Code Compression for Embedded Systems

Haris Lekatsas and Wayne Wolf
Department of Electrical Engineering
Princeton University
{lekatsas,wolf}@ee.princeton.edu

Abstract

Memory is one of the most restricted resources in many modern embedded systems. Code compression can provide substantial savings in terms of size. In a compressed code CPU, a cache miss triggers the decompression of a main memory block, before it gets transferred to the cache. Because the code must be decompressible starting from any point (or at least at cache block boundaries), most file-oriented compression techniques cannot be used. We propose two algorithms to compress code in a space-efficient and simple to decompress way, one which is independent of the instruction set and another which depends on the instruction set. We perform experiments on two instruction sets, a typical RISC (MIPS) and a typical CISC (x86) and compare our results to existing file-oriented compression algorithms.

1 Introduction

Many embedded computing systems are space and cost sensitive. As a result, available memory is limited, posing serious constraints on program size. We are studying ways of reducing the size of stored programs in such systems by using code compression techniques. The reduced executable size can result in significant savings in terms of cost, size, weight and power consumption.

RISC instructions were designed in part to reduce instruction decode overhead. However, in deep-submicron pipelined processors, we believe that most instruction decode time can be hidden (as in the Intel architectures) particularly in the case of compressed code. We decompress our code before inserting it in the cache, following the design first proposed by Wolfe and Chanin [13]. We therefore assume that the processor executes normal uncompressed code and that decompression of a cache line occurs only when there is a cache miss. The loss in performance should therefore depend on the instruction cache hit ratio.

Such a system poses some constraints on the compression algorithms that can be used. Since programs can have jumps, we must decompress individual cache blocks and not the whole program sequentially. Compression techniques which require sequential decompression starting from the beginning of the compressed file cannot be used here. Furthermore the decompression hardware should be fast enough not to severely reduce CPU performance and should be kept as small as possible.

There is an abundance of algorithms for data compression in the literature. Due to the constraints stated above most of the existing state of art algorithms cannot be used directly. In terms of compression ratios the the algorithms that use finite context modelling such as PPM (Prediction by Partial Match) [1], DMC [3], and WORD [8] seem to achieve the best performance. However they require large amounts of memory both for compression and decompression, making them unsuitable for program compression. The Ziv-Lempel family of algorithms [14] use pointers to previous occurences of strings which makes an individual block decompression scheme impossible.

An instruction-based compression approach which overcomes the above problems is described by Kozuch and Wolfe [5], where byte-based Huffman codes [4] are used, yielding a compression ratio around 0.73 (compressed size/original size). Such a scheme is simple to implement, and certainly allows decompression to take place from any place in the code. A shortcoming of this method is the the fact that 8-bit symbols have been used instead of 32-bit symbols corresponding to the size of RISC instructions. This means that all 4 bytes within the same 32-bit word are encoded using the same table. Since instructions have different fields which have different statistical characteristics such a choice increases the entropy of the source significantly. However, if we use 32-bit symbols we need a Huffman table with 2^{32} entries making the size of the decompressor prohibitively large. Futhermore, this method does not take into account dependencies between instructions, limiting the overall compression performance. Another dictionary-based approach is described by Liao et al. [6], where a dictionary is generated for TMS320C25 code. Their approach requires little or no extra hardware, but gives only modest compression ratios.

In this paper, we present two different approaches which produce significantly better compression. We concetrate on the algorithm side to understand the limits of program compressibility as a CAD problem. The related architecture question is the cost in time (and gates) of these techniques on CPUs. We present here some initial analyses which suggest our schemes can be reasonably implemented in hardware, but architectural details remain future work. In section 2, we describe design issues for embedded code compression and we introduce our techniques. Section 3 describes the details of our statistical coding scheme, which is independent of the instruction set and only assumes fixed-size instructions. Section 4 discusses an alternative approach which uses dictionary coding and is instruction set dependent. In section 5 we present our experimental results, and finally in section 6 we conclude and discuss possible future directions.

