Structure-Constrained Microcode Compression

Edson Borin
University of Campinas
Campinas, Brazil
edson@ic.unicamp.br

Guido Araujo
University of Campinas
Campinas, Brazil
guido@ic.unicamp.br

Mauricio Breternitz Jr.
AMD
Sunnyvale, USA
mbreternitz@google.com

Youfeng Wu
Intel Labs
Santa Clara, USA
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. One solution to address the microcode size issue is to store the microcode in a compressed form and decompress it during execution. Furthermore, the reuse of a single hardware building block layout to implement different dictionaries in the two-level microcode compression reduces the cost and the design time of the decompression engine. However, the reuse of the hardware building block imposes structural constraints to the compression algorithm, and existing algorithms may yield poor compression. In this paper, we develop the SC^2 algorithm that considers the structural constraint in its objective function and reduces the area expansion when reusing hardware building blocks to implement different dictionaries. Our experimental results show that the SC^2 algorithm is able to produce similar sized dictionaries and achieves the similar compression ratio to the non-constrained algorithm.**

I. Introduction

Microprogramming is a widely known technique used to implement processor control units. Microcode makes the control unit design process easier, as it can be modified 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 the microcode increase. 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.

One solution to address the microcode size issue is to store the microcode in a transformed representation (compressed) and decompress it during execution. In fact, since the introduction of microprogramming in 1951, by Maurice Wilkes [19], many compression and encoding techniques had been proposed to reduce the microcode size [1], [4], [7], [8], [9], [10], [15], [16], [17], [18], [22]. Most of these techniques date from the sixties and the seventies. With the introduction of RISC machines in the eighties, there was a noticeable decrease of interest in microcode in the research community, as many of the programmable HW functionalities are migrated to SW. However, recent trends have migrated more and more advanced functionalities, such as protection, virtualization and management assistance, to the microcoded portion of a CPU core, and the microcode growth caused by these new features forced the industry to revisit the microcode size problem [4].

A recent work [4] successfully explored a two-level microcode organization to compress the micro-code of high performance processors. This technique, outlined in Section III, replaces the original microinstructions by pointers to dictionaries that hold bit patterns extracted from the microcode. The "pointer arrays" and the "dictionaries" are ROMs that store the pointers and the bit patterns, respectively. The technique allows the microcode columns to be grouped into clusters, so that the number of bit patterns inside the dictionaries is reduced.

When developing the decompression engine in the two-level microcode compression [4], the logic designer must create hardware modules for the "pointer array" and for each dictionary. Although CAD tools are able to automatically generate these pieces (or modules) of hardware from high level descriptions, these automatically generated modules are not power and performance efficient. Therefore, in these cases, part of the hardware must be custom-designed.

Custom-designed hardware is intensive in human resource and is very time consuming. On the other hand, the design of reusable hardware modules allows architects to leverage previously validated modules, considerably reducing the design time. In this spirit, we propose using the same hardware building block for all dictionaries (a.k.a. 'unique pattern arrays') to reduce the compressed microcode design effort. In other words, a single ROM design would be used to implement the different dictionaries. We call this approach Structure-Constrained Microcode Compression.

The contributions of this paper are summarized as follows:

- It proposes the reuse of the same hardware building block to implement different dictionaries in the microcode compression design. This reduces the cost and the designing time of the decompression engine.
- It proposes SC^2, a new structure-constrained clustering algorithm to reduce the wasted area and improve the compression ratio when reusing hardware blocks for the dictionaries. The new algorithm is also applicable to a range of similar problems whose solution set requires equal-sized clustering.

The rest of the paper is organized as follows. Section II outlines the related work. Section III discusses the microcode compression and the structure-constrained compression techniques. Section IV describes the structure-constrained compression (SC^2) algorithm. Section V shows the experimental results, and Section VI concludes the paper.

II. Related Work

Since the introduction of microprogramming in 1951 [19], many compression and encoding techniques had been proposed to reduce the microcode size.

