— Extended Abstract —

Fast Software-managed Code Decompression

Charles Lefurgy and Trevor Mudge
{lefurgy,tnm}@eecs.umich.edu
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
http://www.eecs.umich.edu/~tnm/compress

Abstract

Compressing the instructions of an embedded program is important for cost-sensitive low-power control-oriented embedded computing. Several hardware decompression architectures have been proposed. In this paper, we present a method of decompressing programs using software. One interesting aspect of our method is that it relies on using a software-managed instruction cache under control of the decompressor. In addition, our solution is an order of magnitude faster than a previous software-managed decompression system.

1 Introduction

Many recent code compression studies have suggested that custom on-chip hardware be used to decompress programs. In this paper, we explore doing decompression in software. Software decompression schemes are interesting because they reduce hardware complexity and allow greater choice of compression algorithms late in the product design cycle. Software-managed decompression allows separate programs to use entirely different compression methods. Newly developed compression methods are not constrained to use old decompression hardware. In addition, decompressors can be cheaply implemented on a wide variety of architectures and instruction sets with little effort. The primary challenge in software decompression is to minimize the increased execution time due to running the decompression software. Our technique is an order of magnitude faster than a previously proposed system.

The organization of this paper is as follows. Section 2 reviews previous work in code compression. We present our compression method in section 3. Our simulation environment is presented in section 4. In section 5, we discuss our experimental results. Finally, section 6 contains our conclusions.

2 Previous work

There have been many recent publications about code compression. The Compressed Code RISC Processor [Wolfe92, Kozuch94, Benes98] is a MIPS processor that decompresses instruction cache lines which have been Huffman encoded. Dictionary compression methods [Bell90] have been studied for several processors [Liao95, Lefurgy97]. IBM uses dictionary compression in embedded PowerPC microprocessors [IBM97]. Compression algorithms based on operand factorization and Markov models have been examined [Ernst97]. More complicated compression algorithms have combined operand factorization with Huffman and arithmetic coding [Lekatsas98, Aranjo98]. Compression methods for distributing programs over a network have been proposed [Fraser95, Ernst97, Franz97].

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 1) that the procedure cache be large enough to completely hold the largest procedure and 2) defragmentation be supported when not enough free-space is available. 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 functions of any size. It is faster because it avoids decompressing code that is not executed, does not need to manage cache fragmentation, and uses a simpler decompression algorithm.

3 Compression architecture

We use the instruction cache as a decompression buffer. On a cache miss, compressed instructions are read from main memory, decompressed, and placed in the cache. The instruction cache contents appear identical to a system without compression. This allows the CPU to be unaware of compression. In addition, code performs at native speeds once it is brought into the 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 cache.

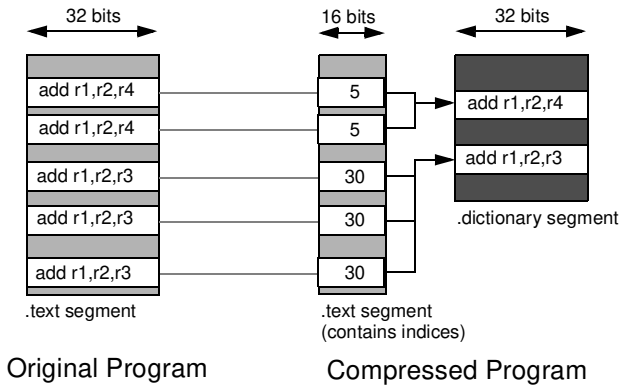


Figure 1: Compressed program

SimpleScalar parameters	Baseline
fetch queue size	1
decode width	1
issue width	1 in-order
commit width	1
LSQ	2
FUs	alu:1, mult:1, memport:1, fpalu:1, fpmult:1
branch pred	bimode 2048 entries
L1 D-cache	8KB, 32B lines, 2-assoc, LRU
L1 I-cache	1KB - 64KB, 16B lines, 2-assoc, LRU
memory latency	10 cycle latency, 2 cycle rate
memory width	64 bits

Table 1: Simulation Parameters

We believe that it is reasonable to expect new processors to provide such capability. These mechanisms have uses beyond just code compression. Jacob et al. proposes using such features so that software-managed address translation may completely replace hardware-translation performed by the translation-lookaside buffer [Jacob97]. Jacob further suggests using software-managed caches to provide fast, deterministic memories in embedded systems [Jacob99]. We believe that software-managed caches may also be useful for dynamic compilation and high-performance interpreters.