2 Design issues

An approach to reduce code size in embedded systems would be to design a new RISC or CISC architecture with denser encoding. Although such an approach may be useful, we chose to follow an alternative method where the memory sys-

35th Design Automation Conference ®
Copyright ©1998 ACM
1-58113-049-x-98/0006/$3.50
DAC98 - 06/98 San Francisco, CA USA

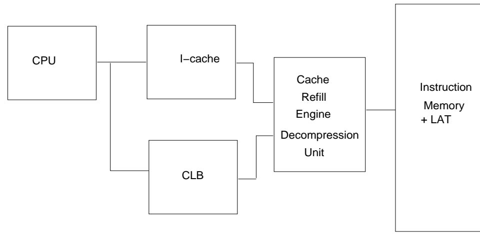

Figure 1: Memory System Organization

tem is redesigned while the processor architecture remains intact. Such an approach has several practical advantages: We can use well-characterized existing processor pipelines, take advantage of existing programming development tools, use existing workstations as development platforms, etc.

Wolfe and Chanin [13] designed a system with run time decompression. Figure 1 shows the memory organization for such a system. The instruction cache is used as a decompression buffer storing recently used decompressed cache blocks. The decompression is being done by the cache refill engine whenever a cache miss occurs. This implies that the compression algorithm must compress on a block-by-block basis. Since the engine must know where to find a compressed cache line in memory (different cache lines will generally have different compressed sizes) a table called LAT (line address table) has been stored in main memory along with the instruction code. This table maps program instruction block addresses into compressed code instruction block addresses. Since accessing the LAT will increase the cache refill time a CLB (Cache line Address Lookaside Buffer) can be used which is essentially identical to a TLB.

Note that we only compress the part of the executable which contains instructions, not any data, tables etc. Compressing data which may also be writable would complicate the design as we would have to provide a fast compressor as well when writing to main memory.

3 SAMC: An ISA-independent method using statistical coding

Our first method uses a binary arithmetic coder [12] driven by a Markov model. In the following we will refer to this method as *SAMC* for Semiadaptive Markov Compression. Since we are compressing cache blocks, an adaptive method cannot be used effectively as the coder will not be able to gather enough statistical information from just one block. A static model to feed the arithmetic coder is therefore required. We will use the term *stream* to refer to a group of bits within instructions. Figure 2 shows an example of stream subdivision for a MIPS instruction set. Although the bits in the figure are adjacent, this is not required by our algorithm. We performed several experiments on this stream subdivision idea, and found that a scheme that works particularly well is to divide instructions into 4 streams of 8 bits each and build Markov models for each one of them. This gives reasonable compression without requiring excessive storage. Here is the two-step compression scheme (n is the number of bits per instruction):

1. **Statistics-gathering phase.**
   (a) Instructions are divided into k streams of bits, each of the streams containing k_i bits, i=0,1,..,k-1, i.e. $n = \sum_{i=0}^{k-1} k_i$

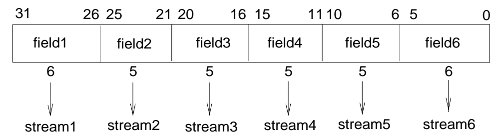

Figure 2: Example stream division for MIPS

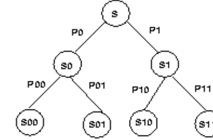

Figure 3: Binary Markov tree

   (b) For each stream we generate a binary Markov tree with $2^{k_i+1}-1$ states. The first state is the initial state corresponding to no input bits. Its left and right child correspond to the states “0 input” and “1 input” respectively. Figure 3 shows such a tree for a stream with 2 bits. State S_0 corresponds to the state “0 input”, state S_{01} corresponds to the state “0 input then 1 input” , etc. Each transition has a probability which is generated by going through the whole subject program. Probability $P_{b_0 b_1 b_2 \ldots b_l}$ corresponds to the probability that the bit sequence $b_0 b_1 b_2 \ldots b_l$ has arrived. We only need to store the probabilities on the left branches of the tree since the probabilities on the right branches are the complements of the probabilities on the left branches. Hence for the tree in figure 3 we need to store 3 probabilities: P_0, P_{00}, P_{10}. In general for a stream of k-bits we need to store $(2^{k+1}-2)/2$ probabilities.