Assuming the original μROM has M unique microinstructions, *maximal encoding* [7] replaces each microinstruction in the μROM by a $\lceil log_2 M \rceil$ bit codeword.

A decoding logic is attached to the output bits of the μROM and whenever a codeword is fetched, the decoding logic converts it into the original microinstruction bits. The decoding logic, added after the compressed μROM, can be very complex and the extra area and delay added by it can nullify the compression benefits.

In 1968, Schwartz [16] proposed a method to reduce the complexity of the decoding logic called *minimal encoding*. This method grouped mutually exclusive micro-operations into microinstruction fields. In this way, the decoding logic could be implemented with simple decoders.

In the *indirect encoding*, or bit steering [1], the microinstruction bits are split into fields and the meaning of a field depends on the value of a control field. The control field makes it possible to classify the microinstructions in formats, which allows a higher degree of encoding. Similarly to the maximum encoding, the decoding logic can become complex, therefore, the indirect encoding must be used carefully.

Stritter and Tredennick [9], [17], [18] investigated the two-level control store to reduce the size of the Motorola MC68000 microcode. In this model, the microcode is stored in two ROMs, the μROM and the *nano*ROM. The μROM contains microinstructions, which are pointers to nanoinstructions stored into the *nano*ROM. The nanoinstructions contain the data path control bits. Menzilcioglu [15] improved the two-level control store by using multiple *nano*ROMs to store the different fields of the nanoinstructions. However, the author did not show any method to identify and group the microcode fields into the *nano*ROMs. Zhao and Papachristou [22] used a similar compressing model and proposed a heuristic to identify and group microcode fields into the *nano*ROMs.

Ishiura and Yamaguchi [11] used a two-level organization to reduce the size of VLIW code in application specific VLIW processors. They also proposed a heuristic to maximize compression by grouping similar VLIW bits into the same ROMs.

Hum *et al.* [10] patented a microcode compression scheme similar to the two-level control store. In their scheme, the microcode columns were grouped into tables, similar to the *nano*ROMs, and the compressed microcode, containing pointers to the tables, was stored into the μROM. Borin *et al.* [4] proposed three algorithms to improve the compression ratio by grouping similar columns into the same tables.

Another approach to the problem was to design algorithms that could efficiently compact operations into microinstructions [8]. These techniques evolved to what is nowadays a set of compiling methods used in code generation for VLIW machines.

Recently, many hardware based code compression techniques had been proposed to reduce the code size in embedded systems [3]. Most of these techniques 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 [13]. Since the microcode storage system does not have a cache memory, such approaches are less desirable when decompressing the performance critical microcode from the μROM.

In this work we use a compression scheme similar to the two-level control store organization. However, instead of μROM and *nano*ROMs, we refer to these structures as "pointer array" and "dictionaries". Next section introduces the two-level storage based microcode compression scheme.

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 Figure 1. In this figure, μAddr is the address of a microinstruction. In the uncompressed form, the μAddr 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 size of the "pointer array" and $M \times L$ is the size of the "dictionary"). 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 approximately 19% reduction in bits. Note: in this discussion we use the number of bits in the μROM as an estimate for its area requirements. Borin *et al.* [4] presented experimental results from layout estimates showing that reductions in actual μROM size are in line with this estimate.

Notice that the "pointer array" and the original μROM have the same number of entries (N). This means that the

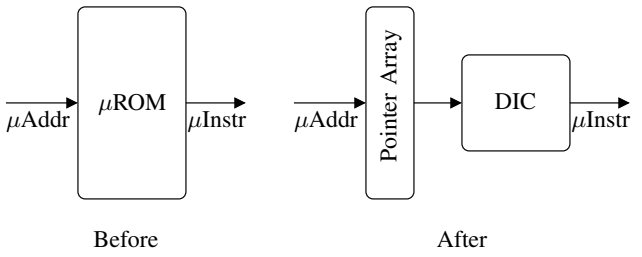

Figure 1. Basic microcode compression idea.

original and the compressed microinstructions have the same address space. Therefore, unlike other code compression techniques, the decompression engine does not require an address translation mechanism [2], [20], [21] and the microcode does not need to be patched [6], [12].

