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 ($\mu$ROM) 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 (SC²) 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 \( \mu \)ROM has \( M \) unique microinstructions, maximal encoding [7] replaces each microinstruction in the \( \mu \)ROM by a \( \lceil \log_2 M \rceil \) bit codeword.

A decoding logic is attached to the output bits of the \( \mu \)ROM and whenever a codeword is fetched, the decoding logic converts it into the original microinstruction bits. The decoding logic, added after the compressed \( \mu \)ROM, 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 \( \mu \)ROM and the nanoROM. The \( \mu \)ROM contains microinstructions, which are pointers to nainoinstructions stored into the nanoROM. The nanoinstructions contain the data path control bits. Menzilcioglu [15] improved the two-level control store by using multiple nanoROMs 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 nanoROMs. Zhao and Papachristou [22] used a similar compressing model and proposed a heuristic to identify and group microcode fields into the nanoROMs.

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 nanoROMs, and the compressed microcode, containing pointers to the tables, was stored into the \( \mu \)ROM. 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 \( \mu \)ROM.

In this work we use a compression scheme similar to the two-level control store organization. However, instead of \( \mu \)ROM and nanoROMs, 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, \( \mu \)Addr is the address of a microinstruction. In the uncompressed form, the \( \mu \)Addr directly access the \( \mu \)ROM 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 \( \mu \)ROM has \( N \) microinstructions each with \( L \) bits, and there are a total of \( M \) unique microinstructions. The original \( \mu \)ROM takes \( N \times L \) bits and the compressed \( \mu \)ROM 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 \( \mu \)ROM uses 1,140,000 bits while the original \( \mu \)ROM uses 1,400,000 bits. This is approximately 19% reduction in bits. Note: in this discussion we use the number of bits in the \( \mu \)ROM as an estimate for its area requirements. Borin et al. [4] presented experimental results from layout estimates showing that reductions in actual \( \mu \)ROM size are in line with this estimate.

Notice that the “pointer array” and the original \( \mu \)ROM have the same number of entries (\( N \)). This means that the
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 in the partitions is reduced. The idea is to group columns that are similar into clusters and go beyond the 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 $10 \times (2 + 2) + 3 \times 3 = 58$ bits, less than 4% reduction from the original 60 bits size.

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 $10 \times (2 + 2) + 3 \times 3 = 58$ bits, less than 4% reduction from the original 60 bits size.

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 $10 \times (1 + 1) + 2 \times 3 = 32$ bits, nearly 50% reduction, significantly better than the basic partitioning 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.
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 $\mu$ROM and always fetch data using the same $\mu$Addr, 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.

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 $1024 \times 32$ bits using two blocks of $512 \times 32$ 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.

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

\[
CR_{str} = \frac{\text{Pointer Array Size} + \text{Dictionaries Size}}{54 \times 2900} = 61\%
\]

While the regular compression ratio is:

\[
CR_{reg} = \frac{\text{Pointer Array Size} + \text{Dictionaries Size}}{500 \times 34} = 42\%
\]

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 \(\mu\)ROM;
- \(N\): 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;
- \(L_1, L_2, \ldots, L_K\): the number of columns in each cluster 1, 2, \ldots, \(K\);
- \(M_1, M_2, \ldots, M_K\): the number of unique patterns in clusters 1, 2, \ldots, \(K\).

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:

\[
F = \frac{\text{Uncomp.Cols}}{N \times L_0} + \sum_{i=1}^{K} \frac{K}{\text{PointerArray}} + \sum_{i=1}^{K} \frac{\text{Dictionary}}{\text{HW Block size}}
\]

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:

\[
F_{str} = \frac{\text{Uncomp.Cols}}{N \times L_0} + \sum_{i=1}^{K} \frac{K}{\text{PointerArray}} + \sum_{i=1}^{K} \frac{\text{Dictionary}}{\text{HW Block size}}
\]

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.

