A Study of CodePack: Optimizing Embedded Code Space

Avishay Orpaz and Shlomo Weiss
EE-Systems, Tel Aviv University
Tel Aviv 69978, ISRAEL

ABSTRACT

CodePack is a code compression system used by IBM in its PowerPC family of embedded processors. CodePack combines high compression capability along with fast and simple decoding hardware. IBM did not release much information about the design of the system and the influence of various design parameters on its performance. In our work we will present the system and its design parameters and investigate how each affects its performance on the compression rate and decoder complexity. We also present a novel efficient algorithm to optimize the class structure of the system.

Keywords: Embedded Systems, CodePack, Code Compression, Optimization, Embedded Software

1. INTRODUCTION

Driven by an expanding market for consumer electronics and communications equipment, embedded software is becoming increasingly complex [1, 2]. In high-end embedded products, 32-bit microprocessor cores provide the computing power needed to run complex algorithms in real-time. Timely development of such complex and large embedded applications requires the use of high-level languages and compilers instead of manually crafted assembly code. From the system point of view, larger applications and compiled code are both factors that add up to a requirement for larger instruction memory.

Another factor is systems software. Currently, many embedded products use real-time operating systems with modest memory requirements, typically in the range of 10KB to 100KB. Embedded versions of Linux and Windows are becoming increasingly popular [3] in high-end 32-bit applications and in products that do not have tight real-time requirements, such as set-top boxes and networked game consoles. Scaled down versions of Linux or Windows NT may require a few megabytes of memory.

The available instruction memory space may be better utilized by encoding embedded software in a compact format. This cost effective approach received a lot of attention in recent years. In this paper we present a detailed study of CodePack – the code compression technology implemented in the IBM PowerPC 405 embedded microprocessor. We look at several design parameters and investigate their performance impact.

CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.

1.1 Related Work

Several approaches have been used to produce compact code. Thumb [4] and MIPS-16 [5] offer extended instruction sets that include short instructions for embedded applications. The use of short instructions adds minimal run-time overhead, but applications compiled for ARM or MIPS must be re-compiled to take advantage of the extended instruction set. A second approach is to reduce the size of the compiled code by generating a custom instruction set, matched to the characteristics of the compiled program [6, 7]. The custom instructions are interpreted at a speed slower by a factor of 2-4 relative to compiler generated code [8], or a tailored decoder may produce the processor's internal signals directly [9].

The third and final approach we discuss here is to compress embedded instructions while maintaining the ability to quickly decompress them, with minimal impact on code execution speed. An important requirement is random access decompression, which can be implemented by modifying the embedded microprocessor core to directly access the compressed instruction memory [10], or by translating addresses produced by the processor to addresses in compressed memory via a block address translation table [11]. The random access requirement limits the choice of compression methods. Prefix coding is an obvious choice, but other compression methods have been also proposed, including arithmetic coding [12], and dictionary compression methods [13, 14].

The simplest prefix coding is based on an order-0 (or context-free) model. Frequent symbols are assigned shorter codes than symbols that occur less frequently. An order-1 fixed-context model uses a single preceding symbol to determine the probability of the next symbol. For an alphabet of n symbols an order-1 model requires a codebook of size n^2. An interesting compromise implemented in [10] is to use a full codebook based on context-free probabilities for all symbols, and an additional small codebook based on the order-1 model for selected symbols.

CodePack, introduced by IBM [15, 16] in 1998, is a prefix coding method used to compress the embedded instruction memory. It takes, however, a different approach for selecting the symbol alphabet and for limiting the codebook size (see Section 2). An evaluation of CodePack [17] provides results

on the performance penalty due to the additional delay incurred by decompressing instructions before execution, and shows that a performance gain is sometimes achievable because compression shortens the transfer time of instruction blocks. The research reported in [17] does not consider, however, varying CodePack design parameters (such as the number of classes and their coding), and does not investigate how these design parameters affect the compression performance.

