4.1: Sample PowerPC code from *vortex*.

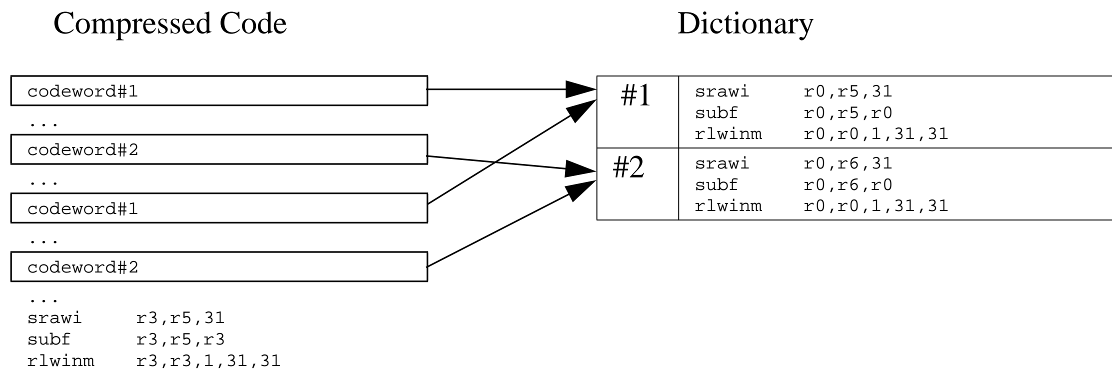

Figure 4.2: Example of dictionary-based compression.

Since all PowerPC instructions are 4 bytes long, the size of this code is 60 bytes. An example of our dictionary-based compression for this code is in Figure 4.2. The code pattern C is not put in the dictionary because it only occurs once in the program. If a sequence with a single occurrence were put into the dictionary, the overall program size would increase by the size of the codeword that replaces it. Therefore, sequence C is not compressed. Assuming that codewords are each 2 bytes long, then the size of the code and dictionary is 44 bytes. This saves 16 bytes over the original code.

Ernst et al. [Ernst97] do code compression by using codewords that represent sequences of instructions. The sequences are actually templates that specify the operation to be performed, but some register and immediate values may not be specified. These unspecified registers and immediate values are supplied by arguments that follow the

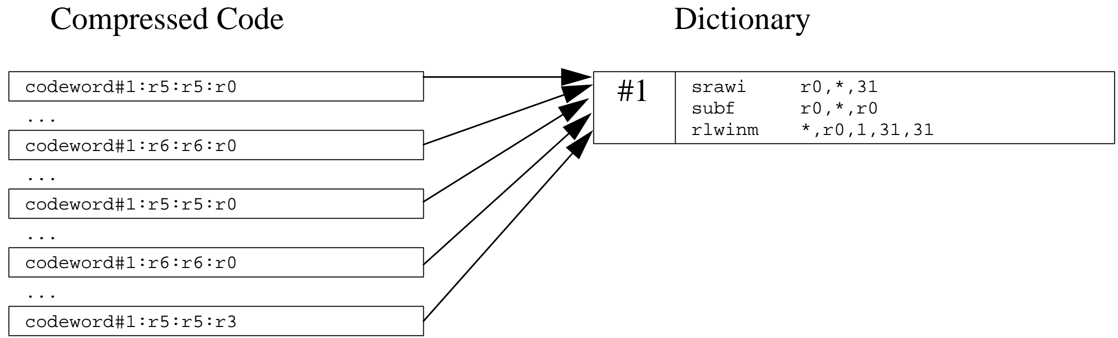

Figure 4.3: Example of template-based compression.

codeword in the program. Figure 4.3 illustrates how template-based compression would represent the sample code. Assuming that the field specifiers in the codewords each take 1 byte, then the size of the code and dictionary is 37 bytes. This is a 7 byte saving over the dictionary-based compression and a 23 byte savings over the original code. In addition, 1 dictionary entry is saved over the dictionary-based compression. Using fewer dictionary entries can help compression by decreasing the number of bits in the codeword. In addition, a small dictionary can simplify the implementation of the microprocessor.