2. **Compression phase.** The probabilities calculated above are done as a preprocessing step. The compressor uses them as predictions on the coming bit to perform the binary arithmetic coding. Hence the compressor goes through the subject program a second time and encodes the message. The final storage requirements are the encoded message and the Markov trees for the all the streams.

Each stream is coded independently and any correlation between streams is not taken into account. Compression performance can be improved by connecting the Markov trees of adjacent streams. This provides some limited memory between streams to the model. The connection between trees is illustrated in figure 4, where for simplicity we have assumed that instructions are 4 bits wide and that we have divided them into 2 streams, 2 bits each.

There are two reasons for dividing instructions into streams. First, it would be impossible to generate one Markov tree for the whole instruction as the number of probabilities we would need to store would be $(2^{32+1}-2)/2 = 2^{32}-1$. The second reason is that we can chooose the streams appropriately so that they have low entropy and thus high compressibility. In order to do the stream division we calculated the correlation factor ρ_{ij} between all bits i and j, i,j = 0,1,...,31, and we tried to group bits that are strongly correlated. Strongly correlated bits will give either high (close to 1) or low (close to 0) probabilities for the Markov tree

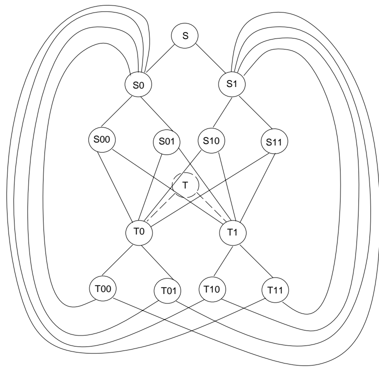

Figure 4: Connected Markov trees

improving the compression performance of the arithmetic coder. Our program combines bits with high correlation to streams and calculates their entropies. It then attempts to exchange some bits between streams randomly and recalculates the entropies. If the new average entropy is lower it accepts this step, otherwise it tries a different combination. In practice, we found that dividing 32-bit instructions into 4 8-bit streams (a stream does not necessarily have adjacent bits) produces results close to optimal.

Once the Markov trees are calculated for all streams the compressor and decompressor use the same steps to perform the arithmetic coding of one cache line. The following lines of pseudo-code illustrate the steps the decompressor must take to decompress one cache line. Note that the interval [min,max] used by our approach has to be reset upon compression and decompression of each block and we must also restart from the initial node of the Markov model. This ensures that we can decompress starting from any block boundary. For the same reason we reset the Markov model to its starting node for each block. We have assumed that the decompression engine works with 24 bits of the compressed code, and that 24 bits is the accuracy of our interval.

```
1    max = 0x1000000; min = 0  // use 24 bits
2    p = top_of_Markov_tree;   // make p point to the start
3    bytes_written = 0;        // of the model
4    val = get_24bits_of_compressed_code();
5    do {
6      c = 0;
7      if (val == (max-1)) break;
8      for (i=0; i<8;i++) {
9        mid = min + (max-min-1)*p->prediction;
10       if (mid == min) mid++;      // fix mid value if
11       if (mid == (max-1)) mid--; // p->prediction=0,1
12       if (val >= mid) {
13         bit = 1;                 // decode 1 bit
14         min = mid;               // update interval
15       }
16       else {
17         bit = 0;                 // decode 1 bit
18         max = mid;               // update interval
19       }
20       if (!bit) then p = p->left // update position
21       else p = p->right          // in Markov tree
22       c = c<<1 | bit;            // store decoded bit
23       while ((max-min) < 0xff) {
24         if(bit)max--;
25         bytes_written++;
26         val = (val << 8) | get_byte(); // get 8 more
27         min = min << 8;                // compressed
28         max = max << 8;                // bits
29         if (min >= max) max = 0x1000000;
30       }
31     }
32     write_byte_to_cache(c);
33   } while (bytes_written<block_size);
```