An improvement of the above approach is to split the microinstruction into a number of fields such that the number of unique patterns for each field is minimized. The 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. Figure 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 $20,000 \times 26 + 5,000 \times 35 + 5,000 \times 35 = 870,000$ bits while the original μROM uses $1,400,000$ bits. This is approximately 38% reduction in number of bits.

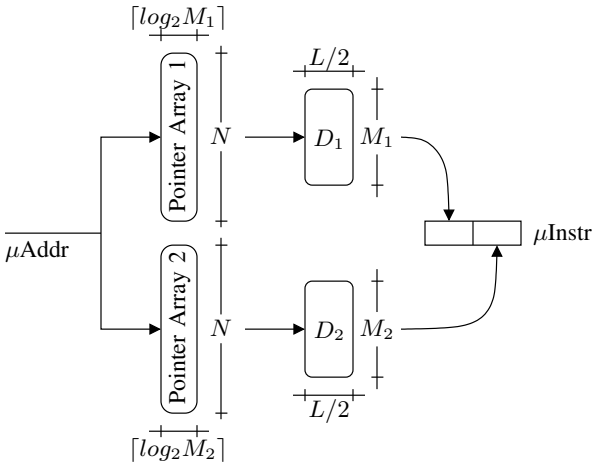

Figure 2. Partitioned compression.

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 instance, Figure 3 shows a simple partitioning of each microinstruction into two fields. With this partitioning, each one of the two partitions has three different patterns and requires two bits to index the unique patterns. Therefore, the compressed form needs $10 \times (2 + 2) + 3 \times 3 + 3 \times 3 = 58$ bits, 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

Figure 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 Figure 3 may be grouped into the two clusters shown in Figure 4, where columns 1, 3, and 5 are grouped into the first cluster and 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, significantly better than 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

Figure 4. Clustering method.

Figure 5 shows how to access the microinstruction in the clustering method. In this case, there is a new component for each cluster, called "spreader", which spreads the dictionary output bits into the appropriate position in the final microinstruction. In other words, the spreader is a rewiring of the original path which 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.

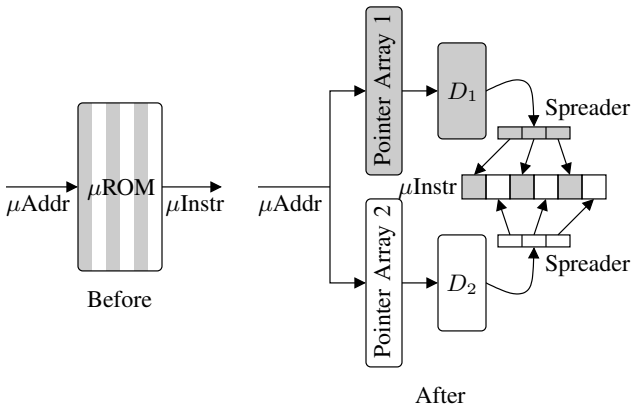

Figure 5. Accessing a microinstruction in the clustering method.

For easy of understanding, Figure 2 and 5 show distinct memory blocks for each "pointer array". However, the "pointer arrays" have the same number of lines as the original μROM and always fetch data using the same μAddr, therefore, they can be stored into a single memory block and the output bits routed to the appropriate "dictionaries". Another interesting observation is that some uncompressed columns can also be placed into the same memory block as the pointer arrays. In this case, the output bits corresponding to the uncompressed columns are routed directly to the output microinstruction bits. Figure 6 provides a pipelined version of the decompression engine representing the pointer arrays and the uncompressed columns in the same memory block.