We will adopt the template compression to define a compression-based instruction set. In the dictionary entries, instructions will have special reserved values that specify the fields that are to be filled in from the bytes following the codeword. For example, any use of register 31 could signal the decoder to use the next byte after the codeword to specify the register that should be used. The instruction decoder fetches each instruction in-order and reads bytes that follow the codeword to fill each unspecified field in the template. When the template is completely decoded, then the decoder is pointing to the next codeword or instruction in the program. Once the instructions in a dictionary entry are decoded, they can be cached so that repeated uses of this dictionary entry can be executed quickly. It may be interesting to consider operators as template fields, but for the first version of this work, we will only use operands.

4.2 Why is template-compression a good idea?

We believe that template compression will provide several benefits over instruction compression. Below are listed some reasons why template compression is a worthwhile approach:

- About 10% of the instructions in the benchmarks were incompressible instructions that only occurred once in the program. Templates could transform these specific instructions into generalized instructions that could be compressed to a single dictionary entry.
- Template compression has the effect of combining together sequences that were in separate dictionary entries in the dictionary-compression. Since the dictionary is made of many small patterns that are only used 2 or 3 times each, it could be advantageous if many entries resembled each other by using templates. Then the dictionary could be made much smaller.
- One template codeword can replace many dictionary codewords by using template fields for the incompressible instructions that are found between dictionary codewords.
- Template compression works well for large patterns that have very few varying inputs.

4.3 A template compression compiler

Template compression has been used as a means of compressing programs after the code generation step of the compiler [Ernst97]. Their code generator does not purposefully create code sequences that are suitable for a template representation. We propose to do template compression inside the compiler and let the compiler directly generate code for a compression based instruction set. Instead of merely discovering templates after code generation, we can explicitly manufacture templates by applying code transformations on the internal representation (IR) of the program. We hope these code transformations will lead to dictionaries with fewer, larger templates that can represent more code. Template compression has already been applied at the compiler IR level [Fraser95]. However, the size of templates was limited to 10 nodes in the LCC IR. We plan to let templates be as large as entire procedures and measure the resulting benefit.

The MIRV project at the University of Michigan is developing a compiler that we will use to demonstrate template compression. The IR of MIRV uses a tree structure for each source level statement. This tree structure has several properties that make it suitable for template generation. The most beneficial templates are large in size and have few fields. We expect the tree structures to contain the types of templates we desire because the tree bundles related instructions producing a single result. Some examples of the types of computations that the trees completely encode are:

- array references
- loop conditions
- function calls

The tree structure explicitly shows the value dependencies between operations. This property will be used to find repeating patterns of computation in programs.

Values with lifetimes that exist completely in the scope of the template can be assigned to any unused register name. Since these templates must work at any point in the program, a simple implementation could reserve some register names to be used only within templates. This removes all conflicts with register names outside the template that hold live values.

Some template values will have lifetimes that extend beyond the template. The template communicates with surrounding code by putting values in registers or memory. It is possible that every instance of code surrounding the template will want to use different register or memory names. These values are the prime candidates for the fields of the template.

4.3.1 Proposed Compilation Algorithm

The modifications we propose to make to the MIRV compiler involve canonically representing the statement trees, applying compression, and generating code for templates. An overview of the compilation phases is given below and illustrated in Figure 4.4. Our additions to MIRV for compression are detailed in steps 3-7 and 9.