Our compression scheme [Lefurgy98] takes advantage of the observation that the instructions in programs are highly repetitive. 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 1 illustrates the compression method. The indices and the dictionary reside in main memory at the position of the original .text segment.

We use 16-bit indices which limits the dictionary to contain only 64K unique instructions. In practice, this is sufficient for many interesting programs, including our benchmarks. However, programs that use more instructions can be accommodated. Such programs are divided into a compressed region and a native code region. When the dictionary is filled the remainder of the program is left in the native code region. An extra flag in the page table entry could identify compressed and native code regions. 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 [IBM97].

4 Simulation environment

We perform our compression experiments on the SimpleScalar 3.0 simulator [Burger97] after modifying it to support compressed code. Our 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 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. Table 1 shows the simulation parameters.

The L1 miss handler (decompressor) is very simple and shown in Figure 2. The handler runs 74 instructions to decompress a cache line of 8 4-byte instructions. The decompression requires less than 3 SimpleScalar instructions per instruction *byte* decoded. In comparison, LZRW1 has a hand-optimized decompressor for 68000 which executes 4 instructions on average per output *byte* [Williams91].

Bench	Dynamic Insns (millions)	Cache miss ratio for 16KB cache	Original size (bytes)	Compressed size (bytes)	Dictionary compression ratio (smaller is better)	LZRW1 Compression ratio
cc1	121	2.9%	1,083,168	654,999	65.4%	60.4%
vortex	154	2.1%	495,248	274,420	65.8%	55.5%
go	133	2.0%	310,576	182,602	69.6%	63.9%
perl	109	1.6%	267,568	162,045	73.7%	60.2%
ijpeg	124	0.1%	198,272	118,131	77.2%	61.5%
mpeg2enc	137	0.01%	119,600	98,688	82.5%	60.5%
pegwit	115	0.02%	88,800	70,608	79.5%	56.7%