1.2 Paper Overview

The CodePack compression scheme, its design parameters, and their effect on compression performance (the CodePack system also includes provisions for address translation – these are not discussed here) is the target of this study. We begin in the next section with an overview of CodePack. In Section 3 we present an efficient algorithm to determine the number of symbols in each class. After describing our experimental setup in Section 4, we present the results in Section 5. We summarize the work and draw conclusions based on it in Section 6.

2. OVERVIEW OF CODEPACK

The CodePack compression scheme, replaces a fixed length instruction code with a variable length compression code. To achieve compression in such a scheme, it is required that more probable instructions will be assigned shorter codes than less probable instructions. One widely known method to build such code is the Huffman method [18].

In CodePack, every instruction is fitted into one of N groups that IBM calls *classes*. Each class has a fixed length and is composed of two fields: the first is the *prefix* – a short, variable length code that identifies the class; the second is a fixed length (for each class) field called *index* that selects a specific instruction out of its class. One more special class exists for instructions that are not compressed – *literals*. These instructions are copied after a prefix that identifies them.

The main advantage of the scheme is that the code length is known by decoding only a few bits, so the next code can be fetched while the first code is still being decoded.

When designing a CodePack-like compression system, several design parameters should be considered:

- *Number of classes (N)* – More classes allow more flexibility in selecting a proper compressed codeword length to instructions according to their probability. However, too many classes will require the prefix to have more bits – which translate to longer decode time and more complex decoder.
- *Codebook size (D)* – All the instructions that are not literals must be stored in a codebook. In order to allow fast decompression, the codebook is usually stored in a fast memory, near the decompression logic. This fact poses severe limit on the size of the codebook. In IBM's implementation the codebook is 1842 byte long.
- *Class Structure* – The class structure defines how many symbols are coded in each class. The class structure is a function of N and D, and must be built to maximize compression rate.
- *Alphabet* – In order to be entropy compressed, the bitstream of the object code must be divided into symbols of fixed length. Large alphabets carry more information, but are hard to handle. IBM divides each 32 bit instruction into two 16 bit halves, each half is compressed independently using its own probabilities.
- *Statistical Method* – A statistical method is used to determine which of the symbols are most likely to occur. IBM mentions nothing about this issue.

3. AN OPTIMIZATION ALGORITHM

To achieve good compression using the CodePack algorithm, a key problem that must be considered is the class structure problem – determining the number of symbols in each class. It is obvious that the brute force approach is an $O(2^N)$ algorithm (N is the number of classes), so applying it for large number of classes is not practical. In this section we present an efficient algorithm to perform this task.

3.1 Definitions

Assume we have a message composed of K different symbols

$$\Sigma = \{S_1, ..., S_K\}. \quad (1)$$

Each symbol occurs f_k times in the message, and it is assumed that Σ so that $f_k \geq f_{k+1}$. We compress it by dividing S into $N+1$ subsets or *classes*

$$C = \{C_1, ..., C_N, C_{N+1}\}. \quad (2)$$

Each class contains c_n symbols, and must be an integral power of two[1], with the possible exception of the last one. To encode a symbol that belongs to class C_n we need a prefix to identify the class and an index having $\log_2 c_n$ bits. This code is actually only a reference to a codebook translating the prefix+index pair into a symbol, and the class C_n will require a total of $c_n \cdot B$ bits in that codebook (B is the number of bits originally used to represent each symbol in Σ). Prefix codes, being relatively few, are encoded using some standard entropy coding such as Huffman or arithmetic code. Said [19] showed that we can separate the problem of prefix and index coding [2] so we will concentrate on the index here. Finally, the symbols are allocated consecutively – the first c_1 symbols belong to class C_1, the next c_2 symbols belong to class C_2, and so on. The last class C_{N+1} is the literal class, which contains all the symbols that do not belong to any other class. Each symbol in the literal class is copied to the compressed message as-is, thus requiring B bits in the compressed stream (none of them are in the codebook).

3.2 Algorithm Description

Using the definitions above, we can now define the optimization problem formally:

Find a set $\{c_1, ..., c_N\}$, which defines a class structure, so that when applied to encode a message over Σ, shall give the shortest possible compressed code.

