Field-Based Microcode Compression for Area and Power Savings

Author 1 and email

Author 1 and email

Affiliation

Abstract

Microcode is a way of using programmability of (micro) architectural structures to enhance functionality and to apply patches to an existing design. As more features get added to a CPU core, the area and power costs associated with microcode increase. Code density is an important issue in memory constrained systems, and compression techniques have been developed to reduce system area and power consumption. Thus, it is desirable to apply compression techniques to microcode.

Microcode poses unique challenges for compression due to the long instruction format, the hand-coded nature of the programs and to stringent performance requirements that require fast decompression. This paper describes a technique for microcode compression that achieves area and power savings, while presenting a streamlined architecture that enables high throughput within the requirements of a high performance CPU. The key idea is to break each microcode word into fields, and take advantage of redundancy in these fields. The codewords for the compressed fields are chosen in order to reduce power consumption by minimizing ROM loading. The paper presents results for microcode compression on a commercial CPU which demonstrates area savings in the order of 32% and estimated power savings of more than 25%.

1 Introduction

Recent trends have migrated more and more advanced functionality to the microcoded portion of a CPU core. For example, a state-of-the art product processor design contains more than 22,000 lines of microcode. This number is due to increase as new technologies such as protection, virtualization and management assistance are added. Designers have the choice to migrate functionality to microcode: a given function may be achieved via dedicated hardware, at an associated hardware and power cost with possible performance benefits, or by microcode enhancements. Typically, functionality that is less performance-critical is relegated to microcode. Microcode bloat causes increased costs in terms of die area and associated power consumption.

The cost for microcode storage is particularly acute in cores for applications requiring small footprint dies and reduced power consumption. These are embedded applications and CPUs that contain arrays of cores in the same die. A recent design targeted at low power and small footprint has estimated the area costs (and associated power consumption) to approach 20% of the die.

Code density is an important issue in embedded systems. The size of flash memory directly affects system cost and therefore program memory is often constrained. Processors with higher code density allow more programs to be supported at lower overall system cost. One solution to address issues with code size in such systems is to apply compression techniques [WolfeChanin92, Breternitz97, Araujo98].

Similarly, one may use compression to reduce the area of microcode ROM (UROM). The idea is to store the microcode in a transformed representation (compressed) and decompress during execution. This enables savings in UROM static size. By judicious design of the decompression mechanism, it is possible to avoid and minimize the performance impact of this approach.

The contributions of the paper are summarized as follows:

- A technique to compress microcode by identifying sub instruction fields suitable for compression;
- A decompression architecture that reduces the performance impact of microcode compression and enables its application to a high performance CPU;
- A technique to reduce power consumption by reducing UROM loading that takes advantage of the compression mechanism.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 discusses microcode compression and Section 4 presents a field-based microcode compression technique. Section 5 discusses area savings and Section 6 describes a power reduction technique by reducing UROM loading. Section 7 concludes the paper and discusses future work.

2 Related Work

There have been several efforts to reduce code size via code compression. We assume that the best effort to remove redundancy by software means has already been applied. These range from the use of classical compiler optimizations such as strength reduction, dead code elimination, tail merging, and common sub-expression elimination [Debray-02]. The compiler optimization called Procedural Abstraction [Fraser-84, Liao-96] is also shown to reduce code size.

In 1992, Wolfe proposed the Compressed Code RISC Processor (CCRP) [WolfeChannin99] using Huffman encoding to compress MIPS R2000 instructions. It achieved a 70% compression ratio with negligible performance loss. To fetch the (variable-sized) compressed words from memory, a translation table "Line Address Table (LAT)" is used. Breternitz/Smith [Breternitz97] enhances on this architecture by pre-processing the program such that I-cache miss addresses point to the fetch address of the compressed program, avoiding the need for a LAT. *Note*: we define compression ratio as the fraction of the compressed program over the original program, taking into account the cost of auxiliary look up tables and structures, so a smaller value is better.