1. Convert source code into tree IR.

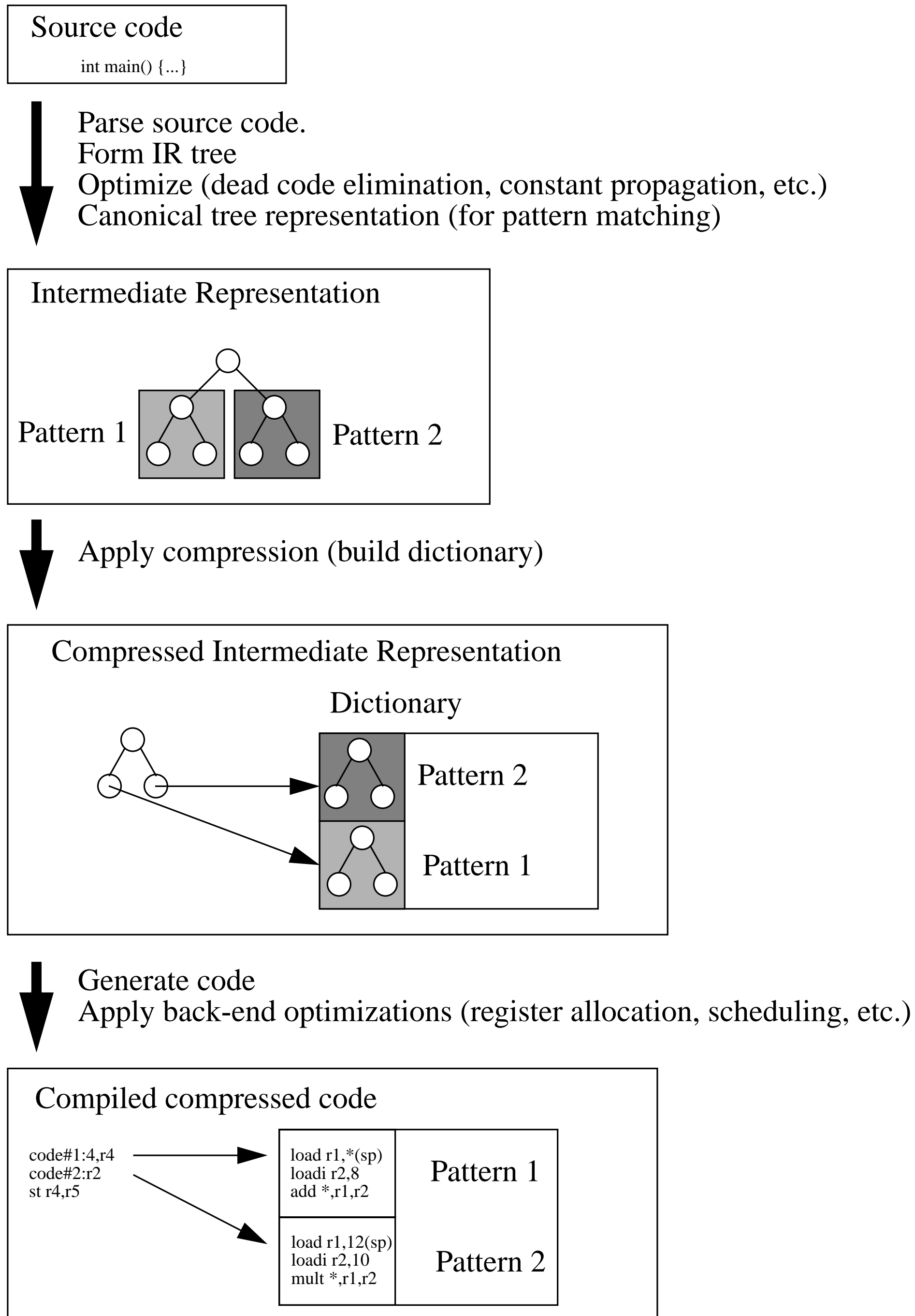

Figure 4.4: Relation between intermediate representation and compression dictionary.

