Clustering-Based Microcode Compression

Edson Borin*, Mauricio Breternitz Jr.†, Youfeg Wu† and Guido Araujo*
*Institute of Computing
University of Campinas - Campinas, SP – Brazil
Email: {borin,guido}@ic.unicamp.br
†Programming System Lab
Intel Corporation - Santa Clara, CA – USA
Email: {mauricio.breternitz.jr,youfeng.wu}@intel.com

Abstract— **Microcode enables 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. A recent Intel internal design targeted at low power and small footprint has estimated the costs of the microcode ROM to approach 20% of the total die area (and associated power consumption). Therefore, 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 the stringent performance requirements that require fast decompression. This paper describes techniques for microcode compression that achieve significant area and power savings, while presenting a streamlined architecture that enables high throughput within the constraints of a high performance CPU. The paper presents results for microcode compression on several commercial CPU designs which demonstrates compression ratios ranging from 50% to 62%.

I. Introduction

Recent trends have migrated more and more advanced functionality to the microcoded portion of a CPU core, such as protection, virtualization and management assistance. Microcode growth causes increased costs in terms of die area and associated power consumption.

The cost of microcode ROM storage (μROM) is particularly critical in cores for applications requiring small footprint dies and reduced power consumption, like embedded processors and CPUs that contain arrays of cores on the same die. A recent Intel internal design targeted at low power and small footprint has estimated the microcode area costs (and associated power consumption) to approach 20% of the die.

One solution to address microcode size issues is to apply code compression techniques [2], [4], [19]. The idea is to store the microcode in a transformed representation (compressed) and decompress it during execution. This enables savings in μROM static size. By judicious design of the decompression mechanism, it is possible to minimize the performance impact of such approach.

Microcode uses a long instruction format comprised of multiple operations and fields. The number of alternative encodings of the operations and fields grows combinatorially. This makes techniques like Instruction-Based Compression [1] 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. Moreover, 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 variable-length encoding compression techniques [1], [14], [19], [20], as these approaches require a more complex decoder, incurring costs in area and power, in addition to longer design and verification times.

The contributions of this paper are summarized as follows:

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

The rest of the paper is organized as follows. Section II outlines related work. Section III discusses microcode compression. Section IV presents a set of clustering-based microcode compression algorithms. Section V shows the experimental results. Section VI discusses area savings. Section VII presents the pipelined decompression engine. Section VIII describes a technique to reduce the μROM loading and Section IX concludes the paper.

II. Related Work

There have been several efforts to reduce code size via code compression. As the microcode is mostly hand-coded, we may assume that the best effort to remove redundancy by means of software 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 [6] to more elaborated techniques, like Procedural Abstraction [8], [16].

In 1992, Wolfe proposed the Compressed Code RISC Processor (CCRP) [19] using Huffman encoding to compress MIPS R2000 instructions. It achieved a 70% compression ratio with minor performance loss. To fetch the (variable-sized) compressed words from memory, a translation table "Line Address Table (LAT)" is used. Breternitz and Smith [4] enhances on this architecture by pre-processing the program such that I-cache miss address points to the fetch address of the compressed program, avoiding the need for a LAT. The use of variable-sized codewords complicates the decoding hardware and makes this technique less desirable for microcode compression. *Note*: we define compression ratio as the fraction of the compressed program over the original program, taking into account the cost of auxiliary tables and structures. Thus the smaller the compression ratio the better.

Lefurgy [14] 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. For microcode, repeated sequences of instructions are less likely due to the many possible long-instruction variants, and the fact that size-conscious microcode programmers combine such sequences in small subroutines.

CodePack [9], [12], [15] 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 have to translate the fetch addresses using a Compression Index Table since the code offset change after the compression. Decompression happens for blocks of 16 instructions at a time, to fill an I-cache line. Their final compression ratio ranges from 60% to 65% and performance from 10% of slowdown to 10% of speedup.

Araujo *et al.* [1] 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% respectively. Again, the complexity of decoding hardware hinders application of this technique to microcode.