Lefurgy [Lefurgy97, Lefurgy99] uses a dictionary to store repeated sequences of instructions in the code. It assigns codewords to these sequences and mixes codewords with uncompressed instructions in the program. The compression ratio ranged from 60%, to 70% for the PowerPC, ARM and i386 architectures.

CodePack [GameBooker98] was designed by IBM for the PowerPC processor using two dictionaries, one for each half (16 bits) of the instructions. The instructions are encoded as two indexes and two tags to specify the index size. They translated the addresses using a Compression Index Table. Their final compression ratio ranges from 60% to 65% and performance from 10% of slowdown to 10% of speedup.

Araujo [Araujo04,Araujo00,Araujo98] presented three methods for code compression: Pattern Based, Tree Based and Instruction Based Compression, achieving a compression ratio of 61.3%, 60.7% and 53.6%.

Most of above techniques are applied to 32-bit processor architectures. Compression of microcode instructions containing many operations is more challenging. The closest similarities are in works compressing DSP programs and VLIW programs. Xie et. Al. [XieWolfe01] developed VLIW code compression techniques, achieving 70% compression rate.

3 Microcode Compression

Microcode has some unique characteristics that challenge compression. First, it uses a long instruction format comprised of multiple independent fields. Multiple operations may be specified in a (micro) instruction and the number of alternative encodings grows combinatorially. This makes techniques like Instruction-Based Compression [Araujo04] less likely to succeed. Second, microcode is usually hand-coded and crafted for performance. As such, it is less likely to contain repeated code patterns such as found in code generated by compilers.

Further, in high-performance processors, it is necessary to provide a steady stream of instructions to the micro engine. So, the decompression engine must have low latency and enable high throughput instruction flow. For such reasons, it is desirable to avoid compression techniques that utilize variable-length codes (e.g., Lempel-Ziv, used in prior art [Craft-98,Ziv-77]), as this approaches require a more complex decoder, incurring costs in area and power, in addition to added design and verification time.

This paper proposes a two-level organization for the compressed microcode structure. The basic idea is to identify the set of unique bit patterns in the microcode program and store these in a table. The actual microcode UROM consists of ‘pointers’ into this table. Two key advantages of this two-level organization are that 1) it enables **pipelining**, which reduces the performance impact of decompression for long microcode sequences, and 2) it allows use of **fixed-sized** ‘pointers’ which facilitate and simplify the decompression hardware. These are the basic ideas in Instruction Based Compression [Araujo04].

An improvement of the above idea is to split the microcode word into fields such that the number of unique patterns for each field is minimized. For example, one approach uses the microinstruction’s operational fields. Such as opcode, source arguments, destination arguments, and immediate values. A later section describes an alternative method that uses hyperspace distance metrics and K-means analysis to group related microcode columns together.

Figure 1 below illustrates a microcode structure for compressed microcode. Figure 1, left, shows the original uncompressed UROM. Solid rectangles represent the ROM array, and dashed rectangles indicate data. Each microinstruction is accessed in the UROM by its address in that table (“uaddr”). One access to the table produces the whole microinstruction with all its component fields (represented as UOP in the figure). Figure 1, right, is a microinstruction composed of two fields, each being an index in a corresponding table of unique patterns. The vertical bar on the compressed UROM array indicates the fact that each data word contains two such pointers. There are two unique pattern tables, one for each field.