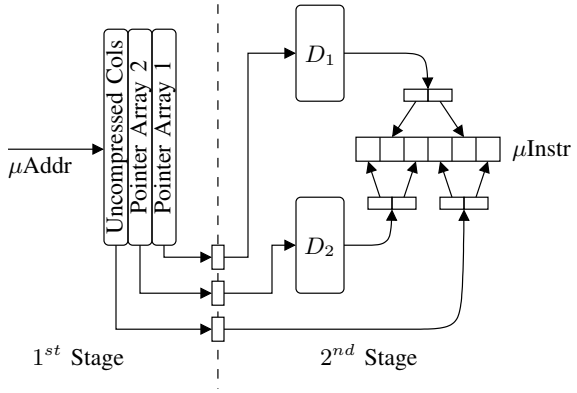

Figure 6. Pipelined decompression engine layout organization.

The clustering-based compression scheme relies on grouping similar columns to minimize the number of patterns in each dictionary. However, some columns may be very different from the other columns and may significantly increase the number of patterns if placed into a dictionary with other columns. Therefore, it is sometime beneficial to place these columns into the uncompressed columns array.

A. Structure-Constrained Compression

The design of the decompression engine makes use of memory blocks to implement the pointer array and the dictionaries. Although computer aided design (CAD) systems can automatically generate these hardware blocks from high level descriptions, in some cases the results are not satisfactory and these blocks must be manually designed. High performance microprocessors are examples of projects that still require these blocks to be manually designed.

Manually designed hardware blocks are very expensive in terms of human resources and time. In order to minimize these costs, previously designed hardware blocks are reused whenever possible. As an example, it is possible to create a memory block with 1024×32 bits using two blocks of 512×32 bits, previously designed and tested, and an extra combinational circuit.

In order to reduce the cost and design time of the decompression engine, we propose the reuse of the same hardware building block to implement different dictionaries. This way, a single ROM design would be used to implement the different dictionaries. We call this approach *Structure-Constrained Microcode Compression*. The main challenge of this approach is that the decompression engine may have many dictionaries with different sizes and the hardware may be under utilized, affecting the final area compression ratio.

As an example, suppose a compressed microcode with two dictionaries, D_1 and D_2. If D_1 has 20 columns and 500 patterns and D_2 has 34 columns and 100 patterns, we could design a memory block with 34 columns and 500 lines and reuse it to implement the two dictionaries. Notice that D_1 would not use 14 of the 34 columns and D_2 would use only 100 out of the 500 lines. Figure 7 shows a decompression engine where the two dictionaries are implemented with the same hardware block. The hashed area shows the hardware block region which is not used by the dictionaries.

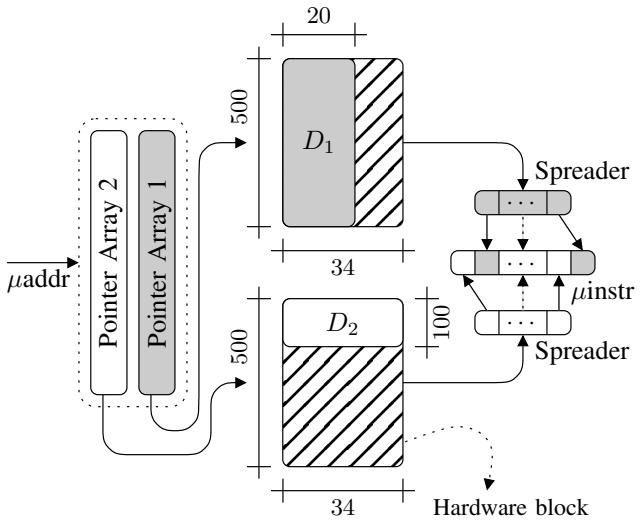

Figure 7. Dictionaries implemented using the same hardware block design.