All techniques above are used to compress 32-bit program code. Microcode compression has been studied in the early '80s, when ROM encoding techniques and design tricks were used to reduce microprogram size [11]. Another approach to the problem, in those days, was to design algorithms that could efficiently compact operations into microinstructions [7]. These techniques evolved to what is nowadays a set of compiling methods used in code generation for VLIW machines. Only recently, due to stringent processor design constraints, has microcode compression been revisited as a way to reduce μROM size. Unfortunately, not much work has been dedicated to compress μROM code, when compared to its VLIW counterpart. Ishiura and Yamagucchi [13] split VLIW instruction up into fields that are optimally compressed, producing a 46%-60% code reduction. Nam *et al.* [17] divide the instruction stream into opcode and operand fields which are mapped into two dictionaries. This approach delivers compression ratios in the range of 63%-71% for a 4-12 issue VLIW architecture. Xie *et al.* [20] proposed an arithmetic encoding based VLIW compression technique, capable of compressing flexible instruction formats. Their approach results in compression ratios in the range of 70%-80%. In another paper, Xie *et al.* [21] use Markov-based Variable-to-Fixed (V2F) encoding to compresses IA-64 and TMS320C6X code to 56% and 70% of its original size.

Most of the work above demands expensive multi-cycle decompression engines and/or control-logic, and in many cases, the decompression engine is placed between the cache and the main memory, so that the performance overhead is hidden on the cache miss penalty [15]. Since the microcode storage system does not have a cache memory, such approaches are less desirable when decompressing the critical microcode from the μROM.

III. Microcode Compression

The basic idea behind microcode compression is to identify a set of unique bit patterns that compose the microinstructions and to store them into a "dictionary" of unique patterns. The original microinstructions are replaced by pointers to the patterns in the "dictionary" as shown in Fig. 1. In this figure, *uaddr* is the address of a microinstruction. In the uncompressed form, the *uaddr* directly access the μROM to fetch a microinstruction. In the compressed form, the unique microinstructions are stored into the "dictionary" (DIC) and, only the index to the pattern in the "dictionary" is stored into the "pointer array". Assume the original μROM has N microinstructions each with L bits, and there are a total of M unique microinstructions. The original μROM takes $N \times L$ bits and the compressed μROM takes only $N \times \lceil log_2 M \rceil + M \times L$ bits (where $N \times \lceil log_2 M \rceil$ is the "pointer array" size and $M \times L$ is the "dictionary" size). For $N = 20\,000$, $M = 12\,000$, and $L = 70$ the compressed μROM uses $1\,140\,000$ bits while the original μROM uses $1\,400\,000$ bits. This is about 19% reduction in bits. *Note*: in this discussion we use the number of bits in the μROM as an estimate for its area requirements. Section VI presents experimental results from layout estimates showing that reductions in actual μROM size are in line with this estimate.

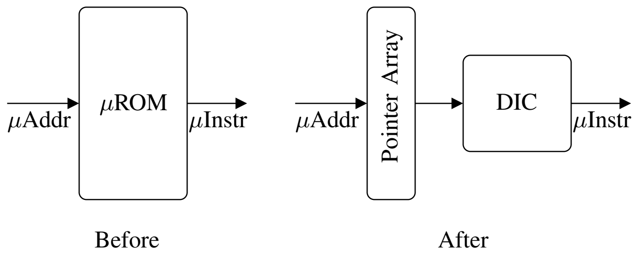

Fig. 1. Basic microcode compression idea.

Notice that the "pointer array" and the original μROM have the same number of entries (N). It means that the original and the compressed microinstructions have the same address space. Therefore, different from other code compression techniques, the decompressor does not require an address translation mechanism [2], [19], [20] and the microcode does not have to be patched [4], [14].

An improvement of the above idea is to split the microinstruction into a number of fields such that the number of unique patterns for each field is minimized. The intuition behind this idea is to take advantage of the entropy within each field. For example, even though a microinstruction 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. Fig. 2 shows an example where each microinstruction is split into two roughly equal-sized fields. Assume M_1 and M_2 are the number of unique patterns for the two halves. The original μROM takes $N \times L$ bits and the compressed μROM takes only $N \times (\lceil log_2 M_1 \rceil + \lceil log_2 M_2 \rceil) + M_1 \times L/2 + M_2 \times L/2$ bits. For $N = 20\,000$, $M_1 = 5\,000$, $M_2 = 5\,000$, and $L = 70$ the compressed μROM uses $20\,000 * 26 + 10\,000 * 35 = 870\,000$ bits while the original μROM uses $1\,400\,000$ bits. This is about 38% reduction in number of bits.