Table 2: Compression ratio of .text section

```
# Load L1 I-cache line with 8 instructions

# Register Use
# r9 : index address
# r10: base address of dictionary
# r11: 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]     # indices base (== .text base)
  mfc0 $10,c0[1]     # dictionary base

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

# index_address = (C0[0] - C0[BADVA]) << 1 + C0[0]
  sub $9,$27,$26  # get offset into .text
  srl $9,$9,1     # transform to offset into indices
  add $9,$26,$9   # load r9 with index

# 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

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

Figure 2: L1 miss exception handler

We add two instructions to SimpleScalar to support software decompression. First, `swic Rx,n(Ry)` stores the word in Rx to the I-cache address Ry+n. Second, `iret` returns from the exception handler to the missed instruction.

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 has a cache-miss itself. Our simulations put the exception handler in its own small on-chip RAM accessed in parallel with the instruction cache. It could also be put in the instruction cache if the cache has the ability to load and lock lines.

5 Results

All results include both application and library code. *Compression ratio* is used to measure the size of the program remaining after compression.

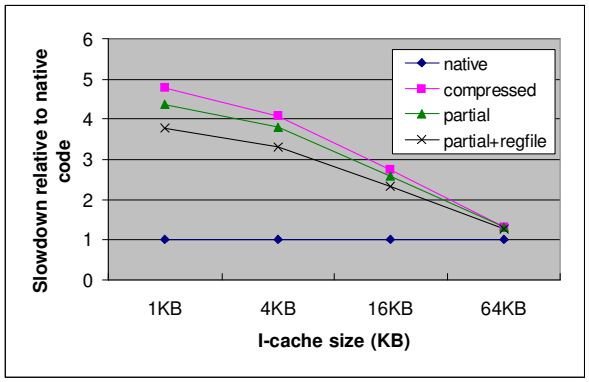
$$\textit{compression ratio} = \frac{\textit{compressed size}}{\textit{original size}} \qquad \textit{(Eq. 1)}$$

The size of the native and compressed programs are given in Table 2. The compressed program size is the sum of the dictionary and index bytes.

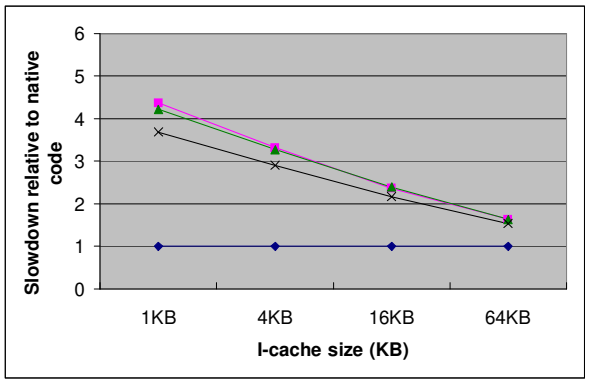
The performance of the benchmarks is shown in Figure 3. The y-axis shows the execution times of the benchmarks scaled to the execution time of the native non-compressed program. For brevity, we only report results for the benchmarks *cc1*, *go*, and *mpeg2enc*. We use *cc1* and *go* because they are among the worst performing benchmarks due to high cache miss ratio. We use *mpeg2enc* as an example of a benchmark with a better cache miss ratio.

In addition to the native and compressed programs, we consider two optimizations for the decompressor.

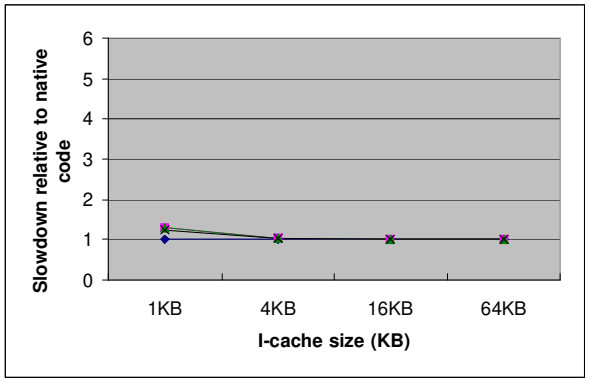
First, we consider only decompressing from the missed instruction until the end of the cache line. We call this *partial decompression*. This avoids work to decompress instructions at the beginning of the cache line which may not actually be executed. Since only part of the cache line is filled on a miss, the CPU must have a method to distinguish valid and invalid instructions in the line. Our simulations assume that the usual cache line valid bit is replaced with a valid bit for each instruction. When a new cache line is allocated, all valid bits are set to *false*. As we decompress, the `swic` instruction sets the valid bits to *true*. The cost is a slight expansion in cache size. Another



(a) cc1



(b) go



(c) mpeg2enc

Figure 3: Performance results. Y-axis shows performance scaled to native code results. *Native*: the native non-compressed program. *Compressed*: The compressed program decompressed with the exception handler in Figure 2. *Partial*: Decompressor optimized to only decompress from missed instruction to end of cache line. *Partial+regfile*: Decompressor optimized with partial decompression and use of a second register file.

method that would not expand the cache size would be to fill the entire cache line with copies of a special instruction when the line is allocated. This special instruction would signal a cache miss whenever it is executed allowing the exception handler to fill the request.

Second, we assume that the CPU has a second register file that the decompressor can use. During an exception, all instructions use the second register file. This allows us to avoid set-up instructions for saving and re-loading registers. In addition, we can store the dictionary and indices base pointers in this second general register file and avoid reloading them each time from system registers.

Partial decompression and the additional register file help most in the 1KB cache simulations due to the high number of instruction cache misses. The partial decompression improves the baseline decompression performance at most 10%. Partial decompression plus the additional register file improve the baseline decompression performance at most 22%.