2. Perform standard code size optimizations. (dead code elimination, constant propagation, etc.)

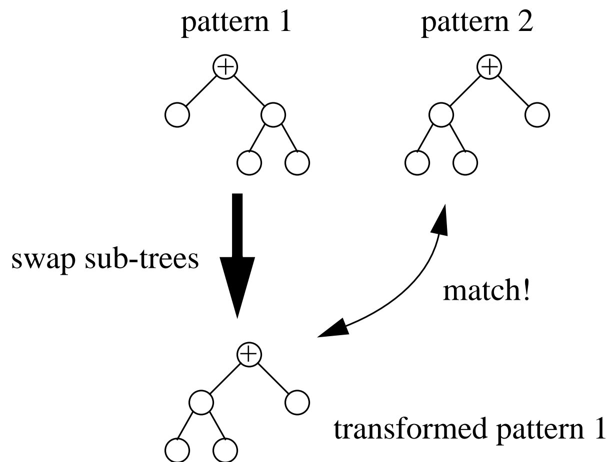

Figure 4.5: Example of canonical form for IR tree.

3. Put trees in canonical form for tree pattern matching on operations. We can borrow pattern matching techniques from common sub-expression elimination to accomplish this.

4. Enumerate all sub-tree patterns. Sub-trees are considered to be identical if the trees have the same shape, operators, and external communications. For the first implementation, we will use only complete sub-trees. For each sub-tree, consider every possible template for it concerning its immediate values.

5. Cover the program with templates picked from the list of all sub-tree patterns. These templates will be in the final dictionary.

   Templates can be chosen greedily from the list of patterns by using a cost function. The cost function will be a factor of 1) estimated code size of template 2) estimated codeword size (including template arguments), and 3) the number of uses of the template, and 4) estimated size of the original non-compressed code.

6. Allocate communication registers to the templates with the highest usage. If possible, give all instances of a dictionary template identical communication registers and specialize the dictionary entry (turn the field into a constant).

7. Allocate registers to the non-compressed code.

8. Generate object code for the program.

9. Do object code compression on the non-compressed instructions. Try to turn the object dictionary into templates for extra compression.

4.4 Template compression experiments

We wish to compare template compression against other techniques that save code size. The proposed template-compression compiler phases raise a number of questions that we hope to answer:

- What is the difference in code size between template-compression and instruction-compression? How do they interact when combined? Do they compress different types of code or the same types of code? Instruction compression on object code should be good at compressing prologue/epilogue code and patterns across abutting trees. In general, object code compression is done after code generation and may have additional instructions to work from that the tree compression did not.

- How much improvement does compression offer over standard code size optimizations? Determine which code size optimizations should be done for the compression instruction set. What is the relative value of each optimization?

- The first version of template compression will use templates that are complete sub-trees in the MIRV IR. What is the value of using templates that are partial sub-trees instead of complete sub-trees? Does this allow more of a program to be covered with templates?

- What algorithm for covering the MIRV IR with templates yields the best compression?

- Is it useful to generalize a template so that it works in more situations, but possible performs some useless work? For example, consider using a multiply-add template. It could cover abutting multiply and add instructions as well as single occurrences of multiply and add instructions. This would allow more of a program to be covered by fewer dictionary entries.

4.5 Software compression

Our preliminary study considers a hardware modification to the fetch and branch units of a microprocessor to support compression. We are also interested in exploring compression options that do not require hardware modification. This would allow compression to be widely applicable to available microprocessors. Some studies have already borrowed the concept of procedure abstraction [Fraser95, Liao95, Ernst97, Lefurgy97]. However, these previous attempts limited the size of the abstracted procedures to a few

instructions or single basic blocks. We are not aware of any study that has measured the applicability of procedure abstraction on a larger level. We propose to add procedure abstraction to the MIRV compiler as a hardware independent method of reducing code size. We will re-use the pattern matching from template compression to find candidates for procedure abstraction. These candidates will be added at the MIRV IR level as functions called with the appropriate arguments. All backends for the MIRV compiler will be able to use this code reducing technique.

