Efficient Execution of Compressed Programs

by

Charles Robert Lefurgy

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2000

Doctoral Committee:

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

There is a right physical size for every idea.

— Henry Moore (1898-1986). English sculptor.

To my parents, Clark and Sarah.

Acknowledgments

Trevor Mudge served as my advisor during the course of my graduate studies. His persistent question-asking helped shape the course of my research.

Eva Piccininni and Ed Kohler developed an ARM microprocessor that supports software-assisted decompression based on the results of my research. Eva wrote the first software version of our CodePack-like compression/decompression software. She also performed many experiments for optimizing hybrid programs in compressed code systems. Eva was especially helpful as a coauthor on our publications which presented the initial research results of this dissertation. Ed developed the first hardware description of a microprocessor with instruction set extensions for executing compressed programs. In addition, he mapped the design onto an FPGA for running large application simulations. Ed and Eva were both helpful in implementing and critiquing initial versions of the software decompression method that is the basis for much of this dissertation.

A significant amount of my time in graduate school was spent working with the MIRV team to build a C compiler. Although I did not ultimately use this compiler for my dissertation research, working on that project has broadened my understanding of computer architecture. I thank the members of the MIRV team, Matt Postiff, David Greene, David Helder, and Kris Flautner, for answering the countless questions I had about compilers.

Finally, I would like to thank my parents, Clark and Sarah, who supported me throughout my academic endeavors. My father sparked my interest in computers when he presented me with my first programming book at age eight.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Embedded microprocessors are highly constrained by cost, power, and size. This is particularly true for high-volume, low-margin consumer applications. Reducing microprocessor die area is an important way to save cost because it not only allows more dies to be put on a wafer, but it can vastly improve die yield. For control-oriented embedded applications, the most common type, a significant portion of the die area is used for program memory. Therefore, using smaller program sizes implies 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].

Research contribution

Both low-cost embedded systems and high-performance microprocessors can benefit from small program sizes. This dissertation focuses on program representations of

embedded applications, where execution speed can be traded for improved code size. We examine *code compression* methods which reduce program code size by using data compression techniques. In general, the usual techniques of text compression, typified by Ziv-Lempel compression, cannot be directly applied to programs because they require that the complete program be compressed or decompressed at once. If a program cannot be incrementally decompressed then the operation of decompressing the complete program will require more (temporary) memory than the uncompressed program — defeating the original reason for employing compression. Incremental decompression avoids this problem. However, incremental decompression requires that information about program labels (jump and call targets) be recoverable incrementally too.

Previous research in this area has suggested that compressed code systems will execute programs slower than native code systems, especially when the decompression is done in software. The primary goal of this research is to demonstrate that compressed programs can still execute with a high level of performance. First, we shows that even simple instruction-level compression methods can attain interesting levels of compression. This is important for building fast decompressors that will operate with high performance. Second, we show a hardware decompression method that can often eliminate the decompression penalty and sometimes execute compressed programs faster than native programs. Third, we examine the performance of decompression in software. Software decompression is interesting because it reduces the die area and cost of embedded microprocessors while allowing greater design flexibility. However, the decompression overhead is extremely high compared to hardware decompression. We present methods that greatly reduce the software decompression overhead and demonstrate that compressed multimedia applications execute with nearly the same performance as native programs.

This chapter introduces data compression and its application to program compression. It concludes with an overview of the organization of this dissertation.

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 use a significantly different set of assumptions in order to retain acceptable levels of performance.

First, programs are small enough to fit in storage, so the compressor can optimize the final compressed representation based on the entire program instead of using only recent history information. However, 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. Therefore, programs must use incremental decompression. As the program executes, units of it are decompressed and executed. As more of the program is decompressed, previously decompressed units must be discarded due to system memory constraints. The unit of compression could be a procedure, a cache line, or an individual instruction. This allows the decompressed unit and the complete compressed program to use less memory than the original native code program.

Second, since decompression will occur as the program is executing, it is desirable to begin decompression at arbitrary points in the program. This is important to consider when choosing the unit of incremental decompression. For example, since program execution can be redirected at branch instructions it would be ideal for the decompression to begin at any branch target. This effectively splits the program into blocks of instructions defined by branch targets. The unpredictable nature of the execution path between program runs is likely to constrain the length of history information available to the compressor.

The third assumption about compressed programs is that a decompressed block may not be entirely executed due to branch instructions that cause early exit from the block. While larger block sizes may improve performance of history-based compression methods, they will also decrease program performance by causing time to be wasted decompressing instructions that are not executed. Therefore, techniques such as Ziv-Lempel compression which rely on the recent history information of large compression buffers may be unsuitable for compressing programs that require high-performance or high levels of compression. Unfortunately, smaller block sizes may also decrease program performance. Small blocks require the decompressor be invoked more frequently. This can be detrimental to performance if the invocation overhead is significant.

Fourth, most microprocessors have alignment restrictions which impose a minimum size on instructions. For example, compressors may restrict their encodings to begin on byte boundaries so that the decompressors can quickly access codewords. This would require the use of pad bits to lengthen the minimum size of codewords.

Finally, an advantage that programs have over typical data is that portions of the program (statements, instructions, etc.) can be rearranged to form an equivalent program. The code-generation phase of a compiler could assist in generating compression-friendly code. This may allow the compressor to find more compressible patterns.

1.2 Text compression

Lossless compression or text compression refers to a class of *reversible* compression algorithms 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, which are linear sequences of instructions. Surveys of text compression techniques have been written by Lelewer and Hirschberg [Lelewer87] and Witten et al. [Witten90]. Compression algorithms that are not lossless are called lossy. These algorithms are used for compressing data (typically images) that can tolerate some data loss in the decompressed message in exchange for a smaller compressed representation. Since computer programs must be executed without ambiguity, lossy compression is not suitable for them.

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:

$$\textit{compression ratio} = \frac{\textit{compressed size}}{\textit{original size}} \qquad \textit{(Eq. 1.1)}$$

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 , 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% 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 Scope

This dissertation only addresses the problem of compressing instruction memory. Some embedded applications require much more data memory than instruction memory. For such applications, compressing data memory may be more beneficial. However, the characteristics of program data are highly application specific. No single compression algorithm is suitable for all types of data. The data size problem is has been partly addressed by the wide variety of application specific data compression formats available (JPEG, MPEG, gzip, etc.). On the other hand, the program code in different applications tends to look similar because the same language, the instruction set, is used to express the

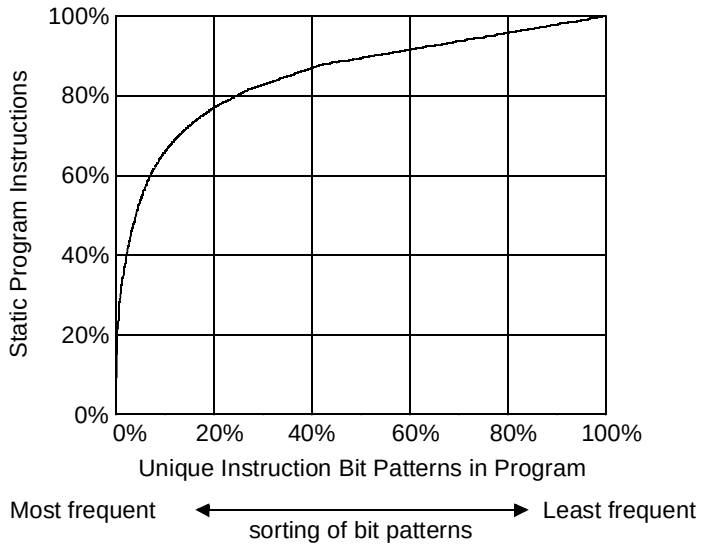


Figure 1.1: Instruction bit patterns
This graph shows the unique instruction bit patterns in a program as a percentage of static program instructions. The data is from the *go* benchmark compiled for PowerPC. The x-axis is sorted by the frequency of bit patterns in the static program.

program. Therefore, a single compression algorithm can compress a range of applications quite well.

Another important issue for embedded systems is power dissipation. Since a compressed program may trade an improved size for a longer running time, it is possible that it will dissipate more power than a larger program that runs for a shorter time. However, requirements for embedded systems vary widely. A cell phone must run on batteries, but a printer is plugged into an outlet. The suitability of code compression depends on the underlying requirements of the embedded system. Regardless, embedded systems can always benefit from lower manufacturing costs made possible by using smaller microprocessor dies. Therefore, the focus of this dissertation is how to utilize die area to execute compressed programs efficiently.

1.5 Organization

The organization of this dissertation is as follows. Chapter 2 reviews previous work to obtain small program sizes. This chapter provides important background for understanding the remainder of the dissertation. Chapters 3, 4 and 5 are largely independent and can be read in any order. Chapter 3 contains the results of a preliminary experi-

ment that applies text compression techniques to programs at the instruction level. Chapter 4 provides an in-depth study of using hardware-managed decompression. Chapter 5 implements decompression in software. In some applications, the overhead of decompression is comparable to that in hardware-managed decompression. However, many applications experience considerable slowdown. This slowdown is addressed in Chapter 6 which presents optimizations for software decompression. Finally, Chapter 7 concludes the dissertation and suggests possible future work.

Chapter 2
Background

This chapter surveys previous program compression techniques. Previous methods have performed compression in the compiler, assembler, linker, or in a separate post-compilation pass. Most program compression systems do compression only during initial program creation and do not re-compress the program as it is running. Decompression can be performed on individual instructions immediately before execution, when a procedure is called, in the memory system as instructions are fetched, or when the program is loaded into memory. Because the point at which decompression occurs varies widely among systems, we organize our presentation according to the decompression methods. Some of the questions to consider when comparing compression schemes are:

- When is compression done? (In compiler or post-compilation pass?)
- What is the compression algorithm? (Ziv-Lempel, arithmetic, etc.)
- What is compressed? (Instructions, data, or both?)
- When does the system detect that decompression must occur? (During a cache miss, during a procedure call, etc.)
- How are instructions decompressed? (With software or hardware assistance?)
- Where is decompressed code stored? (In the instruction cache or main memory?)
- How is the decompressed code managed? (How much decompressed code is available at once? What gets replaced when more instructions are decompressed?)

This chapter begins with an overview of general techniques that are found in many compressed code systems. This is followed by a discussion of individual compressed code systems organized by decompression method. The chapter ends with a discussion on the role of compiler optimizations in producing small programs.

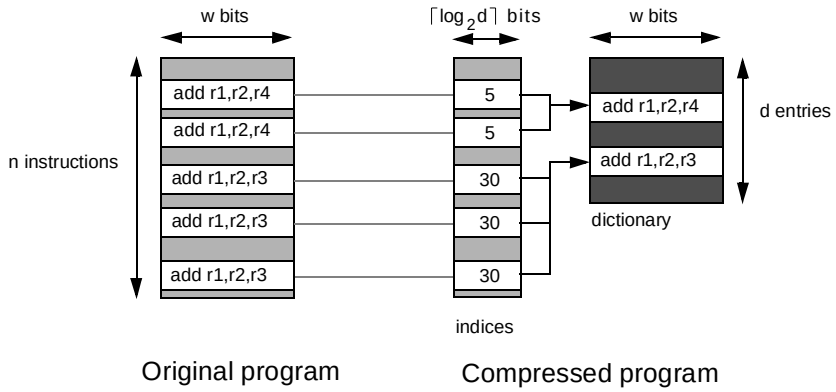


Figure 2.1: Dictionary compression

2.1 General code compression techniques

This section reviews some common techniques that many compressed code systems use.

2.1.1 Dictionary compression

Dictionary compression uses a dictionary of common symbols to remove repetition in the program. A symbol could be a byte, an instruction field, a complete instruction, or a group of instructions. The dictionary contains all of the unique symbols in the program. Each symbol in the program is replaced with an index into the dictionary. If the index is shorter than the symbol it replaces, and the overhead of the dictionary is not large, compression will be realized. Figure 2.1 shows an example of dictionary compression where the symbols are entire machine instructions.

In order for compression to be achieved, Equation 2.1 must hold.

$$nw > n\lceil \log_2(d) \rceil + dw \qquad \textit{(Eq. 2.1)}$$

In this equation, n is the number of static instructions in the program, w is the number of bits in a single instruction, and d is the number of symbols in the dictionary. Of course, this equation does not account for the size of specific implementations of the decompressor which also uses microprocessor die area.

If the dictionary contains all the symbols required to reconstruct the original program, then it is said to be complete. If the dictionary is not complete, then an escape mechanism is necessary to tell the decompressor not to use the dictionary, but instead to interpret some raw bits in the compressed program as symbols for the decompressed program. This will result in some expansion of the compressed program. Using some raw bits in the compressed program prevents the dictionary from growing very large and holding symbols that do not repeat frequently in the original program.

Lefurgy et al. [Lefurgy97] use dictionary entries that each hold one or more instructions. Taunton [Taunton91] divides 32-bit instructions into 24-bit and 8-bit fields. The 24-bit field is encoded with its dictionary entry and the 8-bit field is left raw. This is done because the repetition of the entire 32-bit instruction is lower than the 24-bit field and would cause the dictionary to be very large. Citron considers a similar encoding for 32-bit buses between the CPU and main memory [Citron95].

2.1.2 Difference encoding

A table of values can be compressed by using *difference* or *delta encoding*. The first value is coded explicitly. The next value is encoded as the difference between it and the first value. The remaining values are encoded as the difference between them and the previous value. When the difference between values is small (such as in a sorted list), then the difference can be encoded in fewer bits than the original values. This has been used to compress decoding tables within compressed programs [Taunton91, Wolfe92, IBM98].

2.1.3 Data packing

Codewords can have sizes that do not align to machine-word boundaries. This slows down access to them. Therefore, fields within codewords are often packed in pairs to improve data alignment. For example, consider using 12-bit codewords that consist of a 4-bit control nibble followed by an 8-bit index. Two codewords might be packed together so that the first byte contains both 4-bit nibbles, the second byte contains the first index, and the third byte contains the second index. On a byte-aligned processor, this allows the decompressor to use a natural byte load instruction to access each part of the code word.

A second reason to use data packing is for speed of parsing and simplification of decompression logic. CodePack [IBM98] uses data alignment to quickly parse codewords in the compression stream. Two control fields of either 2 or 3 bits each are followed by two variable-length index fields. The second control field always starts at either the third or fourth bit of the stream, so it is easy to find. If the control and index fields alternated, there would be a much larger range of possibilities for the starting position of the second control field. This would have complicated the decompression logic. However, since the control fields are easy to find and tell the decompressor how long the index fields are, the following codewords can quickly be located in the compressed stream. Data packing may enable multiple codewords to be identified and decompressed in parallel if the time to find codewords is much shorter than the time to decompress the codewords. Taunton mentions the possibility of breaking variable-sized fields in codewords into separate streams to improve load and buffering efficiency [Taunton91]. This allows a single load to be used for each data item, rather than constructing larger items from individual byte loads due to misalignment.

2.2 Decompression of individual instructions

This section reviews decompression techniques that target individual instructions.

A conventional microprocessor must fetch and decode an instruction before it is executed. The decoding process expands the native instruction word into many control signals within the microprocessor. Thus, the encoding of the instruction set can be considered a form of compression since the number of control signals is typically greater than the number of bits in the instruction word. In this case, the decompression of the instruction is done by the hardware decode stage of the microprocessor.

The decoding of an instruction may also be done in software by an interpreted-program system. In this case, the application program is written in a highly encoded instruction set that the underlying processor cannot decode or execute. Instead, an interpreter program written in the native instruction set fetches and decodes the instructions in the application. The interpreter is responsible for executing a series of native code instructions

to emulate the behavior of the interpreted instruction. These native code instructions are then decoded in the normal manner on the microprocessor.

These decompression techniques described below are divided into two categories. The first category covers the design of instruction sets which are decoded in hardware by the decode stage of a conventional microprocessor. The second category covers interpreted-program environments which decode instructions in software. Both systems decode (or decompress) each instruction individually.

2.2.1 Instruction set design

In our experiments, we have observed that the size of programs encoded in conventional instruction sets can differ by a factor of two. This shows that instruction set design is important to achieve a small program size. Bunda et al. [Bunda93] studied the benefit of using 16-bit instructions over 32-bit instructions for the DLX instruction set. 16-bit instructions are less expressive than 32-bit instructions, which causes the number of instructions executed in the 16-bit instruction programs to increase. They report that the performance penalty for executing more instructions was often offset by the increased fetch efficiency.

Dual-mode instruction sets

The work of Bunda et al. can be extended to a microprocessor that utilizes both a 32-bit instruction set and a 16-bit instruction set. A control bit in the processor selects the current instruction set used to decode instructions. This allows a program to have the advantages of wide, expressive instructions for high-performance and short instructions for code density. ARM and MIPS are examples of such dual-mode instruction sets. Thumb [ARM95, Turley95] and MIPS-16 [Kissell97] are defined as the 16-bit instruction set 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 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 instructions 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 eight 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].

Custom encodings

A somewhat different approach was introduced by Larin and Conte [Larin99]. They assume that the embedded processor can be optimized to a specific program. They use the compiler to generate both the compressed program and a custom programmed logic array to fetch and decompress it. One technique they use is to customize the instruction set for the program by shortening instruction fields that are too wide. For example, if only eight registers are used in the program, then the register fields can be shortened to 3 bits. Another technique they use is to Huffman code whole instructions, fields within instructions, or bytes of the instructions. The trade-off here is that as the compressed program becomes smaller, the Huffman decoder becomes larger.

One interesting note is that the compiler removes rare instructions and replaces them with a group of equivalent common instructions that map to shorter Huffman codes. Even though this form of strength-reduction lengthens the final compressed program, it makes the decoder implementation easier by reducing the maximum codeword size that must be decoded.

2.2.2 Interpretation

Interpreted-program environments are another method to attain small code size [Klint81]. Typically, application source code is compiled to a space-efficient 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. Typical interpreted programs for 32-bit instructions have speeds 5-20 times slower than native code and are up to 2 times smaller [Fraser95, Ernst97]. Interpreted code for the TriMedia VLIW processor is 8 times slower than native code and is 5 times smaller [Hooger99].

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 a 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

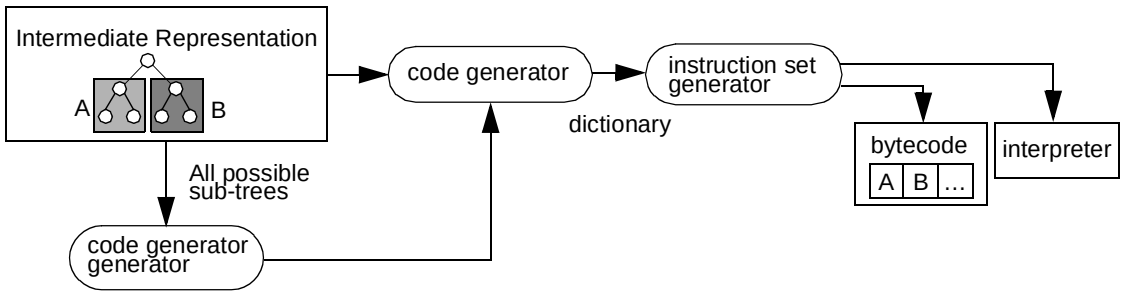


Figure 2.2: Custom instruction sets

current scope can reference. Fields are $\log_2 N$ bits in length for a scope with N objects. For example, if a procedure references eight 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.

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. This process is illustrated in Figure 2.2. 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 programs. However, the programs execute 20 times slower than the original SPARC programs.

BRISC

Ernst et al. [Ernst97] developed BRISC which is an interpretable compressed program format for the Omniware virtual machine (OmniVM). The compression method is illustrated in Figure 2.3. BRISC adds macro-instructions to the OmniVM RISC instruction set. BRISC achieves small code size by replacing repeated sequences of instructions in the

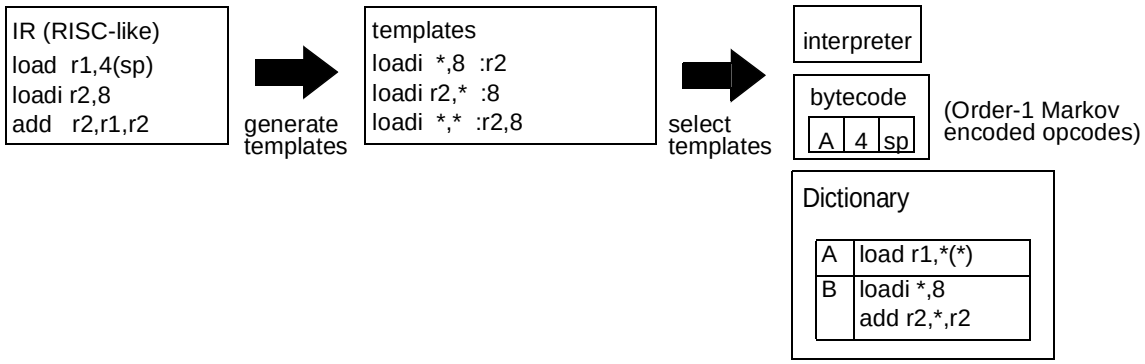


Figure 2.3: BRISC

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 code-word. 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.

More complicated compression algorithms have combined operand factorization with Huffman and arithmetic coding [Lekatsas98, Araujo98].

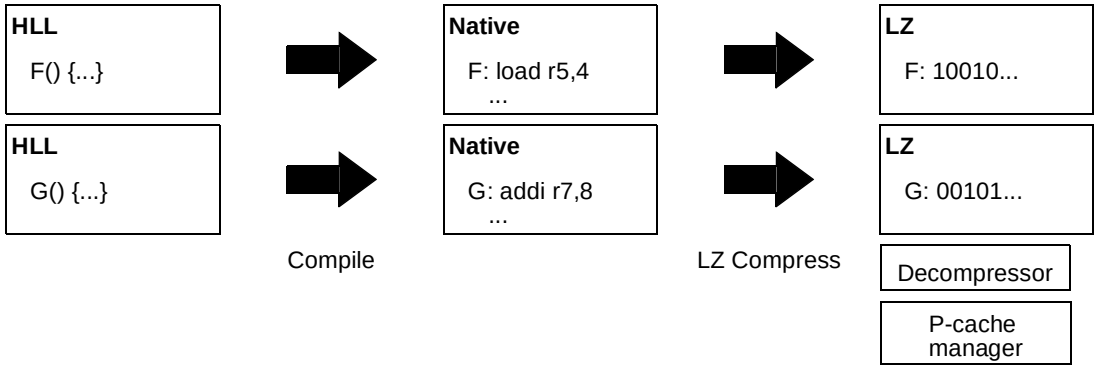


Figure 2.4: Procedure compression

2.3 Decompression at procedure call

Kirovski et al. [Kirovski97] describe a compression method that works at the granularity of procedures. Figure 2.4 illustrates procedure compression. 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. One appealing point of this compression technique is that it can use existing instruction sets and be implemented with minimal hardware support (an on-chip RAM for 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

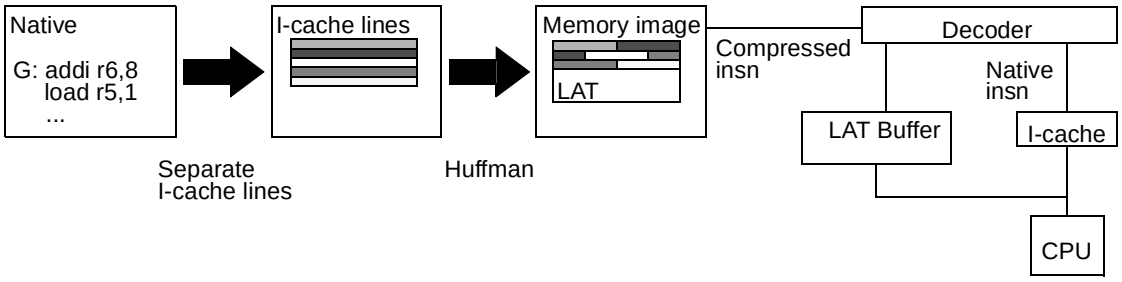


Figure 2.5: Compressed Code RISC Processor

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 possible reason for the high overhead is that whole procedures are decompressed, but few instructions are executed in the procedure. Another reason is that executing procedure calls may cause the caller to be evicted from the procedure cache before it has finished. In this case, when the callee exits, the caller must be decompressed again. A final reason for slow execution is that managing fragmentation in the procedure cache may be expensive.

2.4 Decompression within the memory system

Decompression within the memory system hides the process of decompression from the microprocessor core. This allows the system to reuse the original core design without modification. Decompression is triggered with the microprocessor fetches instructions. From the microprocessor's perspective, decompression appears as a long latency memory access.

2.4.1 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. The CCRP system is illustrated in Figure 2.5. 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 original 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 authors report a 73% compression ratio for MIPS instructions. A working demonstration of CCRP has been completed [Benes97, Benes98]. Implemented in 0.8μm CMOS, it occupies 0.75 mm^2, and can decompress 560 Mbit/s.

Lekatsas and Wolf explore other compression algorithms for CCRP [Lekatsas98]. Their SAMC (Semi-adaptive Markov Compression) algorithm combines a Markov model with an arithmetic coder. Each instruction is partitioned into small blocks. Since each instruction is partitioned the same way, blocks with the same location in each instruction define a stream. All streams are compressed separately. This attains a 57% compression ratio on MIPS programs. Their SADC (Semi-adaptive Dictionary Compression) algorithm creates a dictionary of the instruction fields (opcode, register, immediate, and long immediate). In addition, the dictionary contains entries consisting of common combinations of adjacent fields (such as an opcode and a register). The dictionary indices are Huffman encoded. This attains a 52% compression ratio on MIPS. However, the price of better compression is a more complicated decompressor.

2.5 Decompression at load-time

This section describes systems that decompress programs when they are loaded over a network or from a disk. These techniques are not directly applicable to memory-limited computers because at execution time the compressed programs are expanded to full-size native programs. Although these techniques do not save space at execution time, they provide some of the smallest program representations.

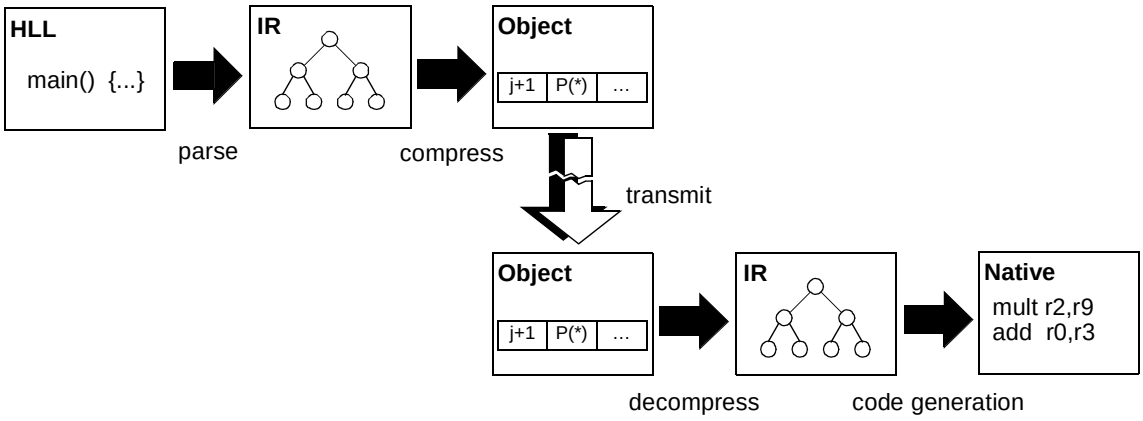


Figure 2.6: Slim binaries

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 process is illustrated in Figure 2.6. 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 from disk 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 reported that loading and generating code for a slim binary is nearly as fast as loading a native binary.

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 size 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.

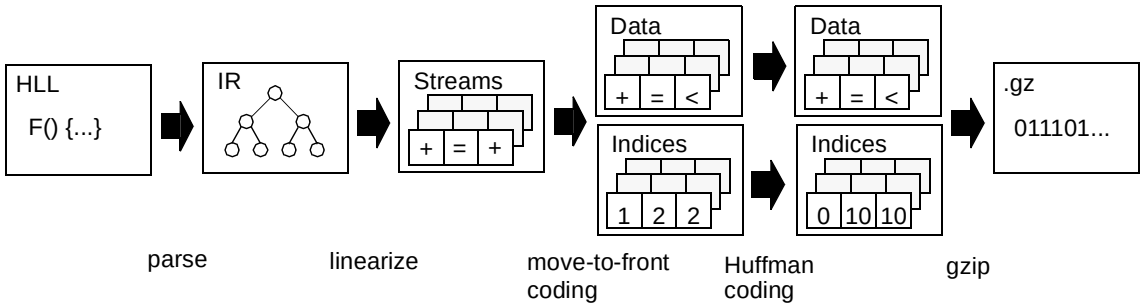


Figure 2.7: Wire code

2.5.2 Wire codes

Ernst et al. [Ernst97] also introduced an encoding scheme that is suitable for transmitting programs over networks. Figure 2.7 illustrates how it works. 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 decompressed and compiled before execution. Therefore, this is not a representation that can be used at execution-time.

2.5.3 RISC iX

Taunton describes the use of dictionary compression for a low-cost computer using the ARM processor [Taunton91]. Each page of text and data in the executable is compressed. The decompressor is integrated with the operating system and each page is

decompressed as it is demand loaded. The compression algorithm saves 45-50% disk space on a RISC iX system and causes programs to load 40% faster from disk. Taunton notices that compressed programs load faster over the network on diskless machines than native code. In addition, the RISC iX virtual memory system does not place text pages in the swap area. Instead, it reloads them through the file system and decompresses them again. This performance penalty was necessary to reduce disk space in a low-cost workstation. Taunton also notes that shared libraries are a form of code compression for saving space on disk and memory.

2.5.4 File system

Techniques used to save space in the file system compress both instructions and data. These techniques could be useful in an embedded system that uses a file system in a long-term, nonvolatile memory, for instance.

Cate and Gross

Cate and Gross [Cate91] investigate using the filesystem to automatically compress the least recently used files and decompress them when the user accesses them. Using a PPM (prediction by partial matching) compression algorithm resulted in a compression factor of 3. They compressed the least recently used 75% of files and kept other files in their original non-compressed format. This resulted in a typical decompression time of 50 seconds per user per day. Thus, compression can double the available disk storage while causing minimum performance degradation.

Coffing and Brown

Coffing and Brown [Coffing97] augmented the Linux filesystem to support automatic compression of files with gzip. Their system is based on that of Cate and Gross, but uses different compression policies. First, they divide files into separate units and have the ability to compress each unit independently. This improves random access time to large files. When part of a file is accessed, the remaining units are speculatively decompressed assuming that the user will want access to other sections of the file. Very large files are never fully decompressed to avoid filling the disk. Small files whose compression may not save at least one disk block are not compressed. One interesting problem the authors found

was that file fragmentation increased significantly as files were decompressed and re-compressed.

Compression cache

Compression has also been used to improve virtual memory management. Douglis [Douglis93] modified the Sprite operating system to compress pages of memory with the LZRW1 algorithm [Williams91]. Before pages are written to backing store (disk), they are first compressed and kept in a *compression cache*. The compression cache is a new level in the virtual memory hierarchy between memory and disk. When the compression cache is full, some compressed pages are written to backing store. In some cases, it allows the working set of a program to remain entirely in main memory and use backing store. When the working set it much larger than available memory, compression reduces the I/O to backing store. Douglis found that applications had speedups ranging from 0.73 to 2.68. Some benchmarks had slowdowns because 1) the number of incompressible pages was very high so that the time taken to compress them was wasted, 2) many pages had low compression ratios so the time to compress and decompress did not offset the I/O time, and 3) the benchmarks had non-sequential data access patterns which did not allow the cost of compression and decompression to be amortized over many data accesses.

2.6 Compiler optimizations for compression

Data compression works by taking advantage of predictability (repetition) in the data stream. The compiler can therefore improve the compressibility of a program by generating code that is more regular and repetitious. Two techniques for accomplishing this are *procedure abstraction* and *register renaming*.

2.6.1 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. Figure 2.8 shows an example of procedure abstraction with a high-level language. Compilers can apply procedure abstraction on an intermediate representation or native instructions. Sequences of code that are

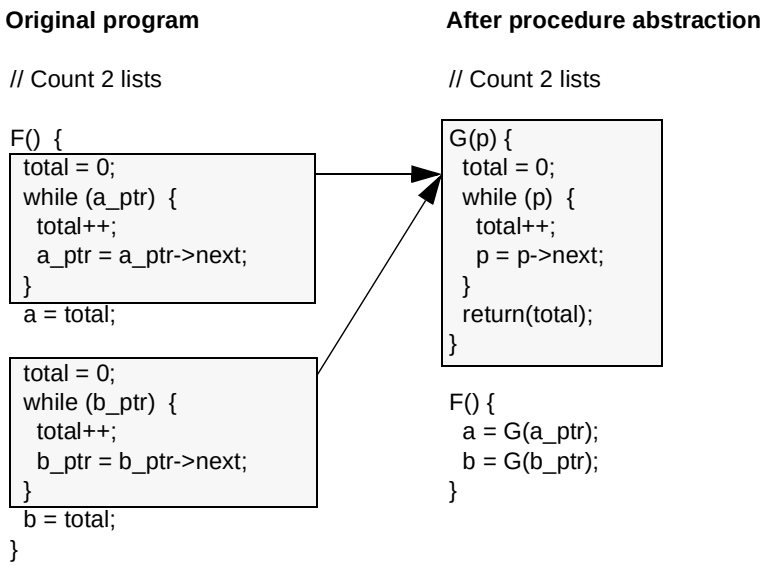


Figure 2.8: Procedure abstraction
Procedure abstraction is an optimization to reduce code size. The sections of code in procedure F are generalized to form procedure G. Procedure F is re-written to call G.

identical, except for the values used, can be bound to the same abstracted function and supplied with arguments for the appropriate values.

Kunchithapadam and Larus

Kunchithapadam and Larus apply procedure abstraction at the level of native instructions by rewriting binary executables [Kunchit99]. They combined this optimization with procedure layout so that the abstracted procedures were adjacent in memory to the function that calls them the most. They find this optimization yields most of its benefit when applied to straightline sequences of instructions. Using larger sequences that included some control-flow only marginally improved their results. For many programs, they found that code size decreased up to 5% and performance improved between 3-9%. However, in some instances, they found that performance could decrease 5-27% which suggests that the effect of procedure abstraction on cache performance needs more study.

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

Original (9 instructions)

```
F:  lbz     r9,0(r28)
    multi   r11,r9,24
    addi    r0,r11,1
    cmpli   cr1,0,r0,8
    ble     cr1,+64
    lbz     r9,0(r28)
    multi   r11,r9,24
    addi    r0,r11,1
    cmpli   cr1,0,r0,8
```

Mini-subroutine (8 instructions)

```
G:  lbz     r9,0(r28)
    multi   r11,r9,24
    addi    r0,r11,1
    cmpli   cr1,0,r0,8
    return

F:  call    G
    ble     cr1,+64
    call    G
```

Call-dictionary (7 instructions)

```
G:  lbz     r9,0(r28)
    multi   r11,r9,24
    addi    r0,r11,1
    cmpli   cr1,0,r0,8


F:  call    G:4
    ble     cr1,+64
    call    G:4
```

Figure 2.9: Mini-subroutines

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.

Figure 2.9 compares the mini-subroutine and call-dictionary methods. A potential problem with these compression methods are that they introduce 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.

2.6.2 Register renaming

Typically, very few opcodes are used by compilers to generate code. Inspite of this, instructions usually differ from one another due to using different register names in their register fields. Register renaming can improve repetition in the program by renaming the registers of an instruction so that it matches another instruction in the program whenever possible. This increases the number of instructions in the program that are identical and allow the compressor to make use of the repetition.

Cooper and McIntosh

Cooper and McIntosh [Cooper99] use register renaming to increase opportunities to apply procedure abstraction and cross-jumping [Wulf75]. They search the entire executable binary for sequences of instructions (possibly spanning several basic blocks) that have similar data-flow and control-flow. They then attempt to make the sequences identical by renaming registers in the live ranges that flow through the sequences. Once the sequences are identical, then procedure abstraction or cross-jumping is applied. They report a 5% average code size reduction and 6% average decrease in dynamic instructions when their code reduction optimizations are applied to optimized programs.

Debray et al.

Debray et al. [Debray99] also combine the techniques of procedure abstraction and register renaming in a binary-rewriting tool. The primary difference is that they use basic blocks, rather than live ranges as the unit of renaming registers. They find basic blocks with matching data-flow graphs and attempt to rename the registers within the basic blocks so that the instructions match. Register move instructions are sometimes inserted before and after the basic blocks to enable renaming. They look at the control-flow between basic blocks so that blocks can be combined into larger units of abstraction. If possible, identical blocks are moved into dominating, or post-dominating blocks to remove copies. Otherwise, the identical basic blocks are then used as candidates for procedure abstraction. Blocks of instructions in the function prologue and epilogue that save and restore registers are treated specially. These are abstracted into procedures. However, the entry point into the abstracted prologue and epilogue changes depending on what registers the caller needs to save and restore. The authors report a compression ratio ranging

from 73% to 81% for Alpha executables produced with gcc 2.7.2.2 with “-O2” optimization (no loop-unrolling and no inlining). They also report performance increases of up to 18% and decreases of up to 28%.

2.7 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.

Chapter 3
Instruction-level compression

In this chapter, we present a study that examines the feasibility of compressing programs at the instruction-level. The results of this study show that ordinary programs contain repetition at the instruction level that can be readily compressed. Programs are compressed with a post-compilation analyzer that examines program object code and replaces common sequences of instructions with a single codeword. A compression-aware microprocessor could execute the compressed instruction sequences by fetching code-words 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.

3.1 Introduction

This chapter studies a compression method similar to the *call-dictionary* scheme [Liao95]. Common sequences of native instructions in object code are replaced with code-words. We extend this work by considering the advantages from using smaller instruction (codeword) sizes. Liao considers the *call-dictionary* instruction to be the size of one or two instruction words. This requires the dictionary to contain sequences with at least two or three 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 only one instruction.

Liao et al. 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, the compression method is described. Second, the experimental results are presented. This chapter ends with conclusions about the compression method and proposals for how it might be improved.

3.2 Overview of compression method

The compression method finds sequences of instructions (some of length one) 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 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 Chapter 3.

3.2.1 Algorithm

The 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. The algorithm has three 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

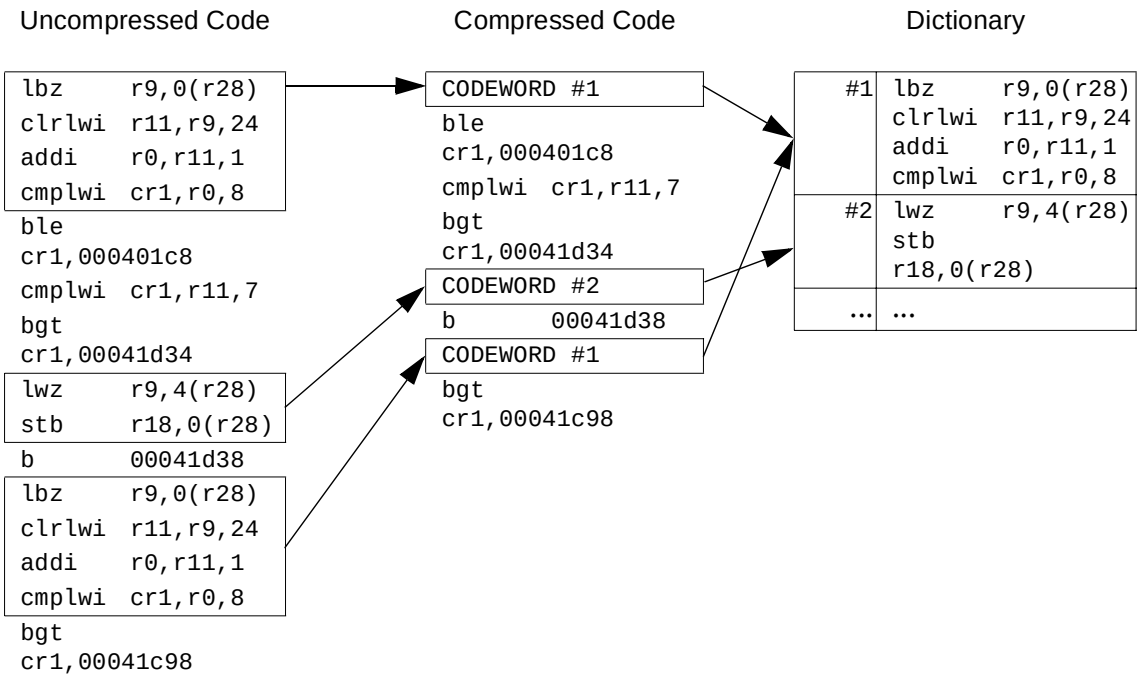


Figure 3.1: Example of compression

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.