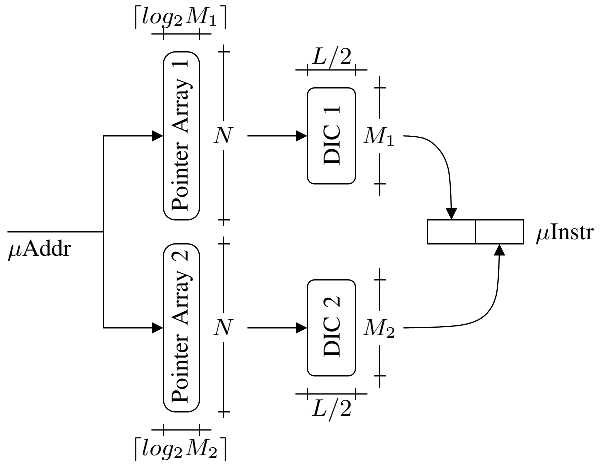

Fig. 2. Partitioned compression.

The key observation from the above approach is that with a proper partitioning of the μROM into subsets of columns, the number of unique patterns in the partitions is reduced and thus the total area will be reduced.

The *clustering-based* compression selectively groups similar columns into clusters, and goes beyond the simple partitioning of the microinstructions into fields composed of adjacent bits. For example, Fig. 3 shows a simple partitioning of each microinstruction into two fields. With this partitioning, the two partitions each have three different patterns and require two bits to index the unique patterns. Therefore, the compressed form needs $10 \times (2 + 2) + 3 \times 3 + 3 \times 3 = 58$, less than 4% reduction from the original 60 bits size.

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
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

Fig. 3. Simple partitioning method.

The clustering-based compression groups columns that are similar to each other into clusters. For example, the sample microcode columns in Fig. 3 may be grouped into the two clusters shown in Fig. 4, where columns 1, 3, and 5 are grouped into the first cluster and the columns 2, 4, and 6 are grouped into the second cluster. With this new clustering, both clusters have only two unique patterns and need only a single bit index. As a result, the compressed form requires only $10 \times (1 + 1) + 2 \times 3 + 2 \times 3 = 32$ bits, nearly 50% reduction and a significant reduction when comparing to the basic partitioning method.

Col 1	Col 3	Col 5	Col 2	Col 4	Col 6
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

Fig. 4. Clustering method.

Fig. 5 shows how to access the microinstruction in the clustering method. In this case, there is a new component called "spreader" for each cluster that spreads the dictionary output bits into the appropriate position in the final microinstruction. The spreader is simply a rewiring of the original path that connects the output to the microinstruction and should not cost any additional die area or power. Notice that this method is not limited by the number of clusters, and sometimes 3 or more clusters are possible.

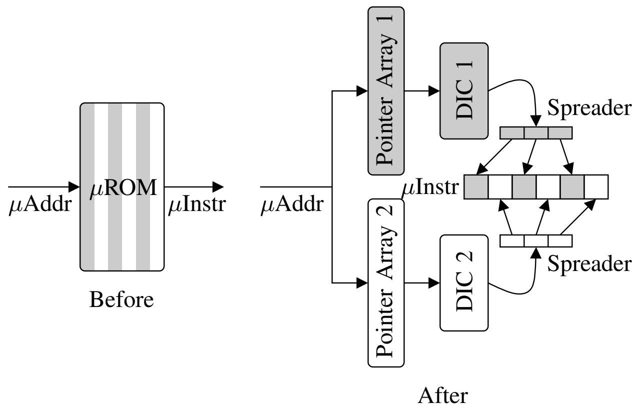

Fig. 5. Accessing a microinstruction in the clustering method.

For easy of understanding, figures 2 and 5 show distinct memory blocks for each "pointer array". However, the "pointer arrays" always fetch data using the same μAddr, therefore, they can be placed in a single memory block and the output bits routed to the appropriate "dictionaries". Other interesting remark is that uncompressed columns also can be placed in the single "pointer array". In this case, the output bits corresponding to the uncompressed columns are routed directly to the output microinstruction bits. Fig. 12 provides a pipelined version of the decompressor representing the "pointer arrays" and the uncompressed columns in a single memory block.

