Evaluation of a High Performance Code Compression Method

Charles Lefurgy, Eva Piccininni, and Trevor Mudge
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
{lefurgy,epiccini,tnm}@eecs.umich.edu

Abstract

Compressing the instructions of an embedded program is important for cost-sensitive low-power control-oriented embedded computing. A number of compression schemes have been proposed to reduce program size. However, the increased instruction density has an accompanying performance cost because the instructions must be decompressed before execution. In this paper, we investigate the performance penalty of a hardware-managed code compression algorithm recently introduced in IBM's PowerPC 405. This scheme is the first to combine many previously proposed code compression techniques, making it an ideal candidate for study. We find that code compression with appropriate hardware optimizations does not have to incur much performance loss. 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.

1 Introduction

Reducing the required footprint for program memory is increasingly important as system on a chip designs become popular in the embedded world. Code compression is one technique to reduce program size by applying compression algorithms to native instruction sets. There are many recent publications suggesting new compressed code representations [Araujo98, Benes97, Benes98, Bunda92, Ernst97, Fraser95, Kozuch94, Lefurgy97, Lekatsas98, Liao96, Wolfe92]. However, the increased instruction density has an accompanying performance cost because the instructions must be decompressed before execution. Although some work has addressed the issue of performance for decompression, on the whole, it remains much less studied than size optimizations for the final compressed program.

In this paper 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?

2 Background and Related Work

In this section we review the methods of code compression relevant to those currently employed by microprocessor manufacturers to reduce the impact of instruction sets on program size. Since we are considering only the performance of hardware-managed compression, we have not covered the many interesting software-managed compression techniques using compiler optimizations and interpretation [Cooper99, Fraser95, Kirovski97].

The metric for measuring compression is *compression ratio* which is defined by the formula:

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

2.1 Compression for RISC instruction sets

Bunda et al. [Bunda92] 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.

Thumb [ARM95] and MIPS16 [Kissell97] are two examples of such instruction sets. Programs compiled for Thumb achieve 30% smaller code in comparison to the standard ARM instruction set, but run 15%-20% slower on systems with ideal instruction memories (32-bit buses and no wait states) [ARM95]. Code written for 32-bit MIPS-III is typically reduced 40% in size when compiled for MIPS16 [Kissell97].

2.2 CCRP

The Compressed Code RISC Processor (CCRP) [Wolfe92, Kozuch94] has an instruction cache that is modified to run compressed programs. At compile-time the cache line bytes are Huffman encoded. At run-time cache lines are fetched from main memory, decompressed, and put in the instruction cache. Instructions fetched from the cache are not compressed and 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 overall compression ratio of CCRP is 73% for MIPS programs.

2.3 CodePack

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 prepend 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.

3 Compression Method

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].

3.1 CodePack algorithm

Figure 1 illustrates the decompression algorithm. To understand it, consider how the compression encoder works (start at the bottom of Figure 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 codeword. 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 decompres-