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 the process of converting tokens to the actual representation of the codewords in the compressed program. Variable-length codewords, (such as those used in the Huffman encoding) 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.

The 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 eight 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 i386 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 two. 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.

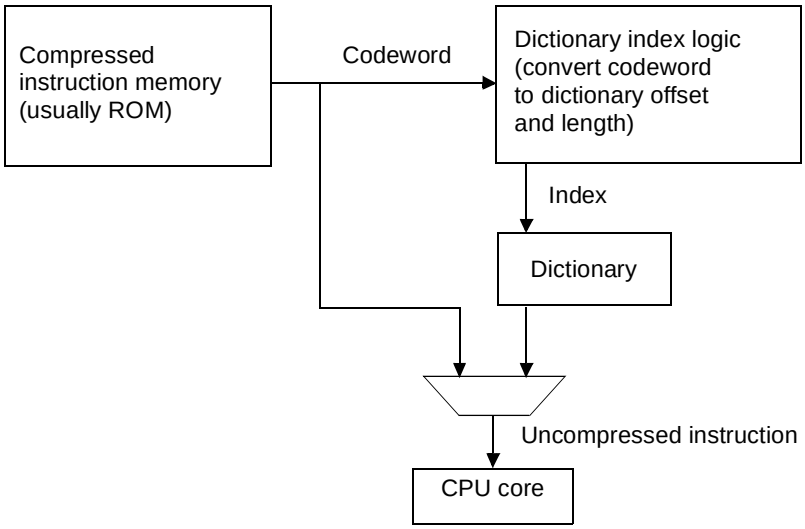


Figure 3.2: Compressed program processor

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.

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 procedure inlining 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 (16 bits long) and the second uses variable-length codewords (8, 12, or 16 bits long). 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 two 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 eight illegal 6-bit opcodes. By using all eight 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 a single 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

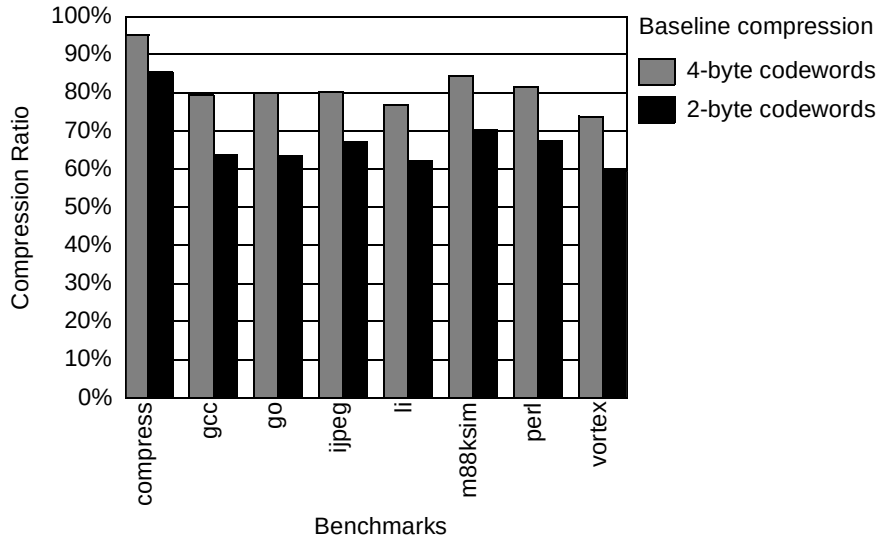


Figure 3.3: Compression ratio using 2-byte and 4-byte codewords
Comparison of baseline compression method with 2-byte and 4-byte codewords.

not beneficial to compress patterns of one 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 one 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 subtracted from the total savings to derive the savings due to using patterns of one instruction. For each benchmark, except vortex, using patterns of one instruction improved the compression ratio by over 6%.

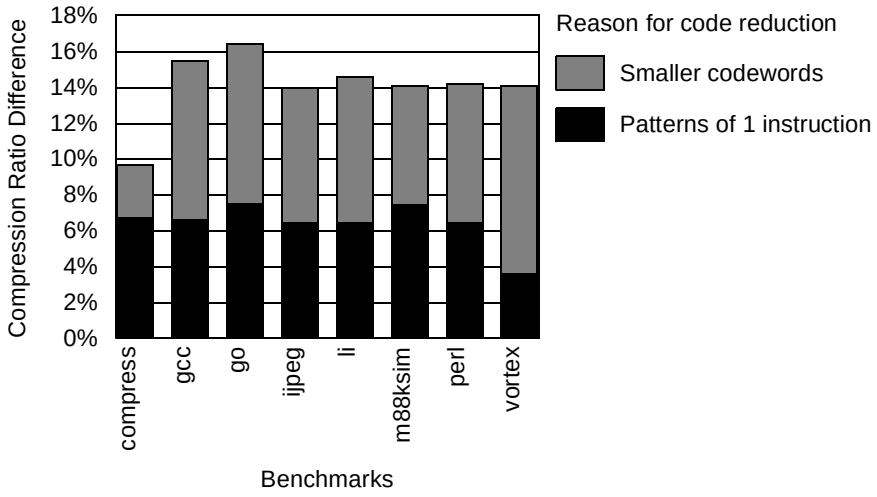


Figure 3.4: Compression ratio difference between 4-byte and 2-byte codewords
Analysis of difference in code reduction between 4-byte codewords and 2-byte codewords in baseline compression method

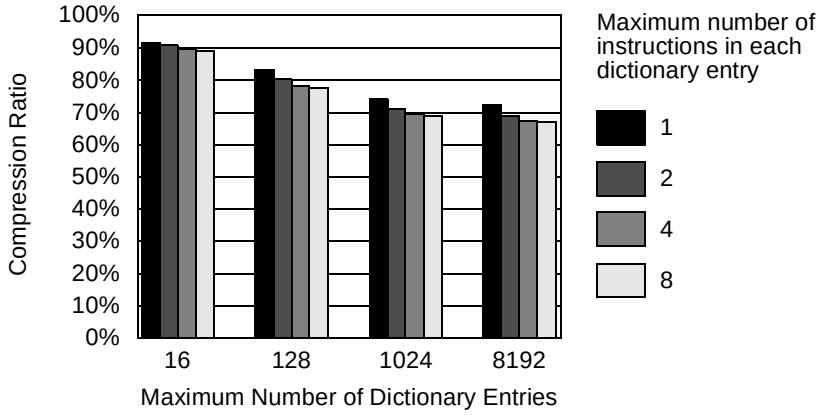


Figure 3.5: Effect of dictionary size and dictionary entry length
Summary of effect of number of dictionary entries and length of dictionary entries in baseline compression method. Compression ratio is averaged over the CINT95 benchmarks.

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, dictionary 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.

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

Table 3.1: Maximum number of codewords used in baseline compression
Maximum dictionary entry size is 4 instructions.

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 from compressing an individual instruction are tiny, but when multiplied over the length of the program, the benefit 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.

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 eight 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 one instruction achieve between 46% and 60% of the

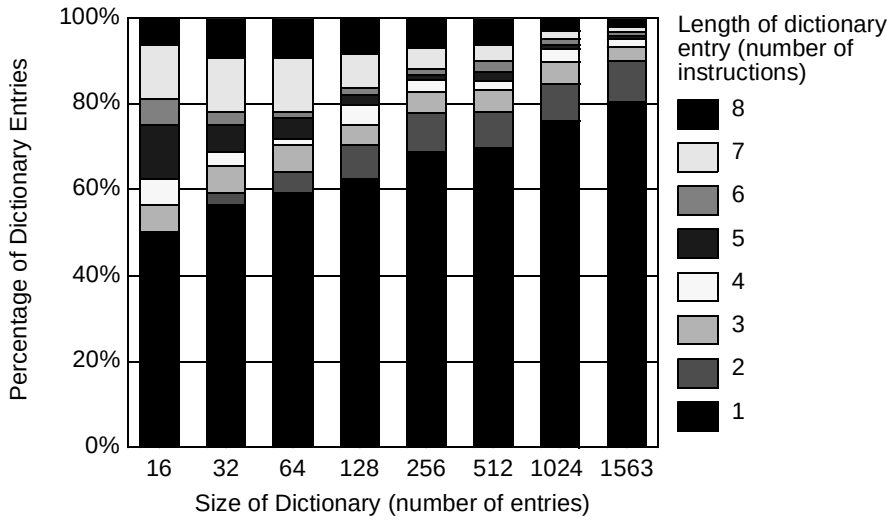


Figure 3.6: Composition of dictionary for *ijpeg*
Longest dictionary entry is 8 instructions.

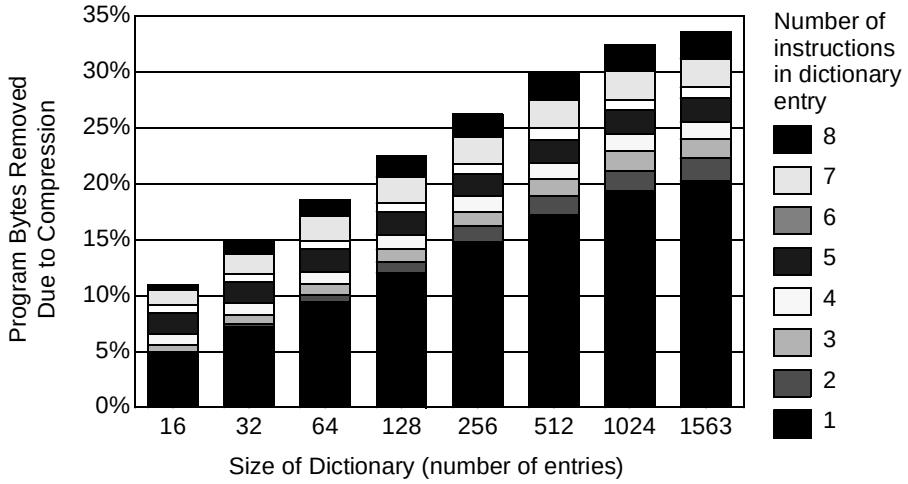


Figure 3.7: Bytes saved according to dictionary entry length
Bytes saved in compression of *ijpeg* according to instruction length of dictionary entry

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.

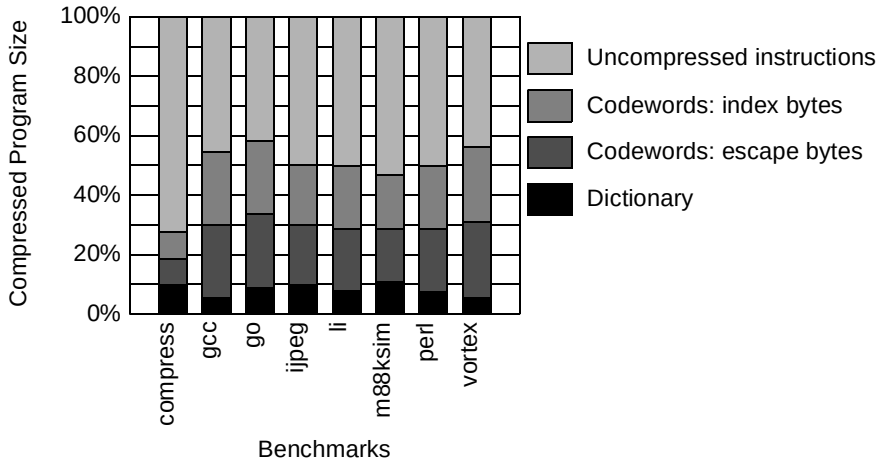


Figure 3.8: Composition of compressed PowerPC programs
Maximum of 8192 2-byte codewords. Longest dictionary entry is 4 instructions.

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 final compressed program contains more codewords than uncompressed instructions.

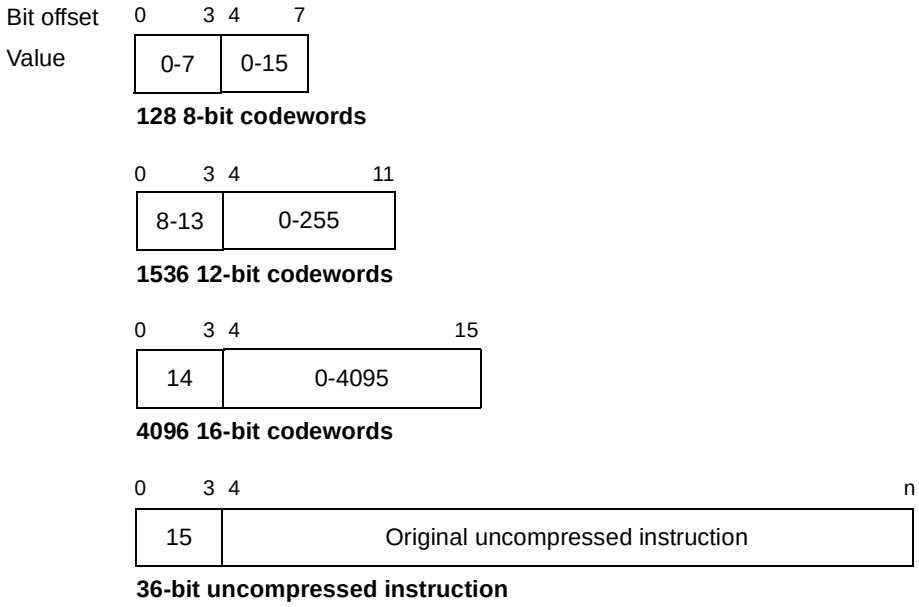


Figure 3.9: Nibble aligned encoding

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

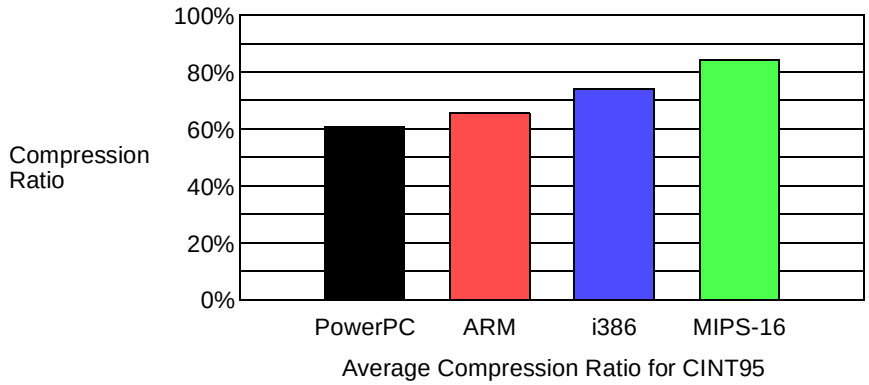


Figure 3.10: Nibble compression for various instruction sets

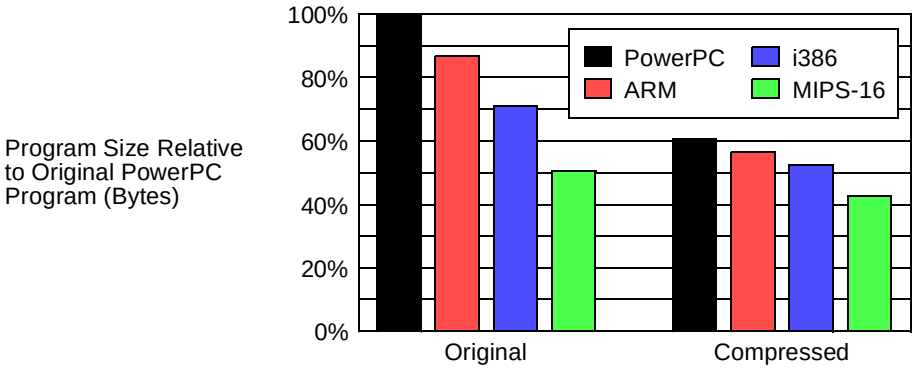


Figure 3.11: Comparison of compression across instruction sets
All program sizes are normalized to the size of the original PowerPC programs.

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 MIPS-16 programs.

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.

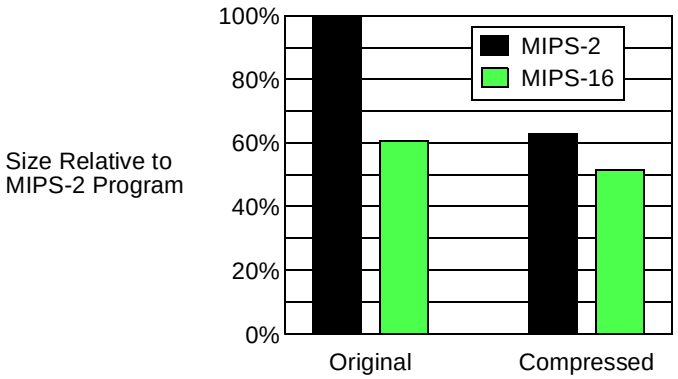


Figure 3.12: Comparison of MIPS-2 with MIPS-16

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 memory is limited. Our approach combines elements of two previous proposals. First we use a dictionary compression method [Liao95] that allows codewords to expand to several instructions. Second, we allow the codewords to be smaller than a single instruction [Kozuch94]. For this method, 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 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 Conclusion

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 a covering algorithm instead of a greedy algorithm.

Chapter 4
Hardware-managed decompression

This chapter investigates the performance of a hardware-managed code compression algorithm. We find that code compression with appropriate hardware optimizations does not have to incur much performance loss over native code, contrary to previous studies. Furthermore, our studies show this holds for architectures with a wide range of memory configurations and issue widths. Surprisingly, we find that a performance increase over native code is achievable in many situations. The performance increase occurs because fetching compressed instructions results in lower memory traffic.

4.1 Introduction

In this chapter we perform an in-depth analysis of one particular compression method supported in hardware: IBM's CodePack instruction compression used in the PowerPC 405. The approach taken by IBM is the first to combine many previously proposed code compression techniques. It is also the first commercially available code compression systems that does more than simply support a 16-bit instruction subset. For these reasons, it makes an ideal study. We do not attempt to precisely model the CodePack as implemented in the PowerPC 405. Instead, we implement CodePack on the SimpleScalar simulator in order to inspect how it performs on various architecture models. Our goal is to determine the performance pitfalls inherent in the compression method and suggest architectural features to improve execution time. We answer the following questions:

- What is the performance effect of decompression?
- How does this performance change over a range of microarchitectures?
- Which steps of the decompression algorithm hinder performance the most?
- What additional optimizations can be made to improve decompression performance?

4.2 Related work

CodePack [IBM98, Kemp98] is used in IBM's embedded PowerPC systems. This scheme resembles CCRP in that it is part of the memory system. The CPU is unaware of compression, and a LAT-like device maps between the native and compressed address spaces. The decompressor accepts L1-cache miss addresses, retrieves the corresponding compressed bytes from main memory, decompresses them, and returns native PowerPC instructions to the L1-cache. CodePack achieves 60% compression ratio on PowerPC. IBM reports that performance change in compressed code is within 10% of native programs — sometimes with a speedup. A speedup is possible because CodePack implements prefetching behavior that the underlying processor does not have.

CodePack uses a different symbol size for compression than previous schemes for 32-bit instructions. CCRP divides each instruction into 4 8-bit symbols which are then compressed with Huffman codes. The decoding process in CCRP is history-based which serializes the decoding process. Decoding 4 symbols per instruction is likely to impact decompression time significantly. Lefurgy et al. proposed a dictionary compression method for PowerPC that uses complete 32-bit instructions as compression symbols [Lefurgy97]. This method achieves compression ratios similar to CodePack, but requires a dictionary with several thousand entries which could increase access time and hinder high-speed implementations. This variable-length encoding scheme is similar to CodePack in that both pre-pend each codeword with a short tag to indicate its size. This should allow implementations with multiple decompressors to quickly extract codewords from an input stream and decompress them in parallel. CodePack divides each PowerPC instruction into 2 16-bit symbols that are then compressed into variable-length codewords. The 16-bit symbols allow CodePack to achieve its compression ratio with only 2 dictionaries of less than 512 entries each.

4.3 Compression architecture

This section gives an overview of the CodePack compression algorithm and discusses its current implementation in PowerPC. The complete CodePack algorithm is described in the CodePack user manual [IBM98].

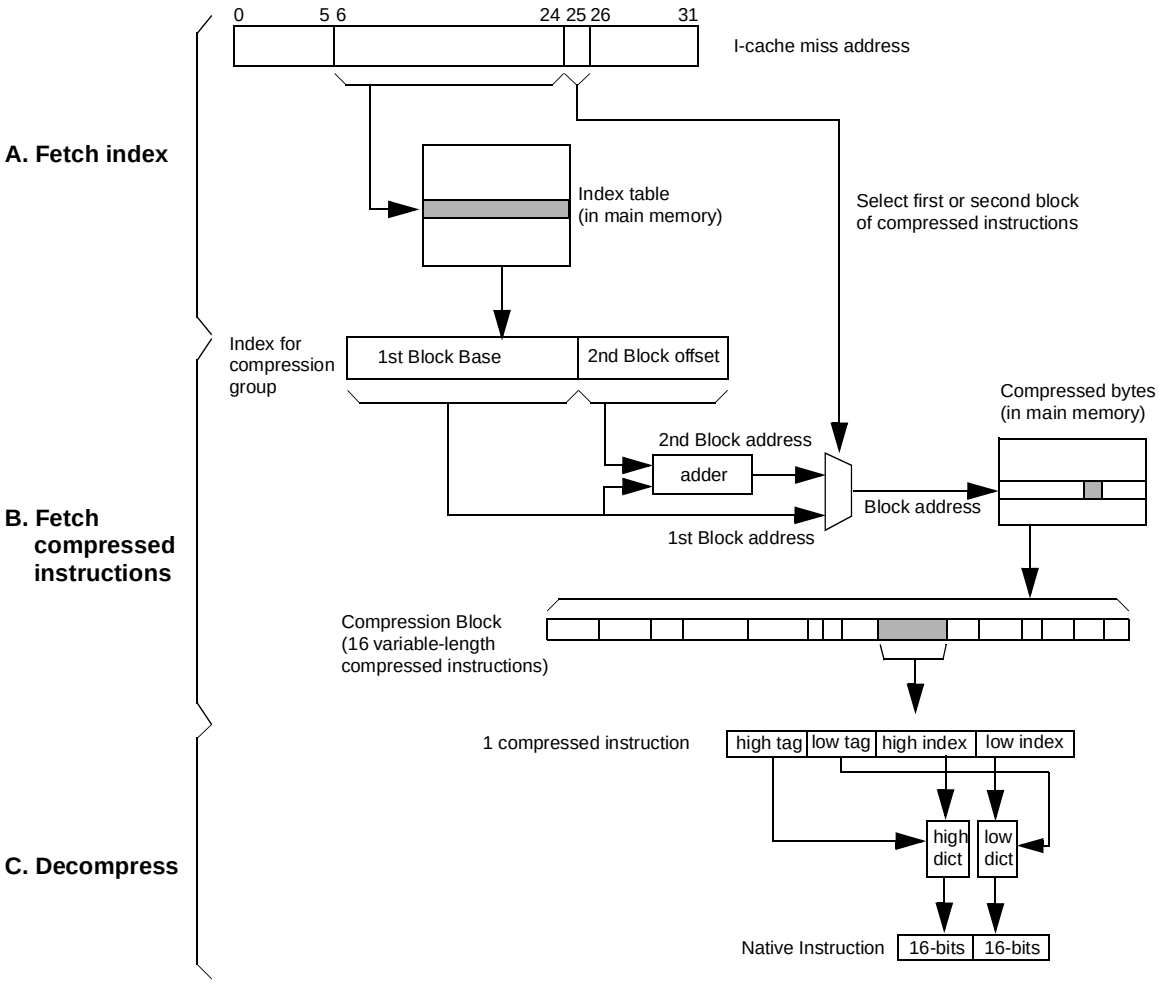


Figure 4.1: CodePack decompression
A) Use instruction address to fetch index from index table. B) Use index to map native instruction address to compressed instruction address and fetch compressed instructions. C) Decompress compressed instructions into native instructions.

4.3.1 CodePack algorithm

Figure 4.1 illustrates the decompression algorithm. To understand it, consider how the compression encoder works (start at the bottom of Figure 4.1). Each 32-bit instruction is divided into 16-bit high and low half-words which are then translated to a variable bit codeword from 2 to 11 bits. Because the high and low half-words have very different distribution frequencies and values, two separate dictionaries are used for the translation. The most common half-word values receive the shortest codewords. The codewords are divided into 2 sections. The first section is a 2 or 3 bit tag that tells the size of the code-

word. The second section is used to index the dictionaries. The value 0 in the lower half-word is encoded using only a 2 bit tag (no low index bits) because it is the most frequently occurring value. The dictionaries are fixed at program load-time which allows them to be adapted for specific programs. Half-words that do not fit in the dictionary are left directly in the instruction stream and pre-pended with a 3 bit tag to identify them as raw bytes instead of compressed bytes.

Each group of 16 instructions is combined into a *compression block*. This is the granularity at which decompression occurs. If the requested I-cache line (8 instructions) is in the block, then the whole block is fetched and decompressed.

The compressed instructions are stored at completely different memory locations from the fixed-length native instructions. Therefore, the instruction address from the cache miss is mapped to the corresponding compressed instruction address by an index table which is created during the compression process. The function of the index table is the same as the LAT in CCRP. Each index is 32-bits. To optimize table size, each entry in the table maps one *compression group* consisting of 2 compressed blocks (32 instructions total). The first block is specified as a byte offset into the compressed memory and the second block is specified using a shorter offset from the first block. This is an example of difference encoding which is described in Section 2.1.2.

4.3.2 Implementation

The IBM implementation of CodePack has several features to enable high-speed decoding. We attempt to model their effects in our simulations.

Index cache

The index table is large enough that it must be kept in main memory. However, the last used index table entry is cached so that an access to the index table can be avoided in the case when the next L1-cache miss is in the same compression group. (There is one index for each compression group and each compression group maps 4 cache lines.) We will discuss the benefit of using even larger index caches.

Burst read

Burst accesses are used to fetch compressed bytes from main memory. There is an initial long latency for the first memory access and shorter latencies for successive accesses to adjacent memory locations.

Dictionaries

Both dictionaries are kept in a 2 KB on-chip buffer. This is important for fast decompression since the dictionaries are accessed frequently (once per instruction).

Decompression

As compressed bytes are returned from main memory, they are decompressed at the rate of one instruction per cycle. This allows some overlap of fetching and decompression operations. We will discuss the benefit of using even greater decompression bandwidth.

Instruction prefetching

On an L1-cache miss, instructions are decompressed and put into a 16 instruction output buffer within the decompressor. Even though the L1-cache line requires eight instructions, the remaining ones are always decompressed. This buffer is completely filled on each L1-cache miss. This behaves as a prefetch for the next cache line.

Instruction forwarding

As instructions are decompressed, they are put in the output buffer and also immediately forwarded to the CPU for decoding and execution.

4.4 Simulation environment

We perform our compression experiments on the SimpleScalar 3.0 simulator [Burger97] after modifying it to support compressed code. We use benchmarks selected from the SPEC CINT95 [SPEC95] and MediaBench [Lee97] suites. The benchmarks *cc1*, *go*, *perl*, and *vortex* were chosen from CINT95 because they perform the worst under CodePack since they have the highest L1 I-cache miss ratios. The benchmarks *mpeg2enc* and *pegwit* are representative of loop-intensive embedded benchmarks. All benchmarks are compiled with GCC 2.6.3 using the optimizations “-O3 -funroll-loops” and are stati-

Bench	Instructions executed (millions)	Input set	I-cache miss rate for 4-issue
cc1	1441	cp-decl.i	6.7%
go	1265	30 12 null.in	6.2%
mpeg2enc	1119	default with profile=1, level=4, chroma=2, precision=0, repeat=0	0.0%
pegwit	1014	11MB file	0.1%
perl	1108	*ref* input without “abortive” and “abruption”	4.4%
vortex	1060	*ref* input with PART_COUNT 400, INNER_LOOP 4, DELETES 80, STUFF_PARTS 80	5.2%

Table 4.1: Benchmarks

cally linked with library code. Table 4.1 lists the benchmarks and the input sets. Each benchmark executes over one billion instructions and is run to completion.

SimpleScalar has 64-bit instructions which are loosely encoded, and therefore highly compressible. We wanted an instruction set that more closely resembled those used in today’s microprocessors and used by code compression researchers. Therefore, we re-encoded the SimpleScalar instructions to fit within 32 bits. Our encoding is straightforward and resembles the MIPS IV encoding. Most of the effort involved removing unused bits (for future expansion) in the 64-bit instructions.

For our baseline simulations we choose three very different architectures. The *1-issue* architecture is a low-end processor for an embedded system. This is modeled as a single issue, in-order, 5-stage pipeline. We simulate only L1 caches and main memory. Main memory has a 64-bit bus. The first access takes 10 cycles and successive accesses take 2 cycles. The *4-issue* architecture differs from the 1-issue in that it is out-of-order and the bandwidth between stages is 4 instructions. We use the *8-issue* architecture as an example of a high performance system. The simulation parameters for the architectures are given in Table 4.2.

Figure 4.2 illustrates the models for L1-miss activity. Figure 4.2-a shows that a native code miss just fetches the cache line from main memory in 4 accesses (32-byte cache lines with a 64-bit bus). We modified SimpleScalar to return the critical word first for I-cache misses. For example, if the fifth instruction in the cache line caused the miss, it will be returned in the first access at t=10. This is a significant advantage for native code programs. Decompression must proceed in a serial manner and cannot take advantage of the critical word first policy. Figure 4.2-b shows the baseline compression system. This model fetches the index table entry from main memory (unless it reuses the previous

SimpleScalar parameters	1-issue	4-issue	8-issue
fetch queue size	1	4	8
decode width	1	4	8
issue width	1 in-order	4 out-of-order	8 out-of-order
commit width	1	4	8
Register update unit entries	2	64	128
load/store queue	2	32	64
function units	alu:1, mult:1, memport:1, fpalu:1, fpmult:1	alu:4, mult:1, mem-port:2, fpalu:4, fpmult:1	alu:8, mult:1, mem-port:2, fpalu:8, fpmult:1
branch pred	bimode 2048 entries	gshare with 14-bit history	hybrid predictors with 1024 entry meta table.
L1 i-cache	8 KB, 32B lines, 2-assoc, lru	16 KB	32 KB
L1 d-cache	8 KB, 16B lines, 2-assoc, lru	16 KB	32 KB
memory latency	10 cycle latency, 2 cycle rate	same	same
memory width	64 bits	same	same

Table 4.2: Simulated architectures

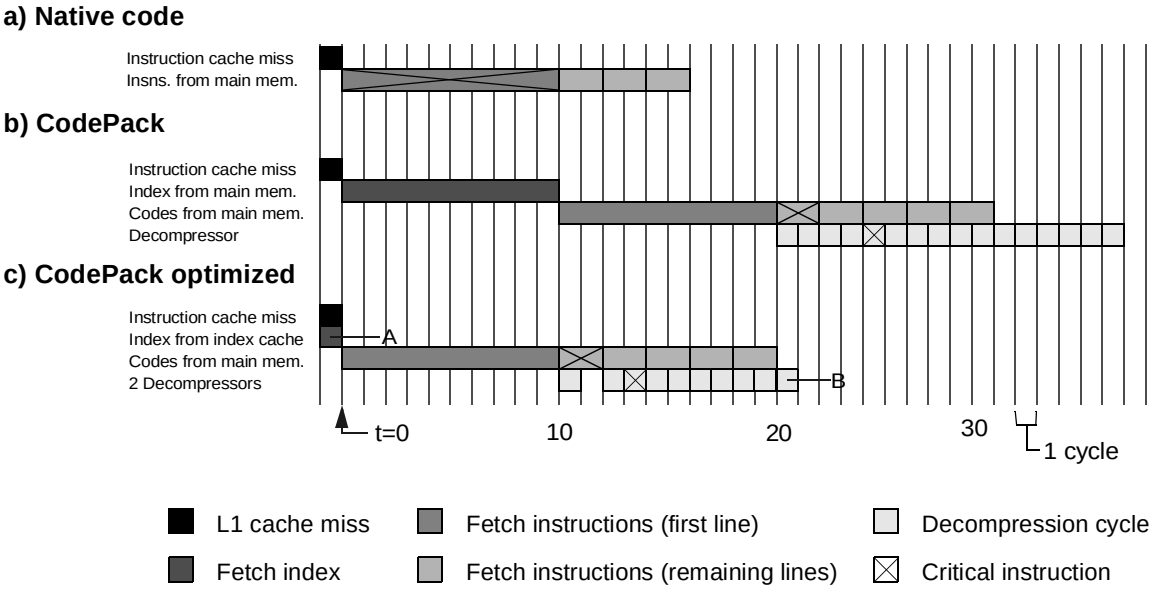


Figure 4.2: Example of L1 miss activity
2-a:The native program can fetch the critical (missed) instruction first and burst read the remainder of the cache line.
2-b:CodePack first fetches the index from the index table in main memory. It then fetches compressed instructions and decompresses them in parallel.
2-c:CodePack optimizations are A) accessing index cache to eliminates index fetch to main memory and B) expanding decompression bandwidth to decompress two instructions per cycle.

index), uses the index to fetch codewords from main memory, and decompresses code-words as they are received. In the example, the consecutive main memory accesses return

Bench	Original size (bytes)	Compressed size (bytes)	Compression ratio (smaller is better)
cc1	1,083,168	654,999	60.5%
go	310,576	182,602	58.8%
mpeg2enc	118,416	74,681	63.2%
pegwit	88,560	54,120	61.3%
perl	267,568	162,045	60.6%
vortex	495,248	274,420	55.4%

Table 4.3: Compression ratio of .text section

compressed instructions in the quantities 2, 3, 3, 3, 3, and 2. The critical instruction is in the second access. Assuming that the decompressor has a throughput of 1 instruction/cycle, then the critical instruction is available to the core at t=25.

Figure 4.2-c shows our improvements to the basic compressed code model. We cache index entries, which often avoids a lengthy access to main memory. We also investigate the effect of increasing the decompression rate on performance. In the example, a decompression rate of 2 instructions/cycle allows the critical instruction to be ready at t=14.

4.5 Results

Our first experiments evaluate the original CodePack algorithm on a variety of architectures to characterize its performance. We then propose optimizations to improve the performance of compressed code. Finally, we vary the memory system parameters to determine the performance trends of the optimizations.

4.5.1 Code size

Table 4.3 shows the size of `.text` section of the original and compressed programs. These results are similar to the typical compression ratio of 60% reported by IBM for PowerPC programs.

Table 4.4 shows the composition of the compressed `.text` section. The **Index table** column represents the bits required to translate cache miss addresses to compression region addresses. The **Dictionary** column represents the contents of the high and low half-word dictionaries. The **Compressed tags** and **Dictionary indices** columns represent the

Bench	Index table	Dictionary	Compressed tags	Dictionary indices	Raw tags	Raw bits	Pad	Total (bytes)
cc1	5.1%	0.3%	22.5%	46.1%	3.9%	20.9%	1.1%	654,999
go	5.3%	1.0%	24.7%	50.9%	2.7%	14.2%	1.2%	182,602
mpeg2enc	5.0%	2.7%	21.9%	46.0%	3.7%	19.9%	1.1%	74,681
pegwit	5.1%	3.4%	26.3%	49.4%	2.7%	14.7%	1.1%	54,120
perl	5.2%	1.1%	22.5%	46.0%	3.8%	20.3%	1.1%	162,045
vortex	5.6%	0.7%	25.1%	50.3%	2.7%	14.3%	1.2%	274,420

Table 4.4: Composition of compressed region

two components of the compressed instructions in the program. The **Raw tags** column represents the use of 3-bit tags to mark non-compressed half-words. The **Raw bits** column represents bits that are copied directly from the original program in either the form of individual non-compressed half-words (preceded by raw tags) or entire non-compressed CodePack blocks. The **Pad** column shows the number of extra bits required to byte-align CodePack blocks. The columns for raw tags and raw bits show that a surprising portion (19-25%) of the compressed program is not compressed. The raw bits occur because there are instructions which contain fields with values that to not repeat frequently or have adjacent fields with rare combinations of values. Many instructions that are represented with raw bits use large branch offsets, unusual register ordering, large stack offsets, or unique constants. Also, CodePack may choose to not compress entire blocks in the case that using the compression algorithm would expand them. These non-compressed blocks are included in the **Raw bits** count, but occur very rarely in our benchmarks. It is possible that new compiler optimizations could select instructions so that more of them fit in the dictionary and less raw bits are required.

4.5.2 Overall performance

Table 4.5 shows the overall performance of CodePack compressed code compared to native code. We also show an optimized decompressor that provides significant speedup over the baseline decompressor and even outperforms native code in many cases. We describe our optimized model in the following sections. The performance loss for compressed code compared to native code is less than 14% for 1-issue, under 18% for 4-issue, and under 13% for 8-issue. The *mpeg2enc* and *pegwit* benchmarks do not produce enough cache misses to produce a significant performance difference between the com-

Bench	1-issue			4-issue			8-issue		
	Native	CodePack	Optimized	Native	CodePack	Optimized	Native	CodePack	Optimized
cc1	0.40	0.35	0.39	1.00	0.82	0.97	1.62	1.42	1.58
go	0.43	0.39	0.44	1.02	0.91	1.07	1.47	1.37	1.53
mpeg2enc	0.51	0.51	0.51	1.48	1.48	1.48	1.76	1.76	1.76
pegwit	0.56	0.56	0.56	2.83	2.82	2.83	4.35	4.35	4.35
perl	0.44	0.38	0.43	1.45	1.19	1.49	2.36	2.23	2.44
vortex	0.43	0.39	0.43	1.58	1.39	1.62	2.51	2.34	2.54

Table 4.5: Instructions per cycle
Native is the original program. *CodePack* is the baseline decompressor. *Optimized* is our optimized CodePack index cache and additional decompression bandwidth.

pressed and native programs. CodePack behaves similarly across each of the baseline architectures provided that the cache sizes are scaled with the issue width. Therefore in the remaining experiments, we only present results for the 4-issue architecture.

4.5.3 Components of decompression latency

Intuition suggests that compression reduces the fetch bandwidth which could actually lead to performance improvement. However, CodePack requires that the compressed instruction fetch be preceded by an access to the index table and followed by decompression. This reduces the fetch bandwidth below that of native code resulting in a potential performance loss.

We explore two optimizations to reduce the effect of index table lookup and decompression latency. These optimizations allow the compressed instruction fetch to dominate the L1 miss latency. Since the compressed instructions have a higher instruction density than native instructions, a speedup should result. In the following subsections, we measure the effects of these optimizations on the baseline decompressor model.

Index table access

We assume that the index table is large and must reside in main memory. Therefore, lookup operations on the table are expensive. The remaining steps of decompression are dependent on the value of the index, so it is important to fetch it efficiently. One way to improve lookup latency is to cache some entries in faster memory. Since a single index maps the location of 4 consecutive cache lines and instructions have high spatial locality, it is likely the same index will be used again. Therefore, caching should be very beneficial. Another approach to reduce the cost of fetching index table entries from main memory is

Number of lines	Line size (index entries)			
	1	2	4	8
1	62.1%	51.9%	42.9%	35.8%
16	53.4%	39.1%	28.0%	19.2%
64	45.5%	29.7%	14.4%	4.6%
256	11.7%	2.7%	0.8%	0.2%

Table 4.6: Index cache miss ratio for *cc1*
Values represent index cache miss ratio during L1 cache miss using CodePack on the 4-issue model. The index cache used here is fully-associative.

Bench	4-issue		
	CodePack	Index Cache	Perfect
cc1	0.82	0.92	0.96
go	0.89	0.99	1.00
mpeg2enc	1.00	1.00	1.00
pegwit	1.00	1.00	1.00
perl	0.82	0.95	0.95
vortex	0.88	0.96	0.98

Table 4.7: Speedup due to index cache
Values represent speedup over native programs. The *Index Cache* column represents a fully-associative cache with 4 indices per entry. The *Perfect* column represents an index cache that never misses.

to burst read several entries at once. We try both approaches by adding a cache for index table entries. Since the index table is indexed with bits from the miss address, it can be accessed in parallel with the L1 cache. Therefore in the case of an index cache hit, the index fetch does not contribute to L1 miss penalty. Table 4.6 shows the miss rate for *cc1* with index caches using the 4-issue model. All index caches are fully-associative. A 64-line cache with 4 indexes per line can reduce the miss ratio to under 15% for the *cc1* benchmark which has the most instruction cache misses. This organization has a miss ratio of under 11% for *vortex* and under 4% for the other benchmarks. We use this cache organization for the index cache in our optimized compression model. The index cache contains 1 KB of index entries and 88 bytes of tag storage. This is about one-eighth the size of the 4-issue instruction cache. It is able to map 32 KB of the original program into compressed bytes. In Table 4.7 the performance of the native code is compared to CodePack, CodePack with index cache, and CodePack with a perfect index cache that always hits. The perfect index cache is feasible to build for short programs with small index tables that can be put in an on-chip ROM. The optimized decompressor performs within 8% of native code for cc1 and within 5% for the other benchmarks.