IV. Clustering-Based Compression Algorithms

A clustering algorithm groups similar columns of a μROM into clusters, such that an objective function is minimized. To describe the objective function, we define the following terms:

- L: the number of columns in the μROM;
- N: the number of bits in each column;
- K: the number of clusters in which the L columns are grouped into;
- $L_1, L_2, \cdots, L_k$: the number of columns in each cluster $1, 2, \cdots, K$;
- $M_1, M_2, \cdots, M_k$: the number of unique patterns in clusters $1, 2, \cdots, K$.

The clustering algorithm finds K clusters such that the following objective function is minimized:

$$F = \sum_{i=1}^{K} \left[\underbrace{N \times \lceil log_2 M_i \rceil}_{PointerArray_i} + \overbrace{M_i * L_i}^{Dictionary_i} \right] \tag{1}$$

For the example in Fig. 4, $N = 10$, $K = 2$, $L_1 = 3$, $L_2 = 3$, $M_1 = 2$, $M_2 = 2$, and $F = 10 \times log_2 2 + 2 \times 3 + 10 \times log_2 2 + 2 \times 3 = 32$.

Intuitively the clustering problem is an NP-hard optimization problem. In order to address the problem we use heuristics. The next sections show the proposed heuristic approaches to solve this problem.

A. Sequential Columns

In order to group the columns in clusters, we divide the problem in two parts:

- Create clusters of columns and attribute to each cluster i a benefit B_i, where B_i is the number of bits saved if cluster is selected. $B_i = N \times (L_i - \lceil log_2 M_i \rceil) - M_i \times L_i$
- Select the best set of clusters in order to maximize the total benefit ($\sum_{i=1}^{k} B_i$).

It would be infeasible to enumerate all the possible clusters [18]. Therefore, we decided to explore only the clusters formed by consecutive columns, an approach we call S.C. (standing for Sequential Columns).

The total number of possible clusters formed by consecutive columns is: $\sum_{i=1}^{L}[L-i] = \frac{L^2-L}{2}$. Thus, for a given microcode with 75 columns, this approach would generate 2 775 clusters.

To select the best clusters we transform the problem into a graph problem. Clusters are vertices with weight B_i, and two vertices have an edge between them if the corresponding clusters conflict (have common columns). The best solution is the independent set with the maximum weight.

Searching for the independent set with the maximum weight in general graphs is a NP-hard problem [10]. However, we can reduce the sequential columns clusters to interval graphs, and consequently solve the problem in polynomial time. Therefore, we can reach the optimum solution when considering only clusters formed by adjacent columns.

Since the clusters have sequential columns, we can represent each cluster with three numbers: the initial column, the final column, and the benefit B_i. The cluster can also be seen as an "interval" with a cost (B_i). Fig. 6 shows an example of a set of intervals (clusters).

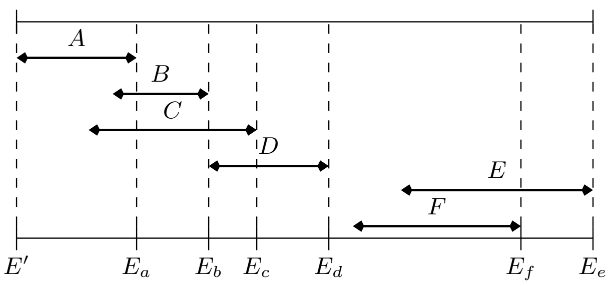

Fig. 6. Clusters represented as intervals. Cluster A contains the columns $0, 1, \cdots, E_a$.