[1] In order to prevent unused indices.

[2] The proof relies on the assumption that the prefix is ideally entropy coded. In the case of Huffman coding, this assumption does not hold. However, we will continue to use it under the assumption that the prefix coding is "close enough" to the ideal.

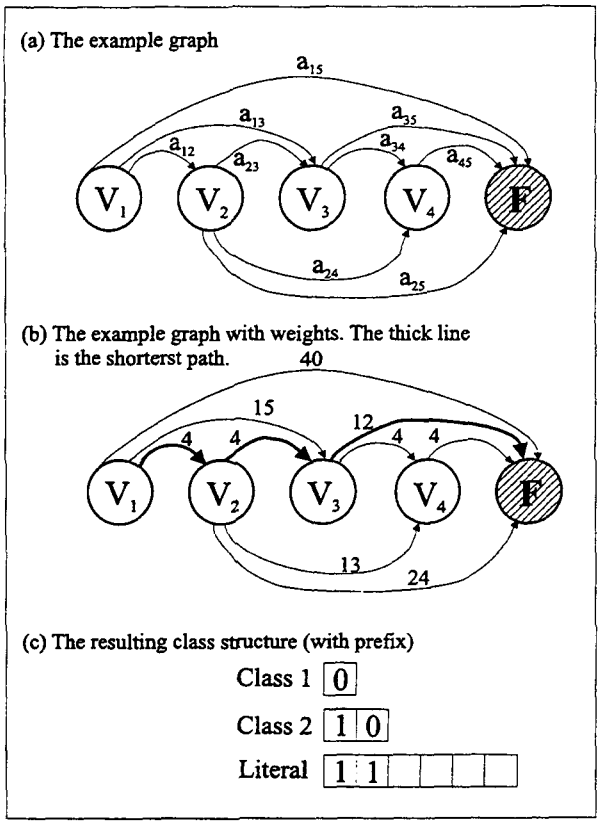

Figure 1: Example graph described in Section 3.3.

To solve this problem we introduce a graph $G = (V, A)$ having $K+1$ nodes – one node corresponding to each symbol, and one additional, *final* node.

$$V = \{V_k \mid 1 \le k \le K+1\} \tag{3}$$

The nodes are connected by arcs of two types – final and non-final.

$$A = A^{\mathrm{NF}} \cup A^{\mathrm{F}} \tag{4}$$

Non-final arcs are defined as:

$$A^{\mathrm{NF}} = \{a_{i,j} \mid \forall i,j, \quad K \ge j > i \text{ and } (j-i) \text{ is an integral power of 2}\} \tag{5}$$

Final arcs are defined as:

$$A^{\mathrm{F}} = \{a_{i,j} \mid \forall i,j, \quad j = (K+1)\} \tag{6}$$

Each arc has a weight:

$$w_{i,j} = \begin{cases} (\sum_{l=i}^{j-1} f_l) \cdot \log_2(j-i) + (j-i) \cdot B & \text{if } a_{i,j} \in A^{\mathrm{NF}} \\ \sum_{l=i}^{K} f_l \cdot B & \text{if } a_{i,j} \in A^{\mathrm{F}} \end{cases} \tag{7}$$

These definitions assert that an arc $a_{i,j}$ that extends from node i to node j corresponds to a class containing the symbols from S_i to S_{j-1}. The weight of the arc is the number of bits such class would have required in the compressed message. Note that this weight includes the number of bits required in the codebook. The arcs that extend from any node i to the final node correspond to a literal class that begins at symbol i.

The message is encoded using $N+1$ classes (of which N are non-literal and one is literal), so its length can be expressed as a sum of weights of arcs, starting with the first node, passing through $N-1$ more nodes and ending at the final node (total $N+1$ nodes).

Let us now write the length (in bits) of the compressed message:

$$L^C = \sum_{l=1}^{N+1} w_{n(l-1),n(l)} \tag{8}$$

$$n(l) = \begin{cases} 1 + \sum_{r=1}^{l} c_r & l \ne 0 \\ 1 & l = 0 \end{cases} \tag{9}$$

The c_n used in the last definition are the length of the n'th arc in the path, and are exactly the values we seek to optimize, subject to the requirement for minimum weight.