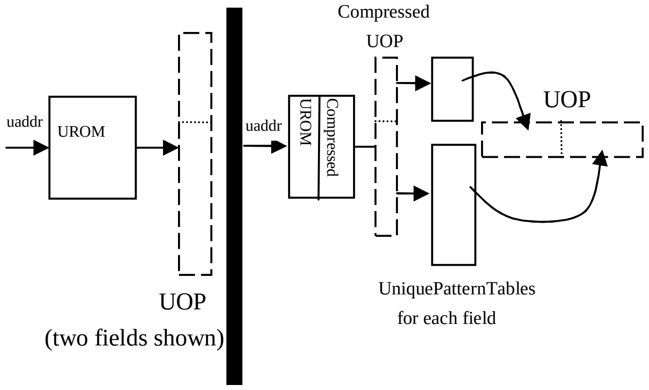

Figure 1 –UnCompressed/Compressed UROM mechanism

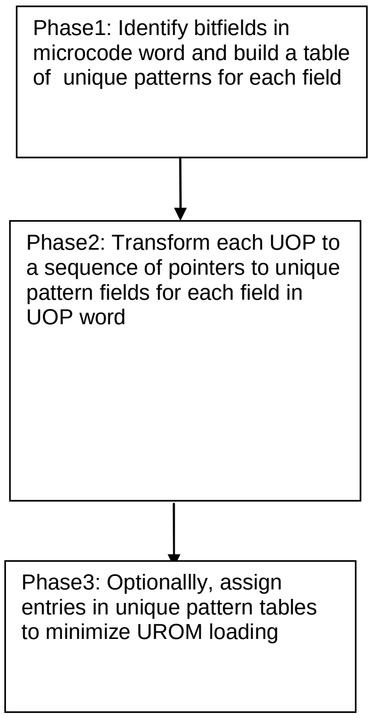
Phase1: Identify bitfields in microcode word and build a table of unique patterns for each field
Phase2: Transform each UOP to a sequence of pointers to unique pattern fields for each field in UOP word
Phase3: Optionallly, assign entries in unique pattern tables to minimize UROM loading

Figure 2. Flowchart of the microcode compression algorithm

Figure 2 illustrates the steps to generate compressed microcode. First, we identify bit fields in the instruction such that the number of unique patterns is minimized, then the microinstruction word is constructed by substituting each field with pointers to the corresponding fields' value in the table of unique patterns for that field. Finally, the tables are organized such so that pointer values minimize UROM loading. This step is described in a later section.

4 Microcode Compression Techniques

4.1 Field-Based Compression

The basic idea for microcode compression is to identify a set of unique bit patterns that compose the microcode word and store these in a table. The microcode programs stores only a (short) *unique id* for each pattern in the original microcode word sequence, as shown in Figure 3. In this figure, uaddr is the address of a microcode word. In the uncompressed form, the uaddr directly access the UROM to fetch a microcode word. In the compressed form, the unique microcode words are stored in the "unique insts" table, and only the index into the table is stored in the original place of the microcode word. Assume the original UROM has N microcode words each with L bits, and there are a total of M unique microcode words. The original UROM takes N*L bits and the compressed UROM takes only $N*\log_2(M)+M*L$ bits. For N=20000, M=12000, and L=70, the compressed UROM uses 1140000 bits while the original UROM uses 1400000 bits. This is about 19% reduction in bits. (Note: in this discussion we use the number of bits in the UROM as an estimate for its area requirements. Section 5 presents experimental results from layout estimates showing that reductions in actual UROM size are in line with this estimate).