We define $Vf(E_i)$ as the first valid interval after column E_i (e.g.: in Fig. 6, $Vf(E_a) = D$ because B and C contains columns $\leq E_a$). v_1, v_2, $\cdots$, v_n are the clusters that contain only columns $\geq E_a$ and conflict with $Vf(E_i)$ (e.g.: for $Vf(E') = A$, $v_1 = B$, and $v_2 = C$).

$$best\_solution = ss(E') \tag{2}$$

$$ss(E_i) = \begin{cases} Max(\, ps(Vf(E_i)), \\ \quad ps(v_1), \\ \quad ps(v_2), & \text{for } Vf(E_i) \neq \emptyset \\ \quad \cdots, \\ \quad ps(v_n)) \\ \quad 0 & \text{for } Vf(E_i) = \emptyset \end{cases} \tag{3}$$

$$ps(v) = v.benefit + ss(E_v) \tag{4}$$

Equation $ss(E_i)$ states that the best sub-solution (from column E_i to the end) involves only the partial-solution ($ps(v)$) of the first valid cluster after E_i ($Vf(E_i)$) and the clusters that conflict with it (v_1, $\cdots$, v_n).

We can see that the partial-solution of a cluster that does not conflict with $Vf(E_i)$ could be improved by adding $Vf(E_i)$ to it. This is true because we filtered out the clusters with negative benefits. Therefore the computation of $ss(E_i)$ only involves clusters that conflict with $Vf(E_i)$.

Every time $ss(E_i)$ is computed, the algorithm updates an array with the result, so that it does not have to recompute it. We can see that $ss(E_i)$ is executed only once for each cluster i. Every time $ss(E_i)$ is executed, the algorithm searches for $Vf(E_i)$ and v_1, $\cdots$, v_n. The clusters can be sorted by the final and initial columns so that this search can be done in $O(logN)$ (where N is the number of clusters). Therefore the algorithm execution time is $O(N \times logN)$.

An interesting observation is that the final clustering may not include some of the original microcode columns. It happens because the heuristic only selects clusters that increases the total benefit. As stated before, the columns that were not included into the clusters, or the uncompressed columns, are placed into the "pointer arrays" and their output bits routed directly to the microinstruction bits.

Since the S.C. heuristic considers only clusters formed by consecutive columns, it does not take advantage of similar columns that are apart from each other. However, we can reorganize the microcode columns, so that similar columns are moved together, and then use the S.C. heuristic to group these columns into clusters. The microcode in Fig. 4 is an example where the similar columns were moved together. The next sections introduce two heuristics to move similar columns together: the *linear ordering* and the *circular ordering* heuristic.

B. Linear Ordering

The linear ordering heuristic reorganizes the microcode by moving similar columns together. Although the heuristic itself does not group columns into clusters, it enables the S.C. heuristic to take advantage of similar columns that were originally apart.

To determine the new columns order, the heuristic starts with a work list (WL) containing one column and grows this list by adding columns to it. The order that the columns are inserted into the list is the new column ordering. Fig. 7 shows the pseudo-code for the linear ordering heuristic. The benefit of $WL \cup \{c\}$ is defined by the number of unique patterns in a cluster formed by the columns in WL and c. Thus, the column with the best benefit is the one that increases fewer patterns when added to WL.

```
1: Initialize $WL$ with one column
2: **while** there are not selected columns **do**
3:    **for** each not selected column $c$ **do**
4:       Compute the benefit of $WL \cup \{c\}$
5:    **end for**
6:    Select the column with the best benefit and append it
      to the end of $WL$.
7: **end while**
8: The columns selection order is the new order.
```

Fig. 7. Linear ordering pseudo-code.

The proposed framework computational time is reasonably fast. Therefore, we can explore the heuristic by taking each of the L columns as the first column in WL.

C. Circular Ordering

The circular ordering approach is based in the linear ordering heuristic. The main differences are that in the circular ordering approach:

- There is a column array that contains the column ordering;
- The work list (WL) grows only until a certain limit W, then it removes the earliest inserted column for every column added to WL. In other words, it moves over the column array;
- The column array is a circle, thus, when WL reaches the end, it continues through the beginning of the column array (initial point).

Fig. 8 shows the work list moving over the column array.

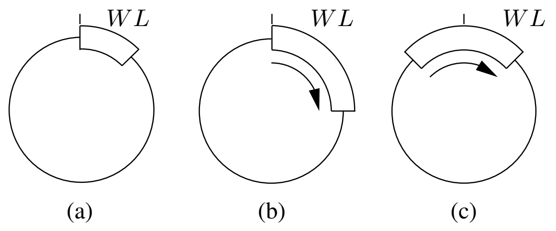

Fig. 8. (a) WL growing from the beginning ($|WL| < W$). (b) WL walking through the column array ($|WL| = W$). (c) When WL reaches the end of the column array it continues selecting columns.