For all benchmarks, the performance cost is no more than 5.1 times the original execution time with a small 1KB cache. The optimized version with a second register file and partial decompression is no worse than 4.2 times slower than the original using 1KB cache. Increasing cache size effectively controls this slowdown. The optimized decompression with a 64KB cache allows the slowest program (*go*) to run with a 54% increase in execution time and the fastest programs (*ijpeg, mpeg2enc,* and *pegwit*) to run with less than a 2% increase in execution time.

In comparison, the decompression used by Kirovski et al. 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 1KB to 64KB sized caches. Our decompression method shows more stability in performance over this range of cache sizes. However, the LZRW1 compression attains better compression ratios. Table 2 shows the compression ratios for LZRW1 when compressing the entire .text section as one unit. LZRW1 has 5-25% better compression ratios on SimpleScalar code compared to our dictionary compression.

Decompression only occurs during a cache miss. Therefore, the instruction cache miss ratio has a strong effect on the performance of the compressed program. Figure 4 shows this effect by plotting the instruction cache miss ratio against the execution time using all of our simulation results for each benchmark and cache size. The key to improving performance of a compressed program is to reduce the instruction cache misses. This can be accomplished in several ways. For example, enlarging the cache, applying code layout optimizations, or applying classical optimizations that improve the native code size. Our results show that once the instruction cache miss ratio is below 1.5%, all decompression implementations perform no worse than twice as slow as the native programs.

Another effective way of controlling loss of execution speed is to use *selective compression*. Highly used procedures should be left un-compressed so that they do not

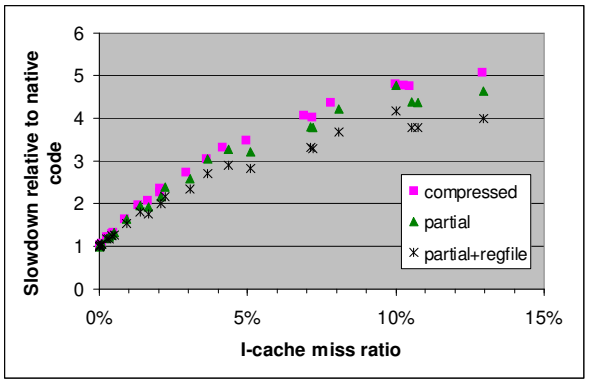


Figure 4: Effect of I-cache miss ratio on performance. Data points represent all benchmarks simulated with all decompressor implementations and cache sizes.

invoke the decompressor. This technique is used in existing compressed code systems [ARM95, IBM98].

6 Conclusions

We have presented a technique of using software-managed caches to support code decompression. In this study we have focused on designing a fast decompressor (rather than generating the smallest code size) in the interest of performance. In some cases, our decompressor is an order of magnitude faster than a previous software decompression scheme at a cost of 5-25% in the compression ratio.

Performance loss due to compression can be mitigated by improving the instruction cache miss ratio. Optimizing the CPU with sub-block valid bits in the instruction cache or by adding a second register file can improve the performance of the decompressor. 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.

Acknowledgments

This work was supported by DARPA grant DABT63-97-C-0047. Simulations were performed on computers donated through the Intel Education 2000 Grant.

References

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

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

[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*, September 1998.

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

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

[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.

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

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

[Jacob97] B. Jacob and T. Mudge, "Software-Managed Address Translation", *Proceedings of the Third International Symposium on High Performance Computer Architecture*, 1997.

[Jacob99] B. Jacob, "Cache Design for Embedded Real-Time Systems", *Proceedings of the Embedded Systems Conference*, Summer 1999.

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

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

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

[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*, 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*, June 1998.

[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.

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

[Williams91] R. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm", *Data Compression Conference*, 1991.

[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.