Reducing Code Size with Run-time Decompression

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

Abstract

Compressed representations of programs can be used to improve the code density in embedded systems. Several hardware decompression architectures have been proposed recently. In this paper, we present a method of decompressing programs using software. It relies on using a software-managed instruction cache under control of the decompressor. This is achieved by employing a simple cache management instruction that allows explicit writing into a cache line. We also consider selective compression (determining which procedures in a program should be compressed) and show that selection based on cache miss profiles can substantially outperform the usual execution time based profiles for some benchmarks.

1 Introduction

Many recent code compression studies have suggested that custom on-chip hardware be used to decompress programs. In this paper, we explore software decompression. Software methods 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. Finally, 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 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 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, Lefurgy99]. 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 [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 procedures 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.

Interpretive programs are another way to achieve small code size [Klint81]. 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 [Hoogerbrugge99]. While we do not achieve the small code sizes attained by interpretation, our programs are much faster. 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 in loop-oriented programs.

3 Compression architecture

This section presents our dictionary-based software decompression, a software decompressor based on IBM's CodePack, and the technique of selective compression for controlling performance degradation due to decompression.

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 I-cache. The I-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 instruction cache. We believe that it is reasonable to expect new processors to provide such capability. For example, the MAJC architecture 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].

3.1 Dictionary compression

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

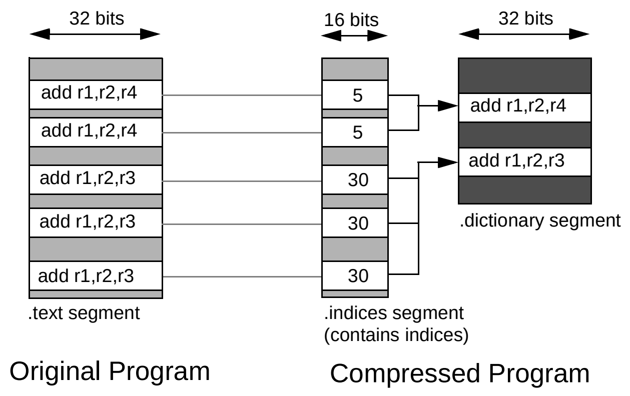


Figure 1: Dictionary compression
In this example, the instruction “add r1,r2,r4” maps to the index “5” and the instruction “add r1,r2,r3” maps to the index “30”.

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.

We use 16-bit indices which limits 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. Such programs are divided into a compressed region and a native code region. This is called selective compression. When the dictionary is filled 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 [IBM97].

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, IBM97]. Our dictionary compression is able to avoid this table lookup because our native instructions and compressed codewords have fixed lengths. Since our 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.

3.2 CodePack

We implemented another software decompressor using the CodePack [IBM98] compression algorithm. This algorithm compresses programs much more than our simple dictionary compression, but the decompressor takes longer to run. We use this as a point of comparison to the dictionary compression and show how selective compression can be used to improve performance.

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.

3.3 Selective Compression

One effective way of controlling loss of execution speed is to use *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.

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.

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.

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 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

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	16KB, 32B lines, 2-assoc, lru
L1 D-cache	8KB, 16B lines, 2-assoc, lru
memory latency	10 cycle latency, 2 cycle rate
memory width	64 bits

Table 1: Simulation Parameters

10 cycles and successive accesses take 2 cycles. Table 1 shows the simulation parameters.

We add three instructions to SimpleScalar to support software decompression. First, `swic Rx,n(Ry)` stores the word in Rx to the address Ry+n in the instruction cache. Rx and Ry are registers and n is an immediate value. Since this instruction writes the cache, it would be difficult to squash and restart the instruction on a miss-speculation. Therefore, we require that the processor is in a non-speculative state before the instruction executes. We accomplish this by flushing the pipeline of preceding instructions. Second, `iret` returns from the exception handler to the missed instruction. Third, `mfc0` (move from co-processor 0) moves a value from a system register into a general purpose register. On a cache miss, the decompressor uses this instruction to get the value of the missed instruction address, which is available in a special system register.

4.1 Decompression

The baseline dictionary decompressor code is shown in Figure 2. The decompressor is 208 bytes (26 instructions) and executes 75 instructions to decompress a cache line of 8 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.

Many embedded processors 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 avoid 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

```
# Load L1 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)
  iret # return from exception handler
```

Figure 2: L1 miss exception handler for dictionary decompression method

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.