We define two compression ratio metrics to measure the

waste of area when applying the structure-constrained compression: *structure-constrained compression ratio* (CR_{str}) and *regular compression ratio* (CR_{reg}). The *structure-constrained compression ratio* assumes that all dictionaries have the same shape ($columns \times lines$) when computing the compression ratio. For instance, assuming that the microcode from Figure 7 has 2000 lines and 54 columns, the *structure-constrained compression ratio* is:

$$CR_{str} = \frac{\overbrace{(\lceil log_2 500 \rceil + \lceil log_2 100 \rceil) \times 2000}^{Pointer\ Array\ Size} + \overbrace{500 \times 34 + 500 \times 34}^{Dictionaries\ Size}}{54 \times 2000} = 61\%$$

While the regular compression ratio is:

$$CR_{str} = \frac{\overbrace{(\lceil log_2 500 \rceil + \lceil log_2 100 \rceil) \times 2000}^{Pointer\ Array\ Size} + \overbrace{500 \times 20 + 100 \times 34}^{Dictionaries\ Size}}{54 \times 2000} = 42\%$$

Notice that there is a 19% difference between the regular compression ratio and the structure-constrained compression ratio. This difference indicates the area wasted by reusing the same hardware block to implement the dictionaries.

In order to reduce the area loss caused by the memory block reuse, it is important to generate dictionaries with similar sizes. However, to the best of our knowledge, the existing clustering-based microcode compressing algorithms [22], [11], [4] do not try to generate similar shaped dictionaries. In fact, our experimental results, shown in Section V, indicate that these algorithms can produce poor structure-constrained compression ratios when compressing the microcode. The main reason is that these algorithms try to minimize an objective function F that does not reflect the waste of area caused by the hardware block reuse. To describe the unconstrained objective function F and the structure-constrained objective function F_{str}, 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_0: the number of uncompressed columns;
- 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 existing clustering-based microcode compression algorithms [22], [11], [4] try to find K clusters (or dictionaries) such that the following objective function is minimized:

$$F = \overbrace{N \times L_0}^{Uncomp.Cols} + \sum_{i=1}^{K} \overbrace{N \times \lceil log_2 M_i \rceil}^{PointerArray_i} + \sum_{i=1}^{K} \overbrace{M_i \times L_i}^{Dictionary_i}$$

The F objective function computes the size of the decompression engine, in terms of bits, assuming that the uncompressed columns and the pointer arrays will be stored together, using a single memory block, as in Figure 6, and each dictionary will be implemented using a hardware block specially designed to match the dictionary size. In this way, minimizing F does not guarantee that the dictionaries will have similar number of columns and patterns.

In order to represent the waste of area caused by the hardware block reuse, we modeled the structure-constrained microcode compression problem using the following objective function:

$$F_{str} = \overbrace{N \times L_0}^{Uncomp.\ Cols} + \sum_{i=1}^{K} \overbrace{N \times \lceil log_2 M_i \rceil}^{Pointer\ Array_i} + \underbrace{K \times (\max_{i=1}^{K}[M_i] \times \max_{i=1}^{K}[L_i])}_{HW\ Block\ size}$$

The F_{str} objective function computes the size of the decompression engine, in terms of bits, when using the same hardware block to implement all the dictionaries. The first two parts of the objective function sum the size of the uncompressed columns and the pointer arrays, and the third part computes the size of the dictionaries by multiplying the number of dictionaries (K) by the size of the common hardware block which is able to store the dictionaries.

Again, the existing clustering-based compression algorithms [22], [11], [4] were not designed to minimize F_{str} and were not able to generate similar dictionaries in our microcode compression experiments. Consequently, we propose the SC^2 algorithm, a Structure-Constrained Clustering algorithm that is able to minimize the objective function F_{str} by generating similar shaped dictionaries. Next section presents the SC^2 algorithm.

IV. The Structure-Constrained Clustering Algorithm

The goal of the SC^2 algorithm is to group similar columns into clusters so that the objective function F_{str} is minimized.