Bench	4-issue		
	CodePack	2 decoders	16 decoders
cc1	0.82	0.87	0.87
go	0.89	0.94	0.94
mpeg2enc	1.00	1.00	1.00
pegwit	1.00	1.00	1.00
perl	0.82	0.86	0.87
vortex	0.88	0.93	0.93

Table 4.8: Speedup due to decompression rate
Values represent speedup over native programs.

Bench	4-issue			
	CodePack	Index	Decompress	Both
cc1	0.82	0.92	0.87	0.97
go	0.89	0.99	0.94	1.05
mpeg2enc	1.00	1.00	1.00	1.00
pegwit	1.00	1.00	1.00	1.00
perl	0.82	0.95	0.86	1.03
vortex	0.88	0.96	0.93	1.03

Table 4.9: Comparison of optimizations
Values represent speedup over native programs. *Index* is CodePack with a fully-associative 64-entry index cache with 4 indices per entry. *Decompress* is CodePack that can decompress 2 instructions per cycle. *Both* shows the benefit of both optimizations together. A slight speedup is attained over native code for *go*, *perl*, and *vortex*.

Instruction decompression

Once the compressed bytes are retrieved, they must be decompressed. Decoding proceeds serially through each block until the desired instructions are found. The baseline CodePack implementation assumes that one instruction can be decompressed per cycle. Since a variable-length compressed instruction is tagged with its size, it is easy to find the following compressed instruction. Wider and faster decompression logic can use this feature for higher decompression throughput. The effect of having greater decoder bandwidth appears in Table 4.8. Using 16 decompressors/cycle represents the fastest decompression possible since compression blocks contain only 16 instructions. In the 4-issue model, we find that most of the benefit is achieved by using only two decompressors.

Performance results

We now combine both of the above optimizations to see how they work together. Table 4.9 shows the performance of each optimization individually and together. In our

Bench	4-issue instruction cache size							
	1 KB		4 KB		16 KB		64 KB	
	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized
cc1	0.76	1.06	0.78	1.01	0.82	0.97	0.96	1.00
go	0.79	1.14	0.84	1.11	0.89	1.05	0.98	1.01
mpeg2enc	0.93	1.01	1.00	1.00	1.00	1.00	1.00	1.00
pegwit	0.99	1.61	0.9	1.38	1.00	1.00	1.00	1.00
perl	0.72	1.13	0.71	1.05	0.82	1.03	0.99	0.99
vortex	0.78	1.25	0.78	1.15	0.88	1.03	0.98	1.00

Table 4.10: Variation in speedup due to instruction cache size
Values represent speedup over native programs using the same instruction cache size. All simulations are based on 4-issue model with different cache sizes. The *16 KB* column is the 4-issue baseline model.

optimized model, the index cache optimization improved performance more than using a wider decompressor. This is because the codeword fetch and decompression stages of the CodePack algorithm cannot begin until the index lookup is complete. However, the decompression stage already executes concurrently with the fetch stage and often must wait for codewords to be fetched. Therefore, improving the decompressor helps less than removing the long latency for the index lookup. When both optimizations are combined, the decompression penalty is nearly removed. In fact, the *go*, *perl*, and *vortex* benchmarks perform better than native code. The reason for this is that using fewer memory accesses to fetch compressed instructions utilizes the memory bus better. This performance benefit is greater than the small decompression penalty and therefore the programs have improved execution time.

4.5.4 Performance effects due to architecture features

The following sections modify the baseline architecture in a number of ways in order to understand in which systems CodePack is useful. For each architecture modification, we show the performance of the baseline decompressor and optimized decompressor relative to the performance of native code.

Sensitivity to cache size

Decompression is only invoked on the L1-cache miss path and is thus sensitive to cache organization. We simulated many L1 instruction cache sizes and show the performance in Table 4.10. The default decompressor has a performance penalty of up to 28% with 1 KB caches. However, the optimized decompressor has up to a 61% performance

Bench	4-issue main memory bus size							
	16 bits		32 bits		64 bits		128 bits	
	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized
cc1	0.94	1.00	0.91	0.99	0.82	0.97	0.76	0.94
go	1.03	1.12	0.98	1.08	0.89	1.05	0.84	1.00
mpeg2enc	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
pegwit	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
perl	0.93	1.05	0.89	1.03	0.82	1.03	0.77	0.97
vortex	1.03	1.09	0.97	1.05	0.88	1.03	0.82	0.97

Table 4.11: Performance change by memory width
Values represent speedup over native programs using the same bus size. All simulations are based on 4-issue model with different bus widths. The *64-bits* column is the 4-issue baseline model.

gain. The optimized decompressor has better performance than the native code in every case. The reason for this is that the dominant time to fill a cache miss is reading in the compressed instructions. Since the optimized decompressor can fetch more instructions with fewer memory accesses, it can fill a cache line request quicker than the native code. As cache size grows, the performance of both decompressors approaches the performance of native code. This is because the performance difference is in the L1-miss penalty and there are few misses with large caches.

Sensitivity to main memory width

Many embedded systems have narrow buses to main memory. Instruction sets with short instruction formats can outperform wider instructions because more instructions can be fetched in less time. Bunda reports similar findings on a 16-bit version of the DLX instruction set [Bunda93]. This suggests that code compression might offer a benefit in such architectures. Our results in Table 4.11 show the performance change for buses of 16, 32, 64, and 128 bits. The number of main memory accesses for native and compressed instructions decreases as the bus widens, but CodePack still has the overhead of the index fetch. Therefore, it performs relatively worse compared to native code as the bus widens. In the optimized decompressor, the index fetch to main memory is largely eliminated so the performance degrades much more gracefully than the baseline decompressor. On the widest buses, the number of main memory accesses to fill a cache line is about the same for compressed and native code. Therefore, the decompress latency becomes important. Native code is faster at this point because it does not incur a time penalty for decompression.

Bench	Main memory latency compared to 4-issue model									
	0.5x		1x		2x		4x		8x	
	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized	CodePack	Optimized
cc1	0.79	0.93	0.82	0.97	0.84	0.97	0.82	0.97	0.81	0.96
go	0.87	0.99	0.89	1.05	0.91	1.09	0.89	1.11	0.88	1.12
mpeg2enc	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
pegwit	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	0.99	1.00
perl	0.80	0.96	0.82	1.03	0.81	1.04	0.78	1.06	0.76	1.04
vortex	0.84	0.97	0.88	1.03	0.91	1.05	0.90	1.05	0.89	1.06

Table 4.12: Performance change due to memory latency
Values represent speedup over native programs using the same memory latency. *1x* column is the 4-issue baseline model.

Sensitivity to main memory latency

It is interesting to consider what happens with decompression as main memory latencies grow. Embedded systems may use a variety of memory technologies. We simulated several memory latencies and show the results in Table 4.12. As memory latency grows, the optimized decompressor can attain speedups over native code because it uses fewer costly accesses to main memory.

4.6 Conclusion

The CodePack algorithm is very suitable for the small embedded architectures for which it was designed. In particular, a performance benefit over native code can be realized on systems with narrow memory buses or long memory latencies. In systems where CodePack does not perform well, reducing cache misses by increasing the cache size helps remove performance loss.

We investigated adding some simple optimizations to the basic CodePack implementation. These optimizations remove the index fetch and decompression overhead in CodePack. Once this overhead is removed, CodePack can fetch a compressed program with fewer main memory accesses and less latency than a native program. Combining the benefit of fewer main memory accesses and the inherent prefetching behavior of the CodePack algorithm often provides a speedup over native code. Our optimizations show that CodePack can be useful in a much wider range of systems than the baseline implementation. In many cases, native code did not perform better than our optimized Code-

Pack except on the systems with the fastest memory or widest buses. Code compression systems need not be low-performance and can actually yield a performance benefit. This suggests a future line of research that examines compression techniques to improve performance rather than simply program size.

The performance benefit provided by the optimized decompressor suggests that even smaller compressed representations with higher decompression penalties could be used. This would improve the compressed instruction fetch latency, which is the most time consuming part of the CodePack decompression. Even completely software-managed decompression may be an attractive option for resource limited computers.

Chapter 5
Software-managed decompression

The previous chapter demonstrated that compressed code systems with hardware decompression can improve code density and often attain better execution time than native code. This was accomplished with decompression hardware that ran concurrently with the application and decompressed missed cache lines directly into the L1 instruction cache.

This chapter presents a method of decompressing programs using software. It relies on using a software-managed instruction cache under control of the decompressor. Decompression is achieved by employing a simple cache management instruction that allows explicit writing into a cache line. Since the decompressor is an ordinary program, it interrupts the execution of the application and always results in a slowdown compared to native code. The challenge is to tune the decompressor software so that performance can remain close to native code speeds.

5.1 Introduction

Software-managed decompression has a higher degree of flexibility than a hardware solution. Software-managed decompression allows separate programs to use entirely different compression methods, whereas a hardware implementation can only tune parameters of the compression algorithm. Newly developed compression methods are not constrained to use old decompression hardware. Software decompression allows the specific compression algorithm to be selected late in the product design cycle. Finally, decompressors can be cheaply implemented on a wide variety of architectures and instruction sets with little effort.

A software implementation also reduces hardware complexity. It is likely that the decompressor will have a smaller physical implementation in software rather than hardware. This is because the software is small (a few hundred bytes) and can be stored in a high-density read-only memory. In addition, the software reuses memory and computation structures (ALU, register file, and buses) that already exist on the processor. The hardware implementation, on the other hand, may duplicate these structures so that the decompressor can execute concurrently with the application, as is done in CodePack.

The primary challenge in software decompression is to minimize the increased execution time due to running the decompression software. Our technique achieves high performance in part through the addition of a simple cache management instruction that writes decompressed code directly into an instruction cache line. Similar instructions have been proposed in the past.

The organization of this chapter is as follows. Section 5.2 reviews previous work in software code compression. We present the software decompressors in Section 5.3 and the compressed code system architecture in Section 5.4 and . The simulation environment is presented in Section 5.5. In Section 5.6, we discuss our experimental results. Finally, Section 5.7 contains our conclusions.

5.2 Related work

Our work is most comparable to a software-managed compression scheme proposed by Kirovski et al. [Kirovski97]. They use a software-managed procedure-cache to hold decompressed procedures. This method requires that 1) the procedure cache be large enough to completely hold the largest procedure, 2) procedure eviction occurs when there is not enough free-space to hold an incoming procedure, and 3) defragmentation occurs when not enough contiguous free-space is available for an incoming procedure. Their compression algorithm is LWRZ1 [Williams91], an adaptive Ziv-Lempel model.

In contrast, our compression scheme works on the granularity of cache lines and can be used with caches of any size and procedures of any size. It is faster because it tends to avoid decompressing code that is not executed, does not need to manage cache fragmentation, and uses a simpler decompression algorithm. When large units such as proce-

dures are decompressed, it becomes increasingly likely that instructions that are never executed will be decompressed. This is an unnecessary increase in decompression overhead. Decompressing shorter sequences that have a high probability of being executed may be more efficient. In addition, there are no fragmentation costs for decompressing a cache-line. This is due to the fact that the instruction cache already allocates and replaces instructions in units of cache lines.

Software decompression can also be compared to interpreted code systems. While we do not achieve the small code sizes attained by interpretation (2-5 times smaller), our programs are much faster. Typical interpreted code systems execute applications 5-20 times slower than native code. In fact, we have native performance for code once it is in the cache since decompression reproduces the original native program. This is particularly effective for loop-oriented programs.

5.3 Software decompressors

This section presents two different software decompressors. The first is a dictionary-based software decompressor designed to be extremely fast. The second is a software version of IBM's CodePack which attains better compression ratios, though it increases execution time.

5.3.1 Dictionary compression

The dictionary compression scheme takes advantage of the observation that the instructions in programs are highly repetitive [Lefurgy97, Lefurgy98]. Each unique 32-bit instruction word in the original program is put in a *dictionary.* Each instruction in the original program is then replaced with a 16-bit index into the dictionary. Because the instruction words are replaced with a short index and because the dictionary overhead is usually small compared to the program size, the compressed version is smaller than the original. Instructions that only appear once in the program are problematic. The index plus the original instruction in the dictionary are larger than the single original instruction, causing a slight expansion from the native representation. Figure 5.1 illustrates the compression method.

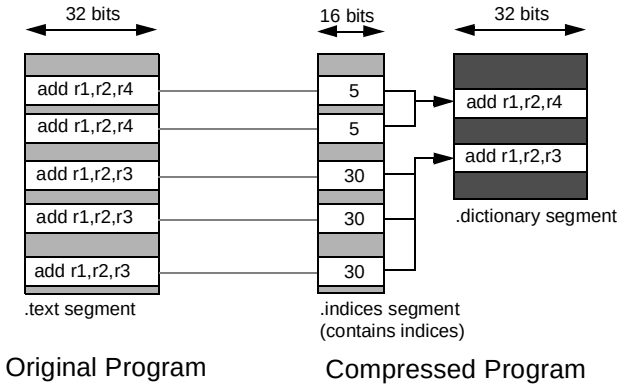


Figure 5.1: Dictionary compression
In this example, the instruction “add r1,r2,r4” maps to the index “5” and the instruction “add r1,r2,r3” maps to the index “30”.

The 16-bit indices limit the dictionary to contain only 64K unique instructions. In practice, this is sufficient for many programs, including our benchmarks. However, programs that use more instructions can be accommodated. Hybrid programs that contain both compressed and native code regions are one solution. When the dictionary is filled, then the remainder of the program is left in the native code region. A cache miss in the native region would use the usual cache controller, but a cache miss in the compressed region would invoke the decompressor. Such a scheme is used in CodePack [IBM98].

Many code compression systems compress instructions to a stream of variable-length codewords. On a cache miss, the compressed code address that corresponds to the native code address must be determined before decompression can begin. This is typically done with a mapping table that translates between the native and compressed code addresses [Wolfe92, IBM98]. The dictionary compression can avoid this table lookup because the native instructions and compressed codewords have fixed lengths. Since the codewords are half the size of instructions, the corresponding codeword can be found at half the distance into the `.indices` segment as the native instruction is into the `.text` segment. Therefore, a simple calculation suffices to map native addresses into compressed addresses and a mapping table is not required.

5.3.2 CodePack

The CodePack software decompressor we present uses the CodePack [IBM98] compression algorithm. This algorithm compresses programs much more than the simple

dictionary compression, but the decompressor takes longer to execute. The CodePack algorithm is described in Section 4.3.1.

The CodePack decompressor is much more complicated than the dictionary method. CodePack works by compressing 16 instructions (2 cache lines) into a group of unaligned variable-length codewords. This constrains the decompressor to serially decode each instruction when the second of the two cache lines is requested. Also, the mapping between the native code addresses and compressed code addresses is complex. A mapping table is used to find the address of the compressed group that corresponds to the missed cache line. This results in one more memory access than our dictionary method.

5.4 Compression architecture

The software decompressors presented here reuse the instruction cache as a decompression buffer. This is similar to the CodePack and CCRP hardware decompressors. On a cache miss, compressed instructions are read from main memory, decompressed, and placed in the instruction cache. The instruction cache contents appear identical to a system without compression. This allows the microprocessor core to be unaware of decompression. The decompression overhead merely appears to be a a long latency cache miss. Compressed instructions execute as quickly as native instructions once they are placed in the instruction cache.

To support software-managed decompression, we require a method to invoke the decompressor on a cache miss and a way to put decompressed instructions into the instruction cache. We can accomplish these by making two modifications to the instruction set architecture. First, the instruction cache miss must raise an exception which invokes the decompression software. Second, there must exist an instruction to modify the contents of the instruction cache. We believe that it is reasonable to expect new processors to provide such capability.

The miss exception is invoked non-speculatively whenever the instruction address misses the cache. This insures that only instructions that are executed will be decompressed and placed in the cache. The miss exception is only raised when the address is in the range of the compressed program. The hardware has two special registers that are set

with the low and high addresses of the program. These addresses are setup before the program is executed. On a miss, the hardware does a simple bounds check using these registers to determine if a miss exception should be taken. This allows systems to have both native and compressed code. When the miss address is outside of the compressed program, then the usual instruction fetch hardware is used to fill the cache. Such systems will be discussed in the next chapter.

Raising an exception on a cache miss is also done in the *informing memory* work by Horowitz et al. [Horowitz98]. On a data cache miss, they raise an exception which runs an exception handler to tune a software data prefetcher. Since the miss exception is non-speculative, it can only be taken after all preceding branches are resolved. They also make the assumption that the branch recovery logic is responsible for redirecting execution to the exception handler on a data cache miss. The exception could also be recognized in the commit stage of the pipeline. The trade-off is that implementing the exception hardware at the commit stage is easier, but the exception handler will be started later. This dissertation assumes that the exception hardware is implemented as part of the branch mis-prediction recovery hardware so that the decompressor can be started as early as possible.

Instructions that modify the instruction cache or instruction memory already exist in microprocessors. For example, the MIPS R10000 `cache` instruction can read and write data, tags, and control bits in the instruction cache. A further example is the MAJC architecture which specifies a special instruction for writing into instruction memory [Gwennap99]. These mechanisms have uses beyond just code compression. Jacob et al. propose using such features to replace hardware-managed address translation performed by the translation-lookaside buffer with software-managed address translation [Jacob97]. Jacob further suggests using software-managed caches to provide fast, deterministic memories in embedded systems [Jacob99].

The following sections describe new instructions added to the SimpleScalar simulator to support software decompression. First, the `swic` instruction for modifying the cache is discussed. This is followed by a discussion of additional instructions that are used in the software decompressors. They are typically found in modern architectures, but were not originally in the SimpleScalar simulator.

5.4.1 Cache modification instruction

swic: store word in instruction cache

Format: `swic rt, offset(rs)`

The address is formed by offset + reg[rs]. The effect of this instruction is to store the value reg[rt] into the instruction cache at the specified address. On a cache miss, the line is allocated and the upper bits of the address are written to the cache line tag. On a cache hit, the appropriate word in the cache line is modified. Since this instruction writes the cache, it would be difficult to squash and restart the instruction on a miss-speculation. Therefore, the processor must be in a non-speculative state before the instruction executes. This is accomplished by allowing the preceding instructions to complete before executing the `swic` instruction.

5.4.2 Synchronization instruction

Dynamic code generation is similar to instruction decompression because both generate code on-the-fly and execute it. In both systems, there must be synchronization between creating the instructions and executing them. This prevents speculatively fetched instructions from executing before the code generation phase has completed writing the instructions.

The PowerPC instruction set serves as an example of how synchronization for dynamic code generation works. A typical instruction sequence used between dynamic code generation and execution on the PowerPC is [Motorola94]:

1. `dcbst` (update memory)
2. `sync` (wait for update)
3. `icbi` (invalidate copy in instruction cache)
4. `isync` (perform context synchronization)

`isync` is an instruction barrier that waits until all previous instructions have completed and then removes any prefetched instructions from the instruction queue before continuing execution.

The above sequence assumes that store instructions have already been used to write instructions into code address space. The instructions may still be in the data cache. Therefore, the first step is to flush all data cache lines that may contain instructions to a

lower level of the memory hierarchy that is visible to instruction memory. This is done with the `dcbst` instruction. The `sync` instruction is a memory barrier that prevents the execution of future instructions until all previous instruction have completed and all previous memory operations are completed. Here, it halts execution until the `dcbst` has completed moving the instructions out of the data cache. In general, the invalidation of the instruction cache with `icbi` is required because the modified code space may already exist in the instruction cache. It is important to remove the mapping so that the new mapping can be fetched from main memory. However, when decompressing instructions, it is already assumed that the instruction is not in the cache — that is why the cache miss exception occurred. Therefore, the software decompressors do not need to invalidate the instruction cache and omit step 3.

We implement `sync` and `isync` in SimpleScalar. The `isync` instruction is always used to prevent cache misses from being fetched before the decompressor is finished. The `sync` instruction is discussed in the next chapter. It is used when accesses to main memory are used to optimize software decompression.

isync: instruction synchronization

Format: `isync`

This instruction halts fetching until all previous instructions have committed. Then all following instructions that have been prefetched are squashed and the instruction fetcher begins fetching at the instruction following the `isync`. This instruction is used to ensure that all `swic` instructions have updated the instruction cache before the decompressor exits and the application attempts to execute decompressed instructions. It also ensures that the application fetches the newly decompressed instructions from the cache and that fetching begins only after all decompression is finished. All of our decompressors execute `isync` at the end of decompression.

5.4.3 Interrupt support instructions

Two instructions, `iret` and `mfc0`, are also used in the software decompressors. On an exception or interrupt, the address of the instruction causing the interrupt is placed in a special machine register. This address is used as the return value when execution of

the exception handler is done. The `iret` instruction restores the program counter from this special register. The decompressor routines locate the compressed code region and associated dictionary by addresses programmed into special registers before the compressed program is executed. At run-time, these special registers are accessed by the decompressor using the `mfc0` instruction.

iret: return from interrupt

Format: `iret`

This instruction moves the interrupt return address (for example, the address of the instruction that caused an instruction cache miss exception) into the program counter. It is the last instruction of every cache miss handler. A similar instruction (rfi) is used on the PowerPC.

mfc0: move from coprocessor-0

Format: `mfc0 rT, rC`

This instruction is similar to the MIPS instruction of the same name. It moves a value from a special register into a general purpose register. The special purpose registers hold information about the compressed program environment. This includes the address of the compressed code, the address of the instruction cache miss, the address of decompression dictionaries and address-translation tables.

5.4.4 Special registers

The software decompressors assume that a few system registers are available to hold parameters used for decompression. The parameters specify where the algorithm can find the compressed code and the associated tables for decompressing it. Software decompression can be used in multi-program environments by merely changing the values of the registers during a context switch. The dictionary decompressor uses 5 special registers. The registers hold 1) the address of the missed instruction, 2) the base address of the indices, 3) the base address of the dictionary, 4) the base address of the original `.text` segment, and 5) the size of the original `.text` segment. The software version of the CodePack decompressor uses an additional register to hold the base address of the mapping table.

SimpleScalar parameters	Values
fetch queue size	1
decode width	1
issue width	1 in-order
commit width	1
Register update unit entries	4
load/store queue	2
function units	alu:1, mult:1, memport:1, fpalu:1, fpmult:1
branch pred	bimode 2048 entries
L1 I-cache	16 KB, 32B lines, 2-assoc, lru
L1 D-cache	8 KB, 16B lines, 2-assoc, lru
memory latency	10 cycle latency, 2 cycle rate
memory width	64 bits

Table 5.1: Simulation Parameters

5.5 Simulation environment

All experiments are performed on the SimpleScalar 3.0 simulator [Burger97] with modifications to support compressed code. The benchmarks come from the SPEC CINT95 and MediaBench suites [SPEC95, Lee97]. The benchmarks are compiled with GCC 2.6.3 using the optimizations “-O3 -funroll-loops” and are statically linked with library code. We shortened the input sets so that the benchmarks would complete in a reasonable amount of time. We run these shortened programs to completion.

SimpleScalar has 64-bit instructions which are loosely encoded, and therefore highly compressible. So as to not exaggerate our compression results, we wanted an instruction set more closely resembling those used in current microprocessors and used by code compression researchers. Therefore, we re-encoded the SimpleScalar instructions to fit within 32 bits. Our encoding is straightforward and resembles the MIPS IV encoding. Most of the effort involved removing unused bits in the 64-bit instructions.

For our baseline simulations we choose a simple architecture that is likely to be found in a low-end embedded processor. This is modeled as a 1-wide issue, in-order, 5-stage pipeline. We simulate only L1 caches and main memory. Main memory has a 64-bit bus. The first access takes 10 cycles and successive accesses take 2 cycles. Table 5.1 shows the simulation parameters.

```
# Load I-cache line with 8 instructions

# Register Use
# r9 : index address
# r10: base address of dictionary
# r11: base of decompressed; index into dictionary
# r12: next cache line addr. (loop halt value)
# r26: indices base and decompressed insn
# r27: insn address to decompress

# Save regs to user stack
# r26,r27 are reserved for OS, do not require saving.
  sw   $9,-4($sp)
  sw   $10,-8($sp)
  sw   $11,-12($sp)
  sw   $12,-16($sp)

# Load system register inputs into general registers
  mfc0 $27,c0[BADVA] # the faulting PC
  mfc0 $26,c0[0]     # decompressed base
  mfc0 $10,c0[1]     # dictionary base
  mfc0 $11,c0[2]     # indices base

# Zero low 5 bits to get cache line addr.
  srl  $27,$27,5
  sll  $27,$27,5     # r27 has the cache line address

# index_address = (C0[BADVA]-C0[0]) >> 1 + C0[2]
  sub  $9,$27,$26    # get offset into decompressed code
  srl  $9,$9,1       # transform to offset into indices
  add  $9,$11,$9     # load r9 with index address

# calculate next line address (stop when we reach it)
  add  $12,$27,32

loop:
  lhu  $11,0($9)     # Put index in r11
  add  $9,$9,2       # index_address++
  sll  $11,$11,2     # scale for 4B dictionary entry
  lw   $26,($11+$10) # r26 holds the instruction
  swic $26,0($27)    # store word in cache
  add  $27,$27,4     # advance insn address
  bne  $27,$12,loop

# Restore registers and return
  lw   $9,-4($sp)
  lw   $10,-8($sp)
  lw   $11,-12($sp)
  lw   $12,-16($sp)
  isync
  iret # return from exception handler
```

Figure 5.2: L1 miss exception handler for dictionary decompression method

5.5.1 Decompression

C language versions of the dictionary and CodePack decompressors are listed in Appendix A.2 and Appendix A.7, respectively. The baseline dictionary decompressor assembly code is shown in Figure 5.2. The decompressor is 208 bytes (26 instructions) and executes 75 instructions to decompress a cache line of eight 4-byte instructions. The size of the CodePack decompressor is 832 bytes (208 instructions) of code and 48 bytes of

data. It decompresses two cache lines on each cache line miss (due to the CodePack algorithm) and takes on average 1120 instructions to do so.

Modern embedded microprocessors, such as ARM, use a second register file to support fast interrupts. During an interrupt or exception, all instructions use the second register file. This allows the interrupt handler to eliminate instructions for saving registers before it begins and restoring registers when it finishes. We have versions of the dictionary and CodePack decompressors that use a second register file in order to measure the benefit for software decompression. The extra registers provided by the second register file also allow us to completely unroll the loop in the dictionary decompressor. This eliminates two add instructions and a branch instruction on each iteration.

It is important that the decompressor not be in danger of replacing itself when it modifies the contents of the instruction cache. Therefore, we assume that the decompressor is locked down in fast memory so that it never incurs a cache-miss itself. Our simulations put the exception handler in its own small on-chip RAM accessed in parallel with the instruction cache.

5.6 Results

This section presents results of our software-managed dictionary and CodePack simulations.

5.6.1 Size results

The size of the native and compressed programs are given in Table 5.2. All results include both application and library code. The dictionary compressed program size includes the indices and dictionary. The CodePack program size includes the indices, dictionary, and mapping table. The decompression code is not included in the compressed program sizes.

Benchmark	Dynamic insns (millions)	16 KB I-cache miss ratio	Original size (bytes)	Dictionary compressed size (bytes)	CodePack compressed size (bytes)	Dictionary compression ratio	CodePack compression ratio	LZRW1 compression ratio
cc1	121	2.93%	1,083,168	707,904	655,216	65.4%	60.5%	60.4%
ghostscript	155	0.04%	1,099,136	762,880	688,736	69.4%	62.7%	61.6%
go	133	2.05%	310,576	216,304	182,816	69.6%	58.9%	63.9%
ijpeg	124	0.07%	198,272	153,104	118,352	77.2%	59.7%	61.5%
mpeg2enc	137	0.01%	118,416	97,424	74,896	82.3%	63.2%	60.2%
pegwit	115	0.01%	88,400	70,144	54,272	79.3%	61.4%	56.2%
perl	109	1.62%	267,568	197,280	162,256	73.7%	60.6%	60.2%
vortex	154	2.05%	495,248	325,920	274,640	65.8%	55.5%	55.5%

Table 5.2: Compression ratio of .text section

Dynamic insns: Number of instructions committed in benchmarks. *Cache miss ratio*: non-speculative cache miss ratio for a 16 KB instruction cache. *Original size*: Size of native code. *Dictionary compressed size*: size of compressed code using dictionary method. *CodePack compressed size*: size of CodePack compressed benchmarks. *Dictionary compression ratio*: Size of dictionary compressed code relative to native code. *CodePack compression ratio*: Size of CodePack compressed code relative to native code. *LZRW1 compression ratio*: Size of whole .text section compressed with LZRW1 algorithm relative to size of native code. This is a lower bound for procedure-based compression using LZRW1.

Benchmark	D	D+RF	CP	CP+RF
cc1	2.99	2.19	17.88	16.91
ghostscript	1.30	1.18	3.46	3.32
go	2.52	1.91	11.14	10.56
ijpeg	1.06	1.03	1.42	1.40
mpeg2enc	1.01	1.00	1.05	1.04
pegwit	1.01	1.01	1.11	1.10
perl	2.15	1.64	11.64	11.02
vortex	2.39	1.80	12.00	11.36

Table 5.3: Slowdown compared to native code

All results are shown as the slowdown of the benchmark when run with compression. A slowdown of 1 means that the code is identical in speed to native code. A slowdown of 2 means that the program ran twice as slow as native code. D: dictionary compression. D+RF: dictionary compression with a second register file. CP: CodePack compression. CP+RF: CodePack compression with a second register file.

5.6.2 Performance results

The performance of the benchmarks is shown in Table 5.3. Programs that use software decompression always have higher execution times than native code versions. Therefore, we present our results in terms of slowdown relative to the speed of native code. A value of 1 represents the speed of native code. A value of 2 means that the benchmark executed twice as slowly as native code. Table 5.2 shows the non-speculative miss ratios for the 16 KB instruction cache used in the simulations.

For all benchmarks, the execution time of dictionary programs is no more than 3 times native code and the execution time of CodePack programs is no more than 18 times native code. Using a second register file reduces the overhead due to dictionary decompression by nearly half. The CodePack algorithm has only a small improvement in performance with a second register file since CodePack does not spend a significant amount of time saving and restoring registers. The multimedia programs (*ghostscript, ijpeg, mpeg2enc*, and *pegwit*) have the best performance among all benchmarks. This is because the benchmarks are loop-oriented and experience fewer cache misses. Therefore, the performance cost of software decompression is amortized over many loop iterations.

Decompression only occurs during a cache miss. Therefore, the way to improve compressed program performance is to make cache misses less frequent or to fill the miss request more quickly. The miss request can be made faster by using hybrid programs which keep some procedures as native code so they will use the hardware cache controller to quickly fill misses. This is discussed in the next chapter. The miss ratio can be reduced by enlarging the cache, increasing cache associativity, applying code placement optimizations to reduce conflict misses, or applying classical optimizations that reduce the native code size.

The instruction cache miss ratio has a strong effect on the performance of the compressed program. We modified the miss ratios of the benchmarks by simulating them with 4 KB, 16 KB, and 64 KB instruction caches. In Figure 5.3, we plot the miss ratios of all the benchmarks under each size of cache against the slowdown in execution time. For dictionary compression, once the instruction cache miss ratio is below 1%, the compressed code is less than 2 times slower than native code. When the miss ratio is below 1% for CodePack programs, the compressed code is less than 5 times slower than native code. Increasing cache size effectively controls slowdown. When considering total memory savings, the cache size should be considered. Having a very large cache only makes sense for the larger programs. Instead of using die area for a larger cache, it may be possible to use the additional area to store some small applications completely as native code.

It is difficult to compare our results with those of Kirovski et al. because different instruction sets and timing models were used. In comparison to our cache-line decompressors, the procedure-based decompression has a much wider variance in performance. They

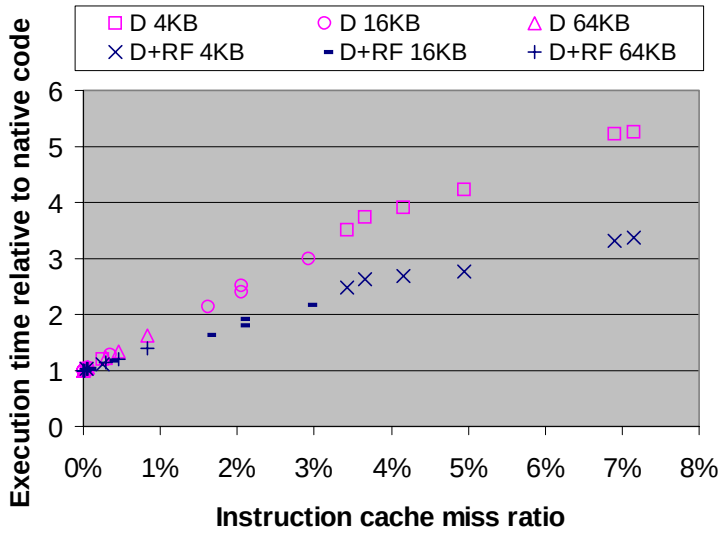
report slowdowns that range from marginal to over 100 times slower (for *cc1* and *go*) than the original programs for 1 KB to 64 KB caches. Both our dictionary and CodePack programs show much more stability in performance over this range of cache sizes. However, the LZRW1 compression sometimes attains better compression ratios. Table 5.2 shows the compression ratios for LZRW1 when compressing the entire `.text` section as one unit. This represents a lower bound for the compression ratio attained when compressing individual procedures. Overall, LZRW1 attains compression ratios similar to CodePack and 5-25% better than dictionary compression.

5.7 Conclusion

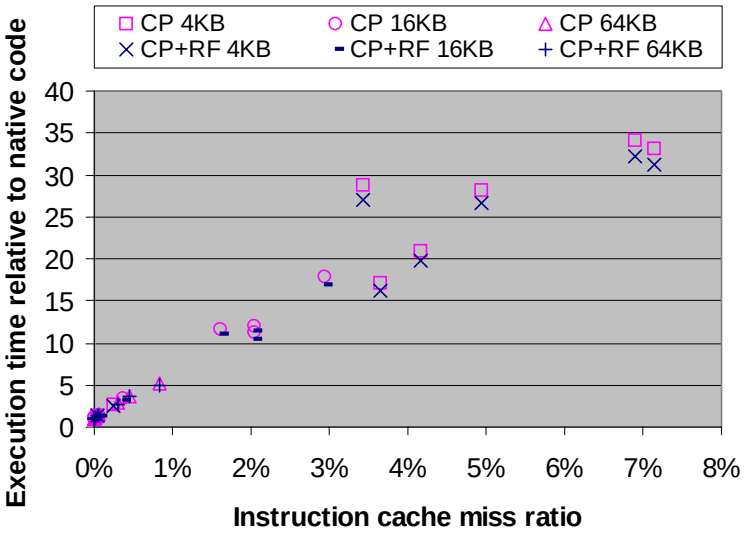
Software decompression allows the designer to easily use code compression in a range of instruction sets and use better compression algorithms as they become available.

We have presented a new technique of using software-managed caches to support code decompression at the granularity of a cache line. In this study we have focused on designing a fast decompressor (rather than generating the smallest code size) in the interest of performance. We have shown that a CodePack software decompressor can perform with significantly less overhead than the software procedure decompression scheme while attaining a similar compression ratio. This is because a decompressor with cache line granularity has an advantage over a decompressor with procedure granularity in that it does not have cache fragmentation management costs and better avoids decompressing instructions that may not be executed. We have shown that a simple highly optimized dictionary compression method can perform even better than CodePack, but at a cost of 5-25% in the compression ratio. The performance of the dictionary method is higher than CodePack because 1) the dictionary and codewords are machine words (or half-words) using their natural alignment as opposed to variable-sized bit-aligned codes, 2) the dictionary compression eliminates the CodePack mapping table by using fixed-length codewords, 3) using a second register file allows the dictionary decompression loop to be unrolled.

Performance loss due to decompression can be mitigated by improving the instruction cache miss ratio or by reducing the amount of time required to service an instruction



(a) Dictionary



(b) CodePack

Figure 5.3: Effect of instruction cache miss ratio on execution time
Data points represent all benchmarks simulated with instruction cache sizes of 4 KB, 16 KB, and 64 KB. a) Dictionary compressed programs. b) CodePack compressed programs.

cache miss. This suggests that incorporating compression with other optimizations that reduce cache misses (such as code placement) could be highly beneficial.

Code decompression can be thought of as interpretation of an instruction set. We believe that decompression fills a gap between interpreted and native code. It attempts to attain the speed of native code and the code density of interpreted code. We presented an instruction, `swic`, for writing instructions into the instruction cache. We believe that such an instruction is not only useful for decompression, but may also be useful for dynamic compilation and high-performance interpreters. It allows instructions be to generated directly into the instruction memory without disturbing the data memory hierarchy with data stores and flushes.

Chapter 6
Optimizations for software-managed decompression

The previous chapter introduced a mechanism for decompressing programs in software. This required modifications to the architecture in the form of an instruction cache miss exception and the `swic` instruction to load instructions into the cache. The experimental data showed that the CodePack software decompressor had very large overhead. CodePack compressed programs were up to 17 times slower than native code. A L1-cache miss takes 10 cycles to copy a cache line from main memory to the cache, but decompressing the same line takes hundreds of cycles. There are several ways to reduce this cost. Chapter 5 demonstrates that using larger instruction caches reduces the need to decompress and therefore improves decompression overhead. This chapter examines software optimizations for improving the overhead of decompression.

6.1 Introduction

From the viewpoint of the microprocessor, decompression overhead looks like a long latency cache miss. One method to reduce decompression overhead is to avoid cache misses. This was addressed in the previous chapter by evaluating caches with different capacities. Another method to reduce overhead is to reduce the latency of the decompressor. This chapter proposes two optimizations to accomplish this. Both optimizations attempt to reduce the latency of the decompressor by reducing the number of times the decompressor is used. One optimization occurs at compile-time guided by an execution profile. The other optimization occurs at run-time using dynamic information about the program execution behavior. The cost of using either of these optimizations to reduce decompression latency is that the compressed program size increases.

The first optimization is to use *hybrid programs* which use both native and compressed code. The compiler or a post-compilation tool generates either native code or compressed code for each procedure. When native code misses the cache, the instruction fetch hardware fetches the missed instructions from a backing memory. This operation is much quicker than executing the software decompressor. This is a static optimization since the selection of compressed and native code is done once before the program is executed. This method also requires that a profile be used to select which code is compressed. This chapter analyzes two profiling methods. Using this optimization increases the size of the compressed program due to the addition of native instructions.

The second optimization, *memoization*, is a well-known program transformation. [Michie68]. The goal of this optimization is to avoid long-latency computations by caching the result of function calls. If the same function is called again with identical input values, the result can be provided from the cache rather than calculated. In the proposed compressed code system, an instruction cache hit can be considered a form of memoization because the decompressor is not called. This can be extended by using a dense backing memory as a memoization table to hold decompressed cache lines. A DRAM memory can hold many more decompressed cache lines per unit area than the SRAM instruction cache can. This avoids decompression when the DRAM holds the requested line. In this case, the requested line is copied from the DRAM into the instruction cache. This is a dynamic optimization because the decision of what code should remain decompressed in the DRAM continually changes as the application is executed. This technique is valuable when a profile for generating a hybrid program is not available. This chapter analyzes different approaches to managing the memoization table. The use of the memoization table requires a system to have additional memory to execute the compressed program. In the experimental results, the size of the memoization table in included in the overall size of the compressed program.

The two optimizations can be combined. A hybrid program can select procedures that should remain highly available and leave them as native code. Memoization will dynamically select the remaining compressed code to fill the memoization table. This chapter shows results for the combined optimizations the performance/area trade-off that can be made.

This chapter will first show the technique of using hybrid programs in Section 6.2. Next, the memoization optimization will be covered in Section 6.3. Finally, performance and area results for the combination of both optimizations will be shown in Section 6.4. A summary of the chapter is given in Section 6.5

6.2 Hybrid programs

Hybrid programs use a combination of native and compressed code to attain a balance between code size and performance. The technique of selecting which procedures in an application should be native code and which should be compressed is called *selective compression*. Typically, a profile of the program execution is used to guide selective compression. Infrequently used procedures will be compressed to improve code density while frequently used procedures are left as native code to reduce the time that the decompressor is executed. While selective compression is not a new technique, little has been written about how it performs over a wide variety of programs. In the following sections, we investigate two methods of selecting the native code functions: *execution-based* selection and *miss-based* selection. We show that selection based on cache miss profiles can substantially outperform the usual execution time based profiles for some benchmarks.

6.2.1 Execution-based selection

Execution-based selection is used in existing code compression systems such as MIPS16 [Kissell97] and Thumb [ARM95]. These systems select procedures to compress with a procedure execution frequency profile [Pittman87, Greenhills98]. Performance loss occurs each time compressed instructions are executed because it typically takes 15%-20% more 16-bit instructions to emulate 32-bit instructions. This is because 16-bit instruction sets are less expressive than 32-bit instruction sets, which causes the number of instructions executed in the 16-bit instruction programs to increase. Therefore, to obtain high performance, the most highly executed procedures should not be compressed.