Every time WL crosses the initial point, the algorithm saves the columns ordering. The work list keeps moving until it reaches the maximum number of iterations. Consequently, for each iteration the algorithm generates a new column ordering. Fig. 9 shows the pseudo-code for the circular ordering heuristic.

```
1: Grows $WL$ until it has $W$ columns
2: **for** each iteration **do**
3:    **for** i=0 to $L$ **do**
4:       $best\_col$ = choose_the_best_col ()
5:       Insert $best\_col$ into the first position in $WL$ and
         remove the column in the last position in $WL$.
6:       Place the removed column right behind $WL$.
7:    **end for**
8:    Saves the column array order
9: **end for**
```

Fig. 9. Circular ordering pseudo-code.

V. Microcode Compression Results

We have applied the clustering algorithms to four different microcodes from production processors. The first microcode is from a Netburst-class architecture and contains 22 528 microinstructions with 75 bits each. The second one is from a Pentium-M-class architecture, and the last two were extracted from mobile-class processors. Table I summarizes the microcodes used in our experiments.

TABLE I
Microcodes descriptions.

μCode	# Cols	# Lines	Size in bits	Class
A	75	22 528	1 689 600	Netburst
B	234	6 656	1 557 504	Pentium-M
C	236	5 632	1 329 152	Mobile
D	240	5 632	1 351 680	

In order to evaluate the heuristics, we used the following criteria:

- Single Cluster: The microcode is compressed by grouping all the columns in a single cluster, such as depicted in Fig. 1.
- Sequential Columns: We apply the sequential columns heuristic to the original microcode.
- Linear Ordering: For a given microcode with L columns, we execute the linear ordering heuristic starting the work list (WL) with each one of the L columns. Consequently, we generate L columns orderings. For each ordering, the microcode is reorganized and the sequential columns algorithm is used to group the new microcode columns.
- Circular Ordering: Like in the linear ordering approach, we use the sequential columns heuristic to group the reorganized microcode columns. In addition to starting WL with each one of the L columns, we vary the parameter W (the maximum size of WL) and fix the number of iterations in L.

Fig. 10 shows the best compression ratios for each clustering approach. Notice that the sequential columns heuristic significantly improves the single cluster approach results. For microcode B, the compression ratio dropped from 93.69% to 54.40%, while in the other microcodes the reduction was about 25%. In fact, it happens because the sequential columns groups the columns in multiple clusters. Moreover, the heuristic is able to identify non-similar columns and place them directly into the "pointer array" as uncompressed columns.

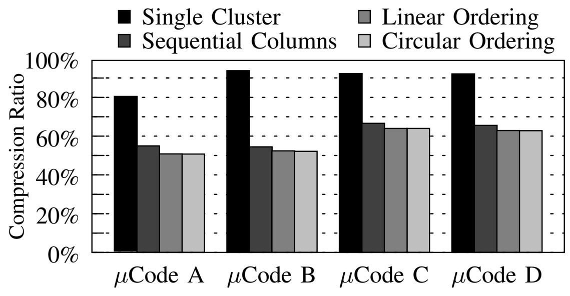

Fig. 10. Compression ratios for the clustering approaches.

The linear ordering approach improved the sequential columns results from 2 to 4%. The circular ordering heuristic also achieved better results, but in this case, the improvement was not significant (less than 0.5%) and the circular ordering execution time was too high.

VI. Die Area Reduction

So far we have discussed compression ratio in terms of the reduction in number of bits stored in the μROM. 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. Moreover, there may be several such arrays and their arrangements affect the layout. Finally, control logic, address decoding structures and the drivers also affect ROM size and do not scale linearly with the number of bits. Fig. 11 illustrates a ROM layout organization.

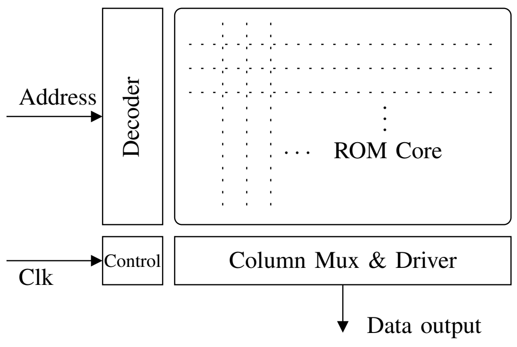

Fig. 11. ROM layout organization.