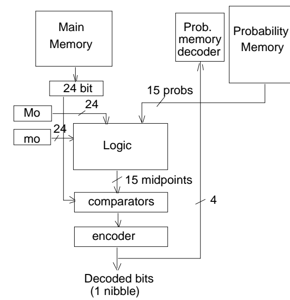

Figure 5: Decoder Block Diagram

The above code performs binary arithmetic decoding. Starting with the interval [min,max] the algorithm uses the probability from the Markov tree (line 9) to calculate a midpoint value. The input value (compressed code) in val is compared with the midpoint and a 0 or a 1 is written to the output (decompressed code). The main drawback of the procedure above is that it works on a bit-by-bit basis. In order to speed it up we can perform midpoint calculations in parallel. However the new midpoint always depends on the previous one, as the new value of max or min are equal to mid (lines 9,14,18). In order to overcome this we can calculate all midpoints for both the cases when $val >= mid$ and $val < mid$. The procedure below gives all the midpoints, where the probabilities $p_{x_0 x_1 \ldots}$ are given by the Markov tree:

$$mid_0 = min_0 + (max_0 - min_0 - 1) \cdot p_0$$
$$mid_1 = mid_0 + (max_0 - mid_0 - 1) \cdot p_{00} \quad , mid_0 < val$$
$$mid_1 = min_0 + (mid_0 - min_0 - 1) \cdot p_{01} \quad , mid_0 > val$$
$$mid_2 = mid_1 + (max_0 - mid_1 - 1) \cdot p_{000} \quad , mid_0 < val$$
$$mid_2 = mid_0 + (max_0 - mid_0 - 1) \cdot p_{010} \quad , mid_0 < val$$
$$mid_2 = mid_1 + (mid_0 - min_0 - 1) \cdot p_{100} \quad , mid_0 > val$$
$$mid_2 = mid_0 + (mid_0 - min_0 - 1) \cdot p_{110} \quad , mid_0 > val$$
...

We can see that all μ_i's can be expressed as functions of M_0, m_0 and p_{ij}'s. Clearly the number of μ_i's and p_{ij}'s grow exponentially. A reasonable solution is to decode 4-bit values which means we need 15 μ_i's and 15 p_{ij}'s. After calculating all μ_i's and p_{ij}'s, we choose the right ones by comparing them with val. Figure 6 shows a block diagram of the decompression engine using the algorithm described above. The "Logic" and "Comparators" calculate midpoints. Note that the output bits are used to determine which probabilities to fetch from the probability memory. To avoid the multiplication in the midpoint calculation unit we can constrain the probability of the less probable symbol to the nearest integral power of $\frac{1}{2}$, thus requiring only shifts. Witten et al [12] showed that the worst-case efficiency is about 95% when we pose this constraint. In our case, in order to save space, we only store the probability of a 0 input. If it is the less probable symbol then only a shift is required, otherwise a shift and a subtraction from the max_i point is required. This greatly simplifies the design of the midpoint calculation unit.

4 SADC: An ISA-dependent method using a dictionary

Another valid choice is to build a dictionary with common sequences of instructions. Dictionaries can be static, semi-adaptive, or dynamic [1]. Dynamic dictionaries are adap-

tive and change while the compressor or decompressor goes through the program. This technique cannot be used here because it does not allow for random access. Static dictionaries are built once and used for all programs, while semiadaptive are built for each subject program. Clearly a semiadaptive dictionary will achieve better compression for a given program as it is specifically designed for that program. The method of this section uses a semiadaptive dictionary to compress opcodes, opcode-register combinations and opcode-immediate combinations. The stream subdivision idea also applies here; instructions are divided into streams in a predetermined way which depends on the instruction set. For example, we divided MIPS instructions into 4 different streams: opcode, register, immediate,long immediate stream (streams have different bit widths in this case). In the following we will use the term $SADC$ (Semiadaptive Dictionary Compression) for our compression scheme.