We use the SC^2 algorithm to explore regularity in the dictionaries when compressing microcode. Let K be the number of dictionaries used in the compression and L_{fix} the fixed number of columns in each dictionary. The pair K, L_{fix} defines a structure-constrained compression configuration. For instance, the pair $\{2, 40\}$ constrains the column clustering process to two clusters, with 40 columns each. To identify the set of possible pairs of configurations, we limit the maximum number of dictionaries to D_{Max} (such as 10).

Then, for i in 2 to D_{Max}, and j in 1 to $\lceil L/i \rceil$, we create the configuration pair i, j. As an example, for a microcode with $L = 80$ columns, we create the following configuration pairs: {2, 1}, {2, 2}, {2, 3}, $\cdots$, {2, 40}, {3, 1}, $\cdots$, {3, 26}, {4, 1}, $\cdots$, {10, 8}. Notice that, the maximum number of configurations $\{i, j\}$ is finite as $i \times j < L$. When $i \times j < L$, the clustering algorithm selects $L - i \times j$ columns as the uncompressed columns.

For each of the configuration pairs, the SC^2 algorithm clusters the microcode columns using the following clustering algorithm, and the compression generated for the configuration with the best compression ratio is chosen as the final result.

For a given configuration pair $\{i, j\}$, the clustering algorithm first takes a set of input constraints and performs a naïve initial grouping of the $i \times j$ columns into i clusters and $L - i \times j$ columns to a separate cluster of uncompressed columns. Then it moves and exchanges columns between clusters so that the initial clustering is improved according to the objective function F_{str}. Figure 8 shows the pseudo-code for the clustering algorithm.

The "naïveClustering" function performs the initial clustering so that the input constraints are not violated. Also, the "selectBestOperation" function only selects move and exchange operations that do not violate the constraints. In this way, the final result always honors the input constraints.

```
1:  BestClustering ← naïveClustering(Constraints)
2:  repeat
3:     CurrentC ← BestClustering
4:     improved ← false
5:     unlock({c_1, c_2, ..., c_L})
6:     repeat
7:        op ← selectBestOperation()
8:        if op.type = EXCHANGE then
9:           exchangeColumns (op.c_1, op.c_2)
10:          lock ({op.c_1, op.c_2})
11:       else if op.type = MOVE then
12:          moveColumn (op.c_1, op.D_2)
13:          lock ({op.c_1})
14:       end if
15:       if F_str(CurrentC) < F_str(BestClustering)
          then
16:          BestClustering ← CurrentC
17:          improved ← true
18:       end if
19:    until there are no more valid operations
20: until improved ≠ true
```

Figure 8. The pseudo-code for the clustering algorithm.

The kernel of the algorithm, comprised by the innermost loop (lines 6-19), continuously moves and exchanges columns between clusters to improve the current clustering. If the operation improves the objective function (F_{str}), the best clustering is updated (lines 15-18). Whenever a column is moved from one cluster to another, the column is locked (lines 10 and 13) and it is not moved to another cluster until it became unlocked. The "selectBestOperation" function does not select operations that invalidate the input constraints or operations that move or exchange locked columns. In this way, when there are no more valid operations, the kernel of the algorithm finishes. If the best clustering was improved (line 20), all the columns are unlocked and the kernel of the algorithm is executed again to improve the current best clustering. When the algorithm is not able to improve the best clustering anymore, it returns the best clustering found so far.

The "selectBestOperation" function (line 7) may select operations that results in negative gains when there are no more positive gain operations. Although these negative gain operations increase F_{str}, subsequent operations may improve it again, making it even better than it was before. The clustering algorithm relies on these negative gain operations to escape from locally optimum results and reach better results.