Optimization can be done using standard method known as *successive approximation* described in detail in [20]. In this method we start by finding the shortest path (weight) from the first node to each node using one arc only, and then in each l'th step we find the shortest path using $l+1$ arcs. Usually we will continue the process until the absolute shortest path is found, but in our case we will stop after $N+1$ steps. We will then have the shortest path starting from node 1, using $N+1$ arcs and ending at the final node. Each arc corresponds to one class and their length is an integral power of 2 by definition. The last arc corresponds to the literal class and its length is not constrained. The weight of the path, which corresponds the length of the compressed message, is minimal.

3.3 Example

The message to be encoded is `ACAAACBDCB`, using $N = 2$ classes. Each symbol is originally represented by $B = 4$ bits. The frequencies of the symbols `A, C, B, D` in this message are $f_1 = 4, f_2 = 3, f_3 = 2$, and $f_4 = 1$ respectively.

The symbols are listed in non-increasing frequency order (`A, C, B, D,`), and represented in Figure 1 by nodes V_1, V_2, V_3, V_4 respectively. As shown in Figure 1a, with the exception of the final arcs, only arcs whose length is a power of 2 are included in the graph. The weight of each arc, calculated using equation (7), is shown in Figure 1b.

For the desired number of $N = 2$ classes, the algorithm selects the optimal path marked with heavy arrows. The length of the arrows in the path determines the number of symbols in each class. Thus Class 1 and Class 2 each consists of a single symbol. The final arc represents the remaining two symbols, assigned to the Literal Class. The classes may be encoded as in Figure 1c.

In the Literal class the 4-bit symbol is appended to the class code. The compressed message length is 28 bits instead of 40 bits in the original message.

3.4 Algorithm Summary

Following are the main steps of the algorithm. Given an alphabet of K symbols that is to be partitioned into N classes and one additional literal class, the algorithm selects

an optimal class structure (i.e., the number of symbols in each class).

1. Construct a graph that consists of K nodes, one node corresponding to each symbol. The nodes are ordered in non-increasing symbol frequency order. Add one additional, final, node, for a total of $K+1$ nodes.
2. Add to the graph all non-final and final arcs as defined by equations (5) and (6).
3. Mark each arc with its weight as defined by equation (7).
4. Using successive approximation, determine the optimal (minimal weight) path starting at node 1, ending at the final node, and having exactly $N+1$ arcs. The optimal path defines the desired class structure.

3.5 Codebook Limit

As mentioned earlier, the cost of the codebook is high in terms of chip space, so we might be interested in limiting its size.

The algorithm, by its essence, selects an optimal set of arcs out of some final set of arcs,which define all the possible options. To prevent selection of options that are not desired, we have to remove the arcs that represent them. Let us remember that a final arc defines the literal group, that is, all the nodes (symbols) that are copied as-they-are and do not require codebook space. A final arc $a_{j,K+1}$ in the optimal path means that all the symbols which are represented by the nodes V_i, $i \geq j$ will be literals. By deleting all the arcs $a_{i,j}$, $i > D$ we assure that the optimal path will include a final arc starting at V_D at most, thus implementing a codebook limit of D symbols ($D \cdot B$ bits).

3.6 Implementation

The implementation starts with the definition of three arrays: `f[k]` which holds f_k; `u[l]` is the shortest path from the first node to the l'th node; and `trace[l,j]` is the number of node through which the shortest path to node `j` in the `l`'t step passes. We initialize `u[l]` with the $w_{1,l}$ according to (7) which is the shortest path from the first node to the l'th node using one step only. If there is no arc between V_1 and V_l, `u[l]` should be given "infinite" value (in practice `MAXINT` can be used).

Next, we run the following loop:

```
for l=2 to N+1
  for j=2 to K
    for i=2 to K
      u[j] ⇐ min(u[i]+ w_{i,j})
      trace[l,j] ⇐ i for which u[j] is minimal
```

After completing, `u[K+1]` will be the value of the shortest path (which is the length of the compressed message without the prefixes). `trace[l,j]` can help us recover the nodes through which that path has passed and that will give us the class structure.