\section*{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: \( \{2, 1\}, \{2, 2\}, \{2, 3\}, \ldots, \{2, 40\}, \{3, 1\}, \ldots, \{3, 26\}, \{4, 1\}, \ldots, \{10, 8\} \). 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: \( \text{BestClustering} \leftarrow \text{ naïveClustering}(\text{Constraints}) \)
2: repeat
3: \( \text{CurrentC} \leftarrow \text{BestClustering} \)
4: \( \text{improved} \leftarrow \text{false} \)
5: unlock(\( \{c_1, c_2, \ldots, c_L\} \))
6: repeat
7: \( \text{op} \leftarrow \text{selectBestOperation}() \)
8: if \( \text{op.type} = \text{EXCHANGE} \) then
9: \( \text{exchangeColumns (op.c_1, op.c_2)} \)
10: \( \text{lock (\{op.c_1\})} \)
11: else if \( \text{op.type} = \text{MOVE} \) then
12: \( \text{moveColumn (op.c_1, op.D_2)} \)
13: \( \text{lock (\{op.c_1\})} \)
14: end if
15: if \( F_{str}(\text{CurrentC}) < F_{str}(\text{BestClustering}) \) then
16: \( \text{BestClustering} \leftarrow \text{CurrentC} \)
17: \( \text{improved} \leftarrow \text{true} \)
18: end if
19: until there are no more valid operations
20: until \( \text{improved} \neq \text{true} \)

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

The kernel of the algorithm, comprised by the inner-most 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>
<thead>
<tr>
<th>( \mu \text{Code} )</th>
<th># Cols</th>
<th># Lines</th>
<th>Size in bits</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>75</td>
<td>22,528</td>
<td>1,689,600</td>
<td>Desktop</td>
</tr>
<tr>
<td>B</td>
<td>234</td>
<td>6,656</td>
<td>1,557,504</td>
<td>Laptop</td>
</tr>
<tr>
<td>C</td>
<td>236</td>
<td>5,632</td>
<td>1,329,152</td>
<td>Mobile</td>
</tr>
<tr>
<td>D</td>
<td>240</td>
<td>5,632</td>
<td>1,351,680</td>
<td></td>
</tr>
</tbody>
</table>
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 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).

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.

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

### Table II

Best regular compression ratio and the respective structure-constrained compression ratios.

<table>
<thead>
<tr>
<th>µCode</th>
<th>Non Constrained Compression</th>
<th>$CR_{reg}$</th>
<th>$CR_{str}$</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>49.54%</td>
<td>52.83%</td>
<td>3.30%</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>50.46%</td>
<td>58.93%</td>
<td>8.47%</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>52.70%</td>
<td>61.29%</td>
<td>8.59%</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>63.03%</td>
<td>85.65%</td>
<td>22.62%</td>
<td></td>
</tr>
</tbody>
</table>

### Table III

Dictionaries sizes for the microcode $D$ compression.

<table>
<thead>
<tr>
<th>Dictionary</th>
<th># Columns</th>
<th># Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>1018</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>923</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>1728</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>501</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>968</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>995</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>999</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>863</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>940</td>
</tr>
</tbody>
</table>

### Table IV

Best structure-constrained and corresponding regular compression ratios for the $SC^2$ algorithm.

<table>
<thead>
<tr>
<th>µCode</th>
<th># Dicts.</th>
<th># Cols per Dict.</th>
<th>$SC^2$ Algorithm</th>
<th>$CR_{reg}$</th>
<th>$CR_{str}$</th>
<th>Diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>32</td>
<td>51.15%</td>
<td>51.67%</td>
<td>0.52%</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>24</td>
<td>51.12%</td>
<td>51.15%</td>
<td>0.03%</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>20</td>
<td>53.66%</td>
<td>55.07%</td>
<td>1.42%</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>9</td>
<td>22</td>
<td>66.33%</td>
<td>66.50%</td>
<td>0.17%</td>
<td></td>
</tr>
</tbody>
</table>

### Table V

Structure-constrained compression cost.

<table>
<thead>
<tr>
<th>µCode</th>
<th>Non Constrained Algorithms</th>
<th>$SC^2$ Algorithm</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$CR_{reg}$</td>
<td>$CR_{str}$</td>
<td>$CR_{reg}$</td>
</tr>
<tr>
<td>A</td>
<td>49.54%</td>
<td>52.83%</td>
<td>51.15%</td>
</tr>
<tr>
<td>B</td>
<td>50.46%</td>
<td>58.93%</td>
<td>51.12%</td>
</tr>
<tr>
<td>C</td>
<td>52.70%</td>
<td>61.29%</td>
<td>53.66%</td>
</tr>
<tr>
<td>D</td>
<td>63.03%</td>
<td>85.65%</td>
<td>66.33%</td>
</tr>
</tbody>
</table>

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