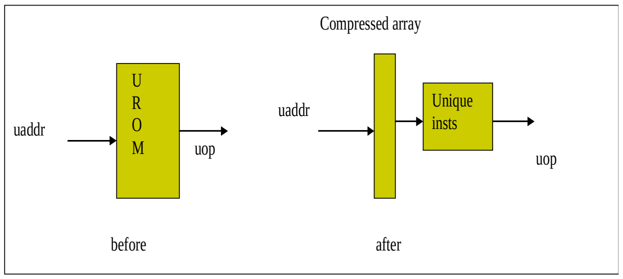

Figure 3. Basic microcode compression idea

An improvement of the above idea is to split the microcode word into a number of sub-words such that the number of unique patterns for each word is minimized. The intuition behind this idea is to take advantage of entropy for each field. For example, even though a microcode word may have, say, upwards of 70 bits, there are fields such as 'opcode' (about 8 bits), in which there is not much variation and in which a few values are dominant. Figure 4 shows an example when each microcode word is split into two roughly equal-sized sub-words. Assume M1 and M2 are the number of unique patterns for the two halves. The original UROM takes N*L bits and the compressed UROM takes only $N*(\log_2(M1)+ \log_2(M2))+M1*L/2 + M2*L/2$ bits. For N=20000, M1=5000, M2=5000, and L=70, the compressed UROM uses 20000*26 + 10000*35 = 870000 bits while the original UROM uses 1400000 bits. This is about 38% reduction in number of bits.

The key observation from the above scheme is that with a proper partitioning of the UROM into subsets of columns, the number of the unique patterns in the partitions is reduced and thus the total area will be reduced. We describe a general clustering technique in the next section.

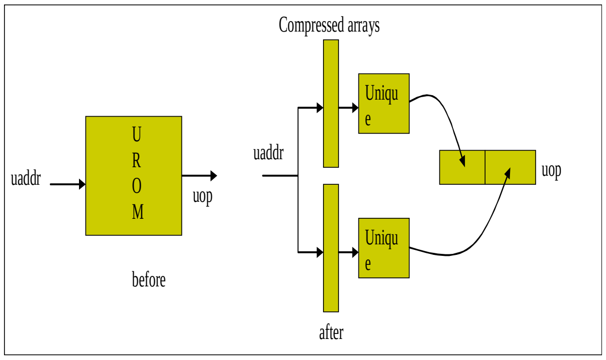

Figure 4. Partitioned compression

4.2 Clustering example

The clustering-based compression selectively groups similar columns into clusters, and goes beyond the simple partitioning of the microcode words into sub-words composed of adjacent bits. For example, Figure 5 shows a simple partitioning of each micro-code word into two sub-words. With this partitioning, the two partitions each have three different patterns and require two bits to index the unique patterns. Using the partitioning method shown in Figure 5, the compressed form needs 10*(2+2)+ 3*3 + 3*3 = 48, about 20% reduction from the original size of 60 bits.

Col1	Col2	Col3	Col4	Col5	Col6
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1

Figure 5. Partitioning method

The clustering-based compression partitions columns that are similar to each other into groups. For example, the sample UROM in Figure 5 may be clustered into the two groups shown in Figure 6, where columns 1, 3, and 5 are clustered into the first group and the columns 2, 4, and 6 are clustered into the second group. With this new clustering, both groups have only two unique patterns and need only a 1 bit index. As a result, the compressed form requires only

10*(1+1) + 2*3 + 2*3 = 32 bits, nearly 50% reduction and a significant reduction vs. the partitioning method.

Col1	Col3	Col5	Col2	Col4	Col6
1	1	1	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
0	0	0	1	1	1

Figure 6. Clustering method

Figure 7 shows how to access the microcode word in the clustering method. There is a new component called "spreader" for each cluster that spreads the unique patterns in a cluster into the appropriate bits in the final microcode word. This spreader is simply a rewiring of the original path that connects the output to the microcode word and should NOT cost any additional die area or power. Although we only show two clusters, this method is not limited by the number of clusters, and we expect sometimes 3 or 4 clusters are possible.

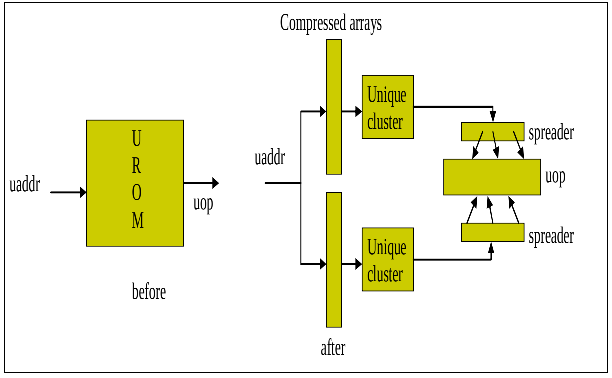

Figure 7. Clustering method

4.3 Clustering problem

A clustering algorithm tries to group the similar columns of UROM into clusters, such that an objective function is minimized. To define the objective function, we define the following items:

L: the number of columns in the UROM

N: the number of bits in each column

K: the number of clusters that the L columns are clustered into

L1, L2, …, Lk: the number of columns in each cluster 1, 2, …, K

M1, L2, …, Mk: the number of unique patterns in clusters 1, 2, …, K

The clustering algorithm tries to find K clusters such that the following objective function is minimized:

$$F = \sum_{i=1}^{K} [N(\log_2 M_i) + M_i * L_i]$$

For the example in Figure 6, N = 10, K = 2, L1 = 2, L2 = 2, M1 = 3, M2 = 3, and F = 10*log2+3*2 + 10*log2 + 3*2 = 32.

This clustering problem is clearly an NP-hard optimization problem. We need to use heuristics to solve it. A general heuristic approach is described in Figure 8. The algorithm iterates from K = 1, to some number X that is less than the number of columns L. For each such K, it first obtains a simple K clusters by equally dividing the L columns into K clusters, and then repeatedly apply heuristics to improve the K clusters until the heuristic termination condition is met. Find, the best clustering among the X clustering results are selected as the final result.

```
For K = 1 to X
    Equally divide the L columns into K clusters
    Repeat
        Heuristic_improvement
    Until heuristic_termination_condition met
Select the K with the best clustering
```

Figure 8. High-level heuristic clustering algorithm

In the following, we present a K-mean based approach to refine the algorithm. Figure 9 shows a k-mean based heuristics that tries to improve the initial clustering by moving columns around until the distance between columns within each cluster cannot be minimized. During each iteration, the algorithm first computes the center for each cluster, then computes the distance from each column to each center, and moves the column to the cluster whose center is closest to the column. The repetition terminates when no column is moved from its current cluster to a different cluster during the current iteration.

```
For K = 1 to X
    Equally divide the L columns into K clusters
    Repeat
        Compute the centers for the K clusters
        For each column, find the center that is closest
to it and assign it to that cluster (if not already in the
cluster)
    Until no column moves to a different cluster
Select the K with the best clustering
```

Figure 9. K-mean based heuristic

Intuitively, the K-mean based algorithm tries to cluster columns that are similar to each other together. Thus the definition of "similarity" or the "distance" between two columns is critical to the algorithm. For the algorithm to identify clusters that minimize our objective function, when the "distance" between the columns in a cluster is small, the number of unique patterns in the cluster must be small. The following definitions of "distance" can be used in the algorithm, where (c1, c2, …, cn) and (d1, d2, …, dn) are two distinct columns.

Hamming distance ((c1, c2, …, cn), (d1, d2, …, dn))

$$= \sum_{i=1}^{n} c_i \wedge d_i \, .$$

Min hamming distance(A,B)

= min(Hamming_distance(A,B), Hamming_distance(A, ~B)), where ~B is the complement of B.

In this paper, we use the standard K-mean algorithm with the Hamming distance. Figure 10 shows the result of the K-mean algorithm applied to production microcode for an actual microprocessor. For this particular processor, the program contains 22,568 microinstructions of 75 bits each. The best clustering is obtained when K=3, with a compression ratio of 61%.