We implement and measure the effect of execution-based selection. First, we profile a benchmark and count the number of dynamic instructions executed per procedure. Then we make a list of procedures and sort them by the number of dynamic instructions.

The procedures with the most instructions are selected from this list until a constraint is met. In our experiments we stop selection once the selected procedures account for 5%, 10%, 15%, 20%, or 50% of all instructions executed in the program. The selected procedures are left as native code and the remaining procedures are compressed. Our constraints produce programs with different ratios of native and compressed code to evaluate the selective compression technique.

6.2.2 Miss-based selection

Our dictionary and CodePack decompressors are only invoked during an instruction cache miss. Thus, all performance loss due to decompression occurs on the cache miss path. In this case, it makes sense to select procedures based on the number of cache misses, rather than the number of executed instructions. Compressed code cache misses take longer to fulfill than native code cache misses. Therefore, we can speed up programs by taking the procedures that have the most cache misses and selecting them to be native code.

Miss-based selection is implemented similarly to execution-based selection. Procedures are sorted by the number of instruction cache misses they cause. Only non-speculative misses are counted since the decompressor is only invoked for non-speculative misses. The selection process continues until the selected procedures account for 5%, 10%, 15%, 20%, or 50% of all cache misses in the program.

6.2.3 Simulation of selective compression

Our compression software has the ability to produce binaries containing both compressed and non-compressed code regions. We profile the benchmarks to measure cache miss frequency and execution frequency for each procedure. The profile is used to determine which procedures are native code and which are compressed. In execution-based selective compression, the top several functions that are responsible for most of the dynamically executed instructions are not compressed. In miss-based selective compression, the top several functions that are responsible for most of the instruction cache misses

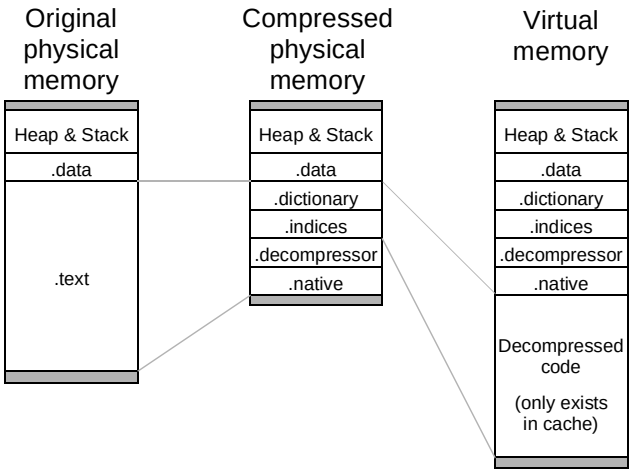


Figure 6.1: Memory layout for dictionary compression
The `.text` segment is compressed into `.dictionary` and `.indices` segments. The `.decompressor` segment holds the decompressor code. Some code in `.text` may be left in `.native` so that decompression does not need to occur for critical code. On a cache miss, the `.dictionary` and `.indices` segments are decompressed and put in the segment marked "Decompressed code". This segment only exists in the instruction cache and does not consume main memory. CodePack programs have an additional segment (not shown) that contains a mapping table to convert cache miss addresses to the addresses of the corresponding variable-length codewords.

are not compressed. Our experiments vary the aggressiveness of the selection algorithm to measure the effect of the trade-off between code size and speed.

Our memory layout for native and compressed programs is shown in Figure 6.1. On a cache miss, we determine if the instruction is in the compressed region or the native region. Our machine model assumes that the regions for compressed and native code can be programmed into special system registers that the microprocessor can use to determine when a cache miss exception should be used. If the cache miss occurs in the compressed region, an exception is raised to invoke the software decompressor. If the instruction is in the native region, then the usual cache controller is used to fill the miss from main memory.

6.2.4 Results

Figure 6.2 shows the results of using both miss-based and execution-based selective compression on dictionary and CodePack programs. The left side of these area/performance curves represent code that is totally compressed. The right side of the curves

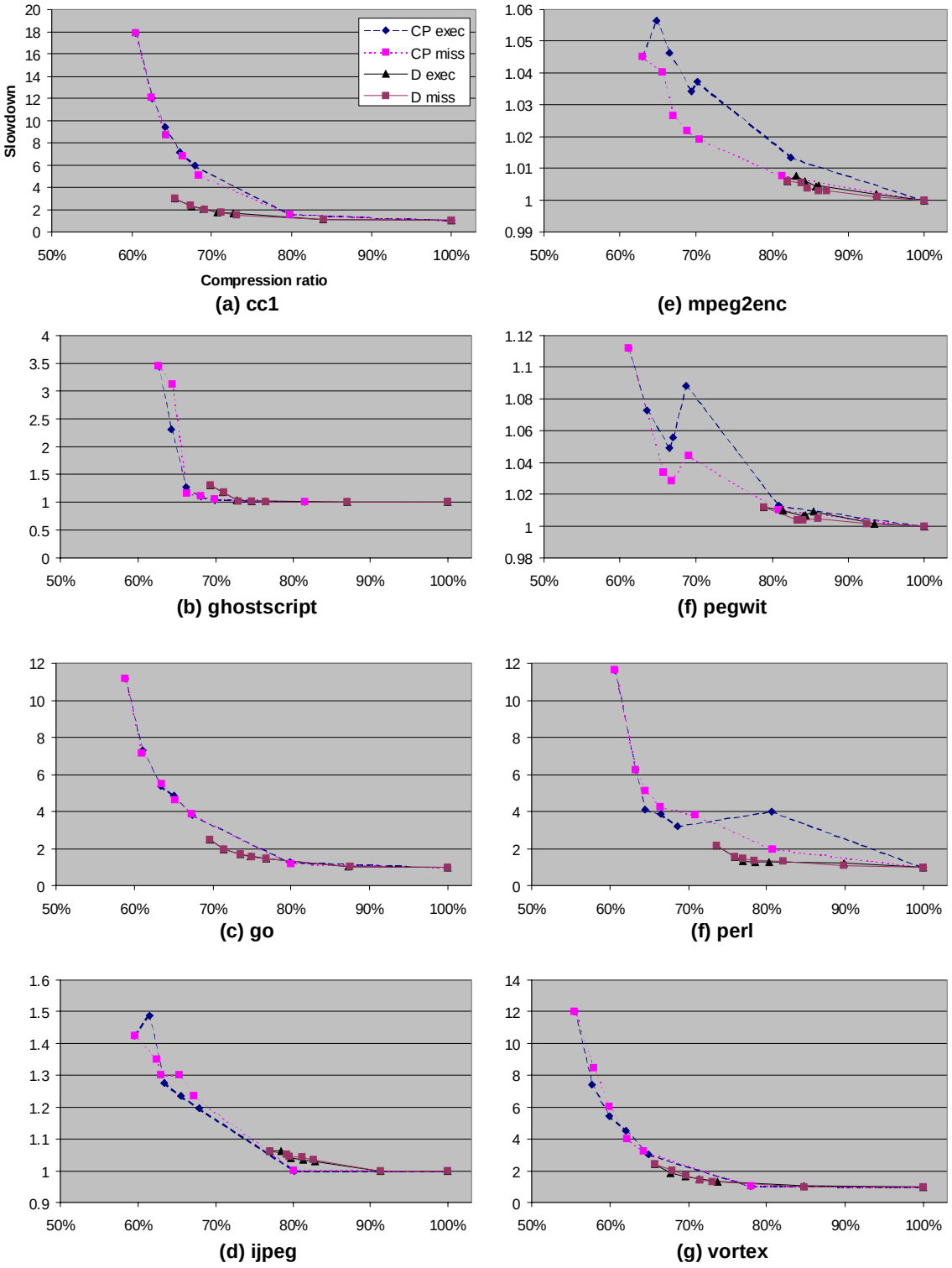


Figure 6.2: Selective compression

These graphs show size/speed curves for CodePack (CP) and dictionary (D) programs for both miss-based (miss) and execution-based (exec) selective compression. The data points from left to right range from fully compressed code to fully native code. Intermediate data points represent hybrid programs with both native and compressed procedures.

represent the original native application. The data points in-between the endpoints represent hybrid programs that have both compressed and native code. The amount of native code in each hybrid program is explained in Section 6.2.1.

Comparison of profiling methods

An interesting result we found is that miss-based profiling can reduce the decompression overhead by up to 50% over execution-based profiling. This occurs in loop-oriented programs such as *mpeg2enc* and *pegwit*. The reason for this is that the profiles cause the selective compression to make opposite decisions in loop regions. Execution-based profiling selects loops to be native code while miss-based profiling compresses loops. The execution-based profiling is helpful for instruction sets such as Thumb and MIPS16 because loops containing 16-bit instructions will be rewritten using fewer 32-bit instructions which execute faster. However, in the software decompressors, the decompression overhead only occurs on an instruction cache miss. The decompressed code will execute as quickly as the native version once it has paid the decompression penalty since decompression recovers the original native instructions. Therefore, it does not make sense to count the number of executed instructions. Loops experience a decompression penalty only on a cache miss and this penalty is amortized over many loop iterations. The miss-based profile more accurately accounts for the cost of the cache miss path and tends to compress loops since they cause few cache misses. *Mpeg2enc* and *pegwit* are the only two benchmarks for which miss-based selection always performs better than execution-based selection. For non-loop programs, the execution-based profiling approximates miss-based profiling since procedures that are called frequently are likely to also miss the cache frequently. Based on these results, we conclude that all compressed code systems that invoke decompression on a cache miss should use miss-based profiling for loop-oriented programs.

Comparison of dictionary and CodePack decompressors

We have already shown that CodePack always attains a better code size than dictionary programs at a cost in performance. However, selective compression shows that CodePack can sometimes provide better size and performance than dictionary compression (*ijpeg* and *ghostscript*). CodePack compresses instructions to a much higher degree

than dictionary compression. When selective compression is used, this allows CodePack to have more native code than dictionary programs, but still have a smaller code size. If the selected native procedures in CodePack provide enough performance benefit to overcome the overhead of CodePack decompression relative to dictionary decompression, then the CodePack program can run faster than the dictionary program. This suggests that it is worthwhile to investigate software decompressors that can attain even higher levels of compression with a higher decompression overhead.

Effect of procedure placement

It seems counterintuitive that some benchmarks (*ijpeg*, *mpeg2enc*, *perl*, and *pegwit*) occasionally perform worse when they use more native code. This is a side-effect of our compression implementation. The assumption that compressed procedures are adjacent to each other in memory simplifies the cache miss exception logic. It can perform a simple bounds check to determine if the miss address is in the compressed region and raise an exception if appropriate. However, distributing the procedures across the native code and compressed code memory regions modifies the order of procedures in the program. Within each region, the procedures have the same ordering as in the original program. However, procedures that were adjacent in the original program may now be in different regions of memory and have new neighbors. This causes the hybrid programs and the original native program to experience different instruction cache conflict misses. It is clear from Figure 5.3 that even small changes in the instruction cache miss ratio can dramatically affect performance of compressed programs. This is because the decompression software greatly extends the latency of each instruction cache miss. Therefore, it is possible that a poor procedure placement could overwhelm the benefit of using hybrid programs. The effect of procedure placement in hybrid programs is illustrated in Figure 6.3.

The effects of procedure placement on performance can be significant. Pettis and Hansen noted that a good procedure placement could improve execution time by up to 10% [Pettis90]. To our knowledge, we are the first to report on the effect of procedure placement on selective compression. Procedure placement is likely to affect selective compression in other compression systems too. For example, IBM's hardware implementation of CodePack also uses compressed and native code regions which will alter proce-

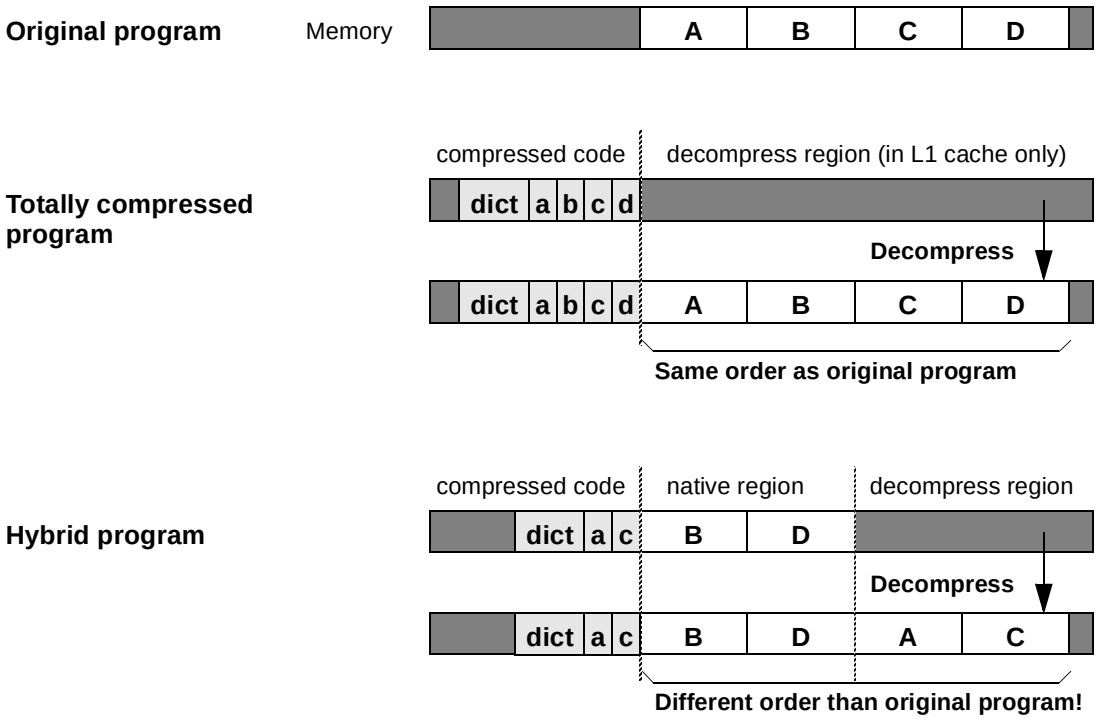


Figure 6.3: Procedure placement in hybrid programs
This example shows that hybrid programs have a different procedure layout than totally compressed and totally native programs. *A*, *B*, *C*, and *D* are native code procedures. The compressed procedures are *a*, *b*, *c*, and *d*. The dictionary is labeled *dict*.

dure placement as procedures are selected to remain as native code. If it were not for the effect of procedure placement, all benchmarks would run faster under miss-based selection than execution-based selection since miss-based selection models the decompression overhead better in our cache-line decompression system. This assumes that the profiles used by the selective compression are representative of the execution behavior of the applications.

One serious problem with miss-based selection is that the selection algorithm uses the cache miss profile from the original code. Once the code is rearranged into native and compressed regions, the new procedure placement will have a different cache miss profile when the program is executed. This new placement may cause more or less cache misses than the original program (and possibly the execution-based selection procedure placement). Nevertheless, we still find miss-based selection useful for loop-oriented benchmarks. These problems suggest that an interesting area for future work would be to develop a unified selective compression and code placement framework.

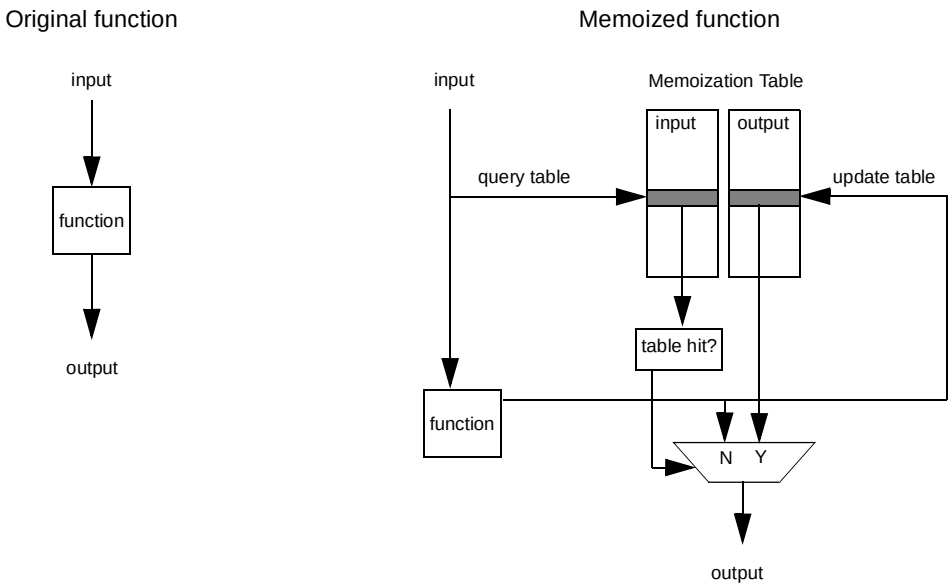


Figure 6.4: Memoization overview
In general, memoization is a function-level optimization. Recent input and output pairs are stored. When future inputs match the stored inputs, the corresponding output value is returned rather than re-computed. This avoids the computational overhead of invoking the original algorithm. When memoization is used in the decompression software, the input value is the address that missed the cache and the output is one or two decompressed cache lines.

6.2.5 Conclusion

Hybrid programs are effective for improving the performance of compressed code. Dramatic results can be achieved by leaving a few procedures as native code and compressing the others. Most of the time, a simple execution profile could be used instead of a cache miss profile (which must be done for each cache organization on which the program will execute). However, we have seen that there can be a benefit for using miss-based profiling on loop-oriented programs such as pegwit and mpeg2enc.

6.3 Memoization

In this section, we examine using memoization to reduce decompressor overhead. Figure 6.4 illustrates the process of memoization. Whenever decompression is done, the results will be cached in the memoization table. The decompression algorithm is modified to first check the table for the decompressed cache line. If the line appears in the table, it can be quickly copied into the instruction cache, thus eliminating most of the decompres-

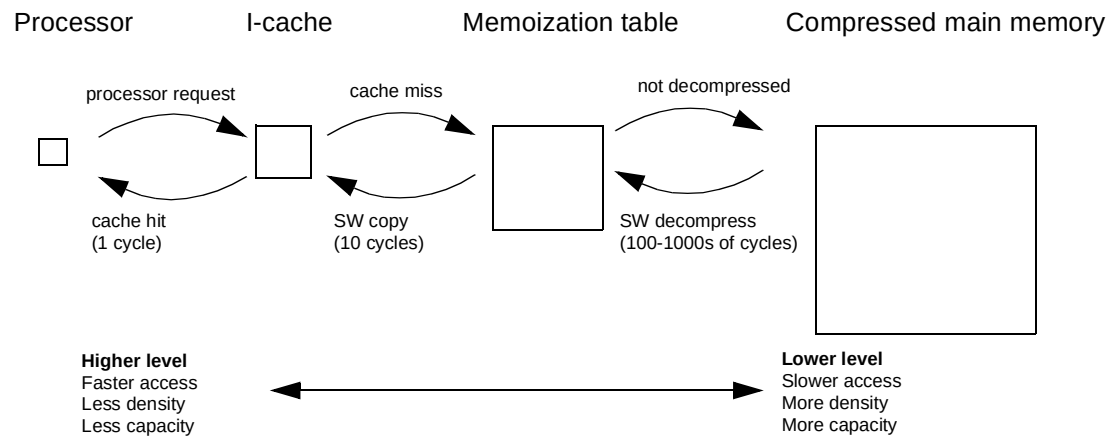


Figure 6.5: Memory hierarchy

sion penalty. We assume that the memoization table is implemented in a smaller, denser, and slower memory than the instruction cache.

The effect of the memoization table is to add another level to the memory hierarchy. This is illustrated in Figure 6.5. When an instruction cache miss occurs, the decompressor first looks in the memoization table. If the instructions are present, they are copied into the instruction cache. If the instructions are not present, then the normal decompression algorithm is invoked. A performance gain over fully compressed code results when the hit ratio of the memoization table is high and the copy operation has less latency than decompressing.

This section shows experimental results for the effect of memoization on compressed program size and performance. In addition, instruction set support for memoization is proposed and analyzed.

6.3.1 Background

Using memoization to improve decompression speed is reminiscent of Douglis's compression cache [Douglis93] discussed in Chapter 2. Both systems view compression as an additional level of the memory hierarchy. The compression cache held compressed memory pages of the virtual memory system before they were swapped to disk. This system allowed more pages to remain in memory (in a compressed form). Performance improved because the virtual memory system could decompress compressed pages in

memory more quickly than it could transfer the original pages from disk. While Douglis compressed both instruction and data pages, we compress only instructions. Instead of using a compression cache to hold compressed code, our work uses a memoization table to hold decompressed code. Copying the decompressed instructions into the instruction cache is faster than running the decompressor and results in performance improvement.

6.3.2 Memoization table management policies

This section describes two policies for managing the instruction cache and memoization table. The *inclusive* policy is designed to shorten the overhead of memoization. The *exclusive* policy is designed to improve utilization of the memoization table over the inclusive policy. The memoization table is organized as a direct-mapped cache (with tag and data sections). This allows hits to the decompressed cache to be fast because only a single tag must be checked.

Inclusive policy

On a memoization table miss, one (dictionary) or two (CodePack) cache lines are decompressed and put into the instruction cache. Simultaneously, the instructions are written to the memoization table along with the instruction cache miss address tag. The memoization table hits when the miss address is found in the tag store. On a hit, one (dictionary) or two (CodePack) lines are copied from the memoization table into the instruction cache.

Exclusive policy

This policy uses the memoization table to hold lines replaced in the instruction cache. On a memoization table miss, replaced lines are moved to the memoization table. The purpose of this is to avoid decompressing them again when the application needs more associativity than the cache can provide. The replaced instruction cache line does not merely hold a copy of the instructions from the memoization table, so the utilization of the memoization table and the cache is higher than in the inclusive cache policy. Since the replaced cache line is stored in the memoization table, the memoization table resembles a victim cache [Jouppi90].

This policy requires that both decompressed lines of the CodePack-compressed algorithm exist in the cache together so they can be both copied into the memoization table. In a typical instruction cache, one line of the CodePack pair might have already been replaced. To prevent this, we explicitly manage the replacement of lines in the instruction cache so that both CodePack lines are replaced together. This insures that on a table hit, both lines will be available in the memoization table to copy into the instruction cache.

On a memoization table hit, one or two cache lines are copied into the instruction cache. This is identical to the behavior of the inclusive policy. This is different than the victim cache policy which would swap the lines in the instruction cache and memoization table. Lines are not swapped so that the common case (hit) executes quickly in software.

6.3.3 Instruction set support for memoization

Chapter 5 introduced the instructions `swic`, `iret`, `mcf0`, and `isync` to support software decompression. This section introduces additional instructions for the SimpleScalar simulator to support memoization of decompression efficiently.

Many modern instruction sets, such as MIPS [MIPS96] and PowerPC [Motorola94], support cache control instructions. Typical instructions allow modification of the tag bits, control bits, and particular words in the cache. In addition entire cache lines may be prefetched, zeroed, or flushed from the cache. We have added `igettag`, `igetctrl`, `swicw`, `lwicw`, `imap`, `imapw`, `iunmap`, and `iunmapw`. A typical decompressor only needs a few of these instructions. Additionally, the `sync` instruction is required for some software decompressors. More discussion on synchronization instructions is in Chapter 5.

Discussion of cache management

There are two reasons for adding new instructions to support memoization. First, the copy operation between the memoization table and the instruction cache must be very fast so that memoization has low latency. This is done by allowing the decompressor to access complete cache lines instead of individual words in the cache. Second, allowing the decompression software to manage the instruction cache replacement policy is necessary

for the CodePack decompressor. The implementation cost of these new instructions should be very small because the existing data paths can be used. For example, the data paths to move complete instruction cache lines into the instruction cache are already supported by the cache fill hardware.

On a hit to the memoization table, the copy operation must be extremely fast to reduce decompression overhead. The following instruction sequence copies one instruction from the memoization table into the instruction cache:

```
load r1, tableEntryAddr
swic r1, missAddr
```

By incrementing the addresses and repeating the sequence several times, an entire cache line can be copied. However, this is inefficient because it only loads one instruction at a time over the bus from the memory. We assume that the bus is wider than a single instruction and could potentially transfer many instructions at once. Therefore, we introduce the `map` instruction that can read an entire cache line from memory into the instruction cache. It uses the full bandwidth of the memory system to move the cache line. The `map` instruction may use multiple bus accesses if the cache line is longer than the bus width. This is identical to how a cache miss would be handled in hardware. An additional benefit is that the transfer can occur directly between the main memory and the instruction cache. This eliminates a series of load instructions that would move the instructions into the data cache and displace application data.

`map` is not merely a simple load instruction. The address of the cache line in the memoization table is different from the address that the microprocessor uses to access the cache. Therefore, the `map` instruction "maps" the cache line to a new address in the cache. In effect, it chooses a line in the cache to replace and updates the tag value with the new address. If the memoization table address and the cache address are the same, then the `map` instruction behaves as a conventional instruction prefetch.

Some variations on the `map` instruction are also used in the software decompressors. The `iunmap` instruction copies an instruction cache line from the cache into the memoization table. This is used in the exclusive policy to store replaced cache lines in the memoization table.

Granularity	Operating mode	
	Hit	Index
1 instruction	`swic`	`swicw, lwicw`
1 cache line	`imap, iunmap`	`imapw, iunmapw, igettag, igetctrl`

Table 6.1: Taxonomy of cache access instructions
The *hit* operating mode checks the tag of the cache line. The *index* operating mode does not check the cache tags. Instead, the specific cache line to access is completely specified by the address. The lower bits of the address specify the way of the cache set to use.

Some instructions have been added to allow the decompression software to control the replacement of cache lines. The `igetctrl` instruction reads the control bits from a cache line. This provides the decompressor with the replacement (LRU) bits so it can determine which line in the instruction to replace when using the exclusive policy. The `igettag` instruction allows the decompressor to read the tag from a particular cache line so it can be placed in the memoization table. The `swicw` and `lwicw` instructions are similar to `swic`, but store and load instruction words in specific cache lines of the cache. The `imapw` and `iunmapw` instructions are similar to `imap` and `iunmap`, but allow the cache way to be specified so particular lines in the cache set can be read or written.

There are two operating modes for the cache instructions. They are similar to those used by the MIPS `cache` instruction. The first operating mode, called *hit*, searches for an address in the cache and acts upon the line in the cache that the address is found. If the tag of the cache does not match the address, then no action is taken. The second operating mode, called *index*, uses the address to determine a particular line in the cache to act upon, regardless of what the contents of the line are. The upper bits of the address select the cache set to use. The low order bits in the address select the way of the set to use. In the simulations, all caches are 2-way set associative. Therefore, the least significant bit of the address selects way-0 or way-1. Table 6.1 shows the operating mode used by each cache instruction. In addition, it shows whether the instruction acts upon one word or a complete cache line.

The new instructions are listed below:

igetctrl: read instruction cache line control bits
Format: `igetctrl rt, offset(rs)`

Operating mode: index

The address is formed by offset + reg[rs]. The control bits corresponding to the selected cache line are written to reg[rt]. The control bits are the replacement bit and the valid bit. When the replacement bit is 0, way-0 is replaced next. When the replacement bit is 1, way-1 is replaced next.

igettag: read instruction cache tag

Format: `igettag rt, offset(rs)`

Operating mode: index

The address is formed by offset + reg[rs]. The tag corresponding to the selected cache line is written to reg[rt].

swicw: store word in instruction cache way

Format: `swicw rt, offset(rs)`

Operating mode: index

This instruction is similar to swic, but uses the index operating mode. The address (offset + reg[rs]) selects the cache set and way. The value reg[rt] is written into the selected cache line at the offset specified in the address. The tag of the cache line is written with the most significant bits of the address.

lwicw: load instruction word way

Format: `lwicw rt, offset(rs)`

Operating mode: index

The address is formed by offset + reg[rs]. The value of the selected instruction cache word in the line is written to reg[rt]. There is no corresponding `lwic` instruction defined in this dissertation because the software decompressors do not require it.

imap: load and map instruction cache line

Format: `imap rt, offset(rs)`

Operating mode: hit

The source address is reg[rt] and the destination address is offset + reg[rs]. The source address is used to read a cache line from the instruction memory hierarchy (bypassing the instruction cache). The cache line is stored in the instruction cache at the destina-

tion address. The cache tag is updated to correspond with the destination address. When the source and destination addresses are identical, the effect is like a conventional instruction prefetch. When the values are different, the data at the source address is effectively mapped to the destination address. The microprocessor can then access the cache at the destination address and find the data from the source address. This instruction is the high-performance counterpart of `swic`. While `swic` transfers one instruction into the cache, `imap` transfers an entire cache line of instructions.

iunmap: store and map instruction cache line

Format: `iunmap rt, offset(rs)`

Operating mode: hit

The source address is offset + reg[rs] and the destination address is reg[rt]. On a cache hit, the entire cache line at the source address in the cache is copied into the next level of the instruction memory hierarchy at the destination address. On a cache miss, the instruction has no effect.

imapw: load and map instruction cache line way

Format: `imapw rt, offset(rs)`

Operating mode: index

This instruction is similar to `imap`, except that it uses the index operating mode to access the cache.

iunmapw: store and map cache line way

Format: `iunmapw rt, offset(rs)`

Operating mode: index

This instruction is similar to iunmap, except that it uses the index operating mode to access the cache.

sync: synchronization

Format: `sync`

This instruction halts fetching until all previous instructions have committed and all previously initiated memory references (stores and loads) have completed. In conventional machines, this instruction is used after a data cache flush to ensure that modifica-

tions to the instruction space will be found by the instruction cache fill unit. The effect of they `sync` is to wait until the instruction modifications have moved from the data memory hierarchy into a level of memory that is visible to instruction memory accesses. In software decompression, the `sync` instruction is used after the `imap` instruction to ensure that the instruction cache is loaded with decompressed instructions before the application begins to fetch and execute them. In this case it is not merely enough to execute `isync` after decompression because the `imap` instruction may have been committed in the execution pipeline, but still be in the process of accessing main memory. If the `imap` instruction is being used as a conventional instruction prefetch (not done in this dissertation), then the `sync` instruction is not required for correct execution. Hardware could be used to identify the execution of `imap` and restrict fetching from instruction cache addresses that conflict with it. However, we assume that code generation is a rare event that should not require special hardware support. Therefore, we take the same approach as PowerPC and MIPS-IV to instruction memory modification and issue synchronization instructions when appropriate.

6.3.4 Memoization implementation

This section explains how the proposed cache instructions are used in the decompressors. Since the memoization policies (inclusive and exclusive) and the access types (word or cache line) are orthogonal, they can be combined to form four different decompressor routines. Each decompressor is described by a two letter name. These four decompressors are called *IW*, *IL*, *EW*, and *EL*. The inclusive and exclusive memoization policies are labeled “I” and “E”, respectively. The decompressors that use word and line accesses to the instruction cache are labeled “W” and “L”, respectively.

This section shows decompressor code listings to illustrate the use of the proposed cache instructions. Since the dictionary and CodePack decompressors use the cache instructions similarly, only examples of the dictionary decompressor are shown. Complete code listings for all decompressors can be found in Appendix A. The dictionary decompressor is discussed in Chapter 5. The cache management instructions to support software decompression appear in **bold** typeface. In the listings, the cache instructions (as well as normal load and store instructions) appear as macro instructions. A complete list of mac-

Memoization Table Size	Instructions	Tags
4.5 KB	4 KB	0.5 KB
9 KB	8 KB	1 KB
18 KB	16 KB	2 KB
36 KB	32 KB	4 KB
72 KB	64 KB	8 KB
144 KB	128 KB	16 KB
288 KB	256 KB	32 KB

Table 6.2: Memoization table contents

ros is given in Appendix A.1. The source code shown assumes the use of a 4 KB 2-way set-associative 32B-line instruction cache and a 18 KB memoization table. Every memoization table used in the experiments has one 4-byte tag for each cache line of eight instructions. Therefore a 18 KB memoization table has 16 KB of instructions and 2 KB of tags. Table 6.2 shows the number of instruction and tag bytes in memoization tables of various sizes.

Figure 6.6 shows the general form of the decompressor and how it is modified to support memoization. The memoized program first checks the memoization table for decompressed code. If a match is found, then the decompressed instructions are copied into the instruction cache. Otherwise, the decompressor generates instructions from compressed code and then updates the memoization table.

All decompressors end with the `isync` instruction. While the decompressor is modifying the instruction cache, it is possibly that the instruction fetch stage is has already fetched instructions beyond the end of the decompressor. This means that the processor is fetching instructions from the cache at the same address in the cache that the decompressor is modifying. The prefetched instructions will signal a cache miss (if the tag was not yet written), or receive invalid instruction words (if the entire cache line was not yet written). The `isync` instruction prevents these instructions from executing by stalling the pipeline until all previous instructions (from the decompressor) have completed and then discarding the contents of the instruction fetch buffer. Instruction fetch is then redirected to the instruction following the `isync`.

Figure 6.7 shows the Memo-IW decompressor. First the data and tag entries in the memoization table are located. The tag value is loaded and compared the miss address. On

a) General form of decompressor

```
{
  DECOMPRESS:
    // Decompression code {...}

  DONE:
    isync;
    iret;
}
```

b) General form of decompressor with memoization

```
{
  MEMO:
    if (memoized)
    {
       // copy memoized decompression into cache
       goto DONE:
    }

  DECOMPRESS:
    // Decompression code {...}

  UPDATE:
    // Update memoization table

  DONE:
    isync;
    iret;
}
```

Figure 6.6: Memoized decompressor

a match, the table data is copied into the instruction cache using `swic` instructions. For convenience the update of the memoization table is done immediately as each instruction is decompressed. Finally, the tag value in the memoization table is updated with the miss address. A significant problem with this program is that many load and `swic` instructions must be issued to copy an entire cache line from the memoization table. The bus from main memory into the instruction cache is not used efficiently because it is only transferring one word at a time.

Figure 6.8 shows the Memo-IL code. The primary change from the Memo-IW code is that the `imap` and `iunmap` instructions are used to move an entire cache line at once. This reduces the impact of issuing multiple `swic` instructions in the Memo-IW code. This code has higher performance because it allows the complete bandwidth of the instruction cache bus to be utilized.

```
// Assume baddr holds the cache block address of missed instruction

MEMO:
  cacheDataBase = (0x20) << 16; // Memo Data at address 0x00200000
  cacheTagBase = (0x21) << 16;  // Memo Tags at address 0x00210000
  cacheTagOffset = ((baddr) << 18) >> 21;
  mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
  cacheDataOffset = (baddr << 18) >> 18;
  cacheDataAddr = cacheDataBase + cacheDataOffset;
  if (cacheTag == baddr) // match in memoization table?
    {
      // Copy instruction 1 of 8
      mem_load_4B(cacheDataAddr,0,iword);
      swic(baddr,0,iword);
      ...
      // Copy instruction 8 of 8
      mem_load_4B(cacheDataAddr,28,iword);
      swic(baddr,28,iword);

      goto DONE;
    }

DECOMPRESS:
  // Decompression code
  {
    // Initializtion code (for indexAddr and dict_base) {...}

    // decompress instruction 1 of 8
    mem_loadu_2B(indexAddr,0,index);
    index <<= 2;
    mem_load_4B_RR(dict_base,index,iword);
    swic(baddr,0,iword);
    mem_store_4B(cacheData,0,iword); // update memo table

    // decompress 7 more instructions (similar to first instruction)
  }

UPDATE:  // miss memoization table
  mem_store_4B_RR(cacheTagBase,cacheTagOffset,baddr); // update tag

DONE:
  isync;
  iret;
```

A

B

C

D

E

Figure 6.7: Memo-IW
A) Load the tag from the memoization table. B) Compute the address the entry in the memoization table that holds the cache line. C) On a memoization table hit, copy the cache line into the cache. A series of eight load and store instructions copy the decompressed cache line from the table to the cache. D) On a miss, decompress the cache line and put a copy into the memoization table. E) Update the tag store for the cache line that was just written into the memoization table.

Figure 6.9 shows the Memo-EW code. It is similar to Memo-IW, but uses the exclusive memoization policy. Note that the update phase precedes the decompression.

```
// Assume baddr holds the cache block address of missed instruction

MEMO:
  // initialize cacheTag as in Memo-IW handler {...}
  if (cacheTag == baddr) // match in memoization table?
    {
      // Copy memo data into cache line
      imap(baddr,0,cacheDataAddr);
      sync;

      goto DONE;
    }

DECOMPRESS:
  // Normal decompression code using swic
  {...}

UPDATE:  // update memoization table
  mem_store_4B_RR(cacheTagBase,cacheTagOffset,baddr); // update tag
  iunmap(baddr,0,cacheDataAddr); // copy new cache line to memo table

DONE:
  isync;
  iret;
```

Figure 6.8: Memo-IL
The difference between Memo-IW and Memo-IL is that some `swic` instructions (as well as some loads and stores) are replaced with `imap` and `iunmap` instructions.

This happens because the memoization table is updated with the contents being replaced in the instruction cache.

The CodePack decompressor must decompress two lines at a time. For ease of implementation, the memoization assumes that either both decompressed lines are in the memoization table, or none of them are. This avoids overhead for supporting the case where one line is in the memoization table, but the other line must be decompressed. This requires that when the memoization table is updated, both cache lines are in the cache to copy into the table. Therefore, when the two cache lines are placed into the cache, they are put in the same way of the cache. This is a simple method to guarantee that one line of the pair is not overwritten by other decompression events. When one line is overwritten, then its pair line will also be overwritten. In the decompressor, the replacement bits of the first cache line are used to choose the way for both cache lines.

Figure 6.10 shows the Memo-EL code. It replaces use of `swic` and `lwicw` in Memo-EW with `imapw` and `iunmap`. This improves instruction cache bus efficiency by transferring entire lines instead of individual instruction words.

```
// Assume baddr holds the cache block address of missed instruction

MEMO:
  // initialize cacheTag as in Memo-IW handler {...}

  // Get the cache tag for the missed address
  compressedBlockAddrTag = baddr>>ICACHE_TAG_SHIFT;

  // Find replacement cache line
  igetctrl(baddr,0,ctrlbits);
  // if lru 0, use this line, else next way
  replAddr = baddr;
  replAddr |= (ctrlbits >> 2); // LRU bit is (ctrlbits >> 2)

  if (cacheTag == compressedBlockAddrTag)
    {
      // copy insn 1/8 into i-cache
      mem_load_4B(cacheDataAddr,0,iword);
      swicw(replAddr,0,iword);
      // copy 7 more insns into i-cache
      ...
      goto done;
    }
```

Figure 6.9: Memo-EW
The `igetctrl` instruction is used to select a cache line to replace. The `swicw` instruction stores native instruction into that selected line.

6.3.5 Results

This section evaluates the effect of memoization on software decompression. We simulate the four decompressors described in the previous section on the baseline model (4 KB instruction cache) with an additional 16 KB memoization table. The experiments in this section only show how memoization can improve performance. The combined effect on area and performance is studied in Section 6.4. For small programs, it is possible that using a large memoization table would increase the size of the compressed program beyond the size of native code. This does not happen in the benchmarks here, but the *mpeg2enc* and *pegwit* benchmarks under dictionary compression are nearly the same size as native code when the memoization table size is considered. This would be unacceptable in an embedded system running a single small compressed program because there would be no size or performance benefit over native code. However, if the system executes mul-

```
UPDATE:

  // Write replaced I-cache line to SW cache

  //Find address of line to be replaced in I-cache
  // Get tag of replaced line to find out where in SW cache to put it.
  igettag(replAddr,0,currentTag);
  // currentAddr is the address of this line
  currentAddr = baddr;
  // Strip top bits to get byte index into cache
  currentAddr &= ((1<<ICACHE_TAG_SHIFT) - 1);
  // Add tag bits on top
  currentAddr |= (currentTag << ICACHE_TAG_SHIFT);

  // Store tag in SW cache for replaced line.
  // find address to store TAG. Just like for faulting address.
  replCacheTagOffset = (currentAddr << 18) >> 21; //strip top bits. SW Cache
parameter.
  // write tag into SW cache
  mem_store_4B_RR(cacheTagBase,replCacheTagOffset,currentTag);

  // Find address in SW cache to store instructions.
  replCacheDataAddr = (currentAddr << 18) >> 18;
  replCacheDataAddr += cacheDataBase;

  // Write instructions in replaced line to SW cache.
  // Move first instruction
  lwicw(replAddr,0,iword);
  mem_store_4B(replCacheDataAddr,0,iword);

  // move 7 more instructions {...}

DECOMPRESS:
  // Normal decompression code using swic
  {...}

DONE:
  isync;
  iret;
```

Figure 6.9 continued: Memo-EW
The `igettag` instruction reads the value of the tag from the replaced line so it can be stored in the memoization table. The `lwicw` instruction loads words from the replacement line to be stored into the memoization table.

tiple small compressed programs, the addition of a memoization table could improve performance of the compressed code and still yield a size benefit over native code when the size and performance of all programs combined is considered.