4.2 Simulation of selective compression

Our compression software has the ability to produce binaries containing both compressed and non-compressed

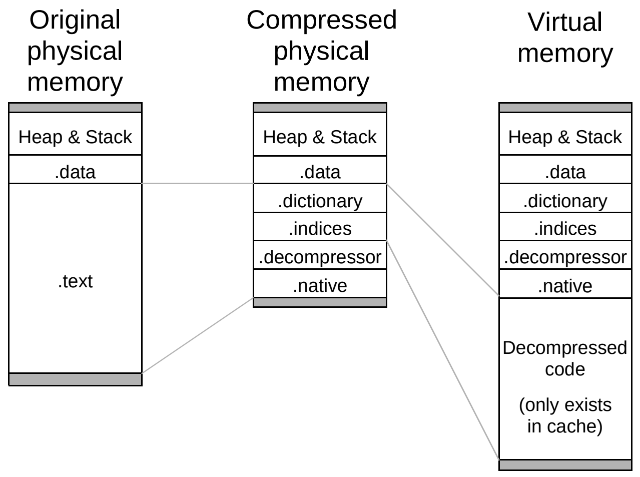


Figure 3: Memory layout
The `.text` segment is compressed into `.dictionary` and `.indices` segments. The `.decompressor` segment holds the decompressor code. Some code in `.text` may be left in `.native` so that decompression does not need to occur for critical code. On a cache miss, the `.dictionary` and `.indices` segments are decompressed and put in the segment marked “Decompressed code”. This segment only exists in the I-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.

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 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 3. 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.

5 Results

This section presents results of our dictionary and CodePack simulations. We also show results for using selective compression to manage performance degradation.

5.1 Size results

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. All results include both application and library code. The dictionary compressed program size is the sum of the dictionary and index bytes. The CodePack program size includes the indices, dictionary, and mapping table. The decompression code is not included in the compressed program sizes.

5.2 Performance results

The performance of the benchmarks is shown in Table 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 2 shows the non-speculative miss ratios for the 16KB 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.

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 selective compression to keep some functions as native code so they will use the hardware cache controller to quickly fill misses. The miss ratio can be reduced by enlarging the cache, increasing cache associativity, applying code placement optimizations to reduce conflict

Benchmark	Dynamic insns (millions)	Cache miss ratio for 16KB cache	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 2: Compression ratio of .text section

Dynamic insns: Number of instructions committed in benchmarks. *Cache miss ratio*: non-speculative cache miss ratio for a 16KB 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 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.

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 4KB, 16KB, and 64KB instruction caches. In Figure 4, 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 performance is less than 2 times slower than native code. When the miss ratio is below 1% for CodePack programs, the performance 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.

It is difficult to compare our results with those of Kirovski et al. since 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 1KB to 64KB 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 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.

5.3 Selective Compression

Selective compression can be used to control performance at a cost in code size. We vary the aggressiveness of our selection policy to measure this trade-off. Figure 5 shows the results of using both miss-based and execution-based selective compression on dictionary and CodePack programs. The left sides of these size/speed curves represent code that is totally compressed. The right side of the curves represent full native code. The data points in between represent hybrid programs that have both compressed and native code. The amount of native code in each hybrid program is explained in Section 3.3.