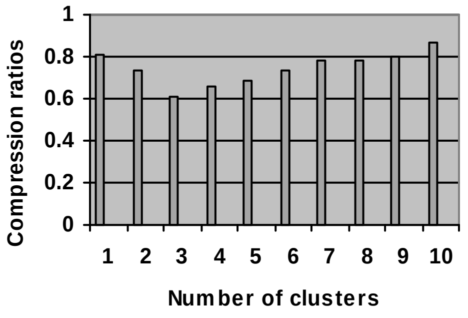

Figure 10. Compression ratios with clusters found by K-mean algorithm

4.4 *Field-Based compression by instruction fields*

In this section we describe application of the field-based compression technique to a production microcode. The program contains 22,568 microinstructions of 75 bits each. The instruction is divided into several fields such as: control of instruction format, opcode, argument designator fields. Argument designators indicate registers, immediate values, branch targets or memory addresses.

To evaluate the automated K-mean clustering technique, we experimented with a manual clustering technique guided by the designer's knowledge of each field. For example we have selected fields that, because of the field semantics, have a reasonable expectation of reduced variation in values. Examples of such fields are the 'opcode' and 'source register 1' fields.

Here we present the experimental results. These are the best results obtained by the designers using knowledge of the expected entropy of each field. The microinstruction word was separated in two parts, respectively with 38 and 37 bits. The first part contains opcode and source argument information fields. The second part contains immediate values and other miscellaneous fields. Of the 22,568 instructions, there are 3359 unique values for the first part (38 bits) and 7356 unique values for the second part (37 bits). The tables containing unique values can be indexed by 12 and 13 bits, respectively. The resulting compression ratio of the hand-clustering technique is 58%. This is better than that achieved by the automated K-mean clustering. However, this is also an indication of potential for future enhancements to our clustering technique. Notice that the automated clustering with K=3 achieved results that are competitive with the best hand-coded results. We intend to investigate enhanced automated clustering techniques, possibly with designer-led 'seeding' of fields to consider clustering first.

5 Die Area Reduction

So far we have discussed compression ratio in terms of the reduction in number of bits stored in the UROM. However on the final layout, actual area reduction may differ from reduction in the number of bits. This is because the memory arrays require a rectangular, regular layout. Imagine, for example, a reduction in a single bit. This may not cause any area reduction because the rectangle area is the same. Unless the reduction in number of bits is enough to change the array dimensions, there may be no noticeable area reduction. Further, there may be several such arrays and their arrangement also affects the actual layout. Finally, the additional control structures and buffers affect the final area.

We applied the previous compression technique to a microcode program and modeled the area for the final layout with help from circuit design engineers. Three cases were considered (the die area is measured in "units" of space):

1. The original UROM with 22,568 entries, each containing a full microcode word. In this technology, the UROM structure contains one array with dimensions 711x800 units, with total area 568,800 units;
2. An organization similar to Figure 1, left, in which a table of unique patterns, considering the whole microcode word, is created. This organization has two ROM arrays; the first array has dimensions 484x214 units and the second array has dimensions 291x1034 units. The total area is 404,470 units;
3. An organization with K=2 (the best hand-developed version) in which the microcode word is broken in two fields. This organization has three arrays, respectively with areas 484x349, 155x286 and 277x634 for a total area of 388,864 units.

The above experiment found an area reduction to 68% of the original area, whereas the best reduction in the number of UROM bits for this program was to 58%. This difference is not unusual, as we discussed above, due to the rectangular form of the arrays. It also demonstrates that our estimates based on bit counts are a good approximation of the actual area savings.

6 Reducing UROM loading

UROM power consumption is determined, in part, by the number of bits set to '1' in the UROM [deAngelSwartzlander97]. Thus, a transformation technique that reduces the number of UROM bits that are set to '1' also reduces its power consumption. Microcode compression may be thought of as a transformation technique.

It is possible to select the bit pattern for the *unique id* of each unique value on a microcode field. The *unique id* is a pointer into the table containing the unique values. We select the bit pattern corresponding to a *unique id* by storing the corresponding pattern in the appropriate entry on the table. For example, the very first entry on the table corresponds to the bit pattern of all zeroes.

As a simple example, assume the field in consideration is the 'opcode' field with only four possible values: ADD, SUB, LOAD, and STORE. Assume a simple program containing twice as many 'ADD' opcodes as 'MUL' opcodes. We reduce the number of bits set to '1' by assigning the bit pattern '00' to 'ADD'.