4.1 Compressing the streams

Opcodes in an instruction sequence, tend to exhibit some dependence between them since compilers typically generate code which uses the same sequences of instructions over and over. Having thrown away immediates and registers we can fully exploit such dependencies by generating new augmented opcodes which combine several opcodes together. We noticed that all our benchmark programs tend to use no more than 50 instructions. Since we cannot allow such opcode augmentation to become very large and in order to keep the opcode value in one byte, we restricted the maximum number of opcodes to 256. This means that we can augment the instruction set by about 200 new opcodes. We are effectively building for each subject program a semiadaptive dictionary which maps indices to opcodes or opcode combinations.

Finding the best dictionary for a given program is shown by Storer [10] to be NP-hard. We therefore do not attempt to generate an optimal dictionary. Once the dictionary is generated we are also faced with the problem of how to parse the subject program and replace occurences of opcode combinations with dictionary indices. Several heuristics for parsing have been proposed with *greedy* parsing being the most popular due to its simplicity and speed. Our approach does dictionary generation and parsing simultaneously. It inserts one candidate for dictionary insertion at a time, and then it modifies the subject file by inserting the dictionary index in the place of the candidate. It then works on the new modified file by finding new candidates and replacing them by their dictionary indices.

In order to decide which opcodes or opcode combinations should be inserted in the dictionary we test the following candidates:

- Adjacent opcode pairs. This attempts to take advantage the correlation between adjacent instructions.
- Generation of new opcodes for instructions which appear frequently with some specific registers or immediates. This means that if, for example, the register R31 in instruction jr R31 for MIPS code appears much more frequently than any other register, we can reduce the register stream size by introducing a new special opcode for jr R31.

Our dictionary generator works as follows:

1. The generator scans the subject program and creates a tree with all the opcodes and their frequencies, all the groups of 2 consecutive opcodes with their frequencies and finally all the groups of 3 consecutive opcodes with their frequencies. It does not attempt to search any bigger groups as that would result in large execution times and memory storage requirements.
2. Going through the tree generated in step 1, the generator inserts all single opcodes in the dictionary. It subsequently follows one of two paths: it either encodes the group of adjacent opcodes which has the largest gain in savings of the encoded opcode stream; or it encodes the opcode with a specific register or immediate which will gives the greatest reduction in register stream or immediate stream size. We explain how to calculate these gains in reduction below.
3. Using the dictionary generated above, the generator encodes the file by storing the dictionary indices for each opcode or group parsed.
4. All dictionary entries and the tree are erased. The generator repeats from step 1 until the dictionary generated has entries equal to the maximum allowed, or the new encoded file isn't smaller than the one of the previous cycle.

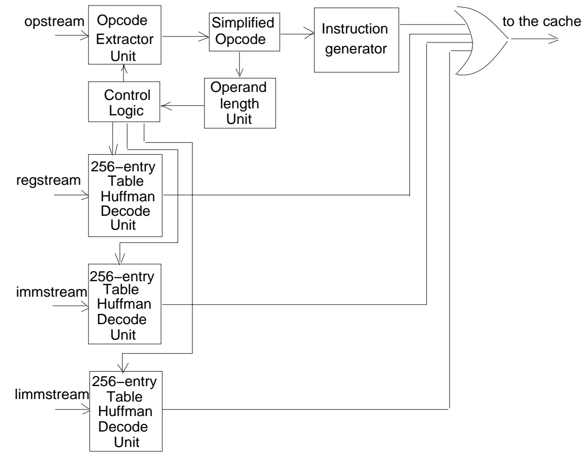

Figure 6: Decompressor Block Diagram for MIPS architecture

Our maximum dictionary size is 256 entries which means we need to store just one byte for the indices. The gain stated in step 2 of the algorithm is calculated as follows:

- Opcode groups. Suppose that the group to be inserted has length n. (Note that this length can be much larger than 2 or 3, since after several cycles of the algorithm the groups generated will contain longer sequences). Also suppose it occurs with frequency f, where f means non-overlapping occurences of the group. Since if it is inserted in the dictionary it will consume n bytes of space, the gain by inserting it in the dictionary and encoding the file will be $g = f \cdot (n-1) - n$.
- Register and immediate selection. The gain here is given by $g_{reg} = f \cdot n_{regs}$ and $g_{imm} = f$, where f is the frequency of an opcode, and n_{regs} is the number of registers selected for integration in the opcode.

The compressor calculates the above gains for all opcodes and picks the group of instructions with the largest gain to include in the dictionary. The final step of our compression is to encode all resulting compressed streams by using Huffman encoding. This tends to further improve overall

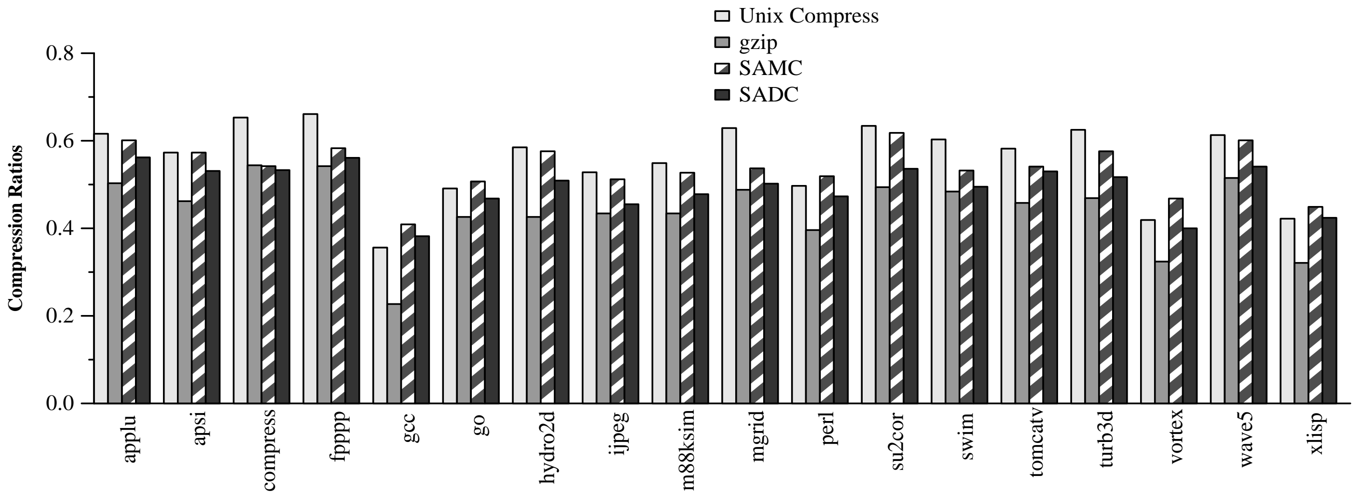

Figure 7: Compression results for MIPS

compression. Figure 6 shows a block diagrams for a MIPS decompression hardware:

- Opcode extractor unit: This takes as input the dictionary indices and produces the 8-bit opcode values contained in the dictionary. For MIPS an instruction generator unit is required since instructions are specified by one or two different groups of bits. The 8-bit opcode values are mapped to the corresponding bits in a real 32-bit instruction.
- Instruction generator. This maps 8-bit (simplified) opcodes to the format required by MIPS. The resulting 32-bit word coming as an output of this unit is ORed with the decompressed register and immediate streams and the final word is sent to the cache. This unit is needed because on MIPS the streams created are not composed from consecutive bits, hence we must put the decompressed bits to their right place.
- Operand length unit: This gives for each opcode how many registers and immediates (if any) must be combined with it to form an instruction.
- Control Logic unit: This sends the appropriate signals to the decompressing units, so that the right number of registers and immediates are combined with each opcode.