The SC^2 algorithm described above was motivated and is illustrated with respect to its application to the microcode compression problem. This solution also applies to computational (and design) tasks that are mapped into a search for equal-sized clusters benefit from the presented solution. One example is VLSI chip partitioning [14] for designs that do not fit in a single-chip solution (or FPGA), parts of the design are separated into similar-sized portions (according to an 'area estimation' function) while minimizing inter-cluster communication.

V. Experimental Results

We have applied the structure-constrained microcode compression algorithm to four different microcodes for production processors. The first microcode, for a desktop processor, contains $22,528$ microinstructions with 75 bits each. The second one is for a laptop processor, and the last two are for 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	Desktop
B	234	6 656	1 557 504	Laptop
C	236	5 632	1 329 152	Mobile-Class
D	240	5 632	1 351 680	

We first apply the existing non-structure constrained algorithms [22], [11], [4] to the microcodes. Table II shows the best regular compression ratio achieved by the existing algorithms and its respective structure-constrained compression ratio for microcodes A, B, C, and D.

Table II
BEST REGULAR COMPRESSION RATIO AND THE RESPECTIVE STRUCTURE-CONSTRAINED COMPRESSION RATIOS.

μCode	Non Constrained Compression [22], [11], [4].		
	CR_{reg}	CR_{str}	Difference
A	49.54%	52.83%	3.30%
B	50.46%	58.93%	8.47%
C	52.70%	61.29%	8.59%
D	63.03%	85.65%	22.62%

The last column of Table II shows the difference between the regular and the structure-constrained compression ratio. Notice that, in microcode D, the structure-constrained compression ratio is more than 22% higher than the regular compression ratio. Again, this difference is due to the irregular size of the dictionaries. In the best regular compression result, microcode D was compressed using nine dictionaries, where the second dictionary had the biggest number of columns (33), and the third one had the biggest number of patterns $(1,728)$. The structure-constrained compression ratio was computed using memory blocks with $33 \times 1,728$ bits for the dictionaries. Table III shows the dictionaries configuration for the microcode D compression.

Table III
DICTIONARIES SIZES FOR THE MICROCODE D COMPRESSION.

Dictionary	# Columns	# Patterns
1	30	1018
2	**33**	923
3	20	**1728**
4	18	501
5	18	968
6	30	995
7	20	999
8	32	863
9	14	940
Uncompressed Columns	21	–

Table IV shows the structure-constrained compression ratios computed from our new SC^2 algorithm when constraining the number of columns per dictionary. The respective regular compression ratios and the number of columns per dictionary are also shown. For example, microcode A was compressed using 2 dictionaries, each containing 32 columns, and the difference between the regular compression ratio and the structure-constrained compression ratio is only 0.52%.

Notice that the SC^2 algorithm does not limit the number of patterns in each dictionary. However, the results at Table IV show that constraining the number of columns in each dictionary is enough to generate dictionaries with similar number of patterns. In fact, the small difference between CR_{reg} and CR_{str} indicates that the numbers of patterns in the dictionaries are similar. Also, notice that the structure-constrained compression ratio achieved by the SC^2 algorithm is consistently better than those produced by the existing algorithms (Table II).

Table IV
BEST STRUCTURE-CONSTRAINED AND CORRESPONDING REGULAR COMPRESSION RATIOS FOR THE SC^2 ALGORITHM.

μCode	# Dicts.	# Cols per Dict.	SC^2 Algorithm		
			CR_{reg}	CR_{str}	Diff.
A	2	32	51.15%	51.67%	0.52%
B	3	24	51.12%	51.15%	0.03%
C	8	26	53.66%	55.07%	1.42%
D	9	22	66.33%	66.50%	0.17%

We define the "structure-constrained compression cost" as the difference between the best structure-constrained compression ratio (CR_{str}) and the best regular compression ratio (CR_{reg}). This cost indicates the extra area required to compress the microcode when using the same hardware block design to implement the different dictionaries. Table V shows the compression ratios for the non-constrained and SC^2 algorithms and the structure-constrained compression cost for each microcode.