We modeled the area of the final layout for the μROM containing the microcode A with help from circuit design engineers. Three cases were considered (the die area is measured in "units" of space):

- The original μROM structure contains one array with dimensions 711×800 units, with total area $568\,800$ units;
- The single cluster organization, with one "pointer array" and one "dictionary" (Fig. 1), is created. This organization has two ROM arrays; the first array has dimensions 484×214 units and the second array has dimensions 291×1034 units. The total area is $404\,470$ units;
- An organization similar to Fig. 2, with $K = 2$ in which the microinstruction is broken in two parts. This organization has three arrays, respectively with areas 484×349, 155×286 and 277×634 for a total area of $388\,864$ units.

The above experiment found an area reduction to 68% of the original area, whereas the reduction in the number of μROM bits for this microprogram was to 57% (when the non-similar columns are included into the clusters). This difference is not unusual, as we discussed above, due to the rectangular form of the arrays and the extra structures that does not scale linearly with the bits reduction. It also demonstrates that our estimates based on bit counts are a good approximation of the actual area savings.

VII. Pipelined Execution

The microcode decompression must be efficient enough to comply with the throughput constraints of a high performance CPU. Compared to the original μROM, the decompression engine implements a two level memory access indirection. It loads the indices from the "pointer arrays", and uses these indices to access the "dictionary" arrays. Notice that this organization may have higher latency than the original μROM, depending on the sizes of the arrays. If the new memory arrays are smaller than the original μROM the new latency should not be as large as twice the original latency.

If the new latency is enough to keep the μROM access out of the cycle time critical path, then the decompression engine is as simple as the organization in Fig. 5 (right). However, if the new latency does increase the CPU critical path, we introduce a pipelined decompression engine.

Fig. 12 depicts the layout organization of the pipelined decompression engine. The structure is divided in two pipeline stages. In the first stage, the indices to access the "dictionary" arrays and the uncompressed columns are fetched from the "pointer arrays". As soon as these arrays are smaller than the original μROM, this stage does not increase the cycle time.

In the second stage, the μInstr bits are fetched from "dictionary" arrays and assembled into the microinstruction. The uncompressed bits are routed directly from the pipeline register to the microinstruction. Again the "dictionary" arrays are smaller than the original μROM, and consequently the time required to fetch the μInstr bits. Notice that the spreaders are basically rewiring, therefore they do not require extra time to reorganize the microcode bits.

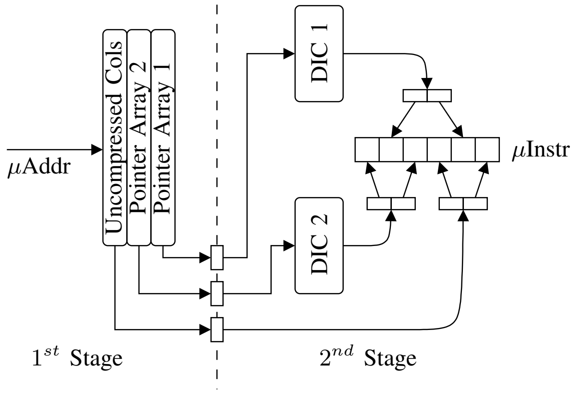

Fig. 12. Pipelined decompression engine layout organization.

The two stage pipelined organization adds only one cycle (to fill the pipeline) when executing a microcode sequence. For very short microcode sequences, this extra cycle may be a compromising overhead. However, most microcode sequences are long, as they correspond to complex instructions. Such instructions are also less frequent, thus being relegated to microcode. In our experiments, adding one cycle to the microcode sequence had negligible performance impact to application programs.

VIII. Reducing μROM Loading

In addition to reducing power due to die area reduction, microcode compression can also be extended to further reduce power consumption of the compressed μROM. The μROM power consumption is determined, in part, by the number of bits set to '1' in the μROM [3], [5]. Thus, a transformation technique that reduces the number of μROM bits that are set to '1' potentially reduces its power consumption. Microcode compression may be thought of as a transformation technique.