A simple algorithm for power reduction is as follows: first, identify the unique values. Next, sort the unique values in descending order of frequency and then assign table positions such that the corresponding bit patterns have the least number of '1' bits set. So, the first position, assigned to the highest-occurring pattern is all zeros. Next, all bit patterns containing one bit set are assigned. (If there are *n* unique patterns, the number of bits to identify the pattern is N= ceiling(log2 n). Thus, there are N patterns with only one bit set. Next, patterns with two bits set are assigned and so forth.

This technique has been applied to a production microcode. The microcode consists of 22528 microinstructions of 75 bits each, with a total of 273,264 bits set to 1. Note that microcode has been hand-coded and takes into consideration UROM loading, by reducing the number of bits set to '1' wherever possible. The compressed organization, with the above power reduction algorithm applied, contains 114,549 bits set. For this microcode sequence, our technique is able to further reduce the *total* number of bits set to 1 by about 60%. Analyzing the UROM area savings and UROM load reduction via spreadsheet analysis, we estimate the power savings for this compressed microcode organization to be more than 25%.

7 Conclusion

In this paper we first show potential benefits and challenges for microcode compression. Then we describe techniques to achieve compression while allowing for a pipelined high-throughput design. Also described are algorithms for separating a microinstruction into fields to achieve higher compression. Then a technique to reduce UROM loading for reduced power consumption is presented. The techniques are illustrated by application to production microcode for an actual microprocessor design.

8 Acknowledgements

Anonymous, etc

9 References

[Araujo04] E.Wanderley Netto, R.Azevedo, P.Centoducatte, G.Araujo," Multi Profile Code Compression", Design Automation Conf., June 2004.

[Araujo00]G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Transactions on VLSI Systems*, Mar. 2000.

[Araujo98]G. Araujo, P. Centoducatte, M. Cˆortes, and R. Pannain. Code compression based on operand factorization. In *Proc. Int'l Symp. on Microarchitecture*, pages 194–201, Dec. 1998.

[Breternitz97] Mauricio Breternitz Jr. and Roger Smith. Enhanced compression techniques to simplify program decompression and execution. ICCD, Oct 1997.

[Craft98] A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions, IBM J.Research and Development, vol 42 no 6, Nov.1998

[deAngel,Swartzlander97] Survey of Low-Power Techniques for ROMS, International Symposium on Low-Power Electronics and Design, 1997

[Debray-02] S. Debray, W. Evans, R. Muth, and B. de Sutter. "Compiler techniques for code compression," ACM Trans. on Programming Languages and Systems, pages 378–415, 2000.

[Fraser-84] C. Fraser, E. Myers, and A. Wendt, "Analyzing and compressing assembly code," SIGPLAN Notices, 19(6):117-121, June 1984

[GameBooker98] M. Game and A. Booker. CodePack: Code Compression for PowerPC Processors. International Business Machines (IBM) Corporation, 1998.

[Lefurgy97] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proc. Int'l Symp. on Microarchitecture*, Dec. 1997.

[Lefurgy99]] C. Lefurgy, E. Piccininni, and T. Mudge. Analysis of a high performance code compression method. In *Proc. Int'l Symp. on Microarchitecture*, Nov. 1999.

[Liao-96] S. Liao. "Code Generation and Optimization for Embedded Digital Signal Processors," Ph.D. thesis, 1996. Massachusetts Institute of Technology.

[WolfeChannin92] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. Int'l Symp. On Microarchitecture*, 1992.

[XieWolfe01] Y. Xie, H. Lekatsas, and W. Wolf. Code compression for VLIW processors. In Proc. Data Compression Conference, March 2001

[Ziv-77] J. Ziv and A Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Transaction on Information Theory, 23 (3), p337-343, May 1977.