Table 6.3 shows the performance results of the memoization optimization. These results are graphed in Figure 6.11. Using the memoization table has a large effect on the

```
MEMO:
  // Initialize as in Memo-EL {...}

  if (cacheTag == compressedBlockAddrTag)
    {
      // copy entire cache line into I-cache
      imapw(replAddr,0,cacheDataAddr);
      sync;
      goto done;
    }

UPDATE:

  // As in Memo-EL {...}

  // Write instructions in replaced line to SW cache.
  // Move first instruction
  iunmapw(replAddr,0,replCacheDataAddr);

DECOMPRESS:
  // Normal decompression code using swic
  {...}

DONE:
  isync;
  iret;
```

Figure 6.10: Memo-EL
The difference between Memo-EW and Memo-EL is that `swicw` and `lwicw` are replaced with `imapw` and `iunmapw` to copy entire cache lines at once. The `sync` instruction allows the `imapw` instruction to complete and load the cache line into the instruction cache before the decompressor ends. This is important so that the instruction fetch hardware accesses a valid instruction word.

running time of compressed programs. This is especially noticeable under CodePack were the minimum improvement is 8% (*mpeg2enc* with Memo-IL) and the maximum improvement is 90% (*pegwit* with Memo-IW). The IL and EL decompressors that can access entire cache lines with `imap`, `iunmap`, `iunmapw`, and `imapw` usually have a distinct performance improvement over the IW and EW decompressors that only use `swic`, `swicw`, and `lwicw`. Using cache line accesses for the dictionary decompressor reduced the decompression overhead by 38% for the inclusive policy and by 31% for the exclusive policy. For the CodePack decompressor, using cache line accesses reduced the decompression overhead by 13% for the inclusive policy and 18% for the exclusive policy. The simple inclusive policy often performs better than the exclusive policy. This is because the exclusive policy has extra overhead to manage the memoization table and often this overhead is greater than the benefit of utilizing the cache better. It is interesting that under

Benchmark	Dictionary	Memo-IW	Memo-IL	Memo-EW	Memo-EL
cc1	4.14	4.36	3.40	4.64	3.54
ghostscript	2.66	2.40	2.01	2.85	2.32
go	3.12	3.50	2.81	3.84	3.05
ijpeg	1.20	1.18	1.11	1.23	1.16
mpeg2enc	1.03	1.02	1.02	1.03	1.02
pegwit	4.43	3.57	1.60	4.42	3.22
perl	3.35	3.31	2.51	3.50	2.70
vortex	3.78	3.85	2.91	4.34	3.27

(a) Dictionary

Benchmark	CodePack	Memo-IW	Memo-IL	Memo-EW	Memo-EL
cc1	27.00	16.58	15.20	16.26	13.90
ghostscript	19.19	5.84	5.38	9.90	8.25
go	17.63	11.86	10.94	12.49	10.68
ijpeg	2.42	1.62	1.56	1.88	1.66
mpeg2enc	1.26	1.10	1.09	1.16	1.11
pegwit	13.67	2.21	1.74	6.81	5.62
perl	21.82	12.15	11.19	11.20	9.47
vortex	25.20	13.15	11.96	14.46	12.37

(b) CodePack

Table 6.3: Performance of memoization
Performance is given as slowdown relative to a native system. For each benchmark, the decompressor with the highest performance is shaded. a) Dictionary. b) CodePack.

CodePack there is not a single decompressor that is better for every benchmark. This strongly motivates the use of per-application decompressors.

Using the memoization table often results in worse performance for the dictionary decompression. This occurs because the dictionary decompressor already has very low overhead. The additional overhead to manage the memoization table and check it for hits often outweighs the performance advantage of using memoization.

6.3.6 Conclusion

Memoization is useful at improving performance of compressed programs. CodePack benefits more than dictionary compression from memoization because CodePack has much more overhead. The memoization routines that access complete cache lines at a time performed better than the ones that move a single instruction at a time. This effect is more pronounced in the dictionary decompression because the movement of instructions between memory and cache represent a larger portion of the decompression overhead than

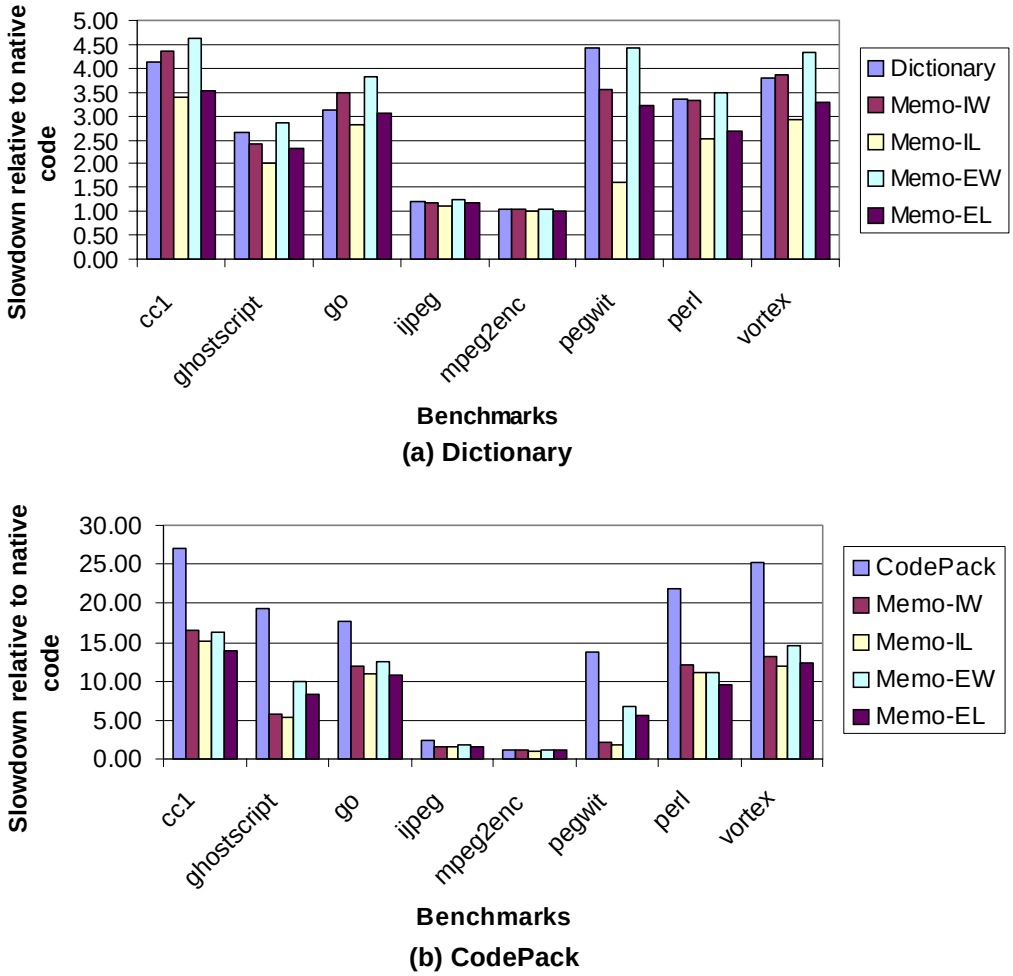


Figure 6.11: Memoization performance results
(a) Dictionary. (b) CodePack.

in the CodePack decompressor. The exclusive policy uses the memoization table more efficiently, but takes more cycles to execute. For the dictionary decompression, the exclusive policy overhead overwhelms the benefit of memoization. Often better performance is achieved by using the simple inclusive policy.

6.4 Memoization and selective compression

Both memoization and hybrid programs reduce decompression overhead. This section compares these two approaches and explores how to combine them. The goal of memoization and hybrid programs is similar. Both predict which instructions will be most frequently decompressed so that they may be stored in native form and the decompression

algorithm does not need to be invoked often. Hybrid programs are a static optimization because the section of code that is decompressed (the native code section of the program) is chosen at compile-time and fixed as the program executes. If the execution behavior of the program does not match the profile used to build the hybrid program, then the native code section may be useless. On the other hand, memoization is a dynamic optimization that can choose which translations to hold in the decompression buffer. As the program executes, memoization can adapt to the execution behavior caused by a particular input data set. However, memoization has a start-up cost because the table must be initialized with recent values. Hybrid programs do not have this cost and can immediately supply native instructions the first time an instruction in the native code section is referenced.

Although the optimizations are similar, they can both be applied to the same program. It is interesting to consider for a fixed die area what mixture of dynamic and static decompression provides the best performance. It is straightforward to construct programs of the same size which use hybrid optimization, memoization, or both. Figure 6.12-A and Figure 6.12-B represent native and compressed applications, respectively. Figure 6.12-C shows the compressed program with memoization. To obtain a hybrid program of the same size, start with the totally compressed program and move procedures into the native code section until the program expands to the size of the program with memoization. This hybrid program is in Figure 6.12-D. A program that uses both optimizations and is of the same size can be made by using a smaller memoization table and then moving compressed code into the native code section until the program is the size of the program only using the larger memoization table. A hybrid program that uses memoization is shown in Figure 6.12-E.

6.4.1 Setup

The experiments in this section simulate a system-on-a-chip (SOC) that uses embedded DRAM (eDRAM) to hold the compressed program and data. We assume that the complete compressed program is loaded from a network or a very slow long-term storage device into the eDRAM prior to execution. The SOC also includes a SRAM instruc-

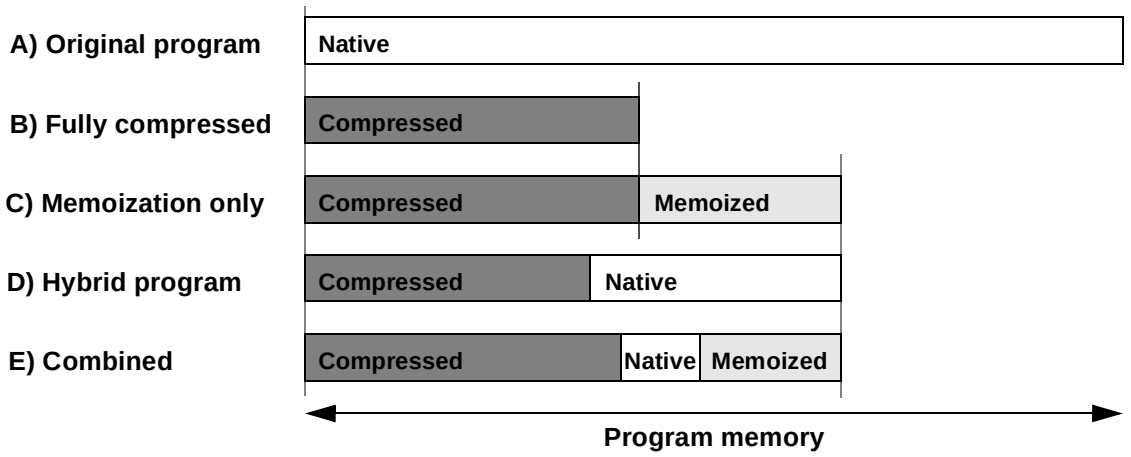


Figure 6.12: Memory usage
An original program and its compressed version are showen in A and B, respectively. C, D, and E show various methods of improving performance at the expense of compressed program size. C) The compressed program with memoization. D) The compressed program is a hybrid program with some native code procedures. Note that the compressed region is smaller because some functions are in the native code region. E) Hybrid programs can also use memoization. In the simulations, programs simulated using techniques C, D, and E all have the same size so that a performance comparison can be made.

tion cache to hold decompressed instructions. The eDRAM also holds decompressed instructions when hybrid programs or memoization is used.

By altering the allocation of die area between the SRAM and eDRAM, different area/performance trade-offs can be made. Performance can be increased by holding more decompressed code in either SRAM or eDRAM. While eDRAM is considerably more dense, it has a slower access time than SRAM. The experiments in this section assume specific densities for the SRAM and eDRAM. The densities are used to calculate a total overall area for the compressed program. The area calculation includes the entire SRAM instruction cache, the portion of the eDRAM that holds the compressed program including any area reserved for memoization or native code. Similar studies by other researchers have failed to take the area of the instruction cache into account when measuring compression ratio. Since the instruction cache has a strong effect on the die area and program performance, it is necessary to include it when making area/performance trade-offs.

The simulations assume that eDRAM has ten times the density of SRAM but that it takes 10 cycles to access. The bus between the eDRAM and SRAM cache is 256 bits. This allows an instruction cache miss to be filled in one access. These parameters are in

Memory	Density	Access time
SRAM I-cache	10,000 bytes/mm^2	1 cycle
eDRAM memory	100,000 bytes/mm^2	10 cycles

Table 6.4: Memory on System-on-Chip

the range of current eDRAM technology [IBM00]. The exact simulation parameters are in Table 6.4.

The hybrid programs used in the following experiments are created using a selective compression algorithm that uses cache-miss frequency as a metric. This algorithm was introduced in Section 6.2.2. The programs that most frequently miss the cache are selected to be in the native code section of the final program. This causes the decompressor to be invoked during fewer cache misses and therefore improves performance.

All memoized programs for the experiments use the Memo-IL policy because it was the only policy that consistently performed well across both dictionary and CodePack compressed programs.

6.4.2 Results

This section contains two experiments. The first experiment shows the area/performance trade-off for allocating more area to the instruction cache. The second experiment shows the area/performance trade-off for allocating more area to the eDRAM for hybrid programs and memoization. A comparison of these two techniques follows.

Chapter 5 used larger caches to show that decompression overhead can be made manageable. In this section, the simulation baseline uses a 4 KB instruction cache that is typical of embedded systems. All experiments in this section try to optimize this baseline system for area and performance. All results for area and slowdown are given as a ratio compared to this system. Area results are given as a compression ratio compared to the area used in the baseline system. Remember that the compression ratio includes the area of SRAM and eDRAM used by the instruction cache, compressed program, and decompression buffer.

The baseline configuration with a 4 KB instruction cache is an arbitrary point selected to illustrate the possible area/performance trade-offs in compressed code systems.

The experiments presented here could use different cache sizes in the baseline configuration. The ratio of the results compared to the baseline results would change, but the same performance and area trends would be evident.

First experiment: increasing budget for instruction cache

Chapter 5 showed that increasing the cache size improved performance of compressed programs. Here the experiment is repeated, but results are given for both performance and area. The results in Table 6.5 through Table 6.12 show how area and performance change as larger instruction caches are used. The area and slowdown given are listed as ratios based on the performance and area of a native program running on a 4 KB cache.

Increasing the cache size slightly can dramatically affect performance. For example, in Table 6.6 CodePack compressed ghostscript has a slowdown of 19.19 with a 4 KB cache, but only a slowdown of 6.42 with a 8 KB cache. The cost of the larger cache is an increase in area from a 64% compression ratio to a 68% compression ratio.

However, using a larger cache can also have a detrimental effect on the compression ratio. In Table 6.8, for example, the *ijpeg* benchmark compressed with CodePack has a compression ratio is 67% when using a 4 KB instruction cache. When the instruction cache is increased to 16 KB, the compression ratio is 121%. This means that the cache size was increased so much that the area savings from compression was lost. The area is 21% larger than a native code application with a 4 KB instruction cache. In addition, the program executes 38% slower than native code. Therefore adding more cache to the compressed program did not optimize it relative to the baseline native code system with the 4 KB cache. It is important to understand that the reason the compressed code is larger than the native code is because the area comparison is against a baseline system with a smaller cache. If the baseline system had a 16 KB cache, then the compression ratio for ijpeg would be less than 100% for the 16 KB cache configuration (less dies area would be used). Note that some small programs (mpeg2enc and pegwit) cannot increase the cache size beyond the original 4 KB without using more die area than the baseline system.

Second experiment: increasing budget for eDRAM

The second experiment increases the die area budget for eDRAM. The additional area can be used to hold 1) native code for a hybrid program, 2) native code for memoization, or 3) native code for both optimizations. In all cases, this extra memory is called the *decompression buffer* because it holds decompressed instructions. Table 6.13 through Table 6.20 show the results of using combinations of hybrid programs and memoization. The tables show the effect on area and performance for adding different size decompression buffers to the baseline system with a 4 KB instruction cache. The various decompression buffer budgets are sized to hold memoization tables of 9 KB through 288 KB. The tables are divided into two parts. The first part (top) shows the slowdown of the program using a decompression buffer with different allocations for the hybrid program and memoization optimizations. The second part (bottom) shows the die area (as a compression ratio) of the program using each size of decompression buffer. All performance results in the same column represent a program utilizing the same amount of die area. The performance result at the top of the column is for a hybrid program with no memoization. The performance result at the bottom of the column is a totally compressed program with some memoization. Results in-between the top and bottom are for hybrid programs that also use memoization. The column is sorted by the size of the memoization table used. Any area in the decompression buffer not used by the memoization table is used to hold the native code section of the hybrid programs.

The previous experiment showed that increasing the instruction cache capacity slightly could often negate the effect of compression. On the contrary, decompression buffer capacities can become quite large before they have the same effect. This is due to the density differences between the memories. For example, the CodePack *ijpeg* can use up to a 72 KB decompression buffer and still have a 97% compression ratio. As we previously observed, instruction cache capacity could only be increased to 8 KB without expanding the die area beyond the baseline system. The 72 KB decompression buffer with a 4 KB cache potentially holds over eight times as many decompressed instructions as the 8 KB cache alone.

Since the eDRAM holds more native code per unit area than the SRAM, performance often increases dramatically when addition area budget is spent on eDRAM. This

occurs even though the eDRAM has a higher access latency. The reason for the performance increase is that eliminating the large decompression penalty (1000s of cycles) is better than eliminating the relatively small increase in access latency (10 cycles).

All results in Table 6.13 through Table 6.20 are graphed in Figure 6.13 through Figure 6.20. Each point on the graphs represents a different allocation of die area for a program. The circled points are programs with the same area budget, but different amounts of memoization in the decompression buffer. For comparison, the results from using larger instruction caches are also plotted. It is immediately obvious that spending area on the decompression buffer provides a better area and performance benefit than increasing the cache capacity. Only for ghostscript and vortex do larger caches come close to competing with an increased decompression buffer size.

There are only a few benchmarks for which using only memoization consistently performs better than using only a hybrid program. These benchmarks are *cc1* under CodePack (18 KB, 36 KB, and 72 KB buffers) and *ijpeg* under CodePack (9 KB buffer). This is likely due to the large initial latency to generate the native code when using memoization. Although hybrid programs give better performance than memoized programs (for the same area budget), more effort must be taken to create them. In this case, a cache simulation profile of each application was collected.

It is interesting that for a particular area budget, the best performing solutions often use both optimizations. This is especially true for the larger applications. Intuitively what happens is that for a particular budget, the working set of a program does not fit into the available native code region of the hybrid program. By allowing some native code to be dynamically selected by memoization, the decompression buffer can adjust itself as the program executes new code regions.

Some compressed programs (for example, ghostscript in Table 6.14) have slowdowns that are less than 1. This means that the programs are running faster than native code on the baseline system. The reason that this happens is because the compressed program has a different procedure placement due to being a hybrid program. A beneficial placement can make a program fit better in the cache and experience fewer cache misses than the original program. This is discussed more thoroughly in Section 6.2.4.

6.5 Conclusion

Both hybrid programs and memoization are useful at improving compressed program performance with a modest loss in compression.

Section 6.2 showed that hybrid programs are very effective at improving the area-performance trade-off in compressed programs. Using profiles of instruction cache misses to guide the selection of native and compressed code in the program was beneficial for loop-oriented programs. Such programs that used these profiles reduced their decompression overhead by 50% over programs that used profiles of dynamic instruction counts.

Section 6.3 demonstrated that modifying the instruction set to move complete cache lines between the memory and instruction cache improved the performance benefit of memoization significantly. This is more noticeable in dictionary compression than CodePack compression because the dictionary decompressor spends a relatively larger amount of time transferring instructions between memory and cache due to the lower latency of the dictionary decompression algorithm. The experiments also showed that the fast inclusive policy often performed better than the slower exclusive policy which attempted to utilize the memoization table better.

Section 6.4 showed that using more eDRAM for memoization or hybrid programs provides better performance and with less area increase than doubling or quadrupling the instruction cache capacity. This is because the greater density of eDRAM stores more native code or decompressed code per unit area. This reduces the number of times the complete decompression algorithm is executed and improves performance. When choosing between hybrid programs and memoization, a hybrid program is usually a better use of area than a fully compressed program with memoization. This is because programs spend most of their time in a few procedures that can be put in a native code section and because hybrid programs do not incur the initial decompression latency of memoization to produce the native code. However the best use of the decompression buffer often combines both optimizations. The multimedia benchmarks (*ghostscript*, *ijpeg*, *mpeg2enc*, and *pegwit*) have the highest performance of all benchmarks under software decompression because they experience fewer cache misses. When they are hybrid programs or use memoization, they execute with nearly the same performance as native programs.

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	4.14	67%	27.00	62%
8 KB	3.19	71%	20.00	66%
16 KB	2.19	78%	12.57	74%
32 KB	1.33	94%	5.57	89%
64 KB	0.96	125%	2.53	120%

Table 6.5: Area and performance as a function of I-cache size (cc1)
Slowdown is the number of times slower a compressed program executes when compared to program running on a 4 KB instruction cache. Area is a compression ratio compared to the area used by a native program running on a 4 KB cache.

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	2.66	71%	19.19	64%
8 KB	1.60	74%	6.42	68%
16 KB	1.20	82%	2.65	76%
32 KB	1.04	97%	1.69	91%
64 KB	0.85	128%	1.03	121%

Table 6.6: Area and performance as a function of I-cache size (ghostscript)

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	3.12	73%	17.63	64%
8 KB	2.50	86%	12.27	76%
16 KB	2.04	110%	8.91	101%
32 KB	1.68	160%	6.35	150%
64 KB	1.30	258%	3.97	248%

Table 6.7: Area and performance as a function of I-cache size (go)

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	1.20	81%	2.42	67%
8 KB	1.10	99%	1.75	85%
16 KB	1.04	135%	1.38	121%
32 KB	1.02	207%	1.26	193%
64 KB	1.00	351%	1.11	336%

Table 6.8: Area and performance as a function of I-cache size (ijpeg)

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	1.03	87%	1.26	73%
8 KB	1.01	114%	1.13	100%
16 KB	1.00	168%	1.04	154%
32 KB	1.00	275%	1.02	261%
64 KB	1.00	488%	1.01	474%

Table 6.9: Area and performance as a function of I-cache size (mpeg2enc)

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	4.43	86%	13.77	74%
8 KB	2.23	119%	1.08	107%
16 KB	0.81	185%	0.88	173%
32 KB	0.81	316%	0.81	304%
64 KB	0.80	577%	0.81	566%

Table 6.10: Area and performance as a function of I-cache size (pegwit)

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	3.35	77%	21.82	66%
8 KB	2.22	91%	13.81	80%
16 KB	1.50	120%	8.15	108%
32 KB	1.12	175%	4.52	164%
64 KB	0.82	287%	1.06	276%

Table 6.11: Area and performance as a function of I-cache size (perl)

I-cache size	Dictionary		CodePack	
	Slowdown	Area	Slowdown	Area
4 KB	3.78	69%	25.20	59%
8 KB	2.61	77%	15.43	67%
16 KB	1.73	93%	8.28	83%
32 KB	1.21	125%	4.18	116%
64 KB	0.91	190%	1.88	180%

Table 6.12: Area and performance as a function of I-cache size (vortex)

	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	3.25	2.96	2.43	1.67	1.25	1.00
4.5 KB	3.90	2.94	2.53	1.72	1.22	1.00
9 KB	4.01	2.97	2.37	1.67	1.19	1.00
18 KB		3.40	2.36	1.67	1.19	1.00
36 KB			2.82	1.73	1.23	1.00
72 KB				2.33	1.29	1.01
144 KB					1.22	1.11
288 KB						2.04
Area (compression ratio)	68%	68%	70%	73%	80%	93%

A) Dictionary

	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	18.66	16.45	12.65	6.76	3.01	1.06
4.5 KB	20.66	14.97	11.64	5.95	2.46	1.02
9 KB	21.29	13.16	10.47	5.09	2.20	1.02
18 KB		15.20	8.42	4.59	2.03	1.02
36 KB			10.08	4.13	2.12	1.07
72 KB				5.69	2.12	1.07
144 KB					4.82	1.27
288 KB						3.28
Area (compression ratio)	63%	64%	65%	69%	75%	88%

B) CodePack

Table 6.13: Decompression buffer performance and area (cc1)

The tables are divided into two parts. The top part of the table holds the performance results. The performance is the slowdown experienced compared to a native code system with a 4KB instruction cache. Each column represents a different die area budget. The top of the column tells how large a decompression buffer is reserved in the eDRAM. The Memoization Table Size column tells how much of the decompression buffer is reserved to memoization. The remainder of the buffer is used to hold native code for hybrid programs. The top of each column is a hybrid program without memoization. The bottom of each column is a totally compressed program that uses only memoization. All performance results in a single column use the same amount of die area. The shaded values in each column show the configurations with the best performance for a particular area budget

The bottom part of the table holds area results. The area is reported as a compression ratio compared to a native code system using a 4 KB instruction cache. The area result includes the size of the instruction cache and amount of eDRAM to hold the compressed program and decompression buffer.

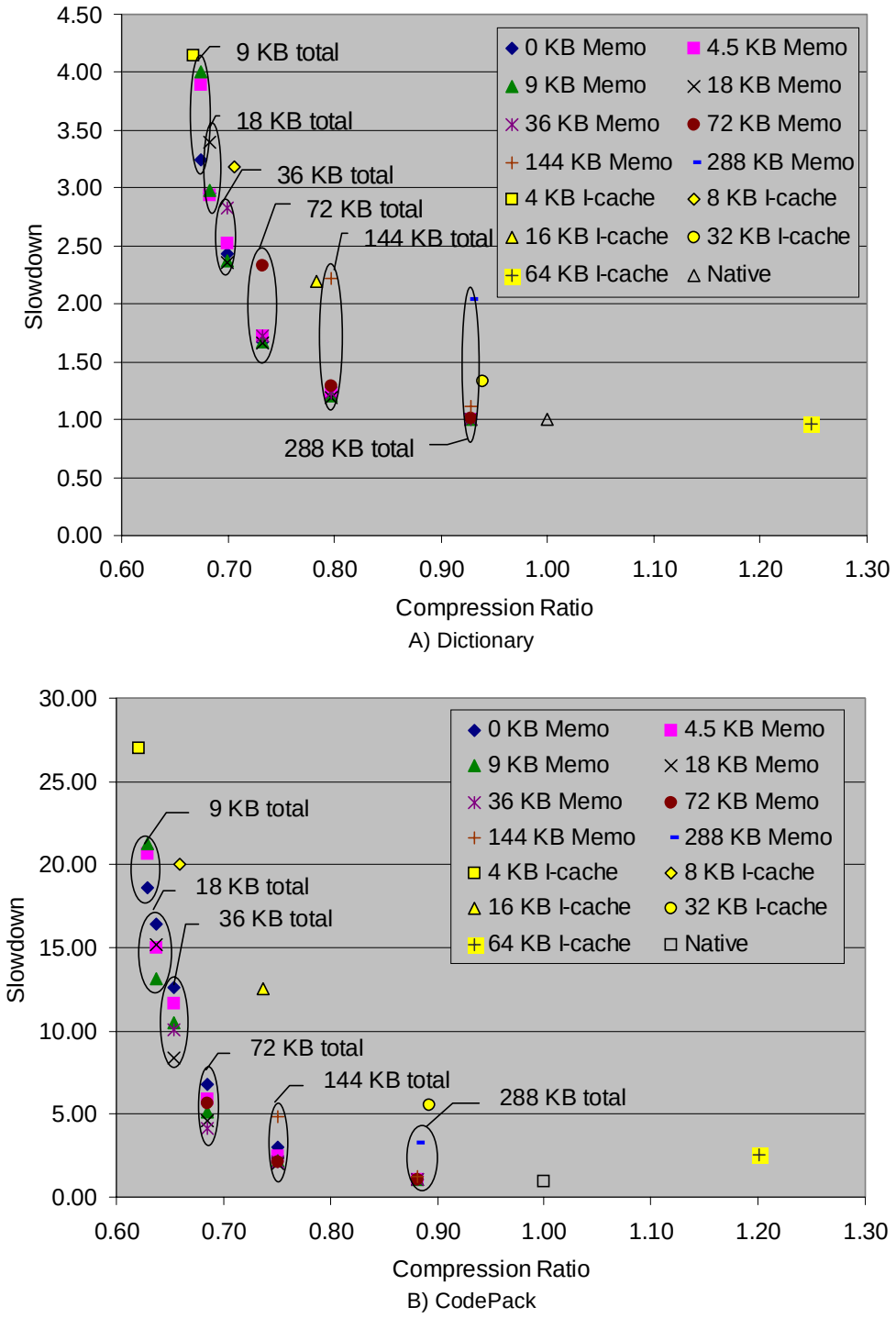


Figure 6.13: Performance and area of decompression buffer (cc1)
The data for these graphs is in Table 6.13. *Native* is the baseline for this graph, a native program executing on a 4 KB I-cache. The *Memo* data points represent hybrid compressed programs (possibly using memoization) executing on 4 KB I-caches. The circled data points use the same die area budget. The number label tells how large the decompression buffer for each budget is. On this graph, the decompression buffers range from 9 KB to 288 KB in size. The *I-cache* data points represent compressed programs running on I-caches with different capacities (the number of cache lines is varied).

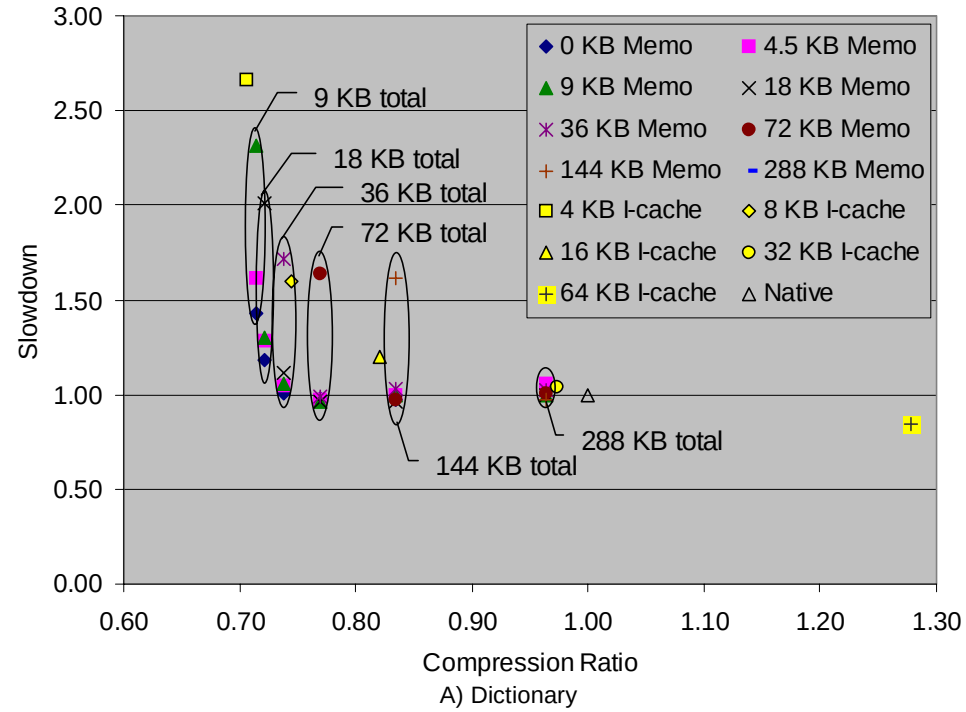
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.43	1.18	1.00	0.98	0.97	1.03
4.5 KB	1.61	1.29	1.05	0.97	1.00	1.06
9 KB	2.31	1.30	1.06	0.96	1.00	1.00
18 KB		2.01	1.11	0.97	0.96	0.99
36 KB			1.71	0.99	1.03	1.02
72 KB				1.64	0.97	1.01
144 KB					1.61	0.97
288 KB						1.60
Area (compression ratio)	71%	72%	74%	77%	83%	96%

A) Dictionary

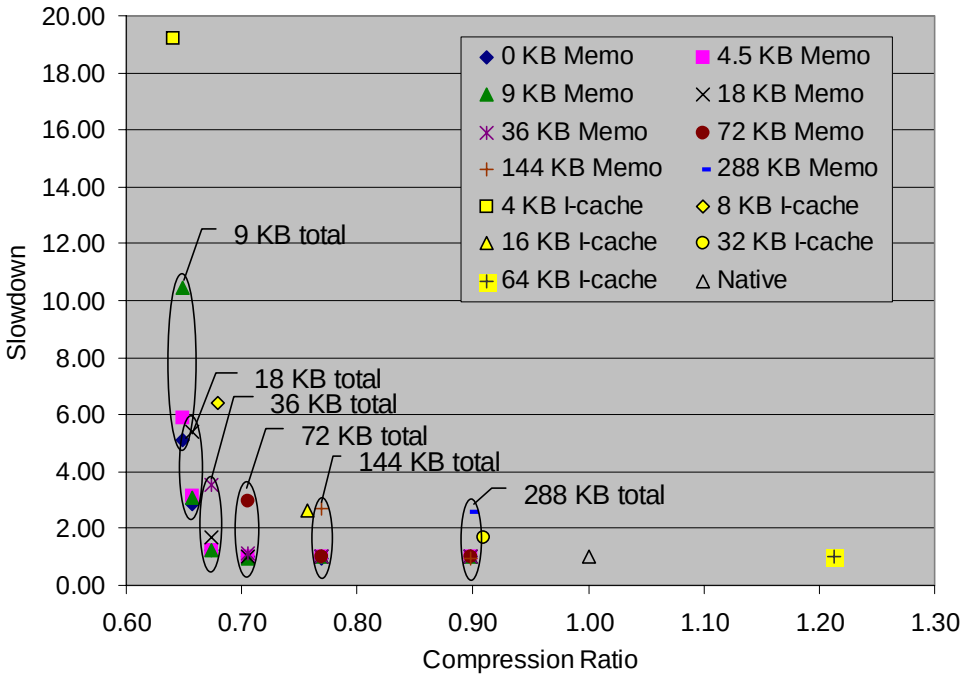
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	5.09	2.85	1.26	1.02	0.98	1.02
4.5 KB	5.91	3.15	1.21	1.00	0.99	1.01
9 KB	10.45	3.10	1.25	0.98	0.99	1.02
18 KB		5.38	1.67	1.02	1.00	1.02
36 KB			3.53	1.10	1.03	1.03
72 KB				2.96	0.99	1.00
144 KB					2.68	0.97
288 KB						2.57
Area (compression ratio)	65%	66%	67%	71%	77%	90%

B) CodePack

Table 6.14: Decompression buffer performance and area (ghostscript)



A) Dictionary



B) CodePack

Figure 6.14: Performance and area of decompression buffer (ghostscript)

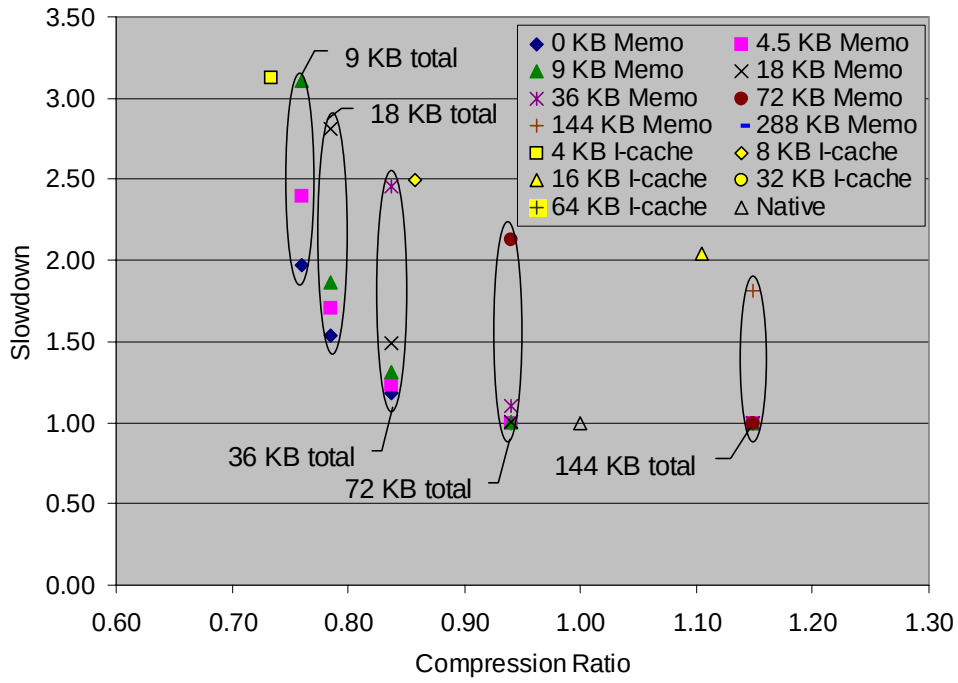
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.97	1.53	1.19	0.99	1.00	1.00
4.5 KB	2.40	1.71	1.23	1.00	1.00	1.00
9 KB	3.11	1.87	1.31	1.00	1.00	1.00
18 KB		2.81	1.49	1.01	1.00	1.00
36 KB			2.45	1.10	1.00	1.00
72 KB				2.13	0.99	1.00
144 KB					1.82	1.00
288 KB						1.68
Area (compression ratio)	76%	79%	84%	94%	115%	156%

A) Dictionary

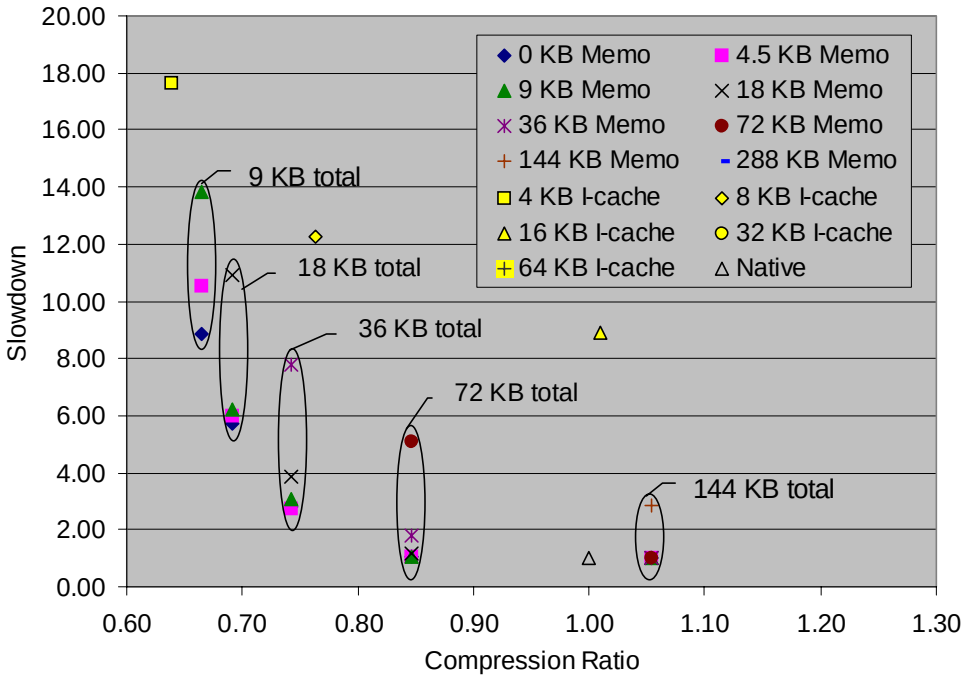
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	8.83	5.69	2.78	1.08	1.00	1.00
4.5 KB	10.51	6.02	2.74	1.09	1.00	1.00
9 KB	13.83	6.22	3.11	1.09	1.00	1.00
18 KB		10.94	3.87	1.20	1.00	1.00
36 KB			7.77	1.78	0.99	1.00
72 KB				5.07	1.01	1.00
144 KB					2.84	1.00
288 KB						1.93
Area (compression ratio)	67%	69%	74%	85%	105%	147%

B) CodePack

Table 6.15: Decompression buffer performance and area (go)



A) Dictionary



B) CodePack

Figure 6.15: Performance and area of decompression buffer (go)