4.6 Improving execution-time

Compressed programs use less memory and can result in fewer instruction cache misses [Chen97a]. If the reduction of instruction cache misses offsets the extra decoding time for compressed instructions, then there will be a performance improvement. Therefore, we plan to measure the effects of our program representations on the instruction cache.

Many of the program representations that we reviewed obtained smaller programs at the cost of execution speed. Some previous studies [ARM95, Fraser95, Liao95] have proposed to balance this trade-off by only applying their representations to the portions of the program that are not frequently executed. This allows the highly executed parts of the program to proceed at the usual execution speed. However, we are unaware of any study that performs this experiment. We propose to modify the MIRV compiler so that we may choose the sections of a program that may be compressed. This will allow us to measure the execution benefit of selective compression.

It may also be possible to use template compression to improve execution time. Function memoization is a code optimization that caches the results of recent function invocations. The next time that the function is called with the same arguments, the result can be quickly provided from the cache instead of computing it [Abelson85]. Sodani [Sodani97] presents a method of improving execution time by memoizing individual machine instructions. He keeps the results of instruction execution in a table and uses these results if the instructions are encountered again with the same source values. This avoids re-executing instructions when the result will be identical to a previous result. It

may be possible to use templates in a similar manner. Caching the result of templates is potentially more valuable than caching the results of individual instructions because templates represent more work. We propose to study the patterns of source values and result values in templates to see if they are useful in avoiding re-execution.

4.7 Conclusion

In this thesis proposal, we have reviewed several methods of program representation to improve space efficiency. We have proposed a representation based on data compression techniques that incorporates elements of previous studies. A preliminary study shows that this representation is competitive with other methods in terms of program size. Additionally, this method allows greater access to the operations of the underlying processor than previous methods. Based upon this preliminary study, we proposed templates as an improved program representation. We have presented a course of research that includes comparison of the template representation with previous methods and exploration of how it might improve microprocessor performance.

Bibliography

[Abelson85] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass., 1985.

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

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

[Benes97] M. Benes, A. Wolfe, S. M. Nowick, "A High-Speed Asynchronous Decompression Circuit for Embedded Processors", *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[Bird96] P. Bird and T. Mudge, *An Instruction Stream Compression Technique*, Technical report 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*, Technical report 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.

[Ernst97] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression", *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, June 1997.

[Flynn83] M. J. Flynn and L. W. Hoevel, "Execution Architecture: The DELtran Experiment", *IEEE Transactions on Computers*, Vol. C-32, No. 2, February 1983.

[Franz94] M. Franz, *Code-Generation On-the-Fly: A Key for Portable Software*, PhD dissertation, Institute for Computer Systems, ETH Zurich, 1994.

[Franz97] M. Franz and T. Kistler, "Slim binaries", *Communications of the ACM*, 40(12):87–94, December 1997.

[Fraser84] C. W. Fraser, E. W. Myers, A. L. Wendt, "Analyzing and Compressing Assembly Code", *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No. 6, June 1984.

[Fraser95] C. W. Fraser, T. A. Proebsting, *Custom Instruction Sets for Code Compression*, unpublished, http://www.cs.arizona.edu/people/todd/papers/pldi2.ps, October 1995.

[Kirovski97] D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure Based Program Compression", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

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

Technical report, Silicon Graphics MIPS Group, 1997.

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

[Lefurgy97] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[Liao95] S. Liao, S. Devadas, K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques", *Proceedings of the 15th Conference on Advanced Research in VLSI*, March 1995.

[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.

[Sodani97] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

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

[Standish76] T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors, *The Irvine Program Transformation Catalogue*, Department of Information and Computer Science, University of California, Irvine, January 1976.

[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.

[Turley95] J. L. Turley. Thumb squeezes arm code size. *Microprocessor Report*, 9(4), 27 March 1995.

[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.