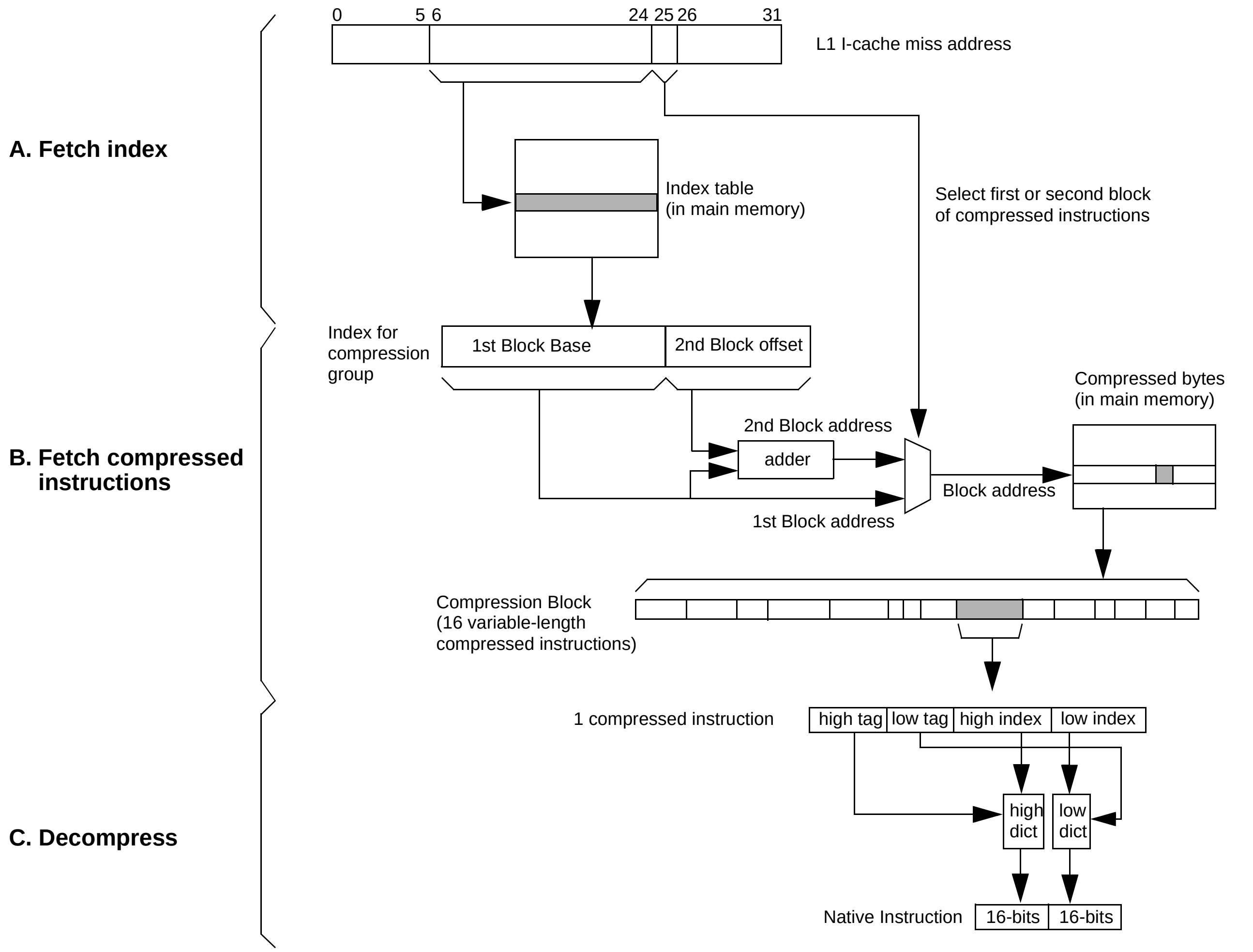

Figure 1: CodePack decompression

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.

sion 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.

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.

Bench	Instructions executed (millions)	Input set	L1 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 1: Benchmarks

Dictionaries. Both dictionaries are kept in a 2KB 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 1 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 8 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 execution.

4 Experimental setup

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 statically linked with library code. Table 1 lists the benchmarks and the input sets. Each benchmark executes over 1 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 2.

Our models for L1-miss activity are illustrated in Figure 2. Figure 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 2-b shows the baseline compression system. This model fetches the index table entry from main memory (unless it re-uses the previous index), uses the index to fetch codewords from main memory, and decompresses codewords as they are received. In the example, the consecutive main memory accesses return 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 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

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, memport:2, fpalu:4, fpmult:1	alu:8, mult:1, memport: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	8KB, 32B lines, 2-assoc, lru	16KB	32KB
L1 d-cache	8KB, 16B lines, 2-assoc, lru	16KB	32KB
memory latency	10 cycle latency, 2 cycle rate	same	same
memory width	64 bits	same	same

Table 2: Simulated architectures

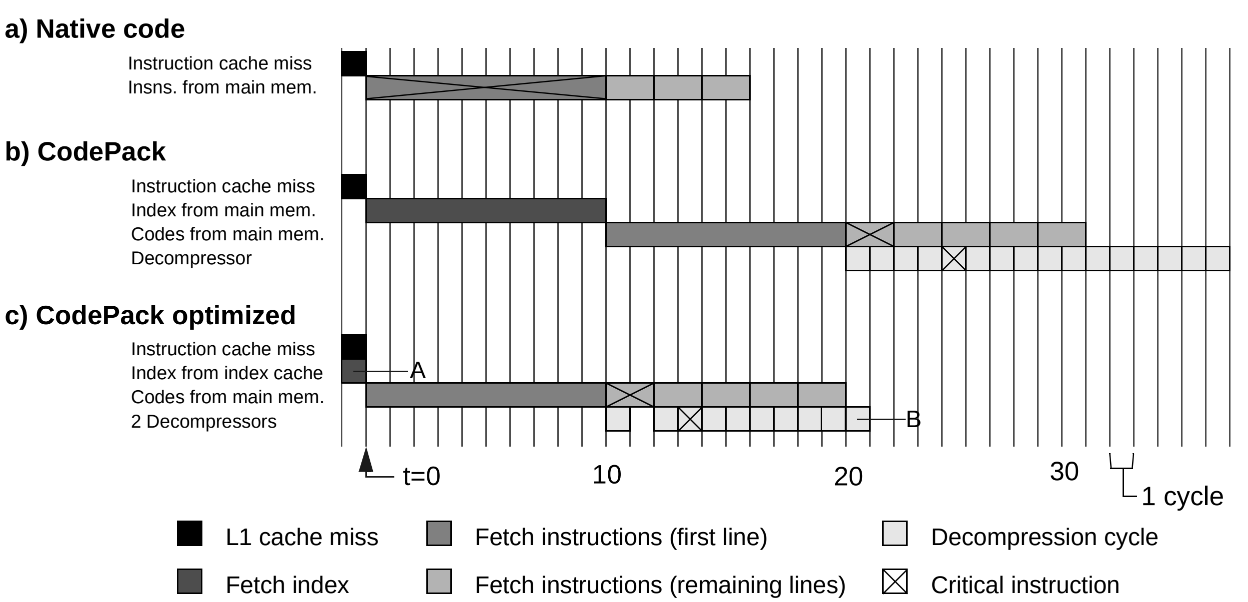

Figure 2: Example of L1 miss activity

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.

instructions/cycle allows the critical instruction to be ready at t=14.