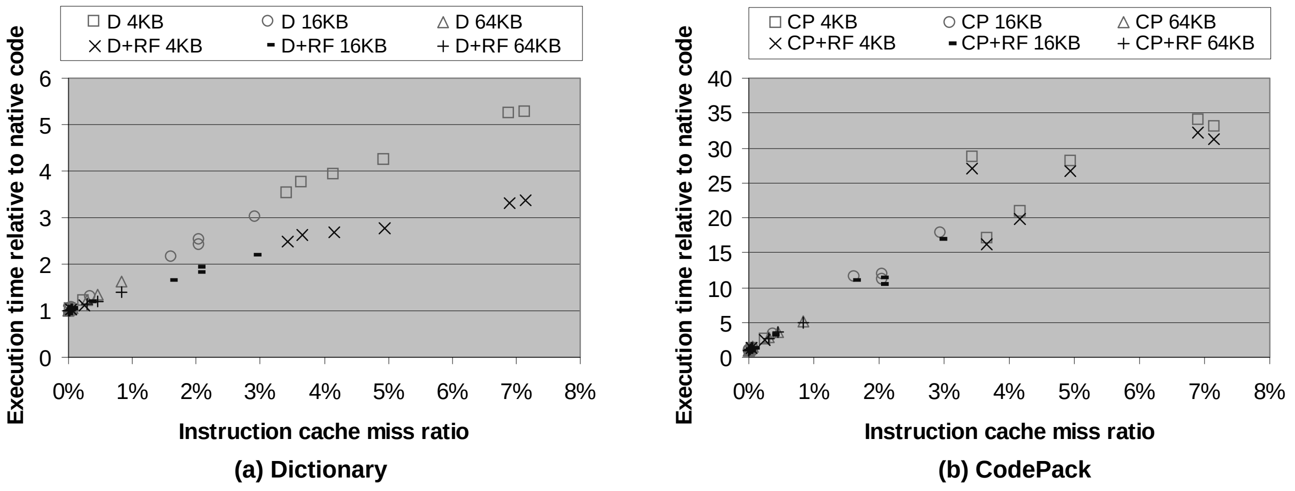


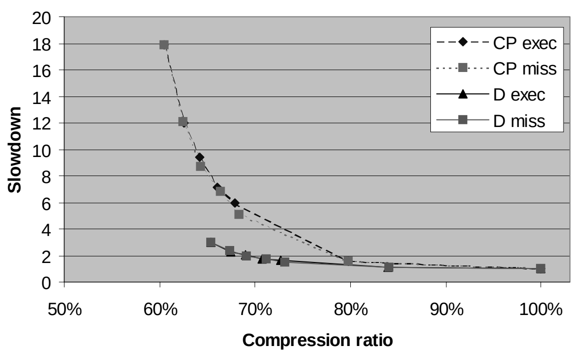
Figure 4: Effect of I-cache miss ratio on execution time
Data points represent all benchmarks simulated with instruction cache sizes of 4KB, 16KB, and 64KB. a) Dictionary compressed programs. b) CodePack compressed programs.

It seems counter-intuitive 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. When the procedures of the benchmark are split between the native and compressed memory regions, it changes the procedure placement of the decompressed program in virtual memory. 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 results in different instruction cache conflict misses occurring than in the original program. It is clear from Figure 4 that even small changes in the instruction cache miss ratio can dramatically affect performance of compressed programs. Therefore, it is possible that a poor procedure placement could overwhelm the benefit of using hybrid programs.

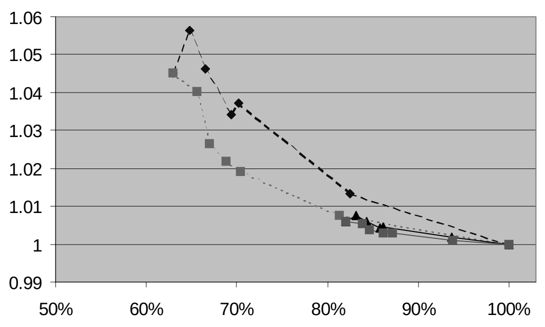
The effects of procedure placement on performance can be significant. Pettis and Hansen noted that a good procedure placement could improve execution time by up to 10% [Pettis90]. To our knowledge, we are the first to report this effect in selective compression. Procedure placement is likely to affect 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 procedure 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.

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.

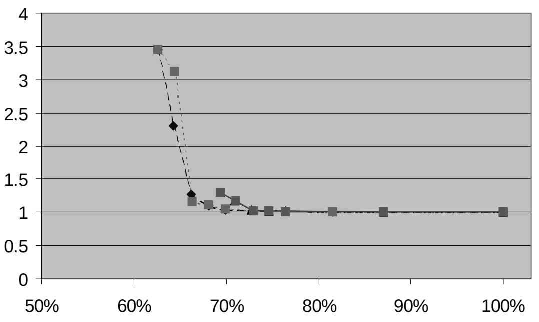
An interesting result we found is that miss-based profiling is a significant improvement over execution-based profiling for loop-oriented programs such as *mpeg2enc* and *pegwit*. The reason is that execution-based profiling selects loops to be native code because they account for most of the executed instructions. This is correct for instruction sets like Thumb and MIPS16 because the loops will be rewritten as 32-bit code which execute faster. However, for our software decompressors, the decompressed code will execute as quickly as the native version once it has paid the decompression penalty. 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. 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 software-managed decompressors that are invoked on cache misses should use miss-based profiling for loop-oriented programs.