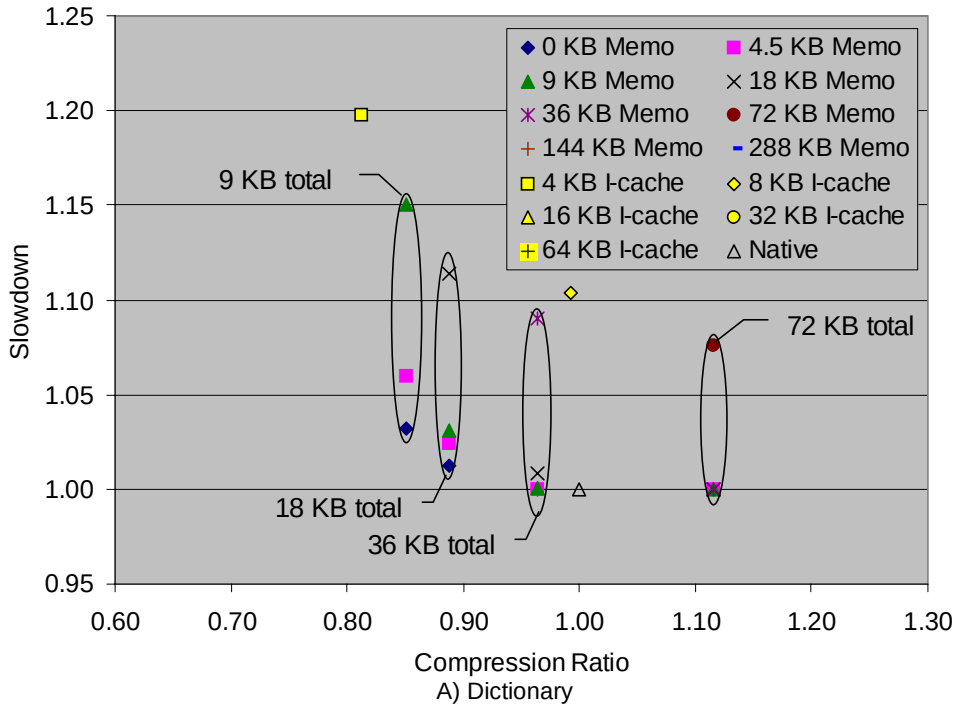
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.03	1.01	1.00	1.00	1.00	1.00
4.5 KB	1.06	1.02	1.00	1.00	1.00	1.00
9 KB	1.15	1.03	1.00	1.00	1.00	1.00
18 KB		1.11	1.01	1.00	1.00	1.00
36 KB			1.09	1.00	1.00	1.00
72 KB				1.08	1.00	1.00
144 KB					1.06	1.00
288 KB						1.06
Area (compression ratio)	85%	89%	96%	112%	142%	203%

A) Dictionary

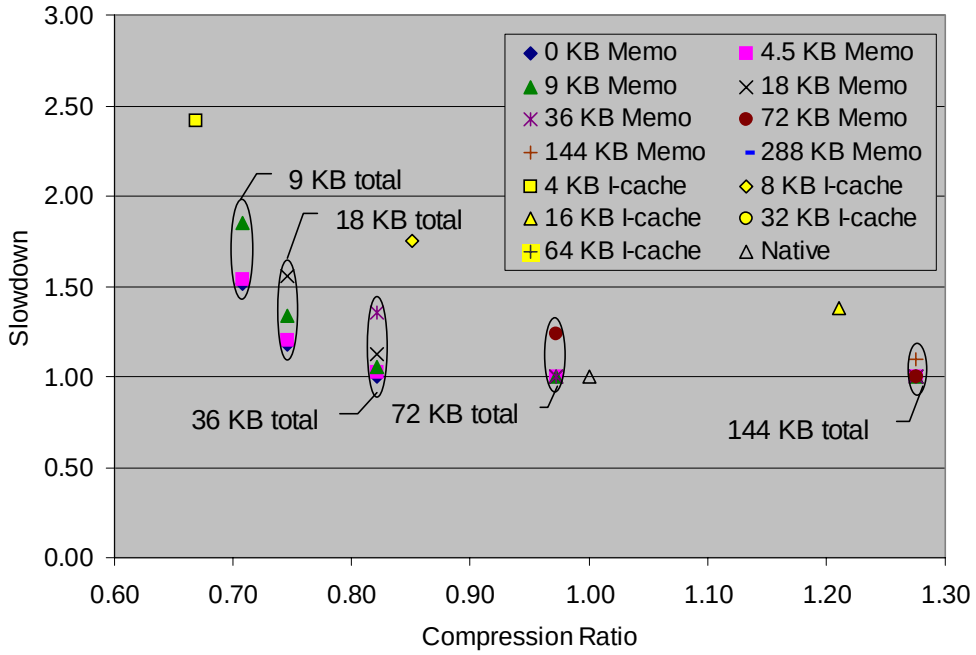
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.52	1.18	1.00	1.00	1.00	1.00
4.5 KB	1.54	1.20	1.02	1.00	1.00	1.00
9 KB	1.85	1.34	1.05	1.00	1.00	1.00
18 KB		1.56	1.12	1.00	1.00	1.00
36 KB			1.35	1.00	1.00	1.00
72 KB				1.24	1.00	1.00
144 KB					1.10	1.00
288 KB						1.09
Area (compression ratio)	71%	75%	82%	97%	128%	188%

B) CodePack

Table 6.16: Decompression buffer performance and area (ijpeg)



A) Dictionary



B) CodePack

Figure 6.16: Performance and area of decompression buffer (ijpeg)

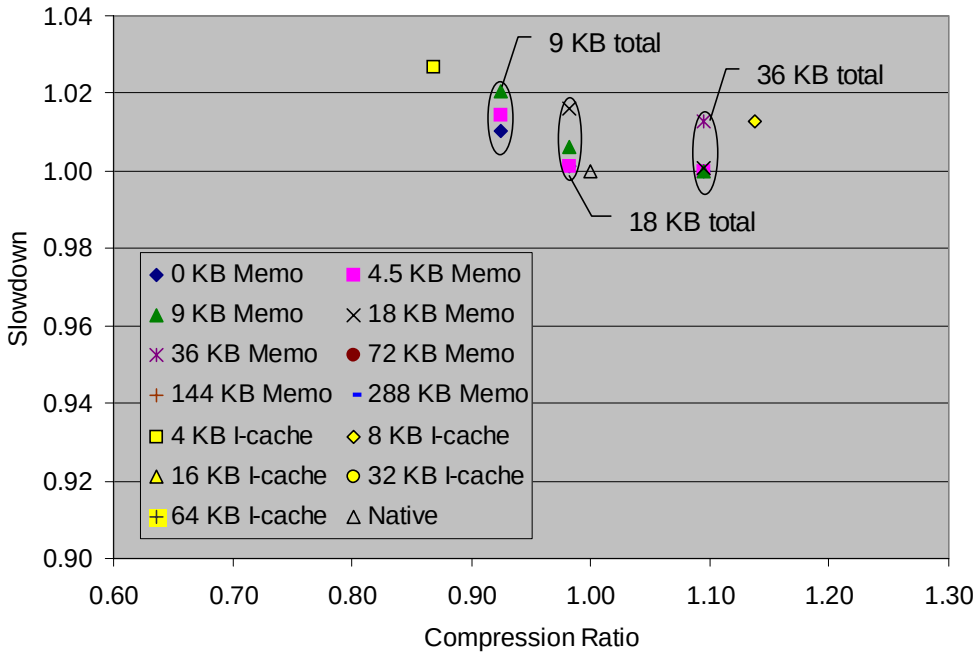
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.01	1.00	1.00	1.00	1.00	1.00
4.5 KB	1.01	1.00	1.00	1.00	1.00	1.00
9 KB	1.02	1.01	1.00	1.00	1.00	1.00
18 KB		1.02	1.00	1.00	1.00	1.00
36 KB			1.01	1.00	1.00	1.00
72 KB				1.01	1.00	1.00
144 KB					1.01	1.00
288 KB						1.01
Area (compression ratio)	93%	98%	109%	132%	177%	268%

A) Dictionary

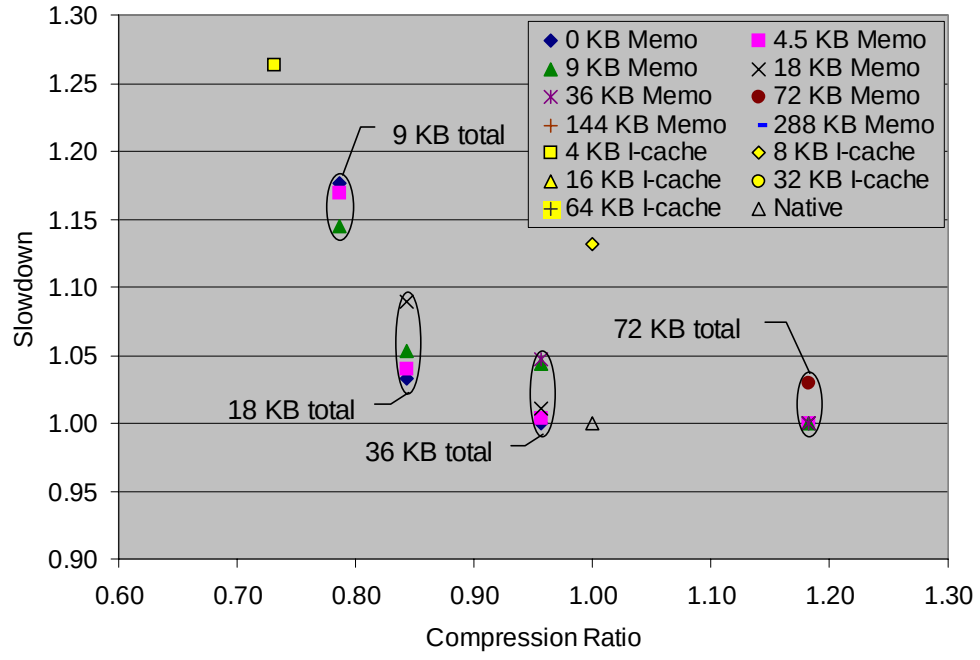
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.18	1.03	1.00	1.00	1.00	1.00
4.5 KB	1.17	1.04	1.00	1.00	1.00	1.00
9 KB	1.14	1.05	1.04	1.00	1.00	1.00
18 KB		1.09	1.01	1.00	1.00	1.00
36 KB			1.05	1.00	1.00	1.00
72 KB				1.03	1.00	1.00
144 KB					1.02	1.00
288 KB						1.02
Area (compression ratio)	79%	84%	96%	118%	163%	254%

B) CodePack

Table 6.17: Decompression buffer performance and area (mpeg2enc)



A) Dictionary



B) CodePack

Figure 6.17: Performance and area of decompression buffer (mpeg2enc)

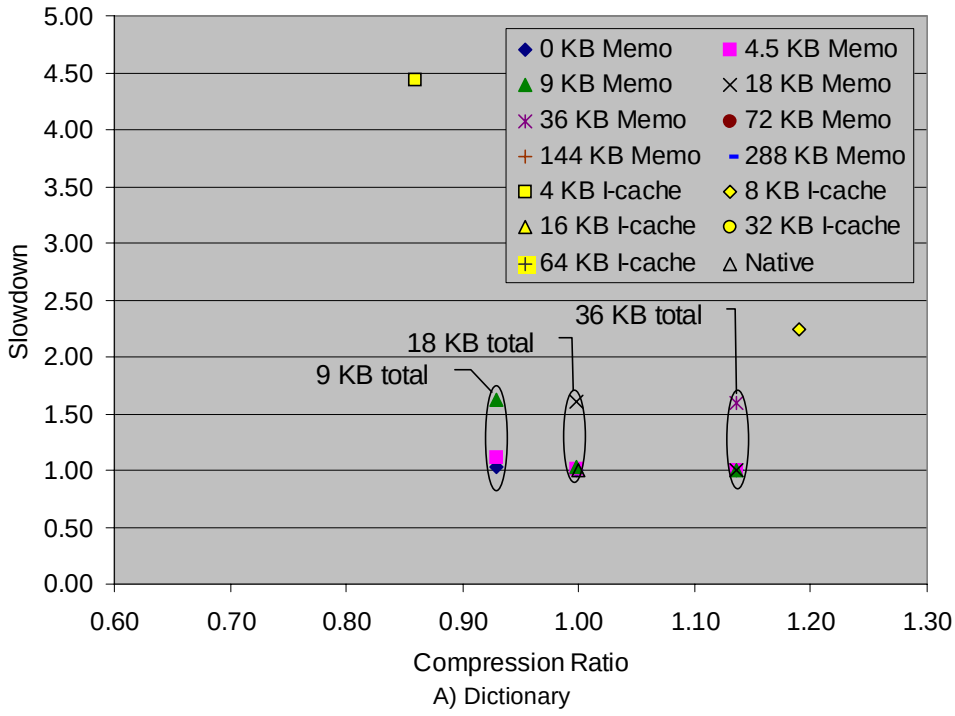
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.03	1.00	1.00	1.00	1.00	1.00
4.5 KB	1.11	1.01	1.00	1.00	1.00	1.00
9 KB	1.63	1.03	1.00	1.00	1.00	1.00
18 KB		1.60	1.00	1.00	1.00	1.00
36 KB			1.59	1.00	1.00	1.00
72 KB				1.59	1.00	1.00
144 KB					1.59	1.00
288 KB						1.59
Area (compression ratio)	93%	100%	114%	141%	197%	307%

A) Dictionary

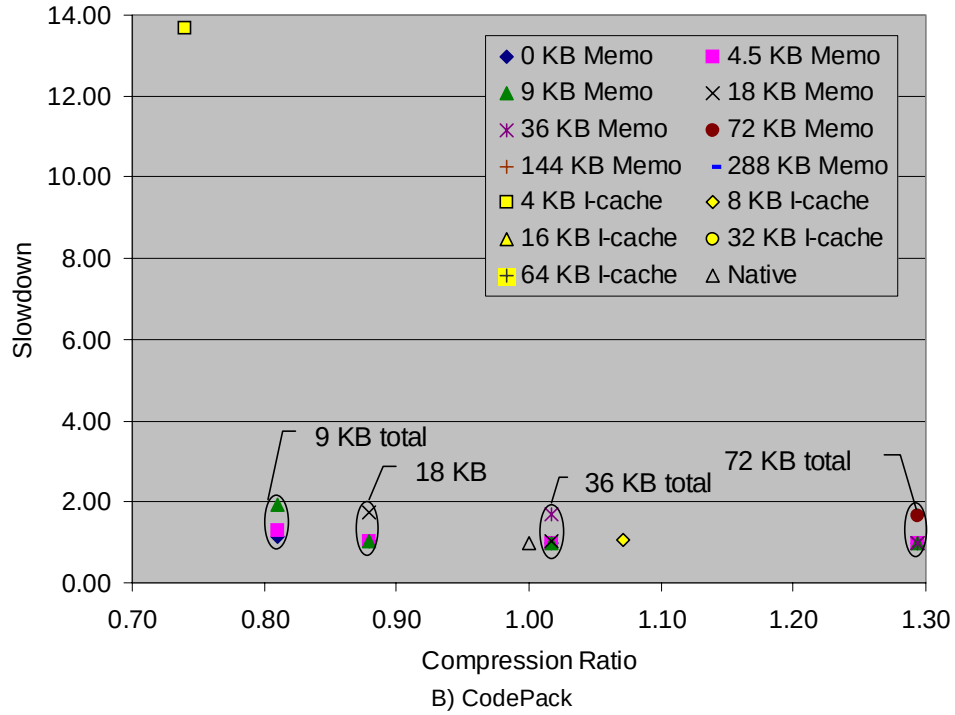
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	1.16	1.02	1.00	1.00	1.00	1.00
4.5 KB	1.30	1.03	1.00	1.00	1.00	1.00
9 KB	1.95	1.04	1.00	1.00	1.00	1.00
18 KB		1.74	1.01	1.00	1.00	1.00
36 KB			1.69	1.00	1.00	1.00
72 KB				1.66	1.00	1.00
144 KB					1.66	1.00
288 KB						1.66
Area (compression ratio)	81%	88%	102%	129%	185%	295%

B) CodePack

Table 6.18: Decompression buffer performance and area (pegwit)



A) Dictionary



B) CodePack

Figure 6.18: Performance and area of decompression buffer (pegwit)

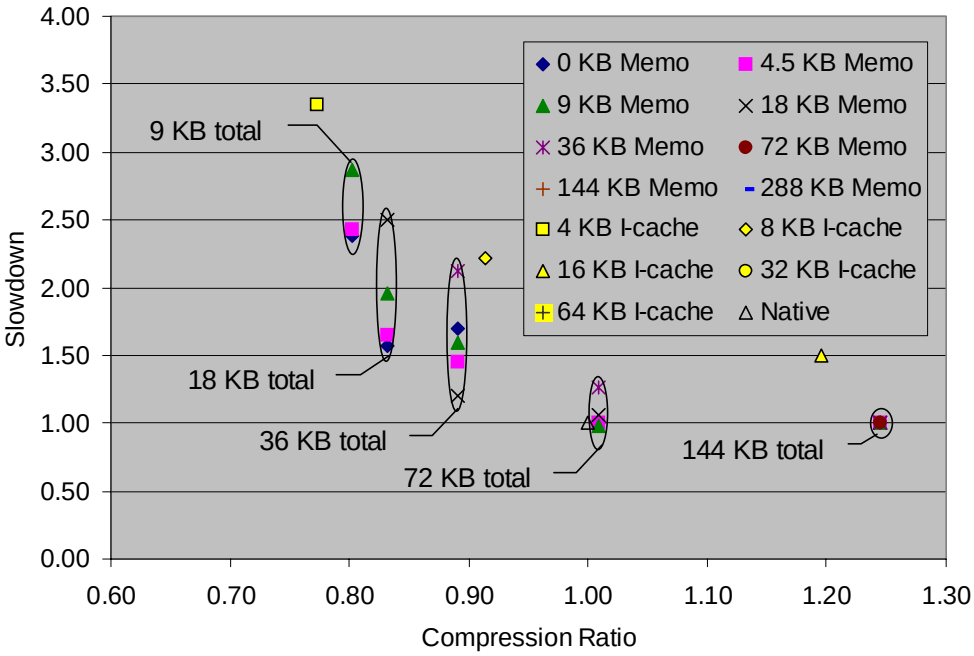
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	2.39	1.57	1.70	1.00	1.00	1.00
4.5 KB	2.43	1.66	1.45	1.00	1.00	1.00
9 KB	2.86	1.96	1.59	0.98	1.00	1.00
18 KB		2.51	1.20	1.06	1.00	1.00
36 KB			2.13	1.27	1.00	1.00
72 KB				2.02	1.00	1.00
144 KB					1.77	1.00
288 KB						1.73
Area (compression ratio)	80%	83%	89%	101%	125%	172%

A) Dictionary

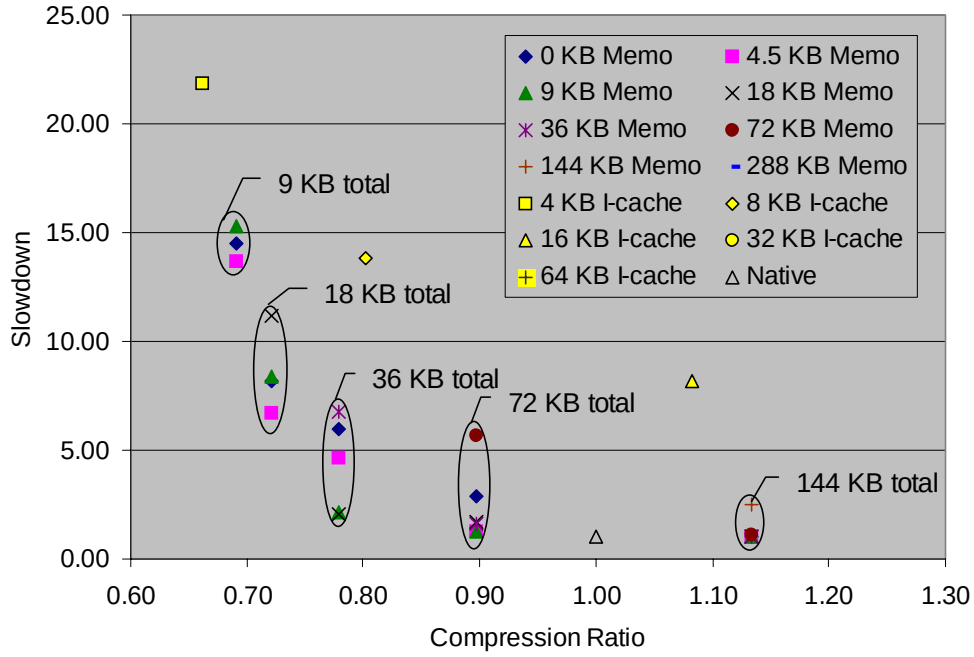
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	14.51	8.18	5.97	2.85	1.00	1.00
4.5 KB	13.68	6.67	4.60	1.22	1.00	1.00
9 KB	15.33	8.35	2.12	1.28	1.00	1.00
18 KB		11.19	2.06	1.70	1.00	1.00
36 KB			6.79	1.61	1.00	1.00
72 KB				5.67	1.09	1.00
144 KB					2.53	1.00
288 KB						2.22
Area (compression ratio)	69%	72%	78%	90%	113%	161%

B) CodePack

Table 6.19: Decompression buffer performance and area (perl)



A) Dictionary



B) CodePack

Figure 6.19: Performance and area of decompression buffer (perl)

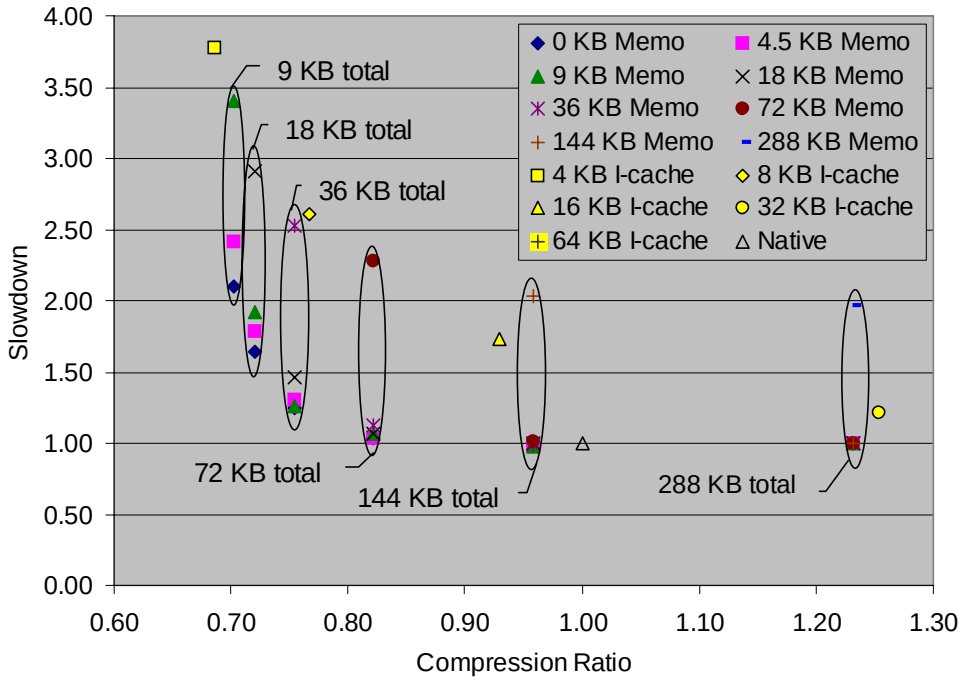
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	2.10	1.64	1.25	1.03	1.00	1.00
4.5 KB	2.42	1.79	1.30	1.03	0.99	1.00
9 KB	3.41	1.92	1.25	1.07	0.98	1.00
18 KB		2.91	1.46	1.07	1.00	1.00
36 KB			2.53	1.13	0.99	1.00
72 KB				2.29	1.01	1.00
144 KB					2.04	1.00
288 KB						1.97
Area (compression ratio)	70%	72%	75%	82%	96%	123%

A) Dictionary

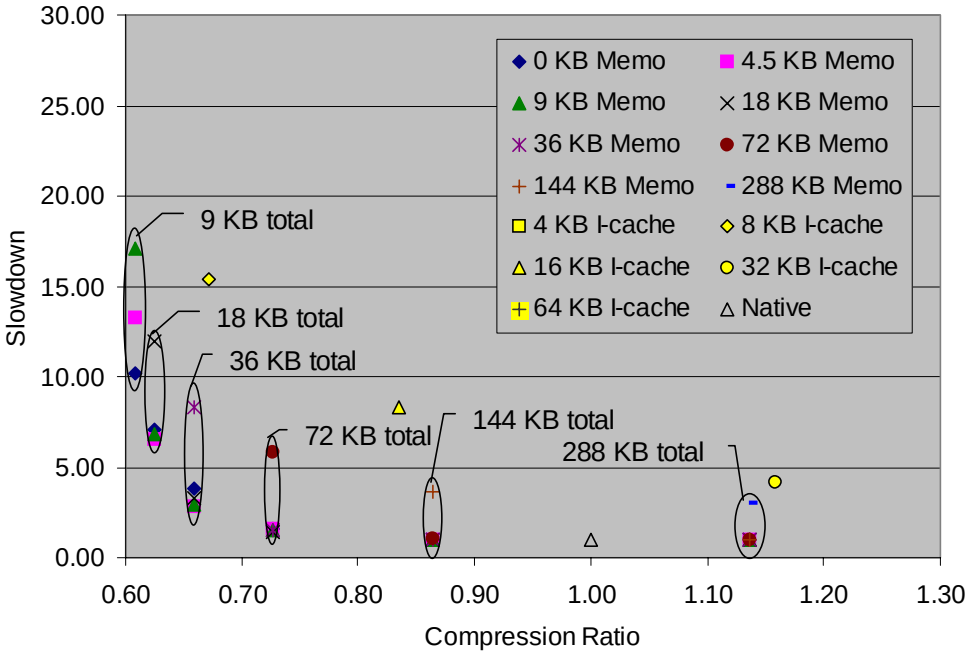
	Size of decompression buffer					
	9 KB	18 KB	36 KB	72 KB	144 KB	288 KB
Memoization Table Size	**Performance (slowdown)**					
0 KB	10.16	7.08	3.83	1.61	1.01	1.00
4.5 KB	13.23	6.55	2.83	1.55	1.02	1.00
9 KB	17.11	6.78	2.93	1.50	1.01	1.00
18 KB		11.96	3.28	1.45	1.00	1.00
36 KB			8.34	1.55	1.01	1.00
72 KB				5.83	1.08	1.00
144 KB					3.64	1.01
288 KB						3.03
Area (compression ratio)	61%	62%	66%	73%	86%	114%

B) CodePack

Table 6.20: Decompression buffer performance and area (vortex)



A) Dictionary



B) CodePack

Figure 6.20: Performance and area of decompression buffer (vortex)

Chapter 7
Conclusion

Integrated circuit technology has reached a point were it is feasible to integrate all components of an embedded system into a single chip. The applications for embedded systems are becoming more complicated as convergence between information appliances occurs. One example is the recent merging of fax, copier, and printer functions into one product. A significant amount of memory in the system-on-a-chip is devoted to storing programs. Therefore it is sensible to devote a small piece of hardware or software to interpret compressed programs. The ability to use a smaller program memory allows a given system-on-a-chip to either use a smaller die area or add more program functionality using the space savings created by compression.

Even with the advent of high-density embedded DRAM, we can expect the use of compressed programs to increase in the future as embedded developers optimize systems for increased functionality at reduced cost. Thumb, MIPS-16, Xtensa, MCore, and SH-4 are new instruction sets designed within the last five years that have given special attention to code size. Another pressure is the increasing complexity of instruction sets that encode conditional execution fields and use larger register files for increased performance. Examples of such instruction sets are TriMedia and IA-64. Compression may help control the code size expansion of these wide instruction formats.

7.1 Research contributions

Code compression is the technique of compressing programs to improve instruction density. This dissertation has explored code compression implementations in both hardware and software and suggested optimizations for each.

Instruction-level compression

We examined code compression systems that use individual instructions as a basic unit of compression. When using dictionary compression, most of the compression benefit comes from compressing instructions individually. Although many groups of instructions that repeated were found in applications, they repeated too infrequently to be conveniently used. A dictionary containing every group of repeating instructions would be enormous. This motivated the examination of a simple, fast dictionary decompressor used in the software-managed decompression studies. It uses compressed programs in which individual instructions are encoded in a compressed format. One interesting result of this work showed that even programs using 16-bit instruction subsets of 32-bit instruction sets have a significant amount of repetition remaining that can be removed with code compression. Clearly, re-encoding the instruction set in such a general manner is not enough to achieve the smallest programs. Compression algorithms have the flexibility to tune the final encoding to individual applications and generate even smaller programs.

Hardware decompression

We studied how decompression could be supported in hardware. We analyzed IBM's CodePack algorithm by executing the decompressor on a cycle-level simulator. The results showed that a significant amount of time was spent accessing an index table and decompressing encoded bytes received from the memory system. The addition of a modest cache and additional hardware to decompress more bytes per cycle removed most of the decompression overhead. The improved code density allowed the program to be fetched across the memory-processor bus in fewer cycles. For most applications, performance improved beyond the original program because the benefit of using fewer bus cycles outweighed the small remaining decompression overhead. The sensitivity of compressed program performance on different architectures was measured. The applications on systems with narrow memory buses and long memory latencies experienced large execution time improvements. When compressed programs did execute slower, using larger caches diminished the slowdown to acceptable levels. Therefore, compressed code systems do not necessarily impose a performance penalty and can often improve performance.

Software decompression

The success with hardware decompression led us to investigate the feasibility of using software to perform decompression. Software decompression is interesting because it reduces hardware complexity, can be implemented at a lower cost, and provides much more flexibility in the system design. Software systems for code decompression have been less studied than hardware solutions. This dissertation is one of the first comprehensive studies on this subject that attempts to optimize both code size and performance simultaneously.

The challenge in software decompression is to maintain application performance. Compressed programs always execute slower in this environment because the application is interrupted while the decompression program generates native code. We proposed adding minor architectural support to the system to efficiently support the decompression program. The hardware support includes the raising of an exception on a cache miss and an instruction to store decompressed instructions directly into the instruction cache. This allows the decompressor to control the contents of the instruction cache and transparently support compressed applications from the viewpoint of the microprocessor core.

The software decompressor is invoked on a cache miss. Executing the decompressor causes a cache miss to appear 10 to 100 times slower than the usual hardware-supported miss. Using larger caches to avoid cache misses reduced the slowdown tremendously. This is encouraging since cache sizes in embedded processors have been increasing over the last few years. Inspite of the large decompression latency, we found that loop-oriented applications (in particular multimedia applications) perform close to native code levels. This occurs because the cost of decompressing loops is amortized when the loop is decompressed into the cache once and executed many times.

We investigated removing the decompression overhead by adding a small amount of native code to the system. One method to do this is to use hybrid programs that contain both compressed and native code. Previous techniques for partitioning a program into compressed and native code used a simple instruction execution profile. We found that using a profile sensitive to cache misses could improve performance by 50%. The reason for this was that our compressed code system only loses performance on cache misses. By leaving code that was likely to cause a cache miss as native code, the decompression pen-

alty was avoided. The other method to improve performance is memoization. This involves the allocation a scratch buffer that can be filled with decompressed code as the program runs. The scratch buffer is kept in dense main memory and can store many more decompressions per unit area than the instruction cache. This avoids the high decompression penalty, but incurs a small cost for copying the decompressed instructions into the instruction cache. Using both hybrid programs and memoization together often resulted in the highest performance for a given die area budget.

This dissertation shows that in many situations, software decompression can perform very close to native code systems. The use of some native code in the compressed program allows a whole continuum of compressed programs to be generated with different trade-offs made for area and performance. This allows any compressed program to perform arbitrarily close to native code although the compression ratio may suffer.

7.2 Future work

The work in this dissertation has uncovered many opportunities for future studies. This section overviews some the major areas for future innovation in compressed code systems.

7.2.1 High-performance computing

While code compression was originally developed for embedded computing applications, it can be used to improve high-performance systems. We have seen that hardware decompression can cause compressed programs to execute faster than native programs. The memory systems in the experiments of this dissertation contained only one level of cache. It is interesting to consider how to implement compression on multilevel cache hierarchies. For example, if code is compressed in a unified L2 cache, then the L2 cache will experience fewer cache miss conflicts between instructions and data. If multiple processors share a L2 cache, then decompressing a shared library into the L2 would make the native code visible for all processors and lower decompression overhead.

The CodePack decompressor prefetched and decompressed instruction cache lines before the miss in the instruction cache occurred. Only a simple next-line prefetch algo-

rithm was used. Examining branches in the decompressed instruction stream and combining the information with advanced instruction prefetch and branch prediction algorithms could also improve the overhead of decompression.

7.2.2 Code generation optimizations for compression

The instruction level compression study showed that there are many unique instructions in a compiled application that do not match other instructions. This is often due to the use of different register names or immediate values. A smart code generator with knowledge about the compression algorithm could optimize sequences of instructions for greater compressibility. For example, using register allocation to create groups of identical instructions so that procedure abstraction can be applied has been previously studied. However, register allocation that targets actual compression algorithms has not yet been examined. This would improve the size of the compressed application by reducing the number of unique instructions that must remain in native form in the compressed program. In the case of hardware decompression which can provide speedup over native code due to improved code density, the application might also become faster than a compressed program that is not optimized for compression.

The hybrid program studies showed that a poor procedure placement could sometimes cause a hybrid program to perform worse as more native code was used. Therefore, it would be interesting to adapt a procedure placement optimization to be aware of hybrid programs so that performance aberrations due to poor placements do not occur.

7.2.3 Low energy consumption

Many embedded systems, especially battery-operated ones, have strict power dissipation requirements. Therefore, it is important to know how code compression affects energy consumption. Some studies have suggested that since code compression uses fewer bus cycles, significant energy savings can be realized due to fewer bus transitions. However, these studies have typically ignored the energy required to execute the decompression algorithm. At this time, the results are inconclusive and more studies are needed to determine if code compression can lower energy consumption.

7.3 Epilogue

This dissertation demonstrates that code compression is a feasible method for improving code density. Hardware decompression of the instruction stream in parallel with the execution of the application by the microprocessor can even speedup compressed programs over native programs. Even software decompression can often execute compressed programs with little performance loss. This dissertation provides microprocessor-based systems designers with an overview of the trade-offs involved in compressed code systems. In addition, it has shown that optimizations for hardware and software decompression are effective in reducing the performance penalties associated with compressed code systems.

Appendix A
Program listings

This appendix provides source code for the decompression programs used in the experiments.

A.1 Macros

The source code listings use many macros to represent inline assembly code. They are used to access regular SimpleScalar PISA instructions and special instructions that are used to support software decompression.

Macro	PISA Assembly code
igetctrl(addr,offset,data)	igetctrl data,offset(addr)
igettag(addr,offset,data)	igettag data,offset(addr)
imap(addr,offset,memaddr)	imap memaddr,offset(addr)
imapw(addr,offset,memaddr)	imapw memaddr,offset(addr)
iret	iret
isync	isync
iunmap(addr,offset,memaddr)	iunmap memaddr,offset(addr)
iunmapw(addr,offset,memaddr)	iunmapw memaddr,offset(addr)
lwicw(addr,offset,data)	lwicw data,offset(addr)
mem_load_1B(addr,offset,data)	lb data,offset(addr)
mem_load_1B_RR(addr1,addr2,data)	lb data,(addr1 + addr2)
mem_loadu_1B(addr,offset,data)	lbu data,offset(addr)
mem_loadu_1B_RR(addr1,addr2,data)	lbu data,(addr1 + addr2)
mem_load_2B(addr,offset,data)	lh data,offset(addr)
mem_load_2B_RR(addr1,addr2,data)	lh data,(addr1 + addr2)
mem_loadu_2B(addr,offset,data)	lhu data,offset(addr)
mem_loadu_2B_RR(addr1,addr2,data)	lhu data,(addr1 + addr2)
mem_load_4B(addr,offset,data)	lw data,offset(addr)
mem_load_4B_RR(addr1,addr2,data)	lw data,(addr1 + addr2)
mem_store_4B(addr,offset,data)	sw data,offset(addr)
mem_store_4B_RR(addr1,addr2,data)	sw data,(addr1 + addr2)
mfc0(dest,src)	mfc0 dest,src
swic(addr,offset,data)	swic data,offset(addr)
swicw(addr,offset,data)	swicw data,offset(addr)
sync	sync

Table A.1: Macros
The labels `addr`, `addr1`, `addr2`, `addrmem`, `data`, and `dest` are general purpose machine registers. The label `src` is a special system register. The label `offset` is a 16-bit constant immediate value.