5 Experimental results

We performed experiments on two architectures, MIPS and x86 (Pentium Pro). For MIPS we divided instructions into 4 streams (opcode, register, 16-bit immediates, 26-bit immediates), while for Pentium we formed 3 streams (opcode, modR/M and SIB, immediate and displacement stream). The Pentium streams are 8 consecutive bits wide. This means the design of a Pentium decompressor would not need an instruction generator.

Figures 7 and 8 show our experimental results on MIPS and Pentium-Pro architectures. To measure compression performance we used the ratio of compressed program size over original program size. Thus a short bar means good compression performance. Since embedded code is hardly portable among architectures we used the SPEC95 benchmarks to measure the compressibility of our algorithms. On both machines we compiled the benchmarks using the default flags provided by the SPEC package. All of our experiments are done assuming a cache block size of 32 bytes. Different cache block sizes have a minimal impact on the results presented.

SAMC on MIPS tends to exhibit compression performance similar to UNIX compress. Gzip generally outperforms SAMC except for the compress benchmark. This is probably due to the small size of compress, which does not allow gzip to take advantage of long repeating sequences. SAMC however, sustains similar performance for both small and large files as it cannot see beyond cache blocks . SADC performs about 4-6% better than SAMC, and for some benchmarks on MIPS it comes close to gzip.

File compression algorithms tend to perform better on the CISC architecture. For SAMC this is expected as we cannot do any stream subdivision (SAMC was designed for fixed-sized RISC instructions). SADC performs better but still fails to come close to gzip. This may be attributed to the rather crude stream subdivision (only 3 byte-based streams). A different stream subdivision working with individual fields and not with whole bytes might improve compression, but one the other hand it would complicate the decompressor's logic.

Figure 9 compares our algorithms with byte-based Huffman coding as presented by Kozuch and Wolfe [5]. The compression ratios presented in figure 9 are the average compression ratios of all SPEC95 benchmarks. For MIPS SAMC and SADC perform substantially better than Huffman, while on Pentium the difference is not as big. SADC on Pentium does not do any stream subdivision and compresses bytes, hence performance similar to Huffman is expected. The results are slightly better than Huffman since the connected Markov trees exploit some of the dependencies between consecutive bytes, while Huffman does not take into account such dependencies. SADC performs much better than SAMC, but results might have been even better with more careful stream subdivision.

6 Conclusions and Future Work

This paper presents two methods to reduce code size for embedded systems. The compression performance results