3.7 Complexity

From the analysis of the algorithm given in Section 3.2, we can simply derive that the time complexity of the algorithm is $O(N \cdot K^2)$. Inspecting carefully the structure of the arcs in the graph, we find that the number of arcs going into node V_k is at most $\log_2 K$, so by removing unnecessary comparisons from the inside of the loop, the time complexity can be reduced even further to $O(N \cdot \log_2 K \cdot K)$.

The space complexity is limited by the size of the `trace` array, and it is approximately $O(N \cdot K)$.

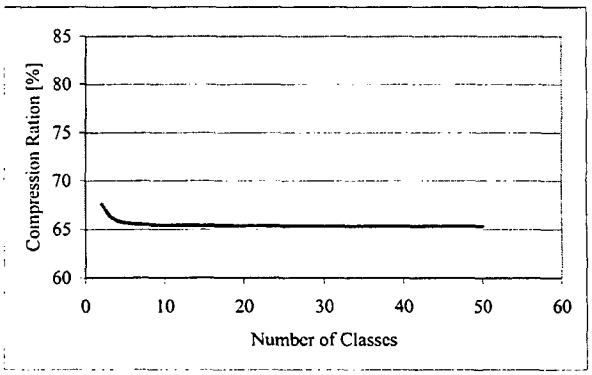

Figure 2: Compression versus the number of classes, without codebook limit.

4. EXPERIMENTAL DESIGN

All our tests were performed on 25 files taken from the SPEC2000 benchmark suite and compiled for Alpha AXP 21264 [21]. From each object file, the instruction part (`.text` section) was extracted and converted to plain binary form, on which our tests where performed.

For each file we built an optimal class structure using each model and the tested parameter set, and then calculated the compressed code size and compression ration. It must be noted here, that in the CodePack system there is a single class structure that serves all the code to be compressed. This fact may lead to slightly lower compression ratios on a real system than described here.

5. RESULTS

5.1 Number of classes

The first parameter we checked is the number of classes vs. compression ratio. The results are shown in figure 2. From the graph it is clearly seen that allowing to have more classes improves the compression ratio, until a certain limit from which more classes have nearly no effect. The limit is in the region of 8 classes. The results shown here were done without codebook limit.

5.2 Codebook Size

We now want to check the effect of codebook limit on attainable compression. N is fixed on 8 and the codebook limit start from 64 to 16K symbols. Figure 3 shows the result. The dashed line shows the compression ratio without codebook limit. Using bigger codebook can result in better compression ration, but doubtfully many applications can allow codebooks of a few kilobytes. We used $N = 8$ in this test.

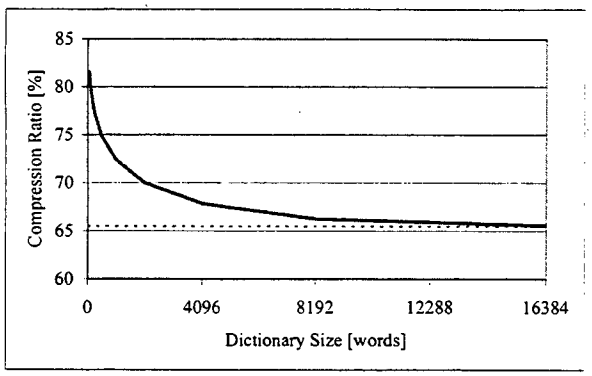

Figure 3: Compression versus codebook size for N = 8 classes.

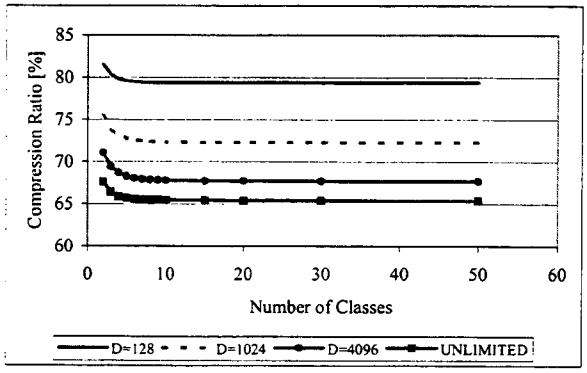

Figure 4: Compression versus the number of classes for several codebook sizes (D).