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

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: Composition of compressed region

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 5: Instructions per cycle

Native is the original program. *CodePack* is the baseline decompressor. *Optimized* is our optimized CodePack model with an index cache and additional decompression bandwidth.

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 3: Compression ratio of .text section

memory system parameters to determine the performance trends of the optimizations.

5.1 Code size

Table 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 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 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 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.

5.2 Overall performance

Table 5 shows the overall performance that compression provides compared to native code. We also show an optimized decompressor that provides significant speedup over the baseline decompressor and even out-performs 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 compressed 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.

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 sub-sections, 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 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 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 under 15% for the *cc1* benchmark which has the most I-cache misses. This organization has a miss ratio of under 11% for *vortex* and under 4% for the other benchmarks. This is the cache organization we use in our optimized compression model. The index cache contains 1KB of index entries and 88 bytes of tag storage. This is about one-eighth the size of the

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%	4.56%
256	11.7%	2.7%	0.8%	0.2%

Table 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 7: Speedup due to index cache

Values represent speedup over native programs. The *Index Cache* column represents a fully-associative 64-entry index cache with 4 indices per entry. The *Perfect* column represents an index cache that never misses.

4-issue instruction cache. It is able to map 32KB of the original program into compressed bytes. In Table 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 possible 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.

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 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 2 decompressors.

Bench	4-issue instruction cache size							
	1KB		4KB		16KB		64KB	
	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 10: Variation in speedup due to I-cache size

Values represent speedup over native programs using the same I-cache size. All simulations are based on 4-issue model with different cache sizes. The *16KB* column is the 4-issue baseline model.

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 8: Speedup due to decompression rate

Values represent speedup over native programs.

Bench	4-issue			
	CodePack	Index	Decompress	All
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 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. *All* shows the benefit of both optimizations together. A slight speedup is attained over native code for *go*, *perl*, and *vortex*.

We now combine both of the above optimizations to see how they work together. Table 9 shows the performance of each optimization individually and together. In our optimized model, the index cache optimization improved performance more than using a wider decompressor.

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 I-cache sizes and show the performance in Table 10. The default decompressor has a performance penalty of up to 28% with 1KB caches. However, the optimized decompressor has up to a 61% performance 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 fewer misses with larger caches.

Sensitivity to main memory width. Many embedded systems have narrow buses to main memory. Instruction sets with short instruction formats can out-perform 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 [Bunda92]. This suggests that code compression might offer a benefit in such architectures. Our results in Table 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 rela-

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 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.

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 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.

tively 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.

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 12. As memory latency grows, the optimized decompressor can attain speedups over native code because it uses fewer costly accesses to main memory.

6 Conclusions and Future Work

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 less main memory accesses and the inherent prefetching behavior of the CodePack algorithm often provides a speedup over native instructions. 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 CodePack 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 to resource limited computers.

Acknowledgments

This work was supported by ARPA grant DABT63-97-C-0047 and equipment donated through Intel Corporation's Technology for Education 2000 Program.

References

[Araujo98] G. Araujo, P. Centoducatte, M. Cortes, and Ricardo Pannain, "Code Compression Based on Operand Factorization", *Proc. 31st Ann. International Symp. on Microarchitecture*, 1998.

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

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

[Benes98] M. Benes, S. M. Nowick, and A. Wolfe, "A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded processors", *Proc. IEEE International Symp. on Advanced Research in Asynchronous Circuits and Systems*, 1998.

[Bunda92] J. Bunda, D. Fussell, and W.C. Athas, "16-bit vs. 32-bit Instructions for Pipelined Microprocessors", *Proc. 20th Ann. International Symp. of Computer Architecture*, 1992.

[Burger97] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", *Computer Architecture News* 25(3), June 1997.

[Cooper99] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors", *Proc. Conf. on Programming Languages Design and Implementation*, 1999.

[Ernst97] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression", *Proc. ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, 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, Oct. 1995.

[IBM98] IBM, *CodePack PowerPC Code Compression Utility User's Manual Version 3.0*, IBM, 1998.

[Kemp98] T. Kemp et. al, "A decompression core for PowerPC", *IBM J. Res. Dev.* 42(6), Nov. 1998.

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

[Kirovski97] D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure Based Program Compression", *Proc. 30th Ann. International Symp. on Microarchitecture*, 1997.

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

[Lee97] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the 30th Ann. International Symp. on Microarchitecture*, 1997.

[Lefurgy97] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques", *Proc. 30th Ann. International Symp. on Microarchitecture*, 1997.

[Lekatsas98] H. Lekatsas and W. Wolf, "Code Compression for Embedded Systems", *Proc. 35th Design Automation Conf.*, 1998.

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

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

[Wolfe92] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proc. 25th Ann. International Symp. on Microarchitecture*, 1992.