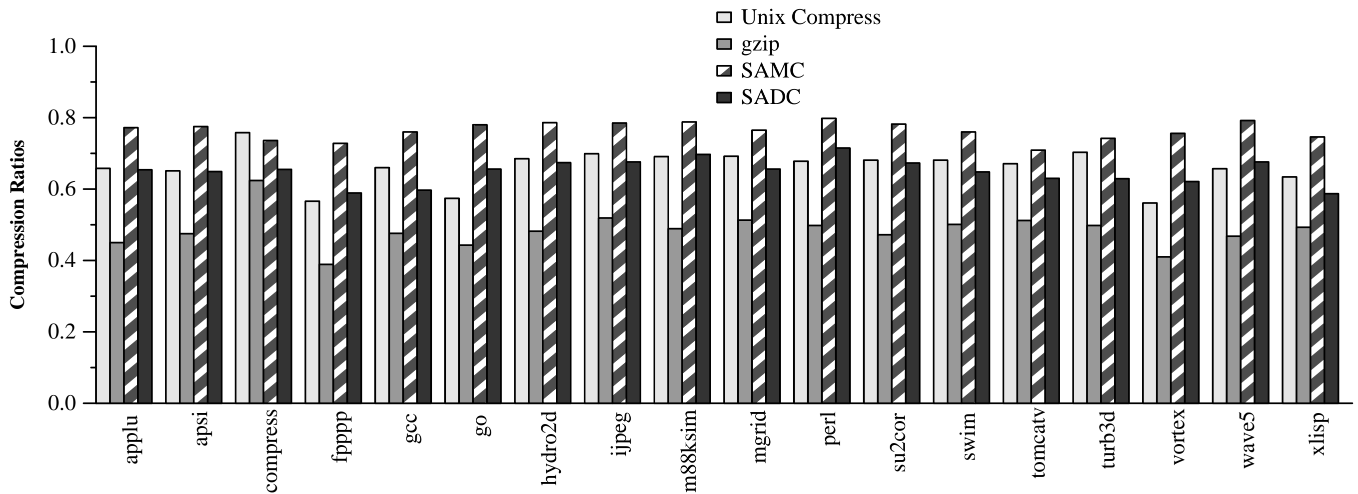

Figure 8: Compression results for Pentium Pro

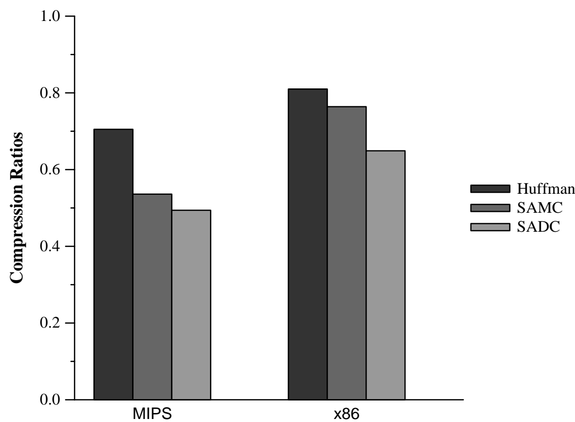

Figure 9: Instruction Compression Algorithms

show significant improvement over previous attempts. Our first method is targeted for RISC instruction sets with fixed-sized instructions and can work for any such architecture. Compression ratios are comparable to UNIX compress. Our second method works on a specific program and instruction set. As a result, it can achieve significantly better compression. Furthermore, as it is a dictionary method, it allows for fast hardware implementations. Its performance is much better than previous instruction compression schemes and is competitive with some file-oriented compression algorithms.

There are several areas that need further work. Some research can be done on how to generate the best Markov model given a subject program for compression. Finally, further research could be made into different and faster decompressor implementations.

Acknowledgments

This work has been supported by the DARPA Small Memory program.

References

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

[2] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, Vol. 29(No 4):pp 320–330, 1986.

[3] G. Cormack and R. Horspool. Data compression using dynamic markov modelling. *The Computer Journal*, Vol. 30(No 6), 1987.

[4] D. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*, volume Vol. 4D, pages pp 1098–1101, September 1952.

[5] M. Kozuch and A. Wolfe. Compression of embedded system programs. *1994 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 1994.

[6] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded dsp processors using data compression techniques. In *Proceedings of the 1995 Chapel Hill Conference on Advanced Research in VLSI*, pages pp 393–399, 1995.

[7] A. Mayne and E. James. Information compression by factoring common strings. *Computer Journal*, Vol. 18(No 2):pp 157–160, 1975.

[8] A. Moffat. Word based text compression. Technical report, Department of Computer Science, University of Melbourne,Parkville,Victoria,Australia, 1987.

[9] J. Rissanen and G. Landon. Universal modeling and coding. *IEEE Transactions on Information Theory*, Vol. 27:pp 12–23, 1981.

[10] J. Storer. NP-completeness results concerning data compression. Technical Report No 234, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J., 1977.

[11] T. Welch. A technique for high-performance data compression. *IEEE Computer*, pages pp 8–19, June 1984.

[12] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the Association for Computing Machinery*, Vol. 30(No. 6):pp 520–540, June 1987.

[13] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. *Proc. 25th Ann. International Symposium on Microarchitecture*, pages pp 81–91, December 1992.

[14] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23(No 3):pp 337–343, May 1977.