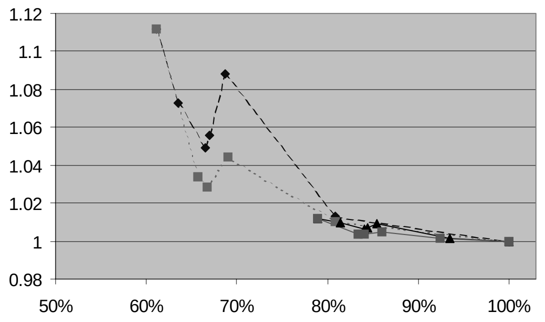
(a) cc1



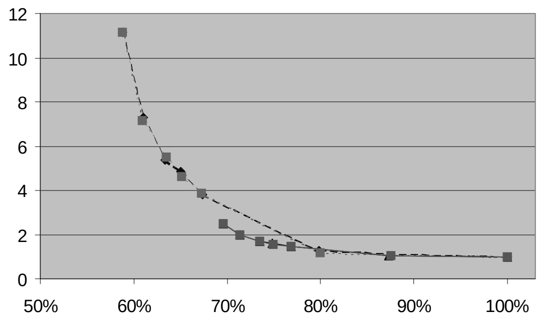
(e) mpeg2enc



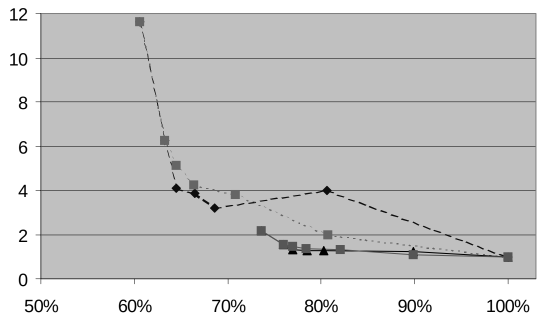
(b) ghostscript



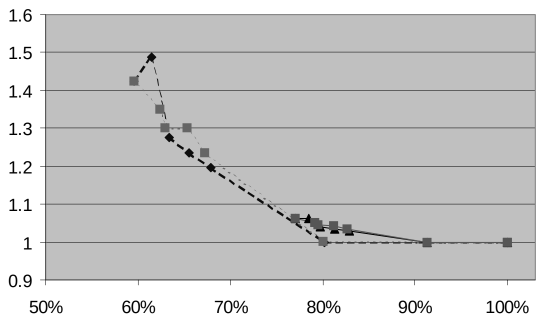
(f) pegwit



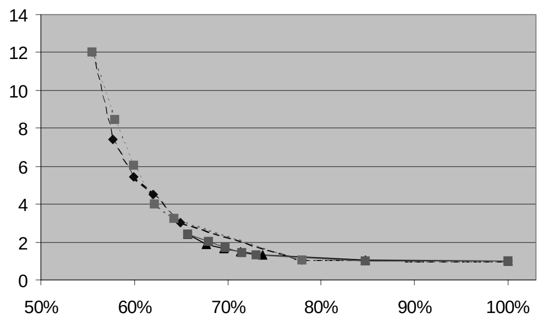
(c) go



(f) perl



(d) ijpeg



(g) vortex

Figure 5: 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.

Mpeg2enc and *pegwit* are the only two benchmarks for which miss-based selection always performs better than execution-based selection. Using miss-based selection for *pegwit* can cause it to have half the decompression overhead of execution-based selection in some cases.

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.

6 Conclusions

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 decompression loop to be unrolled.

Performance loss due to compression can be mitigated by improving the instruction cache miss ratio or by reducing the amount of time required to service an instruction cache miss. This suggests that incorporating compression with other optimizations that reduce cache misses (such as code placement) could be highly beneficial. Selective compression is effective in improving the performance of compressed programs. 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 substantial benefit for using miss-based profiling on loop-oriented programs such as pegwit and mpeg2enc.

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.

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.

[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), Sept. 13, 1999.

[Hoogerbrugge99] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. Van De Wiel, “A Code Compression System Based on Pipelined Interpreters”, *Softw. Pract. Exper.*, 29(11), 1999.

[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.

[Klint81] P. Klint, “Interpretation Techniques”, *Software Practice and Experience*, Vol. 11, No.9, September 1981.

[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.

[Lefurgy99] C. Lefurgy, E. Piccininni, and T. Mudge, “Analysis of a High Performance Code Compression Method”, *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999.

[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.

[Pettis90] K. Pettis and R. Hansen, “Profile Guided Code Positioning”, *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation (PLDI)*, June 1990.

[Pittman87] T. Pittman, “Two-Level Interpreter/Native Code Execution”, *Proceedings of the SIGPLAN’87 Symposium on Interpreters and Interpretive Techniques*, 1987.

[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.