Figure 4 shows the effect of both N and D on a single grid. It is clear from that figure that increasing the codebook limit has greater effect on compression than adding more classes.

5.3 Alphabet

The alphabet model is the way the instruction bitstream is divided into symbols to allow probabilistic properties extraction. We tried several alphabet models which are listed below.

- *Model 1* – The first model is the model used by the original CodePack. Every 32 bit instruction is divided into two 16-bit symbols, and statistics is calculated on each part independently. Then, an optimal class structure is obtained for each part and the code is being compressed. The parameters used here were $N = 10$ (IBM uses $N = 6$) and $D = 512$ symbols for each part.
- *Model 2* – In model 2, the division to two halves is eliminated. The bitstream is divided into 32-bit symbols and a probability is attached to each symbol. Then, the optimal class structure is determined and the compression rate is calculated. The parameters used here are $N = 10$ and $D = 1024$ symbols.

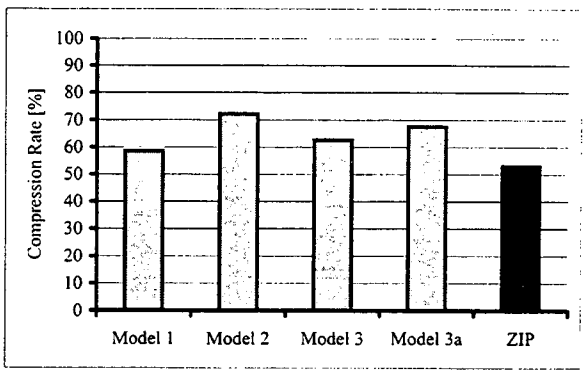

Figure 5: Alphabet Models. Model 1 is CodePack. ZIP is included for reference. The remaining models are defined in the text.

This model can be implemented to be faster than the previous model, since only one step is required to obtain a 32-bit instruction symbol, rather than two steps in the previous model.

- *Model 3, 3a* – In this model, we have tried to improve IBM's method by coding a few of the most probable 32-bit instruction as a whole, while the rest are coded as two 16-bit halves, as in the original scheme. We must introduce here a new design parameter F, the number of instructions coded as 32-bit symbols. Each of these instructions is coded as a single item class, i.e. prefix only. Model 3a is similar, except for the way the 16-bit part probabilities are calculated. In this model, a single probability table is calculated for both the low part and the high part, and a single decoder is used for both parts.

  A decoder for this model may be slightly more complex than the decoder for IBM's model, since it has to deal with variable length output symbols.

Figure 5 shows the results for the different models. Model 1 – which was used by IBM in their system – provides the best compression rate of all the models tested. Also shown on the graph is the code compressed by the commercial compression utility ZIP. This number is given as a reference.

5.4 Literal Compression

In the original CodePack scheme, the literals are copied and not compressed, using actually more bits than used by that symbol in the uncompressed stream. We have tried to improve this situation by compressing them. Each literal is divided into 4 nibbles (4 bit) which were Huffman encoded (using a single encoder for the whole file).

Figure 6 shows the results. Two alphabet models were used – model 1 and model 3a as described in the previous section.

The decoder for these models will be more complex because it has to deal with the decompression of the literals. Its performance may suffer severely, since the main advantage of the original CodePack scheme – knowing the compressed symbol length before completely decoding it – is lost.