A.2 Dictionary

This is the baseline dictionary decompressor.

```
// Dictionary Compression Exception Handler
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//
// Assumptions:
//   I-cache line is 8 32-bit instructions long

// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with a line of 8 instructions.

// Special SimpleScalar instructions
//   mfc0: move from coprocessor-0 register to general purpose register
//   swic: store word in instruction cache
//   isync:      instruction synchronization

#include “eh.h”

void eh();

void eh()
{
  unsigned long baddr;
  unsigned long text_base;
  unsigned long dict_base;
  unsigned long indices_base;
  unsigned short indexAddr;
  unsigned long stopAddr;
  unsigned short index;
  unsigned long iword;


  // Get parameters from system coprocessor
  mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC and put in baddr;
  mfc0(text_base,MD_REG_C0_TEXT);  // where .text starts
  mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
  mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

  // Align miss address to cache-line boundary by zeroing low 5 bits.
  // Assume 32B lines.
  baddr = (baddr >> 5) << 5;

  // Calculate the address of the first index.
  indexAddr = (((baddr - text_base) >> 1) + indices_base);

  // Load 8 instructions.
  //  1. Load the index
  //  2. Scale to 4B access
  //  3. Load instruction from dictionary
  //  4. Store instruction in cache.
```

```
  // instruction 1
  mem_loadu_2B(indexAddr,0,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,0,iword);

  mem_loadu_2B(indexAddr,2,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,4,iword);

  mem_loadu_2B(indexAddr,4,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,8,iword);

  mem_loadu_2B(indexAddr,6,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,12,iword);

  mem_loadu_2B(indexAddr,8,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,16,iword);

  mem_loadu_2B(indexAddr,10,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,20,iword);

  mem_loadu_2B(indexAddr,12,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,24,iword);

  mem_loadu_2B(indexAddr,14,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,28,iword);

  // Context synchronization. Let swic instructions finish before
  // returning from the exception handler and executing the cache
  // line.
  isync;
  iret;
}
```

A.3 Dictionary Memo-IW

```
// Dictionary Compression Exception Handler (IW)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//
// Assumptions:
//   I-cache line is 8 32-bit instructions long
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with a line of 8 instructions.
//
// Special SimpleScalar instructions
//   mfc0: move from coprocessor-0 register to general purpose register
//   swic: store word in instruction cache
//   isync:      instruction synchronization

#include “eh.h”

void eh();

void eh()
{
  unsigned long baddr;
  unsigned long text_base;
  unsigned long dict_base;
  unsigned long indices_base;
  unsigned short indexAddr;

  unsigned long cacheTagOffset;
  unsigned long cacheTag;
  unsigned long cacheDataOffset;
  unsigned long cacheDataAddr;

  unsigned long cacheDataBase;
  unsigned long cacheTagBase;

  unsigned short index;
  unsigned long iword;

  // Get parameters from system coprocessor
  mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;
  // Align miss address to cache-line boundary by zeroing low 5 bits. Assume 32B
lines.
  baddr = (baddr >> 5) << 5;


  // Check SW cache. 16 KB cache.
  //
  // Cache Data is located at address 0x00200000
  // Cache Tags are located at address 0x00210000
  cacheDataBase = (0x20) << 16;
  cacheTagBase = (0x21) << 16;
```

```
cacheTagOffset = ((baddr) << 18) >> 21;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (baddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

if (cacheTag == baddr)
  {
    // hit
    // copy 8 instruction from memoization table into i-cache

    mem_load_4B(cacheDataAddr,0,iword);
    swic(baddr,0,iword);
    mem_load_4B(cacheDataAddr,4,iword);
    swic(baddr,4,iword);
    mem_load_4B(cacheDataAddr,8,iword);
    swic(baddr,8,iword);
    mem_load_4B(cacheDataAddr,12,iword);
    swic(baddr,12,iword);
    mem_load_4B(cacheDataAddr,16,iword);
    swic(baddr,16,iword);
    mem_load_4B(cacheDataAddr,20,iword);
    swic(baddr,20,iword);
    mem_load_4B(cacheDataAddr,24,iword);
    swic(baddr,24,iword);
    mem_load_4B(cacheDataAddr,28,iword);
    swic(baddr,28,iword);

    goto done;
  }

// Miss memoization table

// update tag
mem_store_4B_RR(cacheTagBase,cacheTagOffset,baddr);

mfc0(text_base,MD_REG_C0_TEXT);  // where .text starts
mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

// Calculate the address of the first index.
indexAddr = (((baddr - text_base) >> 1) + indices_base);

// Load new cache line with 8 instructions.
//  1. Load the index
//  2. Scale to 4B access
//  3. Load instruction from dictionary
//  4. Store instruction in cache.

// instruction 1
mem_loadu_2B(indexAddr,0,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,0,iword);
mem_store_4B(cacheDataAddr,0,iword);

mem_loadu_2B(indexAddr,2,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,4,iword);
```

```
  mem_store_4B(cacheDataAddr,4,iword);

  mem_loadu_2B(indexAddr,4,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,8,iword);
  mem_store_4B(cacheDataAddr,8,iword);

  mem_loadu_2B(indexAddr,6,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,12,iword);
  mem_store_4B(cacheDataAddr,12,iword);

  mem_loadu_2B(indexAddr,8,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,16,iword);
  mem_store_4B(cacheDataAddr,16,iword);

  mem_loadu_2B(indexAddr,10,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,20,iword);
  mem_store_4B(cacheDataAddr,20,iword);

  mem_loadu_2B(indexAddr,12,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,24,iword);
  mem_store_4B(cacheDataAddr,24,iword);

  mem_loadu_2B(indexAddr,14,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,28,iword);
  mem_store_4B(cacheDataAddr,28,iword);

 done:
  isync;
  iret;
}
```

A.4 Dictionary Memo-IL

```
// Dictionary Compression Exception Handler (IL)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//
// Assumptions:
//   cache line is 8 32-bit instructions long

// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with a line of 8 instructions.

// Special SimpleScalar instructions
//   mfc0: move from coprocessor-0 register to general purpose register
//   swic: store word in instruction cache
//   isync:      instruction synchronization
//   sync:       memory synchronization

#include “eh.h”

void eh();

void eh()
{
  unsigned long baddr;
  unsigned long text_base;
  unsigned long dict_base;
  unsigned long indices_base;
  unsigned short indexAddr;

  unsigned long cacheTagOffset;
  unsigned long cacheTag;
  unsigned long cacheDataOffset;
  unsigned long cacheDataAddr;

  unsigned long cacheDataBase;
  unsigned long cacheTagBase;

  unsigned short index;
  unsigned long iword;


  // Get parameters from system coprocessor
  mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;
  // Align miss address to cache-line boundary by zeroing low 5 bits. Assume 32B
lines.
  baddr = (baddr >> 5) << 5;

  // Check SW cache. 16 KB cache.
  //
  // Cache Data is located at address 0x00200000
  // Cache Tags are located at address 0x00210000
  cacheDataBase = (0x20) << 16;
```

```
cacheTagBase = (0x21) << 16;
cacheTagOffset = ((baddr) << 18) >> 21;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (baddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

if (cacheTag == baddr)
  {
    // hit

    imap(baddr,0,cacheDataAddr);
    sync;
    goto done;

  }

// update tag
mem_store_4B_RR(cacheTagBase,cacheTagOffset,baddr);

mfc0(text_base,MD_REG_C0_TEXT);  // where .text starts
mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

// Calculate the address of the first index.
indexAddr = (((baddr - text_base) >> 1) + indices_base);

// Load new cache line. 8 instructions.

mem_loadu_2B(indexAddr,0,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,0,iword);

mem_loadu_2B(indexAddr,2,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,4,iword);

mem_loadu_2B(indexAddr,4,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,8,iword);

mem_loadu_2B(indexAddr,6,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,12,iword);

mem_loadu_2B(indexAddr,8,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,16,iword);

mem_loadu_2B(indexAddr,10,index);
index <<= 2;
mem_load_4B_RR(dict_base,index,iword);
swic(baddr,20,iword);

mem_loadu_2B(indexAddr,12,index);
```

```
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,24,iword);

  mem_loadu_2B(indexAddr,14,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,28,iword);

  // copy to SW cache the line just filled in I-cache.
  iunmap(baddr,0,cacheDataAddr);

 done:
  isync;
  iret;
}
```

A.5 Dictionary Memo-EW

```
// Dictionary Compression Exception Handler (EW)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//   SW cache is 16 KB direct-mapped.
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with a line of 8 instructions.
//  The line that is replaced in I-cache is moved into SW cache.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   swicw: index store word in instruction cache
//   lwicw:      index read instruction word from cache
//   igetctrl:   get control bits from I-cache line
//   igettag:    get tag bits from I-cache line
//   mfc0: move from coprocessor-0 register to general purpose register
//   isync:      instruction synchronization

#include “eh.h”

// IMPORTANT: THIS VALUE MUST CHANGE WHEN CACHE ORGANIZATION CHANGES
//   amount to shift address to get the tag. For 32B lines, this is Log(#sets)+5.
//   For example, 4 KB cache with assoc=2,line=32B has 64 sets -->
ICACHE_TAG_SHIFT=11
#ifndef ICACHE_TAG_SHIFT
#error Must define ICACHE_TAG_SHIFT
#endif

void eh();

void eh()
{
  unsigned long baddr;
  unsigned long text_base;
  unsigned long dict_base;
  unsigned long indices_base;
  unsigned short indexAddr;

  unsigned long cacheTagOffset;
  unsigned long cacheTag;
  unsigned long cacheDataOffset;
  unsigned long cacheDataAddr;

  unsigned long cacheDataBase;
  unsigned long cacheTagBase;

  unsigned long currentTag;
  unsigned long ctrlbits;
```

```
unsigned long replAddr;
unsigned long compressedBlockAddrTag;

unsigned long currentAddr;
unsigned long replCacheDataAddr;
unsigned long replCacheTagOffset;

unsigned long iword;
unsigned short index;

// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;
baddr = (baddr >> 5) << 5;

// Get the cache tag for the missed address
// For example, tag has bottom 11 bits removed in 4 KB 2-way 32B-line cache
compressedBlockAddrTag = baddr>>ICACHE_TAG_SHIFT;

// Check SW cache. 16 KB cache.
//
// Cache Data is located at address 0x00200000
// Cache Tags are located at address 0x00210000
cacheDataBase = (0x20) << 16;
cacheTagBase = (0x21) << 16;
cacheTagOffset = ((baddr) << 18) >> 21;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (baddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

// Find replacement cache line
igetctrl(baddr,0,ctrlbits);
// if lru 0, use this line, else next way
replAddr = baddr;
replAddr |= (ctrlbits >> 2); // LRU bit is (ctrlbits >> 2)

if (cacheTag == compressedBlockAddrTag)
  {
    // hit
    // copy 8 instruction from memoization table into I-cache

    mem_load_4B(cacheDataAddr,0,iword);
    swicw(replAddr,0,iword);
    mem_load_4B(cacheDataAddr,4,iword);
    swicw(replAddr,4,iword);
    mem_load_4B(cacheDataAddr,8,iword);
    swicw(replAddr,8,iword);
    mem_load_4B(cacheDataAddr,12,iword);
    swicw(replAddr,12,iword);
    mem_load_4B(cacheDataAddr,16,iword);
    swicw(replAddr,16,iword);
    mem_load_4B(cacheDataAddr,20,iword);
    swicw(replAddr,20,iword);
    mem_load_4B(cacheDataAddr,24,iword);
    swicw(replAddr,24,iword);
    mem_load_4B(cacheDataAddr,28,iword);
    swicw(replAddr,28,iword);

    goto done;
  }
```

```
  // Miss SW cache

  // Write replaced I-cache line to SW cache

  //Find address of line to be replaced in I-cache
  // Get tag of replaced line to find out where in SW cache to put it.
  igettag(replAddr,0,currentTag);
  // currentAddr is the address of this line
  currentAddr = baddr;
  // Strip top bits to get byte index into cache
  currentAddr &= ((1<<ICACHE_TAG_SHIFT) - 1);
  // Add tag bits on top
  currentAddr |= (currentTag << ICACHE_TAG_SHIFT);

  // Store tag in SW cache for replaced line.
  // find address to store TAG. Just like for faulting address.
  replCacheTagOffset = (currentAddr << 18) >> 21; //strip top bits. SW Cache
parameter.
  // write tag into SW cache
  mem_store_4B_RR(cacheTagBase,replCacheTagOffset,currentTag);

  // Find address in SW cache to store instructions.
  replCacheDataAddr = (currentAddr << 18) >> 18;
  replCacheDataAddr += cacheDataBase;

  // Write instructions in replaced line to SW cache.
  lwicw(replAddr,0,iword);
  mem_store_4B(replCacheDataAddr,0,iword);
  lwicw(replAddr,4,iword);
  mem_store_4B(replCacheDataAddr,4,iword);
  lwicw(replAddr,8,iword);
  mem_store_4B(replCacheDataAddr,8,iword);
  lwicw(replAddr,12,iword);
  mem_store_4B(replCacheDataAddr,12,iword);
  lwicw(replAddr,16,iword);
  mem_store_4B(replCacheDataAddr,16,iword);
  lwicw(replAddr,20,iword);
  mem_store_4B(replCacheDataAddr,20,iword);
  lwicw(replAddr,24,iword);
  mem_store_4B(replCacheDataAddr,24,iword);
  lwicw(replAddr,28,iword);
  mem_store_4B(replCacheDataAddr,28,iword);

  // Decompress a new line

  mfc0(text_base,MD_REG_C0_TEXT);  // where .text starts
  mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
  mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

  // Calculate the address of the first index.
  indexAddr = (((baddr - text_base) >> 1) + indices_base);

  // Load new cache line. 8 instructions.

  mem_loadu_2B(indexAddr,0,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,0,iword);
```

```
  mem_loadu_2B(indexAddr,2,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,4,iword);

  mem_loadu_2B(indexAddr,4,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,8,iword);

  mem_loadu_2B(indexAddr,6,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,12,iword);

  mem_loadu_2B(indexAddr,8,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,16,iword);

  mem_loadu_2B(indexAddr,10,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,20,iword);

  mem_loadu_2B(indexAddr,12,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,24,iword);

  mem_loadu_2B(indexAddr,14,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,28,iword);

 done:
  isync;
  iret;
}
```

A.6 Dictionary Memo-EL

```
// Dictionary Compression Exception Handler (EL)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//   SW cache is 16 KB direct-mapped.
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with a line of 8 instructions.
//  The line that is replaced in I-cache is moved into SW cache.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   imapw: read I-cache line from main memory
//   iunmapw: write I-cache line to main memory
//   igetctrl:   get control bits from I-cache line
//   igettag:    get tag bits from I-cache line
//   mfc0: move from coprocessor-0 register to general purpose register
//   isync:      instruction synchronization
//   sync:       memory synchronization

#include “eh.h”

// IMPORTANT: THIS VALUE MUST CHANGE WHEN CACHE ORGANIZATION CHANGES
//   amount to shift address to get the tag. For 32B lines, this is Log(#sets)+5.
//   For example, 4 KB cache with assoc=2,line=32B has 64 sets -->
ICACHE_TAG_SHIFT=11
#ifndef ICACHE_TAG_SHIFT
#error Must define ICACHE_TAG_SHIFT
#endif

void eh();

void eh()
{
  unsigned long baddr;
  unsigned long text_base;
  unsigned long dict_base;
  unsigned long indices_base;
  unsigned short indexAddr;

  unsigned long cacheTagOffset;
  unsigned long cacheTag;
  unsigned long cacheDataOffset;
  unsigned long cacheDataAddr;

  unsigned long cacheDataBase;
  unsigned long cacheTagBase;

  unsigned long currentTag;
```

```
unsigned long ctrlbits;
unsigned long replAddr;
unsigned long compressedBlockAddrTag;

unsigned long currentAddr;
unsigned long replCacheDataAddr;
unsigned long replCacheTagOffset;

unsigned long iword;
unsigned short index;

// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;
baddr = (baddr >> 5) << 5;

// Get the cache tag for the missed address
// For example, tag has bottom 11 bits removed in 4 KB 2-way 32B-line cache
compressedBlockAddrTag = baddr>>ICACHE_TAG_SHIFT;

// Check SW cache. 16 KB cache.
//
// Cache Data is located at address 0x00200000
// Cache Tags are located at address 0x00210000
cacheDataBase = (0x20) << 16;
cacheTagBase = (0x21) << 16;
cacheTagOffset = ((baddr) << 18) >> 21;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (baddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

// Find replacement cache line
igetctrl(baddr,0,ctrlbits);
// if lru 0, use this line, else next way
replAddr = baddr;
replAddr |= (ctrlbits >> 2); // ctrlbits >> 2 == lru

if (cacheTag == compressedBlockAddrTag)
  {
    // hit
    // copy 8 instruction from memoization table into i-cache

    imapw(replAddr,0,cacheDataAddr);
    sync;

    goto done;

  }

// Miss SW cache

// Write replaced I-cache line to SW cache

//Find address of line to be replaced in I-cache
// Get tag of replaced line to find out where in SW cache to put it.
igettag(replAddr,0,currentTag);

//Find address of line to be replaced in I-cache
// Get tag of replaced line to find out where in SW cache to put it.
igettag(replAddr,0,currentTag);
```

```
  // currentAddr is the address of this line
  currentAddr = baddr;
  // Strip top bits to get byte index into cache
  currentAddr &= ((1<<ICACHE_TAG_SHIFT) - 1);
  // Add tag bits on top
  currentAddr |= (currentTag << ICACHE_TAG_SHIFT);

  // Store tag in SW cache for replaced line.
  // find address to store TAG. Just like for faulting address.
  replCacheTagOffset = (currentAddr << 18) >> 21; //strip top bits. SW Cache
parameter.
  // write tag into SW cache
  mem_store_4B_RR(cacheTagBase,replCacheTagOffset,currentTag);

  // Find address in SW cache to store instructions.
  replCacheDataAddr = (currentAddr << 18) >> 18;
  replCacheDataAddr += cacheDataBase;

  // Write instructions in replaced line to SW cache.
  iunmapw(replAddr,0,replCacheDataAddr);

  // Decompress a new line

  mfc0(text_base,MD_REG_C0_TEXT);  // where .text starts
  mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
  mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

  // Calculate the address of the first index.
  indexAddr = (((baddr - text_base) >> 1) + indices_base);

  // Load new cache line. 8 instructions.

  mem_loadu_2B(indexAddr,0,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,0,iword);

  mem_loadu_2B(indexAddr,2,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,4,iword);

  mem_loadu_2B(indexAddr,4,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,8,iword);

  mem_loadu_2B(indexAddr,6,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,12,iword);

  mem_loadu_2B(indexAddr,8,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,16,iword);

  mem_loadu_2B(indexAddr,10,index);
  index <<= 2;
```

```
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,20,iword);

  mem_loadu_2B(indexAddr,12,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,24,iword);

  mem_loadu_2B(indexAddr,14,index);
  index <<= 2;
  mem_load_4B_RR(dict_base,index,iword);
  swic(baddr,28,iword);

 done:
  isync;
  iret;
}
```

A.7 CodePack

This is the baseline CodePack decompressor.

```
// CodePack Exception Handler
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//   C0[MD_REG_C0_LAT]: Location of index table.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with 2 lines of 8 instructions.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   mfc0: move from coprocessor-0 register to general purpose register
//   isync:      instruction synchronization

#include <stdio.h>
#include "eh.h"


unsigned char const indexHiLookup[] = {3, 5, 0, 0, 6, 7, 8, 16};
unsigned char const indexLoLookup[] = {0, 4, 0, 0, 5, 7, 8, 16};
unsigned short const ihalfHiLookup[] = {0x008, 0x020, 0, 0, 0x040, 0x080, 0x100,
0};
unsigned short const ihalfLoLookup[] = {0, 512+0x010, 0, 0, 512+0x020, 512+0x080,
512+0x100, 0};

void eh();

void eh()
{

  unsigned long ihalfHiLookupAddr;
  unsigned long ihalfLoLookupAddr;
  unsigned long indexHiLookupAddr;
  unsigned long indexLoLookupAddr;

  unsigned long loop;
  unsigned long loopByte;

  unsigned long insnNumber;

  unsigned long groupNumber;
  unsigned long blockNumber;
  unsigned long whichBlock;

  unsigned long indexTableEntry;

  unsigned long groupOffset;
```

```
unsigned long secondBlockOffset;
unsigned long groupAddr;
unsigned long blockAddr;
unsigned long iword32;

unsigned long baddr;
unsigned long compressedBlockAddr;

unsigned long text_base;
unsigned long dict_base;
unsigned long lat_base;
unsigned long indices_base;

// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;
mfc0(text_base,MD_REG_C0_TEXT);  // where compressed region starts
mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
mfc0(lat_base,MD_REG_C0_LAT); // where line address table starts
mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

// remove bottom portion, get line address from word address;
baddr = (baddr >> 5) << 5;
compressedBlockAddr = (baddr >> 6) << 6;

insnNumber = (baddr - text_base) >> 2; /* (b-ltb/4) 4=sizeof SS32 insn */

/* Determine what compression group and block the requested
   instruction is in.  32 instructions per compression group, 16
   instructions per compression block, 2 compression blocks per
   group.  */
groupNumber = insnNumber >> 5; /* / 32; */
blockNumber = insnNumber >> 4; /* / 16; */

/* Determine whether the requested address is in the first or second
   compression block in the group.  */
whichBlock = blockNumber % 2;

/* Fetch index table entry from memory */
mem_load_4B_RR(lat_base,groupNumber*4,indexTableEntry);

/* offset of compression group is upper 26 bits of index table entry
   shifted right */
groupOffset = indexTableEntry & ~0x3f;
groupOffset = groupOffset >> 6;

/* offset of second block in compression group is lower six bits of
   index table entry */
secondBlockOffset = indexTableEntry & 0x3f;

/* address of compression group is base address of text plus group
   offset */
groupAddr = indices_base + groupOffset;

/* check whether first or second block in group is being accessed */
if (whichBlock == 0) /* first block */
  {
    /* block address is same as group address */
    blockAddr = groupAddr;
    if (secondBlockOffset <= 1)
```

```
      {
        /* if block offset of index table entry is 0 or 1, first
           block is not compressed */
        goto copy;
      }
  }
else /* second block */
  {
    if (secondBlockOffset <= 1)
      {
        /* if block offset of index table entry is 0 or 1, second
           block address is address of first block + 64 */
        blockAddr = groupAddr + 64;
        if (secondBlockOffset == 0)
          {
            /* if block offset of index table entry is 0, second
       block is not compressed */
            goto copy;
          }
      }
    else
      {
        /* if block offset of index table entry is greater than 1,
           second block address is address of first block plus block
           offset (lower 6 bits of index table entry) */
        blockAddr = groupAddr + secondBlockOffset;
      }
  } /* second block */

  ihalfHiLookupAddr = (long) &ihalfHiLookup;
  ihalfLoLookupAddr = (long) &ihalfLoLookup;
  indexHiLookupAddr = (long) &indexHiLookup;
  indexLoLookupAddr = (long) &indexLoLookup;


/* decompress 2 cache lines */
{
  unsigned long tagHi, tagLo;
  unsigned long tagHiLength, tagLoLength;
  unsigned long indexHi, indexLo;
  unsigned long indexHiLength, indexLoLength;
  unsigned short ihalfHi16, ihalfLo16;

  unsigned long inputWord;
  unsigned long tempWord;
  unsigned long bitPosition = 32 - (8 * (blockAddr & 0x03));
  unsigned long addr = blockAddr & ~0x03;
  unsigned k;

  /* fetch first word from memory into inputWord */
  mem_load_4B(addr,inputWord);

  for (loop = 0; loop < 16; loop++)
    /* fetch and decompress 16 instructions (one block) */
    {
      /* determine length of high tag by looking at next bit of
         input word */
      tempWord = inputWord << (32-bitPosition);
```

```
/* tempWord is 0 or 1. If 0, length = 3, else length=2. */
tagHiLength = 2 + (tempWord>>31);
k = 32 - tagHiLength;
if (bitPosition > tagHiLength)
  {
    /* if all the bits for the high tag are available in this
       input word, grab them and readjust bitPosition */
    tempWord = tempWord >> k;

    tagHi = tempWord;
    bitPosition -= tagHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    tagHi = tempWord;

    /* get the rest of the bits for the high tag from the next
       word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagHi |= tempWord;
       }
  }

/* repeat above for low tag */
tempWord = inputWord << (32-bitPosition);

/* tempWord is 0 or 1. If 0, length = 3, else length=2. */
tagLoLength = 2 + (tempWord>>31);
k = 32-tagLoLength;
if (bitPosition > tagLoLength)
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;
    bitPosition -= tagLoLength;
  }
else
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;

    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagLo |= tempWord;
       }
  }

/* determine length of indexes according to the tag values */
```

```
mem_load_1B_RR(indexHiLookupAddr,tagHi,indexHiLength);
mem_load_1B_RR(indexLoLookupAddr,tagLo,indexLoLength);

/* get high index value from memory */
tempWord = inputWord << (32-bitPosition);
k = 32 - indexHiLength;
if (bitPosition > indexHiLength)
  {
    /* if the rest of the bits for the high index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexHi = tempWord;
    bitPosition -= indexHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexHi = tempWord;

    /* get the rest of the bits for the high index from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexHi |= tempWord;
       }
  }

/* repeat above for low index value */
tempWord = inputWord << (32-bitPosition);
k = 32-indexLoLength;
if (bitPosition > indexLoLength)
  {
    /* if the rest of the bits for the low index are available
       in this input word, grab them */
    tempWord = tempWord >> k;

    indexLo = tempWord;
    bitPosition -= indexLoLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexLo = tempWord;

    /* get the rest of the bits for the low index from the
        next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
```

```
      tempWord = inputWord >> bitPosition;
      indexLo |= tempWord;
            }
        }

      /* Now we have the indexes into the decode lookup table for
         this instruction.  Look up the high and low half-words and
         store into temps.  Each entry in the dictionary is 2 bytes
         long.  The dictionary for the high index starts at
         dict_base.  The dictionary for the low index starts at
         dict_base + 512. */

      if (tagHi != 7)
        {
          unsigned temp;
          mem_load_2B_RR(ihalfHiLookupAddr,tagHi<<1,temp);
          temp += indexHi;
          mem_load_2B_RR(dict_base,2*temp,ihalfHi16);
        }
      else
        {
          ihalfHi16 = indexHi;
        }

      if ((tagLo != 7) && (tagLo != 0))
        {
          unsigned temp;
          mem_load_2B_RR(ihalfLoLookupAddr,tagLo<<1,temp);
          temp += indexLo;
          mem_load_2B_RR(dict_base,2*temp,ihalfLo16);
        }
      else if (tagLo == 0)
        {
          ihalfLo16 = 0;
        }
      else
        {
          ihalfLo16 = indexLo;
        }

      /* put together the whole 32-bit instruction and put it into
         temporary storage */
      iword32 = (ihalfHi16 << 16) | ihalfLo16;
      swic(compressedBlockAddr,0,iword32);
      compressedBlockAddr+=4;
     }

 }
 goto done;


copy:
 /* block is not compressed, we can just copy it directly from memory
    with no decoding */
 {
   unsigned char temp;
   unsigned long addr = blockAddr;

   for (loop = 0; loop < 16 ; loop++)
```

```
      {
        unsigned long iword=0;
        /* fetch 16 instructions (one block, 64 bytes) and put them
           into temporary storage */
        for (loopByte = 0; loopByte < 4; loopByte++)
          {
            mem_load_1B(addr,0,temp);
            addr++;
            ((unsigned char *)&iword)[loopByte] = temp;
          }
        swic(compressedBlockAddr,0,iword);
        compressedBlockAddr+=4;
      }

  }

 done:
  isync;
  iret;

}
```

A.8 CodePack Memo-IW

```
// CodePack Exception Handler (IW)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//   C0[MD_REG_C0_LAT]: Location of index table.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with 2 lines.
//
// Memoization
//   Store all decompressed instructions in a software cache.  Look in
//   this cache first for decompressed instructions and copy them into
//   I-cache with swic.  If the instructions are not found in the
//   software cache, then do decompression as normal and put a copy
//   into the software cache.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   mfc0: move from coprocessor-0 register to general purpose register
//   isync:      instruction synchronization

#include “eh.h”

unsigned char const indexHiLookup[] = {3, 5, 0, 0, 6, 7, 8, 16};
unsigned char const indexLoLookup[] = {0, 4, 0, 0, 5, 7, 8, 16};
unsigned short const ihalfHiLookup[] = {0x008, 0x020, 0, 0, 0x040, 0x080, 0x100,
0};
unsigned short const ihalfLoLookup[] = {0, 512+0x010, 0, 0, 512+0x020, 512+0x080,
512+0x100, 0};

void eh();

void eh()
{
  unsigned long loop;
  unsigned long loopByte;

  unsigned long insnNumber;

  unsigned long groupNumber;
  unsigned long blockNumber;
  unsigned long whichBlock;

  unsigned long indexTableEntry;

  unsigned long groupOffset;
  unsigned long secondBlockOffset;
  unsigned long groupAddr;
  unsigned long blockAddr;
```

```
unsigned long iword32;

unsigned long baddr;
unsigned long compressedBlockAddr;

unsigned long text_base;
unsigned long dict_base;
unsigned long lat_base;
unsigned long indices_base;

unsigned long cacheTagOffset;
unsigned long cacheTag;
unsigned long cacheDataOffset;
unsigned long cacheDataAddr;

unsigned long cacheDataBase;
unsigned long cacheTagBase;


unsigned long ihalfHiLookupAddr;
unsigned long ihalfLoLookupAddr;
unsigned long indexHiLookupAddr;
unsigned long indexLoLookupAddr;


// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;

// remove bottom portion, get line address from word address;
baddr = (baddr >> 5) << 5;
compressedBlockAddr = (baddr >> 6) << 6;

// Check SW cache. 16 KB cache.
//
// Cache Data is located at address 0x00200000
// Cache Tags are located at address 0x00210000
cacheDataBase = (0x20) << 16;
cacheTagBase = (0x21) << 16;
/* Note: Use 21 for 1-line granularity. Use 22 for 2-line granularity*/
cacheTagOffset = ((compressedBlockAddr) << 18) >> 22;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (compressedBlockAddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

if (cacheTag == compressedBlockAddr)
  {
    // hit
    // copy 16 instruction from software cache into I-cache

    /* ASSUME: if the first line is here, then the second line is
       too since we always decompress 2 lines at a time.  So we
       really only need half the tag store! Therefore, we don't need
       to check for a hit in the second line */

    unsigned long iword;

    mem_load_4B(cacheDataAddr,0,iword);
    swic(compressedBlockAddr,0,iword);
    mem_load_4B(cacheDataAddr,4,iword);
```

```
    swic(compressedBlockAddr,4,iword);
    mem_load_4B(cacheDataAddr,8,iword);
    swic(compressedBlockAddr,8,iword);
    mem_load_4B(cacheDataAddr,12,iword);
    swic(compressedBlockAddr,12,iword);
    mem_load_4B(cacheDataAddr,16,iword);
    swic(compressedBlockAddr,16,iword);
    mem_load_4B(cacheDataAddr,20,iword);
    swic(compressedBlockAddr,20,iword);
    mem_load_4B(cacheDataAddr,24,iword);
    swic(compressedBlockAddr,24,iword);
    mem_load_4B(cacheDataAddr,28,iword);
    swic(compressedBlockAddr,28,iword);
    mem_load_4B(cacheDataAddr,32,iword);
    swic(compressedBlockAddr,32,iword);
    mem_load_4B(cacheDataAddr,36,iword);
    swic(compressedBlockAddr,36,iword);
    mem_load_4B(cacheDataAddr,40,iword);
    swic(compressedBlockAddr,40,iword);
    mem_load_4B(cacheDataAddr,44,iword);
    swic(compressedBlockAddr,44,iword);
    mem_load_4B(cacheDataAddr,48,iword);
    swic(compressedBlockAddr,48,iword);
    mem_load_4B(cacheDataAddr,52,iword);
    swic(compressedBlockAddr,52,iword);
    mem_load_4B(cacheDataAddr,56,iword);
    swic(compressedBlockAddr,56,iword);
    mem_load_4B(cacheDataAddr,60,iword);
    swic(compressedBlockAddr,60,iword);

    goto done;
  }

// Miss SW cache

/* update tag */
mem_store_4B_RR(cacheTagBase,cacheTagOffset,compressedBlockAddr);

mfc0(text_base,MD_REG_C0_TEXT);  // where compressed region starts
mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
mfc0(lat_base,MD_REG_C0_LAT); // where line address table starts
mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

insnNumber = (baddr - text_base) >> 2; /* (b-ltb/4) 4=sizeof SS32 insn */

/* Determine what compression group and block the requested
   instruction is in.  32 instructions per compression group, 16
   instructions per compression block, 2 compression blocks per
   group.  */
groupNumber = insnNumber >> 5; /* / 32; */
blockNumber = insnNumber >> 4; /* / 16; */

/* Determine whether the requested address is in the first or second
   compression block in the group.  */
whichBlock = blockNumber % 2;

/* Calculate address of index table entry.  Each index table entry
   is 4 bytes long and contains info for one compression group */
/* Fetch index table entry from memory */
```

```
mem_load_4B_RR(lat_base,groupNumber*4,indexTableEntry);

/* offset of compression group is upper 26 bits of index table
   entry shifted right */
groupOffset = indexTableEntry & ~0x3f;
groupOffset = groupOffset >> 6;

/* offset of second block in compression group is lower six
   bits of index table entry */
secondBlockOffset = indexTableEntry & 0x3f;

/* address of compression group is base address of text plus group
   offset */
groupAddr = indices_base + groupOffset;

/* check whether first or second block in group is being accessed */
if (whichBlock == 0) /* first block */
  {
    /* block address is same as group address */
    blockAddr = groupAddr;
    if (secondBlockOffset <= 1)
      {
        /* if block offset of index table entry is 0 or 1, first block
           is not compressed */
        goto copy;
      }
  }
else /* second block */
  {
    if (secondBlockOffset <= 1)
      {
        /* if block offset of index table entry is 0 or 1, second block
           address is address of first block + 64 */
        blockAddr = groupAddr + 64;
        if (secondBlockOffset == 0)
          {
            /* if block offset of index table entry is 0, second
       block is not compressed */
             goto copy;
          }
      }
    else
      {
        /* if block offset of index table entry is greater than 1,
           second block address is address of first block plus block
           offset (lower 6 bits of index table entry) */
        blockAddr = groupAddr + secondBlockOffset;
      }
  } /* second block */

  ihalfHiLookupAddr = (long) &ihalfHiLookup;
  ihalfLoLookupAddr = (long) &ihalfLoLookup;
  indexHiLookupAddr = (long) &indexHiLookup;
  indexLoLookupAddr = (long) &indexLoLookup;


/* decompress 2 cache lines */
{
  unsigned long tagHi, tagLo;
```

```
unsigned long tagHiLength, tagLoLength;
unsigned long indexHi, indexLo;
unsigned long indexHiLength, indexLoLength;
unsigned short ihalfHi16, ihalfLo16;

unsigned long inputWord;
unsigned long tempWord;
unsigned long bitPosition = 32 - (8 * (blockAddr & 0x03));
unsigned long addr = blockAddr & ~0x03;
unsigned k;

/* fetch first word from memory into inputWord */
mem_load_4B(addr,inputWord);

for (loop = 0; loop < 16; loop++)
  /* fetch and decompress 16 instructions (one block) */
  {
    /* determine length of high tag by looking at next bit of
       input word */
    tempWord = inputWord << (32-bitPosition);

    /* tempWord is 0 or 1. If 0, length = 3, else length=2. */
    tagHiLength = 2 + (tempWord>>31);
    k = 32 - tagHiLength;
    if (bitPosition > tagHiLength)
      {
        /* if all the bits for the high tag are available in this
           input word, grab them and readjust bitPosition */
        tempWord = tempWord >> k;

        tagHi = tempWord;
        bitPosition -= tagHiLength;
      }
    else
      {
        /* otherwise, get the rest of the bits in this input word */
        tempWord = tempWord >> k;

        tagHi = tempWord;

        /* get the rest of the bits for the high tag from the next
           word */
        addr += 4;
        mem_load_4B(addr,inputWord);
        bitPosition += k;
         if (bitPosition < 32)
           {
    tempWord = inputWord >> bitPosition;
    tagHi |= tempWord;
          }
      }

    /* repeat above for low tag */
    tempWord = inputWord << (32-bitPosition);

    /* tempWord is 0 or 1. If 0, length = 3, else length=2. */
    tagLoLength = 2 + (tempWord>>31);
    k = 32-tagLoLength;
    if (bitPosition > tagLoLength)
```

```
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;
    bitPosition -= tagLoLength;
  }
else
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;

    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagLo |= tempWord;
       }
  }

/* determine length of indexes according to the tag values */
mem_load_1B_RR(indexHiLookupAddr,tagHi,indexHiLength);
mem_load_1B_RR(indexLoLookupAddr,tagLo,indexLoLength);

/* get high index value from memory */
tempWord = inputWord << (32-bitPosition);
k = 32 - indexHiLength;
if (bitPosition > indexHiLength)
  {
    /* if the rest of the bits for the high index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexHi = tempWord;
    bitPosition -= indexHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexHi = tempWord;

    /* get the rest of the bits for the high index from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexHi |= tempWord;
       }
  }

/* repeat above for low index value */
tempWord = inputWord << (32-bitPosition);
k = 32-indexLoLength;
if (bitPosition > indexLoLength)
  {
```

```
    /* if the rest of the bits for the low index are available
       in this input word, grab them */
    tempWord = tempWord >> k;

    indexLo = tempWord;
    bitPosition -= indexLoLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexLo = tempWord;

    /* get the rest of the bits for the low index from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexLo |= tempWord;
       }
  }

/* now we have the indexes into the decode lookup table for
   this instruction.  look up the high and low half-words and
   store into temps.  each entry in the dictionary is 2 bytes
   long.  the dictionary for the high index starts at
   dict_base.  the dictionary for the low index starts at
   dict_base + 512. */

if (tagHi != 7)
  {
    unsigned temp;
    mem_load_2B_RR(ihalfHiLookupAddr,tagHi<<1,temp);
    temp += indexHi;
    mem_load_2B_RR(dict_base,2*temp,ihalfHi16);
  }
else
  {
    ihalfHi16 = indexHi;
  }

if ((tagLo != 7) && (tagLo != 0))
  {
    unsigned temp;
    mem_load_2B_RR(ihalfLoLookupAddr,tagLo<<1,temp);
    temp += indexLo;
    mem_load_2B_RR(dict_base,2*temp,ihalfLo16);
  }
else if (tagLo == 0)
  {
    ihalfLo16 = 0;
  }
else
  {
    ihalfLo16 = indexLo;
  }
```

```
        /* put together the whole 32-bit instruction and put it into
           temporary storage */
        iword32 = (ihalfHi16 << 16) | ihalfLo16;
        swic(compressedBlockAddr,0,iword32);
        compressedBlockAddr+=4;
        /* update the cache */
        mem_store_4B(cacheDataAddr,0,iword32);
        cacheDataAddr+=4;
      }

  }

  goto done;


 copy:
  /* block is not compressed, we can just copy it directly from
     memory with no decoding */
  {
    unsigned char temp;
    unsigned long addr = blockAddr;

    for (loop = 0; loop < 16 ; loop++)
      {
        unsigned long iword=0;
        /* fetch 16 instructions (one block, 64 bytes) and put them
           into temporary storage */
        for (loopByte = 0; loopByte < 4; loopByte++)
          {
            mem_load_1B(addr,0,temp);
            addr++;
            ((unsigned char *)&iword)[loopByte] = temp;
          }
        swic(compressedBlockAddr,0,iword);
        compressedBlockAddr+=4;
        /* update the cache */
        mem_store_4B(cacheDataAddr,0,iword);
        cacheDataAddr+=4;
      }

  }

 done:
  isync;
  iret;

}
```

A.9 CodePack Memo-IL

```
// CodePack Exception Handler (IL)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//   C0[MD_REG_C0_LAT]: Location of index table.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with 2 lines.
//
// Memoization
//   Store all decompressed instructions in a software cache.  Look in
//   this cache first for decompressed instructions and copy them into
//   I-cache with swic.  If the instructions are not found in the
//   software cache, then do decompression as normal and put a copy
//   into the software cache.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   imap:       load I-cache line from main memory
//   iunmap:     store I-cache line to main memory
//   mfc0: move from coprocessor-0 register to general purpose register
//   sync:       memory synchronization
//   isync:      instruction synchronization


#include “eh.h”

unsigned char const indexHiLookup[] = {3, 5, 0, 0, 6, 7, 8, 16};
unsigned char const indexLoLookup[] = {0, 4, 0, 0, 5, 7, 8, 16};
unsigned short const ihalfHiLookup[] = {0x008, 0x020, 0, 0, 0x040, 0x080, 0x100,
0};
unsigned short const ihalfLoLookup[] = {0, 512+0x010, 0, 0, 512+0x020, 512+0x080,
512+0x100, 0};

void eh();

void eh()/* block address to access */
{

  unsigned long ihalfHiLookupAddr;
  unsigned long ihalfLoLookupAddr;
  unsigned long indexHiLookupAddr;
  unsigned long indexLoLookupAddr;

  unsigned long loop;
  unsigned long loopByte;

  unsigned long insnNumber;
```

```
unsigned long groupNumber;
unsigned long blockNumber;
unsigned long whichBlock;

unsigned long indexTableEntry;

unsigned long groupOffset;
unsigned long secondBlockOffset;
unsigned long groupAddr;
unsigned long blockAddr;
unsigned long iword32;

unsigned long baddr;
unsigned long compressedBlockAddr;

unsigned long text_base;
unsigned long dict_base;
unsigned long lat_base;
unsigned long indices_base;

unsigned long cacheTagOffset;
unsigned long cacheTag;
unsigned long cacheDataOffset;
unsigned long cacheDataAddr;

unsigned long cacheDataBase;
unsigned long cacheTagBase;

unsigned long swicAddr;


// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;

// remove bottom portion, get line address from word address;
baddr = (baddr >> 5) << 5;
compressedBlockAddr = (baddr >> 6) << 6;

// Check SW cache. 16 KB cache.
//
// Cache Data is located at address 0x00200000
// Cache Tags are located at address 0x00210000
cacheDataBase = (0x20) << 16;
cacheTagBase = (0x21) << 16;
/* Note: Use 21 for 1-line granularity. Use 22 for 2-line granularity*/
cacheTagOffset = ((compressedBlockAddr) << 18) >> 22;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (compressedBlockAddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

if (cacheTag == compressedBlockAddr)
  {
    // hit
    // copy 16 instruction from software cache into I-cache

    /* ASSUME: if the first line is here, then the second line is
       too since we always decompress 2 lines at a time.  So we
       really only need half the tag store! Therefore, we don't need
       to check for a hit in the second line */
```

```
      imap(compressedBlockAddr,0,cacheDataAddr);
      imap(compressedBlockAddr,32,cacheDataAddr+32);
      sync;

      goto done;
    }

  // Miss SW cache

  mfc0(text_base,MD_REG_C0_TEXT);  // where compressed region starts
  mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
  mfc0(lat_base,MD_REG_C0_LAT); // where line address table starts
  mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

  swicAddr = compressedBlockAddr; // SWIC insn uses this as current iword pointer
to write.

  /* update tag */
  mem_store_4B_RR(cacheTagBase,cacheTagOffset,compressedBlockAddr);

  insnNumber = (baddr - text_base) >> 2; /* (b-ltb/4) 4=sizeof SS32 insn */

  /* Determine what compression group and block the requested
     instruction is in.  32 instructions per compression group, 16
     instructions per compression block, 2 compression blocks per
     group.  */
  groupNumber = insnNumber >> 5; /* / 32; */
  blockNumber = insnNumber >> 4; /* / 16; */

  /* Determine whether the requested address is in the first or second
     compression block in the group.  */
  whichBlock = blockNumber % 2;

  /* Calculate address of index table entry.  Each index table entry
     is 4 bytes long and contains info for one compression group */
  /* fetch index table entry from memory */
  mem_load_4B_RR(lat_base,groupNumber*4,indexTableEntry);

  /* offset of compression group is upper 26 bits of index table entry
     shifted right */
  groupOffset = indexTableEntry & ~0x3f;
  groupOffset = groupOffset >> 6;

  /* offset of second block in compression group is lower six bits of
     index table entry */
  secondBlockOffset = indexTableEntry & 0x3f;

  /* address of compression group is base address of text plus group
     offset */
  groupAddr = indices_base + groupOffset;

  /* check whether first or second block in group is being accessed */
  if (whichBlock == 0) /* first block */
    {
      /* block address is same as group address */
      blockAddr = groupAddr;
      if (secondBlockOffset <= 1)
        {
```

```
        /* if block offset of index table entry is 0 or 1, first block
           is not compressed */
        goto copy;
      }
    }
  else /* second block */
    {
      if (secondBlockOffset <= 1)
        {
          /* if block offset of index table entry is 0 or 1, second block
             address is address of first block + 64 */
          blockAddr = groupAddr + 64;
          if (secondBlockOffset == 0)
            {
              /* if block offset of index table entry is 0, second block
           is not compressed */
              goto copy;
            }
        }
      else
        {
          /* if block offset of index table entry is greater than 1,
             second block address is address of first block plus block
             offset (lower 6 bits of index table entry) */
          blockAddr = groupAddr + secondBlockOffset;
        }
    } /* second block */

    ihalfHiLookupAddr = (long) &ihalfHiLookup;
    ihalfLoLookupAddr = (long) &ihalfLoLookup;
    indexHiLookupAddr = (long) &indexHiLookup;
    indexLoLookupAddr = (long) &indexLoLookup;


  /* decompress 2 cache lines */
  {
    unsigned long tagHi, tagLo;
    unsigned long tagHiLength, tagLoLength;
    unsigned long indexHi, indexLo;
    unsigned long indexHiLength, indexLoLength;
    unsigned short ihalfHi16, ihalfLo16;

    unsigned long inputWord;
    unsigned long tempWord;
    unsigned long bitPosition = 32 - (8 * (blockAddr & 0x03));
    unsigned long addr = blockAddr & ~0x03;
    unsigned k;

    /* fetch first word from memory into inputWord - use “big-endian” word load
*/
    mem_load_4B(addr,inputWord);

    for (loop = 0; loop < 16; loop++)
      /* fetch and decompress 16 instructions (one block) */
      {
        /* determine length of high tag by looking at next bit of
           input word */
        tempWord = inputWord << (32-bitPosition);
```

```
/* tempWord is 0 or 1. If 0, length = 3, else length=2. */
tagHiLength = 2 + (tempWord>>31);
k = 32 - tagHiLength;
if (bitPosition > tagHiLength)
  {
    /* if all the bits for the high tag are available in this
       input word, grab them and readjust bitPosition */
    tempWord = tempWord >> k;

    tagHi = tempWord;
    bitPosition -= tagHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    tagHi = tempWord;

    /* get the rest of the bits for the high tag from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagHi |= tempWord;
       }
  }

/* repeat above for low tag */
tempWord = inputWord << (32-bitPosition);

/* tempWord is 0 or 1. If 0, length = 3, else length=2. */
tagLoLength = 2 + (tempWord>>31);
k = 32-tagLoLength;
if (bitPosition > tagLoLength)
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;
    bitPosition -= tagLoLength;
  }
else
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;

    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagLo |= tempWord;
       }
  }

/* determine length of indexes according to the tag values */
mem_load_1B_RR(indexHiLookupAddr,tagHi,indexHiLength);
```

```
mem_load_1B_RR(indexLoLookupAddr,tagLo,indexLoLength);

/* get high index value from memory */
tempWord = inputWord << (32-bitPosition);
k = 32 - indexHiLength;
if (bitPosition > indexHiLength)
  {
    /* if the rest of the bits for the high index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexHi = tempWord;
    bitPosition -= indexHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexHi = tempWord;

    /* get the rest of the bits for the high index from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexHi |= tempWord;
       }
  }

/* repeat above for low index value */
tempWord = inputWord << (32-bitPosition);
k = 32-indexLoLength;
if (bitPosition > indexLoLength)
  {
    /* if the rest of the bits for the low index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexLo = tempWord;
    bitPosition -= indexLoLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexLo = tempWord;

    /* get the rest of the bits for the low index from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexLo |= tempWord;
```

```
            }
          }

        /* now we have the indexes into the decode lookup table for
           this instruction.  look up the high and low half-words and
           store into temps.  each entry in the dictionary is 2 bytes
           long.  the dictionary for the high index starts at
           dict_base.  the dictionary for the low index starts at
           dict_base + 512. */

        if (tagHi != 7)
          {
            unsigned temp;
            mem_load_2B_RR(ihalfHiLookupAddr,tagHi<<1,temp);
            temp += indexHi;
            mem_load_2B_RR(dict_base,2*temp,ihalfHi16);
          }
        else
          {
            ihalfHi16 = indexHi;
          }

        if ((tagLo != 7) && (tagLo != 0))
          {
            unsigned temp;
            mem_load_2B_RR(ihalfLoLookupAddr,tagLo<<1,temp);
            temp += indexLo;
            mem_load_2B_RR(dict_base,2*temp,ihalfLo16);
          }
        else if (tagLo == 0)
          {
            ihalfLo16 = 0;
          }
        else
          {
            ihalfLo16 = indexLo;
          }

        /* put together the whole 32-bit instruction and put it into
           temporary storage */
        iword32 = (ihalfHi16 << 16) | ihalfLo16;
        swic(swicAddr,0,iword32);
        swicAddr+=4;
      }

 }

 goto writeback;


copy:
 /* block is not compressed, we can just copy it directly from memory
    with no decoding */
 {
   unsigned char temp;
   unsigned long addr = blockAddr;

   for (loop = 0; loop < 16 ; loop++)
     {
```

```
        unsigned long iword=0;
        /* fetch 16 instructions (one block, 64 bytes) and put them
           into temporary storage */
        for (loopByte = 0; loopByte < 4; loopByte++)
          {
            mem_load_1B(addr,0,temp);
            addr++;
            ((unsigned char *)&iword)[loopByte] = temp;
          }
        swic(swicAddr,0,iword);
        swicAddr+=4;
       }

  }

 writeback:
  // hardback to SW cache the two lines just filled in I-cache.
  iunmap(compressedBlockAddr,0,cacheDataAddr);
  iunmap(compressedBlockAddr,32,cacheDataAddr+32);

 done:
  isync;
  iret;

}
```