Table V
STRUCTURE-CONSTRAINED COMPRESSION COST.

μCode	Non Constrained Algorithms [22], [11], [4].		SC^2 Algorithm		Cost
	CR_{reg}	CR_{str}	CR_{reg}	CR_{str}	
A	49.54%	52.83%	51.15%	51.67%	2.13%
B	50.46%	58.93%	51.12%	51.15%	0.69%
C	52.70%	61.29%	53.66%	55.07%	2.37%
D	63.03%	85.65%	66.33%	66.50%	3.47%

These results indicate that the cost of structure-constrain is small. Therefore, besides making the dictionaries similar, the SC^2 algorithm can also produce good compression ratios.

VI. CONCLUSIONS

In this paper we investigated the reuse of a single hardware building block layout to implement different dictionaries in the microcode compression. This approach reduces the cost and the design time of the decompression engine. We also introduced the SC^2 algorithm, a new structure-constrained column clustering algorithm to improve the similarity of the dictionaries when compressing the microcode. The SC^2 algorithm significantly reduces the area expansion when reusing hardware building blocks to implement different dictionaries. Our experimental results show that the SC^2 algorithm is able to produce similar sized dictionaries without compromising the microcode compression ratio.

REFERENCES

[1] Agrawala, A., and Rauscher, T. Microprogramming: Perspective and status. IEEE Transactions on Computers, C-23(8):817-837, 1974.

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

[3] Beszdes, ., Ferenc, R., Gyimthy, T., Dolenc, A., and Karsisto, K. Survey of code-size reduction methods. ACM Computing Surveys, 35(3):223-267, 2003.

[4] Borin, E., Breternitz, M. J., Wu, Y., and Araujo, G. Clustering-Based Microcode Compression. In ICCD'06. p. 189-196, 2006.

[5] Borin, E. Microcode Compression Algorithms (in portuguese). PhD. Thesis, Unicamp, Brazil, 2007.

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

[7] Dasgupta, S. The organization of microprogram stores. ACM Computing Surveys, 11(1):39-65, 1979.

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

[9] Gunter, T., and Tredennick, H. Two-Level control store for microprogrammed data processor. U.S. Patent n. 4,325,121, 1982.

[10] Hum, H., Breternitz, M., Wu, Y. and Kim, S. Compressing microcode. U.S. Patent n. 7,095,342, 2006.

[11] Ishiura, N. and Yamaguchi, M. Instruction code compression for application specific VLIW processors based on automatic field partitioning. In the 7th Workshop on Synthesis and System Integration of Mixed technologies. p. 105-109, 1997.

[12] Lefurgy, C., Bird, P., Chen, I.-C., and Mudge, T. Improving code density using compression techniques. In Proceedings of MICRO-30, p. 194-230, 1997.

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

[14] Leighton, T., Makedon, F. and Tragoudas, S. G. Approximation Algorithms for VLSI partition problems. In Proceedings of ISCAS90. p. 2865-2868, 1990.

[15] Menzilcioglu, O. A case study in using two-level control stores. In Proceedings of MICRO-20. p. 142-146, 1987.

[16] Schwartz, S. An algorithm for minimizing read only memories for machine control. In Conference Record of 1968 9th Annual Symposium on Switching and Automata Theory. P. 28-33, 1968.

[17] Stritter, S., and Tredennick, N. Microprogrammed implementation of a single chip microprocessor. In MICRO-11. p. 8-16, 1978.

[18] Tredennick, H., and Gunter, T. Microprogrammed control apparatus having a two-level control store for data processors. U.S. Patent n. 4,307,445, 1981.

[19] Wilkes, M. The best way to design an automatic calculating machine. In Manchester University Computer Inaugural Conference. p. 16-18, 1951.

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

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

[22] Zhao, W., and Papachristou, C. Architectural partitioning of control memory for application specific programmable processors. In ICCAD'95. p. 521-26, 1995.