Notice that it is possible to choose the "dictionary" position where a given pattern is placed. As a consequence, the "pointer array" indices are also changed. For instance, if a given pattern is placed into the first position of the "dictionary" (address 0x00h), the "pointer array" entries that points to this pattern will contain the value 0x00h. Therefore, we can assign the most frequent patterns to the "dictionary" addresses with fewer bits to reduce the number of '1' bits in the "pointer arrays".

A simple algorithm to reduce the compressed μROM loading is as follows: first, sort the dictionary patterns in descending order of frequency and then assign dictionary positions such that the most frequent patterns have the least number of '1' bits in its address. So, the first position, assigned to the highest-occurring pattern is all zeros. Next, all addresses containing one bit set are assigned, and so on.

We achieved about 60% reduction on the number of '1' bits when applying this algorithm to the microcodes from Section V. Note that these microcodes have been hand-coded and take into consideration μROM loading by reducing the number of bits set to '1' wherever possible.

As an example, the microcode A contains $273\,264$ bits set to '1'. Using the μROM loading reduction algorithm, the compressed organization for the same microcode contains only $114\,549$ bits set. For this microcode sequence, our technique is able to further reduce the total number of bits set to '1' by 58%, considering all ROM arrays.

IX. 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 the μROM loading for reduced power consumption is presented. The techniques are illustrated by application to microcode from production microprocessor designs, achieving compression ratios ranging from 50% to 62%.

Acknowledgments

This work was partially supported by a fellowship grant from FAPESP (02/08139-3) and research awards from FAPESP (00/15083-9) and CNPq (301731/2003-9). We would like to thank Herbert Hum and the anonymous reviewers for their valuable comments and discussions. We also appreciate the support provided by Jesse Fang at the Programming System Laboratory at Intel.

References

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

[2] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression based on operand factorization. In *Proceedings of MICRO-31*, p. 194–201, 1998.

[3] E. de Angel and J. Earl E. Swartzlander. Survey of low power techniques for ROMs. In *Proceedings of ISLPED'97*, p. 7–11, 1997.

[4] M. Breternitz Jr. and R. Smith. Enhanced compression techniques to simplify program decompression and execution. In *Proceedings of ICCD'97*, p. 170–176, 1997.

[5] Y.-S. Chang, B.-I. Park, and C.-M. Kyung. Conforming inverted data store for low power memory In *Proceedings of ISLPED'99*, p. 91–93, 1999.

[6] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[8] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. In *Proceedings of SIGPLAN'84*, p. 117–121, 1984.

[9] M. Game and A. Booker. CodePack: Code compression for PowerPC processors. *MicroNews 5(1)*, 5(1), 1999. IBM.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[11] K. M. Guttag. Compressing control ROM for VLSI microprogrammed microprocessors. In *Proceedings of MICRO-13*, p. 115–121, 1980.

[12] IBM. *CodePack PowerPC Code Compression Utility Users Manual Version 3.0*. IBM, 1998.

[13] N. Ishiura and M. Yamaguchi. Instruction code compression for application specific VLIW processors based on automatic field partitioning. In *Proceedings of The Workshop on Synthesis and System Integration of Mixed technologies*, p. 105–109, 1997.

[14] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proceedings of MICRO-30*, p. 194–203, 1997.

[15] C. Lefurgy, E. Piccininni, and T. Mudge. Evaluation of a high performance code compression method. In *Proceedings MICRO-32*, p. 93–102, 1999.

[16] S. Y.-H. Liao. *Code generation and optimization for embedded digital signal processors*. PhD thesis, MIT, 1996.

[17] S.-J. Nam, I.-C. Park, and C.-M. Kyung. Improving dictionary-based code compression in VLIW architectures. *IEICE Transactions on Fundamentals of Electronics*, E82-A(11):2318–2324, 1999.

[18] G.-C. Rota. The Number of Partitions of a Set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[19] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *SIGMICRO Newsletter* , 23(1-2):81–91, 1992.

[20] Y. Xie, W. Wolf, and H. Lekatsas. Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures. In *Proceedings of ASIC'01*, p. 337–340, 2001.

[21] Y. Xie, W. Wolf, and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. In *Proceedings of ISSS'02*, p. 138–143, 2002.