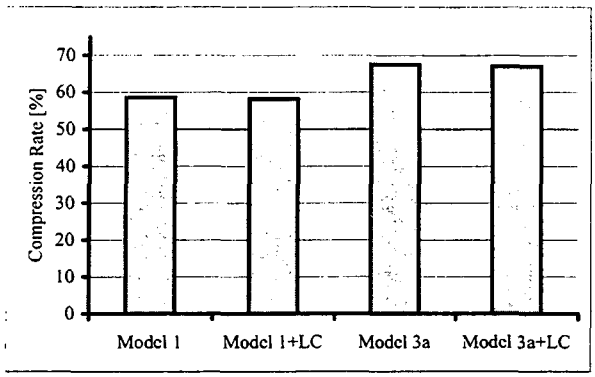

Figure 6: Literal Compression. Two of the models with the literals not compressed (Model 1 and Model 3a) and with Huffman encoded literals (Model 1+LC and Model 3a+LC).

6. CONCLUSIONS

The main advantage of class-based coding is that the code length is known by decoding only a few bits. This provides both simple decoding and good performance because several symbols may be decoded in parallel. To design class-based coding, one must partition the symbol alphabet into classes, according to the symbol frequencies. We have introduced an algorithm that selects an optimal class structure for a given number of classes. The algorithm complexity is $O(N \cdot \log_2 K \cdot K)$ for N classes and K symbols.

Using the new algorithm, we have compared the compression performance of several alphabet models as a function of the number of classes and the codebook size. For all codebook sizes, the compression ratio flattens out after $N = 10$ classes. From the models and parameter sets we tested, the IBM CodePack model gives good compression rates while retaining simplicity. It is possible to improve the compression rates slightly, but the cost in hardware complexity and performance loss might be considerable.

As for future work, in the algorithm described here we have given equal weight to storage bits needed for the compressed message and for the codebook. In practice, the codebook is likely to be stored in high-speed ROM, while the compressed program might be stored in slower but denser flash memory. An extension of this work would be to study the impact of different memory technologies for codebook and program storage.

7. REFERENCES

[1] A.C. Lear. Shedding light on embedded systems. *IEEE Software*, 16(1):122–125, January/February 1999.

[2] E.A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(9):18–26, September 2000.

[3] B. Santo. Embedded battle royale. *IEEE Spectrum*, 38(12):36–41, December 2001.

[4] J.L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4), March 1995.

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

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

[7] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, 1999.

[8] C.W. Fraser and T.A. Proebsting. Finite-state code generation. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 270–280, May 1999.

[9] S.Y. Larin and T.M. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. In *Proc. Int'l Symp. on Microarchitecture*, pages 82–92, November 1999.

[10] A. Miretsky, A. Ben-Efraim, V. Sukonik, A. Saper, A. Ginsberg, R. Natan, and A. Dor. RISC code compression model. In *Proc. Embedded Systems Conference*, Chicago, Illinois, March 1999.

[11] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. Int'l Symp. on Microarchitecture*, pages 81–91, 1992.

[12] H. Lekatsas and W. Wolf. Random access decompression using binary arithmetic coding. In *Proc. Data Compression Conference*, pages 306–315, March 1999.

[13] S. Liao, S. Devadas, and K. Keutzer. A text compression based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, January 1999.

[14] G. Araújo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Transactions on VLSI Systems*, 8(5):530–533, October 2000.

[15] J.L. Turley. PowerPC adopts code compression. *Microprocessor Report*, 12(14):26–29, Oct 1998.

[16] M. Game and A. Booker. *CodePack: Code Compression for PowerPC Processors.* International Business Machines (IBM) Corporation, 1998.

[17] C. Lefurgy, E. Piccininni, and T. Mudge. Evaluation of a high-performance code compression method. In *Proc. Int'l Symp. on Microarchitecture*, pages 93–102, Haifa, Israel, November 1999.

[18] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9):1098–1101, September 1952.

[19] A. Said and W.A. Pearlman. Low-complexity waveform coding via alphabet and sample-set partitioning. In *Visual Communications and Image Processing '97, Proc. SPIE Vol. 3024*, pages 25–37, Feb. 1997.

[20] E.L. Lawler. *Combinatorial Optimization.* Holt, Rinehart and Winstone, July 1976.

[21] C.T. Weaver. Spec 2000 Binaries. *www.eecs.umich.edu/~chriswea/benchmarks/spec2000.html.*