A.10 CodePack Memo-EW

```
// CodePack Exception Handler (EW)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//   C0[MD_REG_C0_LAT]: Location of index table.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with 2 lines.
//
// Memoization
//   First check the SW cache for the requested cache lines. If they
//   are there, copy them into the I-cache and return.  Otherwise,
//   begin full decompression.  Before decompression, if the both
//   lines being replaced in the I-cache are in the same compression
//   block, then store them both in the SW cache.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   swicw: index store word in instruction cache
//   lwicw:      index read instruction word from cache
//   igetctrl:   get control bits from I-cache line
//   igettag:    get tag bits from I-cache line
//   mfc0: move from coprocessor-0 register to general purpose register
//   isync:      instruction synchronization


// IMPORTANT: THIS VALUE MUST CHANGE WHEN CACHE ORGANIZATION CHANGES
//   amount to shift address to get the tag. For 32B lines, this is Log(#sets)+5.
//   For example, 4 KB cache with assoc=2,line=32B has 64 sets -->
ICACHE_TAG_SHIFT=11
#ifndef ICACHE_TAG_SHIFT
#error Must define ICACHE_TAG_SHIFT
#endif

#include “eh.h”

unsigned char const indexHiLookup[] = {3, 5, 0, 0, 6, 7, 8, 16};
unsigned char const indexLoLookup[] = {0, 4, 0, 0, 5, 7, 8, 16};
unsigned short const ihalfHiLookup[] = {0x008, 0x020, 0, 0, 0x040, 0x080, 0x100,
0};
unsigned short const ihalfLoLookup[] = {0, 512+0x010, 0, 0, 512+0x020, 512+0x080,
512+0x100, 0};

void eh();

void eh()/* block address to access */
{

  unsigned long ihalfHiLookupAddr;
```

```
unsigned long ihalfLoLookupAddr;
unsigned long indexHiLookupAddr;
unsigned long indexLoLookupAddr;

unsigned long loop;
unsigned long loopByte;

unsigned long insnNumber;

unsigned long groupNumber;
unsigned long blockNumber;
unsigned long whichBlock;

unsigned long indexTableEntry;

unsigned long groupOffset;
unsigned long secondBlockOffset;
unsigned long groupAddr;
unsigned long blockAddr;
unsigned long iword32;

unsigned long baddr;
unsigned long compressedBlockAddr;

unsigned long text_base;
unsigned long dict_base;
unsigned long lat_base;
unsigned long indices_base;

unsigned long cacheTagOffset;
unsigned long cacheTag;
unsigned long cacheDataOffset;
unsigned long cacheDataAddr;

unsigned long cacheDataBase;
unsigned long cacheTagBase;

unsigned long currentTag;
unsigned long currentTag2;
unsigned long ctrlbits;
unsigned long replAddr;
unsigned long compressedBlockAddrTag;

unsigned long currentAddr;
unsigned long replCacheDataAddr;
unsigned long replCacheTagOffset;

// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;

// remove bottom portion, get line address from word address;
baddr = (baddr >> 5) << 5;
compressedBlockAddr = (baddr >> 6) << 6;

// For example, tag has bottom 11 bits removed in 4 KB 2-way 32B-line cache
compressedBlockAddrTag = compressedBlockAddr>>ICACHE_TAG_SHIFT;

// Check SW cache. 16 KB cache.
//
```

```
// Cache Data is located at address 0x00200000
// Cache Tags are located at address 0x00210000
cacheDataBase = (0x20) << 16;
cacheTagBase = (0x21) << 16;
cacheTagOffset = ((compressedBlockAddr) << 18) >> 22;
mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
cacheDataOffset = (compressedBlockAddr << 18) >> 18;
cacheDataAddr = cacheDataBase + cacheDataOffset;

/* Find replacement cache line */
igetctrl(compressedBlockAddr,0,ctrlbits);
/* if lru 0, use this line, else next way */
replAddr = compressedBlockAddr;
replAddr |= (ctrlbits >> 2); /* ctrlbits >> 2 == lru */

if (cacheTag == compressedBlockAddrTag)
  {
    // hit
    // copy 2 cache lines (16 instructions) into I-cache

    /* ASSUME: if the first line is here, then the second line is
       too since we always decompress 2 lines at a time.  So we
       really only need half the tag store! Therefore, we don't need
       to check for a hit in the second line */

    mem_load_4B(cacheDataAddr,0,iword32);
    swicw(replAddr,0,iword32);
    mem_load_4B(cacheDataAddr,4,iword32);
    swicw(replAddr,4,iword32);
    mem_load_4B(cacheDataAddr,8,iword32);
    swicw(replAddr,8,iword32);
    mem_load_4B(cacheDataAddr,12,iword32);
    swicw(replAddr,12,iword32);
    mem_load_4B(cacheDataAddr,16,iword32);
    swicw(replAddr,16,iword32);
    mem_load_4B(cacheDataAddr,20,iword32);
    swicw(replAddr,20,iword32);
    mem_load_4B(cacheDataAddr,24,iword32);
    swicw(replAddr,24,iword32);
    mem_load_4B(cacheDataAddr,28,iword32);
    swicw(replAddr,28,iword32);
    mem_load_4B(cacheDataAddr,32,iword32);
    swicw(replAddr,32,iword32);
    mem_load_4B(cacheDataAddr,36,iword32);
    swicw(replAddr,36,iword32);
    mem_load_4B(cacheDataAddr,40,iword32);
    swicw(replAddr,40,iword32);
    mem_load_4B(cacheDataAddr,44,iword32);
    swicw(replAddr,44,iword32);
    mem_load_4B(cacheDataAddr,48,iword32);
    swicw(replAddr,48,iword32);
    mem_load_4B(cacheDataAddr,52,iword32);
    swicw(replAddr,52,iword32);
    mem_load_4B(cacheDataAddr,56,iword32);
    swicw(replAddr,56,iword32);
    mem_load_4B(cacheDataAddr,60,iword32);
    swicw(replAddr,60,iword32);

    goto done;
```

```
    }

  // Miss SW cache

  // Write replaced I-cache line to SW cache

  igettag(replAddr,0,currentTag);
  igettag(replAddr,32,currentTag2);

  if (currentTag == currentTag2)  /* writeback decompressed code if we still have
both lines */
    {
      /*Find address*/
      currentAddr = compressedBlockAddr;
      /* Strip top bits to get byte index into cache */
      currentAddr &= ((1<<ICACHE_TAG_SHIFT) - 1);
      /* add tag bits on top */
      currentAddr |= (currentTag << ICACHE_TAG_SHIFT);

      /* find address to store TAG. Just like for faulting address */
      replCacheTagOffset = (currentAddr << 18) >> 22; /* strip top bits. DRAM
PARAMETER. */
      /* push tag. Only need to push 1 tag because both lines are loaded and
stored together */
      mem_store_4B_RR(cacheTagBase,replCacheTagOffset,currentTag);
      /* push current cache contents to memory.  Replace same way for
         both lines */

      // Find address in SW cache to store instructions.
      replCacheDataAddr = (currentAddr << 18) >> 18;
      replCacheDataAddr += cacheDataBase;

      lwicw(replAddr,0,iword32);
      mem_store_4B(replCacheDataAddr,0,iword32);
      lwicw(replAddr,4,iword32);
      mem_store_4B(replCacheDataAddr,4,iword32);
      lwicw(replAddr,8,iword32);
      mem_store_4B(replCacheDataAddr,8,iword32);
      lwicw(replAddr,12,iword32);
      mem_store_4B(replCacheDataAddr,12,iword32);
      lwicw(replAddr,16,iword32);
      mem_store_4B(replCacheDataAddr,16,iword32);
      lwicw(replAddr,20,iword32);
      mem_store_4B(replCacheDataAddr,20,iword32);
      lwicw(replAddr,24,iword32);
      mem_store_4B(replCacheDataAddr,24,iword32);
      lwicw(replAddr,28,iword32);
      mem_store_4B(replCacheDataAddr,28,iword32);
      lwicw(replAddr,32,iword32);
      mem_store_4B(replCacheDataAddr,32,iword32);
      lwicw(replAddr,36,iword32);
      mem_store_4B(replCacheDataAddr,36,iword32);
      lwicw(replAddr,40,iword32);
      mem_store_4B(replCacheDataAddr,40,iword32);
      lwicw(replAddr,44,iword32);
      mem_store_4B(replCacheDataAddr,44,iword32);
      lwicw(replAddr,48,iword32);
      mem_store_4B(replCacheDataAddr,48,iword32);
      lwicw(replAddr,52,iword32);
```

```
    mem_store_4B(replCacheDataAddr,52,iword32);
    lwicw(replAddr,56,iword32);
    mem_store_4B(replCacheDataAddr,56,iword32);
    lwicw(replAddr,60,iword32);
    mem_store_4B(replCacheDataAddr,60,iword32);
  }

mfc0(text_base,MD_REG_C0_TEXT);  // where compressed region starts
mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
mfc0(lat_base,MD_REG_C0_LAT); // where line address table starts
mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

/* miss SW cache */

insnNumber = (baddr - text_base) >> 2; /* (b-ltb/4) 4=sizeof SS32 insn */

/* Determine what compression group and block the requested
   instruction is in.  32 instructions per compression group, 16
   instructions per compression block, 2 compression blocks per
   group.  */
groupNumber = insnNumber >> 5; /* / 32; */
blockNumber = insnNumber >> 4; /* / 16; */

/* Determine whether the requested address is in the first or second
   compression block in the group.  */
whichBlock = blockNumber % 2;

/* Calculate address of index table entry.  Each index table entry
   is 4 bytes long and contains info for one compression group */
/* fetch index table entry from memory */
mem_load_4B_RR(lat_base,groupNumber*4,indexTableEntry);

/* offset of compression group is upper 26 bits of index table entry
   shifted right */
groupOffset = indexTableEntry & ~0x3f;
groupOffset = groupOffset >> 6;

/* offset of second block in compression group is lower six bits of
   index table entry */
secondBlockOffset = indexTableEntry & 0x3f;

/* address of compression group is base address of text plus group
   offset */
groupAddr = indices_base + groupOffset;

/* check whether first or second block in group is being accessed */
if (whichBlock == 0) /* first block */
  {
    /* block address is same as group address */
    blockAddr = groupAddr;
    if (secondBlockOffset <= 1)
      {
        /* if block offset of index table entry is 0 or 1, first block
           is not compressed */
        goto copy;
      }
  }
else /* second block */
  {
```

```
      if (secondBlockOffset <= 1)
        {
          /* if block offset of index table entry is 0 or 1, second block
             address is address of first block + 64 */
          blockAddr = groupAddr + 64;
          if (secondBlockOffset == 0)
            {
              /* if block offset of index table entry is 0, second block
           is not compressed */
               goto copy;
            }
        }
      else
        {
          /* if block offset of index table entry is greater than 1,
             second block address is address of first block plus block
             offset (lower 6 bits of index table entry) */
          blockAddr = groupAddr + secondBlockOffset;
        }
    } /* second block */

    ihalfHiLookupAddr = (long) &ihalfHiLookup;
    ihalfLoLookupAddr = (long) &ihalfLoLookup;
    indexHiLookupAddr = (long) &indexHiLookup;
    indexLoLookupAddr = (long) &indexLoLookup;


  /* decompress 2 cache lines */
  {
    unsigned long tagHi, tagLo;
    unsigned long tagHiLength, tagLoLength;
    unsigned long indexHi, indexLo;
    unsigned long indexHiLength, indexLoLength;
    unsigned short ihalfHi16, ihalfLo16;

    unsigned long inputWord;
    unsigned long tempWord;
    unsigned long bitPosition = 32 - (8 * (blockAddr & 0x03));
    unsigned long addr = blockAddr & ~0x03;
    unsigned k;

    /* fetch first word from memory into inputWord - use “big-endian” word load
*/
    mem_load_4B(addr,inputWord);

    for (loop = 0; loop < 16; loop++)
      /* fetch and decompress 16 instructions (one block) */
      {
        /* determine length of high tag by looking at next bit of
           input word */
        tempWord = inputWord << (32-bitPosition);

        /* tempWord is 0 or 1. If 0, length = 3, else length=2. */
        tagHiLength = 2 + (tempWord>>31);
        k = 32 - tagHiLength;
        if (bitPosition > tagHiLength)
          {
            /* if all the bits for the high tag are available in this
               input word, grab them and readjust bitPosition */
```

```
    tempWord = tempWord >> k;

    tagHi = tempWord;
    bitPosition -= tagHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    tagHi = tempWord;

    /* get the rest of the bits for the high tag from the next
       word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagHi |= tempWord;
      }
  }

/* repeat above for low tag */
tempWord = inputWord << (32-bitPosition);

/* tempWord is 0 or 1. If 0, length = 3, else length=2. */
tagLoLength = 2 + (tempWord>>31);
k = 32-tagLoLength;
if (bitPosition > tagLoLength)
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;
    bitPosition -= tagLoLength;
  }
else
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;

    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagLo |= tempWord;
      }
  }

/* determine length of indexes according to the tag values */
mem_load_1B_RR(indexHiLookupAddr,tagHi,indexHiLength);
mem_load_1B_RR(indexLoLookupAddr,tagLo,indexLoLength);

/* get high index value from memory */
tempWord = inputWord << (32-bitPosition);
k = 32 - indexHiLength;
if (bitPosition > indexHiLength)
```

```
  {
    /* if the rest of the bits for the high index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexHi = tempWord;
    bitPosition -= indexHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexHi = tempWord;

    /* get the rest of the bits for the high index from the
       next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexHi |= tempWord;
      }
  }

/* repeat above for low index value */
tempWord = inputWord << (32-bitPosition);
k = 32-indexLoLength;
if (bitPosition > indexLoLength)
  {
    /* if the rest of the bits for the low index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexLo = tempWord;
    bitPosition -= indexLoLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexLo = tempWord;

    /* get the rest of the bits for the low index from the
       next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexLo |= tempWord;
      }
  }

/* now we have the indexes into the decode lookup table for
```

```
         this instruction.  look up the high and low half-words and
         store into temps.  each entry in the dictionary is 2 bytes
         long.  the dictionary for the high index starts at
         dict_base.  the dictionary for the low index starts at
         dict_base + 512. */

      if (tagHi != 7)
        {
          unsigned temp;
          mem_load_2B_RR(ihalfHiLookupAddr,tagHi<<1,temp);
          temp += indexHi;
          mem_load_2B_RR(dict_base,2*temp,ihalfHi16);
        }
      else
        {
          ihalfHi16 = indexHi;
        }

      if ((tagLo != 7) && (tagLo != 0))
        {
          unsigned temp;
          mem_load_2B_RR(ihalfLoLookupAddr,tagLo<<1,temp);
          temp += indexLo;
          mem_load_2B_RR(dict_base,2*temp,ihalfLo16);
        }
      else if (tagLo == 0)
        {
          ihalfLo16 = 0;
        }
      else
        {
          ihalfLo16 = indexLo;
        }

      /* put together the whole 32-bit instruction and put it into
         temporary storage */
      iword32 = (ihalfHi16 << 16) | ihalfLo16;
      swic(compressedBlockAddr,0,iword32);
      compressedBlockAddr+=4;
     }

 }

 goto done;


copy:
 /* block is not compressed, we can just copy it directly from
    memory with no decoding */
 {
   unsigned char temp;
   unsigned long addr = blockAddr;

   for (loop = 0; loop < 16 ; loop++)
     {
       unsigned long iword=0;
       /* fetch 16 instructions (one block, 64 bytes) and put them
          into temporary storage */
       for (loopByte = 0; loopByte < 4; loopByte++)
```

```
          {
            mem_load_1B(addr,0,temp);
            addr++;
            ((unsigned char *)&iword)[loopByte] = temp;
          }
        swic(compressedBlockAddr,0,iword);
        compressedBlockAddr+=4;
      }

  }

 done:
  isync;
  iret;

}
```

A.11 CodePack Memo-EL

```
// CodePack Exception Handler (EL)
//
// Inputs
//   C0[MD_REG_CO_INDICES]: address base of indices
//   C0[MD_REG_C0_DICTIONARY]: address base of dictionary
//   C0[MD_REG_C0_TEXT]: address of .text segment
//   C0[MD_REG_C0_BADVA]: This is the location to load the I-cache.
//   C0[MD_REG_C0_LAT]: Location of index table.
//
// Assumptions:
//   I-cache is 2-way associative with 8 32-bit instructions in each line
//
// Output
//   Load I-cache at address C0[MD_REG_C0_BADVA] with 2 lines.
//
// Memoization
//   First check the SW cache for the requested cache lines. If they
//   are there, copy them into the I-cache and return.  Otherwise,
//   begin full decompression.  Before decompression, if the both
//   lines being replaced in the I-cache are in the same compression
//   block, then store them both in the SW cache.
//
// Special SimpleScalar instructions
//   swic: store word in instruction cache
//   imapw: index read I-cache line from main memory
//   iunmapw: index write I-cache line to main memory
//   igetctrl:   get control bits from I-cache line
//   igettag:    get tag bits from I-cache line
//   mfc0: move from coprocessor-0 register to general purpose register
//   isync:      instruction synchronization
//   sync:       memory synchronization

#include “eh.h”

// IMPORTANT: THIS VALUE MUST CHANGE WHEN CACHE ORGANIZATION CHANGES
//   amount to shift address to get the tag. For 32B lines, this is Log(#sets)+5.
//   For example, 4 KB cache with assoc=2,line=32B has 64 sets -->
ICACHE_TAG_SHIFT=11
#ifndef ICACHE_TAG_SHIFT
#error Must define ICACHE_TAG_SHIFT
#endif

unsigned char const indexHiLookup[] = {3, 5, 0, 0, 6, 7, 8, 16};
unsigned char const indexLoLookup[] = {0, 4, 0, 0, 5, 7, 8, 16};
unsigned short const ihalfHiLookup[] = {0x008, 0x020, 0, 0, 0x040, 0x080, 0x100,
0};
unsigned short const ihalfLoLookup[] = {0, 512+0x010, 0, 0, 512+0x020, 512+0x080,
512+0x100, 0};

void eh();

void eh()/* block address to access */
{

  unsigned long ihalfHiLookupAddr;
```

```
unsigned long ihalfLoLookupAddr;
unsigned long indexHiLookupAddr;
unsigned long indexLoLookupAddr;

unsigned long loop;
unsigned long loopByte;

unsigned long insnNumber;

unsigned long groupNumber;
unsigned long blockNumber;
unsigned long whichBlock;

unsigned long indexTableEntry;

unsigned long groupOffset;
unsigned long secondBlockOffset;
unsigned long groupAddr;
unsigned long blockAddr;
unsigned long iword32;

unsigned long baddr;
unsigned long compressedBlockAddr;

unsigned long text_base;
unsigned long dict_base;
unsigned long lat_base;
unsigned long indices_base;

unsigned long cacheTagOffset;
unsigned long cacheTag;
unsigned long cacheDataOffset;
unsigned long cacheDataAddr;

unsigned long cacheDataBase;
unsigned long cacheTagBase;

unsigned long currentTag;
unsigned long currentTag2;
unsigned long ctrlbits;
unsigned long replAddr;
unsigned long compressedBlockAddrTag;

unsigned long currentAddr;
unsigned long replCacheDataAddr;
unsigned long replCacheTagOffset;

// Get parameters from system coprocessor
mfc0(baddr,MD_REG_C0_BADVA);   // Get missed PC about put in baddr;

// remove bottom portion, get line address from word address;
baddr = (baddr >> 5) << 5;
compressedBlockAddr = (baddr >> 6) << 6;


// For example, tag has bottom 11 bits removed in 4 KB 2-way 32B-line cache
compressedBlockAddrTag = compressedBlockAddr>>ICACHE_TAG_SHIFT;

// Check SW cache. 16 KB cache.
```

```
  //
  // Cache Data is located at address 0x00200000
  // Cache Tags are located at address 0x00210000
  cacheDataBase = (0x20) << 16;
  cacheTagBase = (0x21) << 16;
  cacheTagOffset = ((compressedBlockAddr) << 18) >> 22;
  mem_load_4B_RR(cacheTagBase,cacheTagOffset,cacheTag);
  cacheDataOffset = (compressedBlockAddr << 18) >> 18;
  cacheDataAddr = cacheDataBase + cacheDataOffset;

  /* Find replacement cache line */
  igetctrl(compressedBlockAddr,0,ctrlbits);
  /* if lru 0, use this line, else next way */
  replAddr = compressedBlockAddr;
  replAddr |= (ctrlbits >> 2); /* ctrlbits >> 2 == lru */

  if (cacheTag == compressedBlockAddrTag)
    {
      // hit
      // copy 2 cache lines (16 instructions) into I-cache

      /* ASSUME: if the first line is here, then the second line is
         too since we always decompress 2 lines at a time.  So we
         really only need half the tag store! Therefore, we don't need
         to check for a hit in the second line */

      imapw(replAddr,0,cacheDataAddr);
      imapw(replAddr,32,cacheDataAddr+32);
      sync;

      goto done;
    }

  igettag(replAddr,0,currentTag);
  igettag(replAddr,32,currentTag2);

  if (currentTag == currentTag2)  /* writeback decompressed code if we still have
both lines */
    {
      /*Find address*/
      currentAddr = compressedBlockAddr;
      /* Strip top bits to get byte index into cache */
      currentAddr &= ((1<<ICACHE_TAG_SHIFT) - 1);
      /* add tag bits on top */
      currentAddr |= (currentTag << ICACHE_TAG_SHIFT);

      /* find address to store TAG. Just like for faulting address */
      replCacheTagOffset = (currentAddr << 18) >> 22; /* strip top bits. DRAM
PARAMETER. */
      /* push tag. Only need to push 1 tag because both lines are loaded and
stored as 1 unit */
      mem_store_4B_RR(cacheTagBase,replCacheTagOffset,currentTag);
      /* push current cache contents to memory.  Replace same way for
         both lines */

      // Find address in SW cache to store instructions.
      replCacheDataAddr = (currentAddr << 18) >> 18;
      replCacheDataAddr += cacheDataBase;
```

```
    iunmapw(replAddr,0,replCacheDataAddr);
    iunmapw(replAddr,32,replCacheDataAddr+32);
  }

mfc0(text_base,MD_REG_C0_TEXT);  // where compressed region starts
mfc0(dict_base,MD_REG_C0_DICTIONARY); // where dictionary starts
mfc0(lat_base,MD_REG_C0_LAT); // where line address table starts
mfc0(indices_base,MD_REG_C0_INDICES); // where indices start (the codewords)

/* miss SW cache */

insnNumber = (baddr - text_base) >> 2; /* (b-ltb/4) 4=sizeof SS32 insn */

/* Determine what compression group and block the requested
   instruction is in.  32 instructions per compression group, 16
   instructions per compression block, 2 compression blocks per
   group.  */
groupNumber = insnNumber >> 5; /* / 32; */
blockNumber = insnNumber >> 4; /* / 16; */

/* Determine whether the requested address is in the first or second
   compression block in the group.  */
whichBlock = blockNumber % 2;

/* Calculate address of index table entry.  Each index table entry
   is 4 bytes long and contains info for one compression group */
/* fetch index table entry from memory */
mem_load_4B_RR(lat_base,groupNumber*4,indexTableEntry);

/* offset of compression group is upper 26 bits of index table entry
   shifted right */
groupOffset = indexTableEntry & ~0x3f;
groupOffset = groupOffset >> 6;

/* offset of second block in compression group is lower six bits of
   index table entry */
secondBlockOffset = indexTableEntry & 0x3f;

/* address of compression group is base address of text plus group
   offset */
groupAddr = indices_base + groupOffset;

/* check whether first or second block in group is being accessed */
if (whichBlock == 0) /* first block */
  {
    /* block address is same as group address */
    blockAddr = groupAddr;
    if (secondBlockOffset <= 1)
      {
        /* if block offset of index table entry is 0 or 1, first block
           is not compressed */
        goto copy;
      }
  }
else /* second block */
  {
    if (secondBlockOffset <= 1)
      {
        /* if block offset of index table entry is 0 or 1, second block
```

```
            address is address of first block + 64 */
          blockAddr = groupAddr + 64;
          if (secondBlockOffset == 0)
            {
              /* if block offset of index table entry is 0, second block
         is not compressed */
              goto copy;
            }
        }
      else
        {
          /* if block offset of index table entry is greater than 1,
             second block address is address of first block plus block
             offset (lower 6 bits of index table entry) */
          blockAddr = groupAddr + secondBlockOffset;
        }
    } /* second block */

    ihalfHiLookupAddr = (long) &ihalfHiLookup;
    ihalfLoLookupAddr = (long) &ihalfLoLookup;
    indexHiLookupAddr = (long) &indexHiLookup;
    indexLoLookupAddr = (long) &indexLoLookup;


  /* decompress:*/
  {
    unsigned long tagHi, tagLo;
    unsigned long tagHiLength, tagLoLength;
    unsigned long indexHi, indexLo;
    unsigned long indexHiLength, indexLoLength;
    unsigned short ihalfHi16, ihalfLo16;

    unsigned long inputWord;
    unsigned long tempWord;
    unsigned long bitPosition = 32 - (8 * (blockAddr & 0x03));
    unsigned long addr = blockAddr & ~0x03;
    unsigned k;

    /* fetch first word from memory into inputWord - use “big-endian” word load
*/
    mem_load_4B(addr,inputWord);

    for (loop = 0; loop < 16; loop++)
      /* fetch and decompress 16 instructions (one block) */
      {
        /* determine length of high tag by looking at next bit of
           input word */
        tempWord = inputWord << (32-bitPosition);

        /* tempWord is 0 or 1. If 0, length = 3, else length=2. */
        tagHiLength = 2 + (tempWord>>31);
        k = 32 - tagHiLength;
        if (bitPosition > tagHiLength)
          {
            /* if all the bits for the high tag are available in this
               input word, grab them and readjust bitPosition */
            tempWord = tempWord >> k;

            tagHi = tempWord;
```

```
    bitPosition -= tagHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    tagHi = tempWord;

    /* get the rest of the bits for the high tag from the next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagHi |= tempWord;
       }
  }

/* repeat above for low tag */
tempWord = inputWord << (32-bitPosition);

/* tempWord is 0 or 1. If 0, length = 3, else length=2. */
tagLoLength = 2 + (tempWord>>31);
k = 32-tagLoLength;
if (bitPosition > tagLoLength)
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;
    bitPosition -= tagLoLength;
  }
else
  {
    tempWord = tempWord >> k;
    tagLo = tempWord;

    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
tagLo |= tempWord;
       }
  }

/* determine length of indexes according to the tag values */
mem_load_1B_RR(indexHiLookupAddr,tagHi,indexHiLength);
mem_load_1B_RR(indexLoLookupAddr,tagLo,indexLoLength);

/* get high index value from memory */
tempWord = inputWord << (32-bitPosition);
k = 32 - indexHiLength;
if (bitPosition > indexHiLength)
  {
    /* if the rest of the bits for the high index are
       available in this input word, grab them */
    tempWord = tempWord >> k;
```

```
    indexHi = tempWord;
    bitPosition -= indexHiLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexHi = tempWord;

    /* get the rest of the bits for the high index from the
       next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexHi |= tempWord;
       }
  }

/* repeat above for low index value */
tempWord = inputWord << (32-bitPosition);
k = 32-indexLoLength;
if (bitPosition > indexLoLength)
  {
    /* if the rest of the bits for the low index are
       available in this input word, grab them */
    tempWord = tempWord >> k;

    indexLo = tempWord;
    bitPosition -= indexLoLength;
  }
else
  {
    /* otherwise, get the rest of the bits in this input word */
    tempWord = tempWord >> k;

    indexLo = tempWord;

    /* get the rest of the bits for the low index from the
       next word */
    addr += 4;
    mem_load_4B(addr,inputWord);
    bitPosition += k;
     if (bitPosition < 32)
       {
tempWord = inputWord >> bitPosition;
indexLo |= tempWord;
       }
  }

/* now we have the indexes into the decode lookup table for
   this instruction.  look up the high and low half-words and
   store into temps.  each entry in the dictionary is 2 bytes
   long.  the dictionary for the high index starts at
   dict_base.  the dictionary for the low index starts at
```

```
         dict_base + 512. */

      if (tagHi != 7)
        {
          unsigned temp;
          mem_load_2B_RR(ihalfHiLookupAddr,tagHi<<1,temp);
          temp += indexHi;
          mem_load_2B_RR(dict_base,2*temp,ihalfHi16);
        }
      else
        {
          ihalfHi16 = indexHi;
        }

      if ((tagLo != 7) && (tagLo != 0))
        {
          unsigned temp;
          mem_load_2B_RR(ihalfLoLookupAddr,tagLo<<1,temp);
          temp += indexLo;
          mem_load_2B_RR(dict_base,2*temp,ihalfLo16);
        }
      else if (tagLo == 0)
        {
          ihalfLo16 = 0;
        }
      else
        {
          ihalfLo16 = indexLo;
        }

      /* put together the whole 32-bit instruction and put it into
         temporary storage */
      iword32 = (ihalfHi16 << 16) | ihalfLo16;
      swic(compressedBlockAddr,0,iword32);
      compressedBlockAddr+=4;
     }

 }

 goto done;


copy:
 /* block is not compressed, we can just copy it directly from memory
    with no decoding */
 {
   unsigned char temp;
   unsigned long addr = blockAddr;

   for (loop = 0; loop < 16 ; loop++)
     {
       unsigned long iword=0;
       /* fetch 16 instructions (one block, 64 bytes) and put them
          into temporary storage */
       for (loopByte = 0; loopByte < 4; loopByte++)
         {
           mem_load_1B(addr,0,temp);
           addr++;
           ((unsigned char *)&iword)[loopByte] = temp;
```

```
          }
        swic(compressedBlockAddr,0,iword);
        compressedBlockAddr+=4;
       }

  }

 done:
  isync;
  iret;

}
```

Bibliography

[Araujo98] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code Compression Based on Operand Factorization", *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pp. 194-201, 1998.

[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*, pp. 219-236, September 1997.

[Benes98] M. Benes, S. M. Nowick, and A. Wolfe, "A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded processors", *Proceedings of the IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 43-56, September 1998.

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

[Bunda93] J. Bunda, D. Fussell, R. Jenevein, and W.C. Athas, "16-Bit vs. 32-Bit Instructions for Pipelined Microprocessors", *Proceedings of the 20th Annual International Symposium of Computer Architecture*, pp. 237-246, 1993.

[Burger97] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", *Computer Architecture News* 25(3), pp. 13-25, June 1997.

[Cate91] V. Cate and T. Gross, "Combining the Concepts of Compression and Caching for a Two-Level Filesystem", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 200-209, 1991.

[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.

[Citron95] D. Citron, "Creating a Wider Bus Using Caching Techniques", *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pp. 90-99, January 1995.

[Coffing97] C. Coffing and J. Brown, "A System for Transparent File Compression With Caching Under Linux", unpublished.

[Cooper99] Keith D. Cooper and Nathaniel McIntosh, “Enhanced code compression for embedded RISC processors”, *Proc. Conf. on Programming Languages Design and Implementation*, pp. 139-149, 1999.

[Debray99] S. Debray, W. Evans, and R. Muth, “Compiler Techniques for Code Compression”, *Proc. 2nd Workshop on Compiler Support for Systems Software* (WCSSS-99), May 1999.

[Douglis93] F. Douglis, “The Compression Cache: Using On-line Compression to Extend Physical Memory”, *Proceedings of the Winter 1993 USENIX Conference*, pp. 88-94, 1993.

[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)*, pp. 358-365, June 1997.

[Flynn83] M. J. Flynn and L. W. Hoevel, “Execution Architecture: The DELtran Experiment”, *IEEE Transactions on Computers*, Vol. C-32, No. 2, pp. 156-175, February 1983.

[Franz94] M. Franz, *Code-Generation On-the-Fly: A Key for Portable Software*, Ph.D. 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.

[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.

[Greenhills98] Greenhills Software, *Optimizing Speed vs. Size using The CodeBalance Utility For ARM/THUMB and MIPS16 Architectures*, white paper, 1998.

[Gwennap99] L. Gwennap, “MAJC Gives VLIW a New Twist”, *Microprocessor Report*, 13(12), pp. 13-15:22, Sept. 13, 1999.

[Hooger99] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. Van De Wiel, “A Code Compression System Based on Pipelined Interpreters”, *Softw. Pract. Exper.*, 29(11), pp. 1005-1023, 1999.

[Horowitz98] M. Horowitz, M. Martonosi, T. Mowry, M. Smith, “Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications”, *ACM Transactions on Computer Systems*, 16(2):170-205, May 1998.

[IBM98] IBM, *CodePack PowerPC Code Compression Utility User’s Manual Version 3.0*, IBM, 1998.

[IBM00] IBM, *ASIC SA-27E*, Databook, February 2000.

[Jacob97] B. Jacob and T. Mudge, “Software-Managed Address Translation”, *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pp. 156-167, 1997.

[Jacob99] B. Jacob, “Cache Design for Embedded Real-Time Systems”, *Proceedings of the Embedded Systems Conference*, Summer 1999.

[Jouppi90] N. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364-373, June 1990.

[Kemp98] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach, “A decompression core for PowerPC”, *IBM Journal of Research and Development* 42(6), November 1998.

[Kirovski97] D. Kirovski, J. Kin, and W. H. Mangione-Smith, “Procedure Based Program Compression”, *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp.204-211, December 1997.

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

[Klint81] P. Klint, “Interpretation Techniques”, *Software Practice and Experience*, Vol. 11, No.9, pp. 963-973, September 1981.

[Kozuch94] M. Kozuch and A. Wolfe, “Compression of Embedded System Programs,” *IEEE International Conference on Computer Design*, pp. 270-277, 1994.

[Kunchit99] K. Kunchithapadam and J. Larus, *Using Lightweight Procedures to improve Instruction Cache Performance*, CS-TR-99-1390, University of Wisconsin, 1999.

[Larin99] S. Larin and T. Conte, “Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors”, *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 82-92, November 1999.

[Lee97] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[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*, pp. 194-203, December 1997.

[Lefurgy98] C. Lefurgy and T. Mudge, *Code Compression for DSP*, CSE-TR-380-98, University of Michigan, November 1998.

[[Lekatsas98] H. Lekatsas and W. Wolf, “Code Compression for Embedded Systems”, *Proceedings of the 35th Design Automation Conference*, pp. 516-521, June 1998.

[Lelewer87] D. A. Lelewer and D. S. Hirschberg, “Data Compression”, ACM Computing Surveys, 19(3):261-296, September 1987.

[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.

[Michie68] D. Michie, “*Memo* Functions and Machine Learning”, *Nature*, Vol. 218, pp. 19-22, April 6, 1968.

[MIPS96] MIPS Technologies, Inc., *MIPS R10000 Microprocessor User’s Manual*, Version 2.0, 1996.

[Motorola94] Motorola, *PowerPC Microprocessor Family: The Programming Environments*, Sept. 1994.

[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*, pp. 169-183, October 1996.

[Pettis90] K. Pettis and R. Hansen, “Profile Guided Code Positioning”, *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation (PLDI)*, pp. 16-27, June 1990.

[Pittman87] T. Pittman, “Two-Level Interpreter/Native Code Execution”, *Proceedings of the SIGPLAN’87 Symposium on Interpreters and Interpretive Techniques*, pp. 150-152, 1987.

[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.

[Taunton91] M. Taunton, “Compressed Executables: an Exercise in Thinking Small”, *Proceedings of the Summer 1991 USENIX Conference*, pp. 385-403, 1991.

[Turley95] J. L. Turley, “Thumb squeezes arm code size”, *Microprocessor Report*, 9(4), pp. 1-5, 27 March 1995.

[Williams91] R. Williams, “An Extremely Fast Ziv-Lempel Data Compression Algorithm”, *Data Compression Conference*, pp.362-371, 1991.

[Witten90] Witten, R. Neal, and J. Cleary, “Arithmetic Coding for Data Compression”, *Communications of the ACM*, Vol. 30(6), pp. 520-540, June 1987.

[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.

[Wulf75] W. Wulf, R. Johnsson, C. Weinstock, S. Hobbs, and C. Geschke, *The Design of an Optimizing Compiler*, North Holland, 1975.